Programming as an Interactive Experience

Craig Innes, Student Number: S0929508

MInf Project (Part 2) Report
Master of Informatics
School of Informatics
University of Edinburgh

2014

Abstract

This project explores the feasibility of creating programming tools in Haskell which update their result **immediately** when the user makes a change. Such tools would allow programmers to alter a program's source code and instantly perceive the effects of their changes without saving, compiling, and running the program themselves. Such a system could also provide **two-way** interaction by letting the programmer manipulate the *result* of a running program. Such manipulations should lead automatically and instantly to the corresponding changes in the source code.

The main contribution of this project was the development of a programming tool for simple 2D graphics and animation which embodies the above principles. The tool was written in Haskell, for Haskell programs. Implementing the system in a purely functional language provides the opportunity to use techniques such as higher-order functions. Additionally properties of the language, such as the absence of side-effects, simplifies particular problems while making others more challenging. The project also includes a 12 person user evaluation of the above tool to assess the usefulness of such interactive programming tools.

Table of Contents

1 Introductory Synopsis **5**
1.1 Main Inspiration 5
1.2 Project Goals 7

2 Related Work **11**
2.1 Plugging Haskell In 11
2.2 Other "Inventing on Principle" Implementation attempts 12
2.2.1 Don Kirkby's LivePy 12
2.2.2 Gabriel Florit's LiveCoding.io 14
2.3 Live Coding 15
2.4 Haskell School of Expression 16
2.5 Bi-directional Transformations and Lenses 16
2.6 Continuous Feedback GUIs 17

3 Previous work carried out **19**
3.1 The SOE Graphics Language 20
3.2 The Compiler, UI, and Render Threads 22
3.3 Tracking Where Regions were Created 23
3.4 Updating the Source by Manipulating the Image 26
3.5 Limitations of Updating 28

4 This Year's Implementation Work **33**
4.1 Architecture Improvements 33
4.1.1 Compilation and Rendering Threads 34
4.1.2 Offering feedback on failure 37
4.2 Wider range of transformations 38
4.2.1 Rotation and Scaling Operations 38
4.2.2 Providing more tracking information 39
4.2.3 Updating Combined Regions 42
4.2.4 Avoiding Overwriting Important Information 44
4.2.5 Unsupported Operations 45
4.3 Animation 46
4.3.1 Support for Basic Animation 46
4.3.2 Immediate Updates at the Correct Point in Time 47
4.3.3 Onion Skinning 48
4.3.4 Time Manipulation 50

5 Evaluation **51**
5.1 Comparison against the "Gold Standard" 52
5.2 User Evaluation 54
5.2.1 Task Description 54
5.2.2 Results and Analysis 57
5.2.3 User Evaluation Conclusions 69

6 Conclusion **71**
6.1 Project Summary 71
6.2 Ideas for Future Work 72

7 Appendix **73**
7.1 User Evaluation Task Script 73
7.2 User Evaluation "Cheat Sheet" 74
7.3 User Evaluation Template Files 77
7.3.1 TreeExercise.hs 77
7.3.2 Figure8Exercise.hs 77
7.3.3 SolarSystemExercise.hs 78

Bibliography **79**

Chapter 1

Introductory Synopsis

The system created for this project was a development environment for programming simple graphics and animations using Haskell. The system embodies the main principles of *immediate feedback* and *two-way interaction*. This initial chapter outlines the main inspiration, goals and success criteria for the project. The following chapters then set the context for this year's work by discussing related work and the implementation details of work done in the first year.

1.1 Main Inspiration

The idea for this project came from a talk by Bret Victor titled "Inventing on Principle" [1] in which he demonstrates five ideas for interactive programming environments. The principle in question, which he argues all tools should embody, is that "Creators should have an immediate connection to what they create".

One of the examples he presents is a JavaScript graphics program. The left side of the screen shows a JavaScript drawing canvas with a picture of a tree drawn on it. The right side of the screen shows the JavaScript source code used to render the tree. Several features of this demonstration show how the principle of providing the user with an immediate connection between their source code and its result could work:

- The moment a user makes a change to the JavaScript source code, the drawing of the tree is updated.
- Hovering over a part of the tree in the canvas (e.g the trunk) highlights in red the line of source code which produced that shape (and vice versa).
- Hovering over a numeric value in the source code produces a *range slider* which can be moved up and down to allow the user to experiment with how changes in that value change the result.
- Hovering over a hexadecimal value produces a colour swatch, allowing the user to see which colour that hexadecimal value represents and giving them the opportunity to pick similar colours.

- The user can directly manipulate parts of the tree (resizing a branch for example), which causes the system to immediately update the source code in order to reflect this change.

- As the user is typing functions or objects, the program attemps to guess what the user is going to type, offers a suggestion as to the probable functions they might want, and provides an immediate preview of what the new tree would look like if they picked the most likely suggestion.

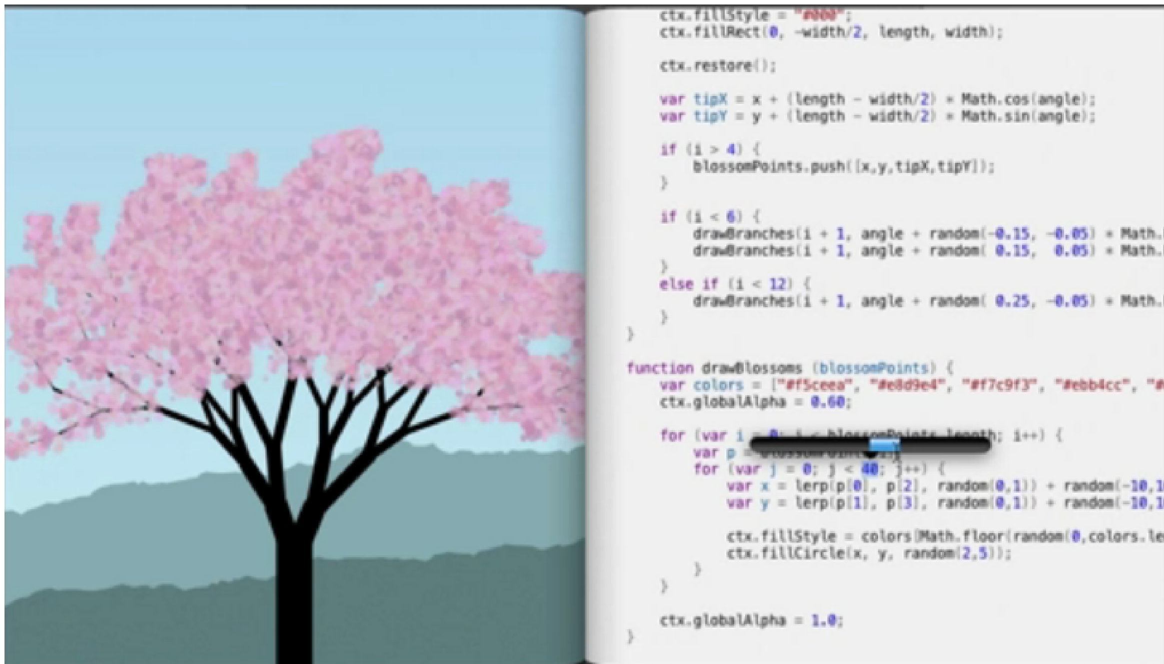


Figure 1.1: Screenshot of the Bret Victor demonstration. On the right of the screen is some JavaScript graphics code to render a tree. On the left of the screen is a JavaScript canvas which shows the result of executing the graphics code. Here, the user is altering a numeric variable in a loop counter, and watching the tree picture change in response

Other examples in the talk show similar principles of immediate feedback applied to the domains of animation and games.

It is not clear whether the examples presented by Bret Victor were part of a fully implemented system, or merely superficial toy examples of what such a system *could* be capable of. In particular - and unusually for him - no source code or technical explanation for the demonstration has been released [2].

Part of the drive of this project was therefore to demonstrate whether it was possible to implement a fully working version of such an environment in Haskell, and how feasible each of the features introduced in the graphics and animation demonstrations actually were.

1.2 Project Goals

The main aim of this project was to explore the technical feasibility and usefulness of such immediate feedback programming systems in Haskell by implementing one for simple graphics and animation.

The were two ways in which the final implemented system was assessed:

- Taking the ideal system presented in the Bret Victor demonstration as a "Gold Standard", and comparing the features in the final Haskell implementation against the list of features in this ideal system.
- Conducting a *User Evaluation* to gauge the effectiveness of the system when used to complete simple tasks.

At the beginning of the project, I identified the following list of key features from the Bret Victor "Gold Standard" system:

- Changes made to the program code are immediately reflected in the graphical result of a program
- There is a direct connection between the source code and the graphical result which it produces - hovering over a shape in the graphic will highlight the line in the source code which is responsible for the creation of that shape and vice versa
- Source code can be changed in a variety of ways which encourage experimentation with values. For example, numeric values in the code may be associated with slider widgets which allow the user to sweep across a range of values, or colour values may be associated with a colour palette.
- Features which current Integrated Development Environments (IDEs) provide should be augmented to provide immediate feedback. For example, auto-complete suggestions for functions should show a preview of what the graphical result would look like if the given suggestion were applied.
- The user is able to directly manipulate the graphical result in a number of ways. This includes translation, scaling, rotation, changing the colour or texture of a shape, re-arranging the z-ordering of shapes etc.
- Changes made via direct manipulation of the graphical result should be immediately reflected in the source code. In other words, the source code should update to reflect the new state of the graphical result.

In the second year of the project, the scope of the system was extended to include support for basic animations. Bret Victor's second demonstration for a hypothetical immediate feedback game editor outlines key features that an ideal system should support for animation:

- Changes to the source should be immediately reflected in the resulting animation, but with the additional constraint that the animation should be able to con-

tinue running without interruption from the point at which the change to the source code was made.

- The user should have the ability to see how their changes affect the entire span of the animation immediately. One way of accomplishing this may be to have a single image view of the whole animation which displays the trajectory the objects will take throughout the duration of the animation.
- The user should be able to directly manipulate the animation. The unique component which is present in animation as compared to a static graphic is the passage of time, so such manipulations should revolve around changing, reversing and pausing time.

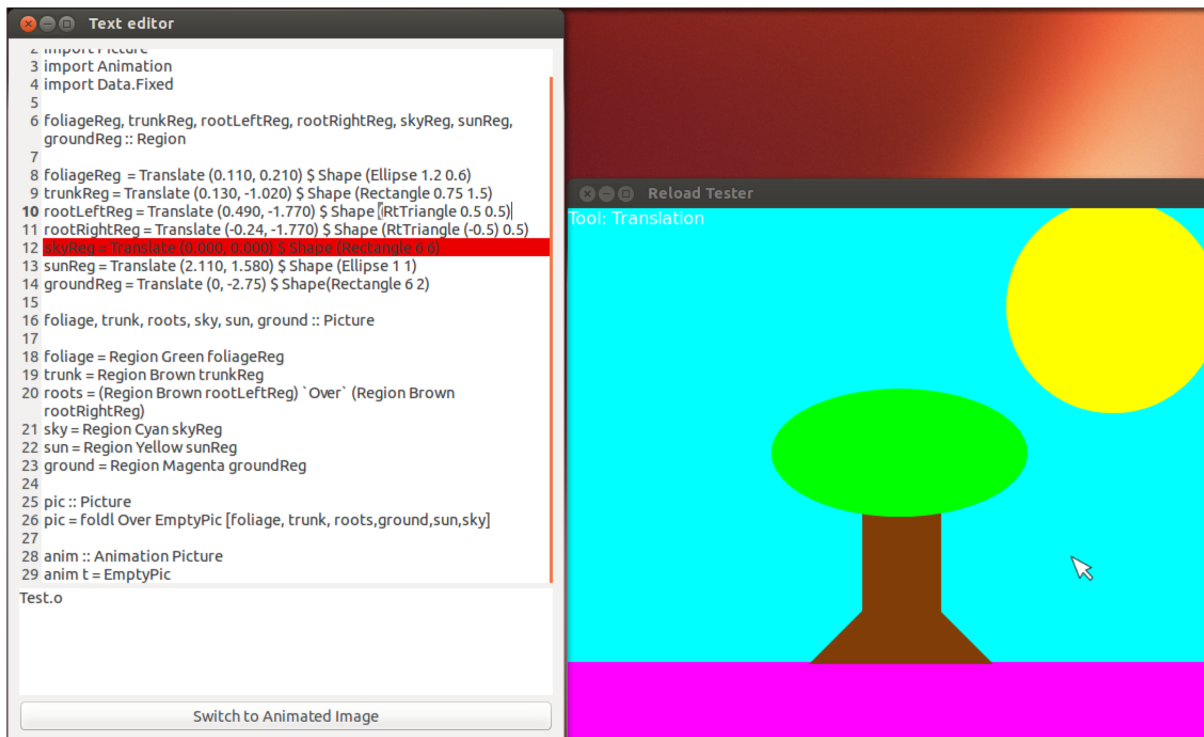


Figure 1.2: Screenshot of this project's finished system. The left side of the screen shows Haskell code for drawing a tree scene. The right side shows the resulting image. The user is currently hovering over the sky, so the system has highlighted the line which constructed the sky in red

These lists were used to assess the final project implementation by seeing how closely the final implementation matches the specification of our ideal system in terms of the number of listed features that it actually supports and, for the features I did manage to implement, how well they perform based on the following evaluation criteria:

Evaluation Criteria

- How *"immediate"* is the feedback (i.e. How fast is it?)
- Does the implemented solution provide robust / accurate results? (Are users able to break the system through normal / antagonistic use)?

- Is the solution *useful*? In other words, are tasks completed faster with the system than without? Does the system provide a deeper understanding of the source code? Does the system encourage experimental creation?
- Is the system easy to use?

While some of these criteria can be assessed by simply inspecting the final system, or by comparing it to alternative approaches, many cannot. For this reason, a user evaluation was carried out to assess criteria such as the usability of the system. This user evaluation and its results are discussed later in the report.

Chapter 2

Related Work

The following sections set the context for understanding the implementation details and the value of the system developed as part of this project. Some sections discuss techniques or tools which were directly used as part of this project, others discuss ideas which were used as inspiration, while others discuss alternative implementations and techniques which were not used as part of this project.

2.1 Plugging Haskell In

A substantial part of this project involves instantly re-loading changes which the user makes to their source code while the system is still running. A solution to this problem is proposed in Don Stewart's paper: "Plugging Haskell In" [3].

In this paper, Stewart argues that applications such as Emacs have become popular in part because they allow users to easily extend their functionality. They way they do this is by having a small "core" architecture which stays static, and allowing the user to "plug in" new functionality by dynamically loading in new components. Stewart argues that such systems are valuable because the user does not have to re-compile or even understand the source code of the entire system each time they want to make a change, they only have to understand the small piece of functionality that they want to add in.

The paper goes on to explain the implementation details of the Haskell library "hs-plugins", which allows you to do the following things:

- Dynamically compile a Haskell file from inside another running Haskell program.
- Load the newly compiled Haskell code into your currently running program. On susequent changes to the Haskell code, the system is smart enough to *only* reload the parts which have changed. It will not reload every single module each time.
- Extract a particular value/function from the newly compiled/loaded Haskell file and "plug in" that value/function into the appropriate place in your currently

running program.

As explained later, this library was used in the implementation of this project's graphics programming tool in order to reload changes the user makes to their source code.

2.2 Other "Inventing on Principle" Implementation attempts

During the 2 years that this project has been in development, a number of similar projects which also attempt to implement some of the immediate feedback features discussed in Bret Victor's Inventing on Principle [1] talk have begun to appear. As far as I am aware however, no other tool exists specifically for Haskell programs, or indeed any purely functional language. Additionally, none of these tools make any attempt to provide *two-way* interaction between the source code and the result.

2.2.1 Don Kirkby's LivePy

Don Kirkby's LivePy [5] project aims to bring the principle of "Immediate Feedback" to Python. It consists of two parts: A live "turtle" drawing library, and a plugin for Emacs and Eclipse.

The live "turtle" module allows you to draw basic lines and shapes using the popular "turtle" [6] library. It differs from using the regular library in that there is no need to save and manually re-interpret your Python code. The LivePy system immediately updates the resulting picture to reflect any changes made to the Python turtle drawing source code.

The emacs/eclipse plugin is intended for more general purpose programming. When turned on, the plugin will automatically reinterpret and run your python source code any time you make a change. Not only that, but it will print out the value of every variable mentioned in the source code.

LivePy is similar to this project's Haskell system in that both provide immediate feedback when the user makes a change to their source code. However, the Haskell system provides many features which LivePy does not. For example:

- LivePy does not provide any form of *two-way* interaction (i.e It is not possible to interact with the *result* of a python script). In fact, the *only* method of interaction LivePy provides is to manually edit the source code.
- While LivePy makes some attempt to provide information on the connection between the source code the user has written and the resulting output, this information is limited (printing the values of variables at each step).
- LivePy only provides support for static graphics using the turtle module. It does not support animations.

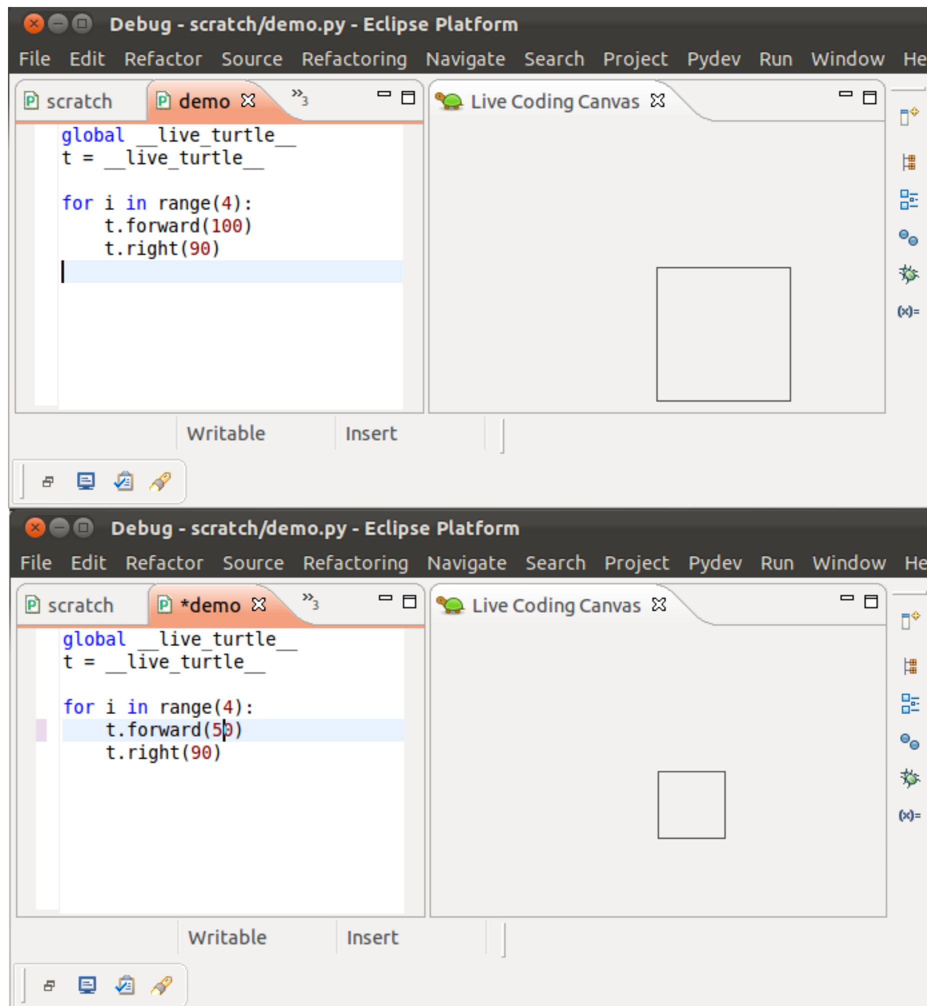


Figure 2.1: Don Kirkby’s LivePy Turtle drawing module. Here we see the user make a change to the size of the square which is reflected immediately in the resulting image

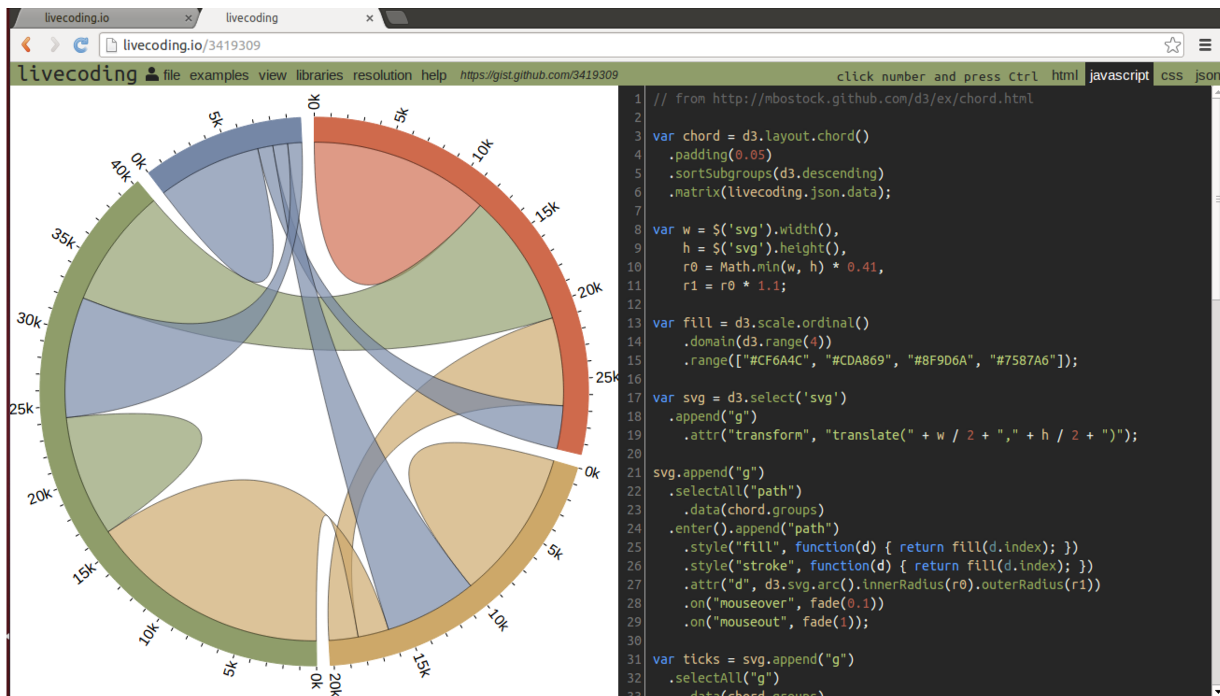


Figure 2.2: Gabriel Florit's LiveCoding.io

The ways in which the Haskell system implements the features lacking in the LivePy project are discussed later in the report.

2.2.2 Gabriel Florit's LiveCoding.io

Gabriel Florit's LiveCoding.io [7] application is in many ways a more fully realised product than both LivePy and this project's Haskell system.

LiveCoding.io is an online, browser-based editor for JavaScript. In a similar vein to the Bret Victor demonstration, it consists of a JavaScript source code editor on one side of the screen, and a JavaScript drawing canvas on the other side to display the resulting compiled code. It has the following key features:

- When a change is made to the JavaScript source code, the canvas updates immediately *without* reloading the page.
- Popular JavaScript libraries can be loaded into the editor with ease.
- Popup sliders are provided for altering for numerical variables and hexadecimal color strings.
- Snippets of code written using the tool can be shared with friends.

However, while this is indeed a polished application, its focus differs from this project's Haskell system in many ways. LiveCoding.io provides no methods for *two-way* interaction, provides no information on the connection between the source code and its resulting output, and while theoretically it is possible to write JavaScript for animations

Figure 2.3: Screenshot of a live visual performance using the "Fluxus" language

using the tool, there is no attempt made to "keep the system running". In other words, if you create an animation using LiveCoding.io, then change the colour of the animation, the entire animation will be recompiled and reset, whereas it would be preferable to have the changed animation continue from where the previous animation left off.

Again, the way in which the Haskell's features differ will become clearer when discussing the implementation details.

2.3 Live Coding

"Live coding" is a discipline which shares some of the immediate feedback goals of this project. The discipline is concerned with finding methods of dynamically generating music and visuals from code as it is being written for the sake of performance art. [8]

Many domain specific languages have come out of this community, such as Fluxus [9], Overtone [10], and ChucK [11]. These languages and compilers for these languages are focused on making strong guarantees about the maximum amount of time between making a change to your source code and having the resulting change carry through to the resulting audio/visuals. In other words, they are "strongly timed".

For this project, the focus was on providing a wide range of interactions in two directions for a more general programming language (Haskell). Additionally, providing guarantees on the immediacy of the feedback was not a concern as long as it was "good

enough" that it was not noticeable by the user. For these reasons, a domain-specific live coding language was not used in this project.

2.4 Haskell School of Expression

Paul Hudak's book "The Haskell School of Expression" [12] is an introductory text to the Haskell language, but also provides useful information on implementing basic shape generation and manipulation in a functional style.

The graphics library used in the book, named *SOEGraphics* [13] provides a framework for drawing and displaying basic shape primitives on screen in Haskell. Its source code was adapted and extended for use in this project.

As well as making use of the associated graphics library. The book also introduced several concepts that were later used in the project's implementation. One of the main concepts used was the idea of treating animations as first class functions which took a time value and produced a corresponding picture as output.

The book also briefly explores the paradigm of *functional reactive programming* (FRP). FRP functions take a stream of events or time varying signals as their input. While such techniques were not explored in the implementation of this project's system, extending the system to use FRP may make for interesting future work.

2.5 Bi-directional Transformations and Lenses

Research in the field of bi-directional transformations involves finding transformations which express both a way to get from a specific input to a specific output, and a way to go in reverse from that specific output back to the input [14]. Lenses are pieces of code which define a function mapping inputs to outputs when read left to right, but define a function mapping outputs to inputs when read right to left.

The main application of bi-directional transformations in current research has been in manipulating structured data such as database records or XML. Here researchers are largely concerned with the "View Update Problem" : If we have some abstract view onto a data set, and make a change to the abstract view, how do we propagate those changes back to the original underlying concrete data structure? Libraries such as Benjamin Pierce's *Boomerang* [15] attempt to solve this problem.

Since this project is in part concerned with providing two-way interaction between source code and its resulting graphical output, it is feasible that research in this area could be applicable if we were to think of the graphical output as an "abstract view" of the source code. Work in this area may help resolve some of the ambiguities identified in section 3.5 ("Limitations of Updating").

2.6 Continuous Feedback GUIs

The paper "It's Alive! Continuous Feedback in UI Programming" by Sebastian Burckhardt describes a language for creating imperative programs where the user interface can be updated on-the-fly. The key feature of this language is that code pertaining to the User Interface (the "View") is explicitly kept separate from all the other code/state in the program.

When a change is made to the User Interface code, the system treats this change as an event which causes it to save the state of the rest of the program, throw out the old user interface, and reload the new user interface while keeping the state of the rest of the program intact.

This paper describes a similar approach to immediate feedback which was used in this project's Haskell system - namely that of having a static core system and plugging in the part of the system which you want to change.

The paper dedicates a lot of space to describing how they guarantee that changes can be made to the User Interface while keeping the state of the rest of the running program intact. Our system sidesteps this problem by being written in a purely functional language. In other words, there is no mutable state to keep track of in the first place.

Chapter 3

Previous work carried out

In the first year of the project, much of the foundational work for the implementation was done in order to provide the groundwork for further expansion in the second year.

At the end of the first year I had a bare-bones environment for writing simple Haskell graphics programs with immediate feedback in two directions: changing the source code to make the result change, and changing the result to make the source code change.

The first year implementation had the following key features:

- A GUI built using the *Haskell GTK* Library. This allowed users to write Haskell programs and provided some basic text editor functionality such as line numbering and bracket matching.
- A drawing canvas for displaying graphics rendered using the SOE Library [13] (from the book "Haskell School of Expression")
- A multi-threaded system which had three separate threads for responding to user input, rendering graphics to the screen, and periodically checking if the user's source code needed to be dynamically recompiled.
- The ability to track which line of code was responsible for the construction of a particular shape.
- Support for interaction with the resulting image which allowed the user to move the shapes around on screen (translation), and have the source code update to reflect that translation operation.

What follows is a brief summary of the implementation details of some of the features present in the first year of the project. This will provide information on the key decisions and limitations faced during the development of these features. Additionally, this knowledge will help put into context the further extensions which were implemented in the second year.

Figure 3.1: Screenshot of system at the end of the first year of development

3.1 The SOE Graphics Language

Users of this project's system use the Haskell graphics library SOEGraphics [13] to write code which draws simple graphics onto the canvas. Images in SOEGraphics are built up in 3 steps: Creating a *Shape*, creating a *Region*, then creating a *Picture*.

The first step is to create a *Shape* value. Shapes represent basic geometric primitives such as a Rectangle, Ellipse or Triangle and are constructed in the following way:

```
s1 :: Shape
s1 = Rectangle 3 2
```

This code creates a rectangle with a width of 3 and a height of 2. It has type Shape and the name s1.

The next step is to construct a Region using an existing Shape. Regions are constructed using the constructor "Shape":

```
r1 :: Region
r1 = Shape s1
```

This code takes our previous Shape value s1 and uses it as a parameter to construct a new value r1 of type Region. Regions differ from Shapes in two ways. Firstly, regions can be transformed by various wrappers such as Translate, Scale, and Rotate. For example, the following code creates a value which is 2 units to the right of r1 and 1 unit above it:

```
translatedRegion :: Region
translatedRegion = Translate (2,1) r1
```

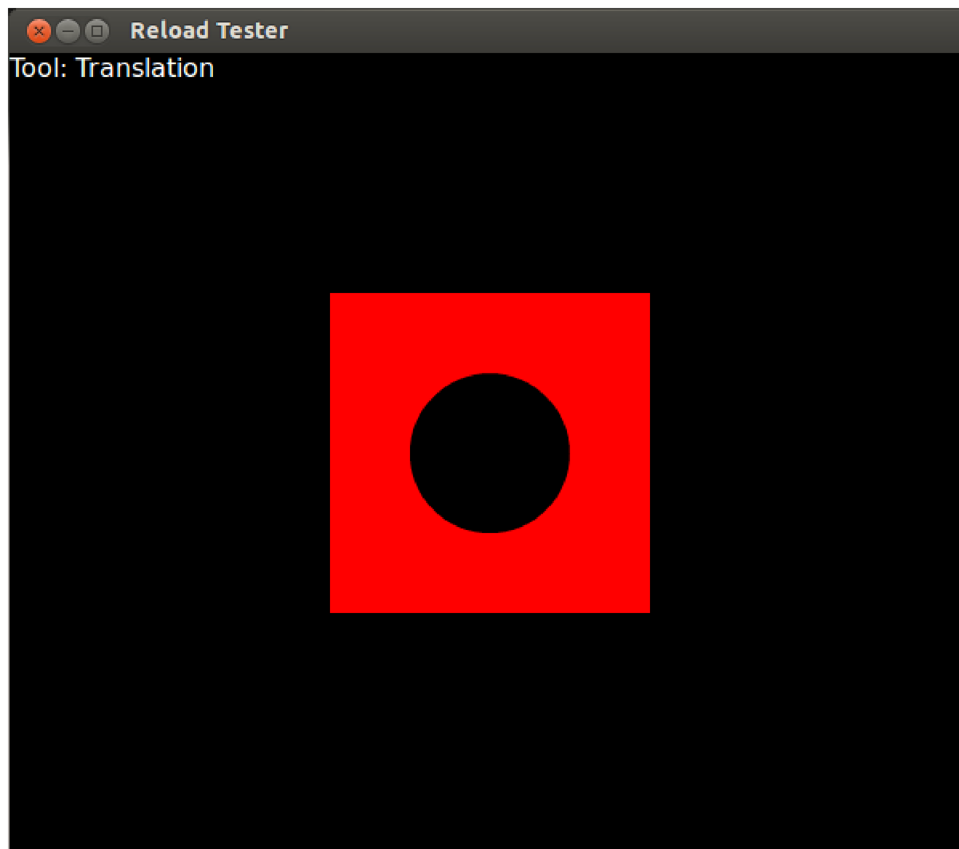


Figure 3.2: Picture of what “squareWithHole” would look like

The second way in which Regions differ from Shapes is that multiple Regions can be combined through various combination operations to create new Regions. Such combination operations are “Union”, “Xor”, and “Intersection”. For example, the following code creates a region representing a square with a circular hole cut out of the centre:

```
smallCircle :: Region
smallCircle = Shape (Ellipse 1 1)

largeSquare :: Region
largeSquare = Shape (Rectangle 4 4)

squareWithHole :: Region
squareWithHole = largeSquare `Xor` smallCircle
```

The final step in creating an image is to take a Region and use it to construct a Picture. A Picture is just a Region with a colour assigned to it. Pictures are constructed using the keyword “Region”:

```
redTriangle :: Picture
redTriangle = Region Red (Shape (RtTriangle 1 2))

blueCircle :: Picture
blueCircle = Region Blue (Shape (Ellipse 1.5 1.5))
```

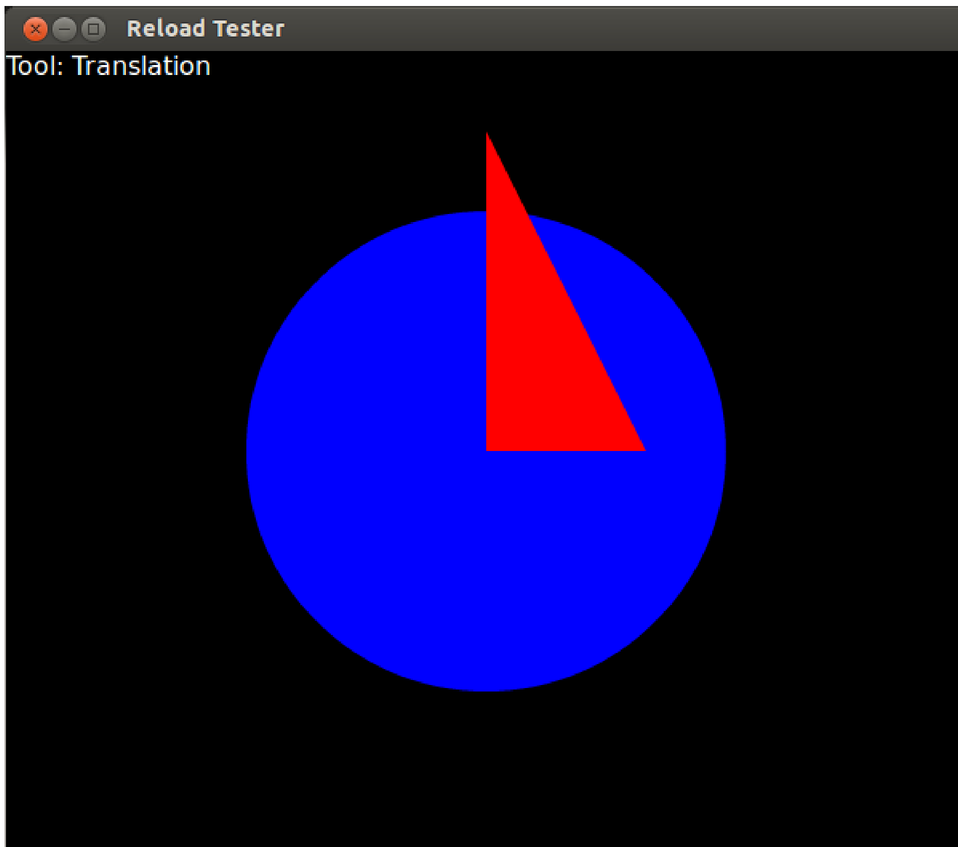


Figure 3.3: redTriangle over blueCircle

Pictures can then be displayed on the canvas by the system. Simlarly to Regions, multiple Pictures can be combined using the “Over” constructor:

```
finalPicture :: Picture
finalPicture = redTriangle `Over` blueCircle
```

Several other Haskell graphics libraries exist, such as Cairo [16] and Diagrams [17], but SOEGraphics was chosen for this project for two main reasons. Firstly, SOEGraphics consists of only a small number of shape primitives and operations, making it easier to support and extend than more complex libraries such as Diagrams. Secondly, it provides a simple layer of abstraction for constructing and transforming shapes, unlike libraries like Cairo which operate on a much lower level of abstraction.

3.2 The Compiler, UI, and Render Threads

The overall structure of this project's system is split into three threads: a rendering thread, a thread for responding to user input to the GUI, and a compile loop thread.

The user input thread is mostly used to listen for keyboard input and allow the user to write/edit source code in the left-hand window.

The rendering thread is responsible for continually rendering the user's picture to screen, and also handles user input if the user tries to directly manipulate the graphics on the canvas.

The compile loop thread is responsible for managing changes to the user's source code. At regular short intervals, this thread checks the source code text buffer to see if it has changed since the last time it checked. If it has, it will take steps to change what is being drawn on the canvas to reflect the newly changed source code. It does this by saving the text buffer to a file, then using the hs-plugins [4] library (mentioned in the "Related Work" section above) to dynamically re-compile and load the user's newly changed source code from within the running system. It then kills the current rendering thread and replaces it with a new rendering thread which has been set up to render the picture from the newly re-compiled source code instead of the previous version.

This was the state of the architecture at the end of the first year of implementation, and it suffered from several inadequacies. One problem was that the overhead of creating and destroying rendering threads each time a change was made incurred a noticeable delay in displaying the graphics. This meant that the time to get feedback on some changes to the user's source code would occasionally be a second or so longer than what could really be considered "immediate". Another problem was that because the rendering thread was essentially being interrupted and destroyed during the middle of computation, careful coding was required to avoid the risk of undefined behaviour. These problems were fixed by improvements in the second year of development.

3.3 Tracking Where Regions were Created

In order for certain features to work - such as highlighting the line of code responsible for displaying a particular shape, or being able to update the source code when the user moved a shape - the system needed to be given certain pieces of "tracking information". This was done by augmenting the source code with extra information before compilation in a way that is invisible to the user.

Before the system re-compiles and re-renders the user's source code, it first searches through the code for every occurrence of the "Shape" keyword (the keyword which is used to construct a Region), and replaces it with a `TrackedShape <lineNumber>`. This constructor is identical to "Shape", but encodes information about the line number on which it was written directly into the data type. As an example, if we had the following code on lines 34 and 35:

```
r1 :: Region
r1 = Shape (Ellipse 2 3)
```

our pre-processor would convert it to:

```
r1 :: Region
r1 = TrackedShape 35 (Ellipse 1 0.5)
```

Now that the system has this information, if a user hovers over a particular region, the

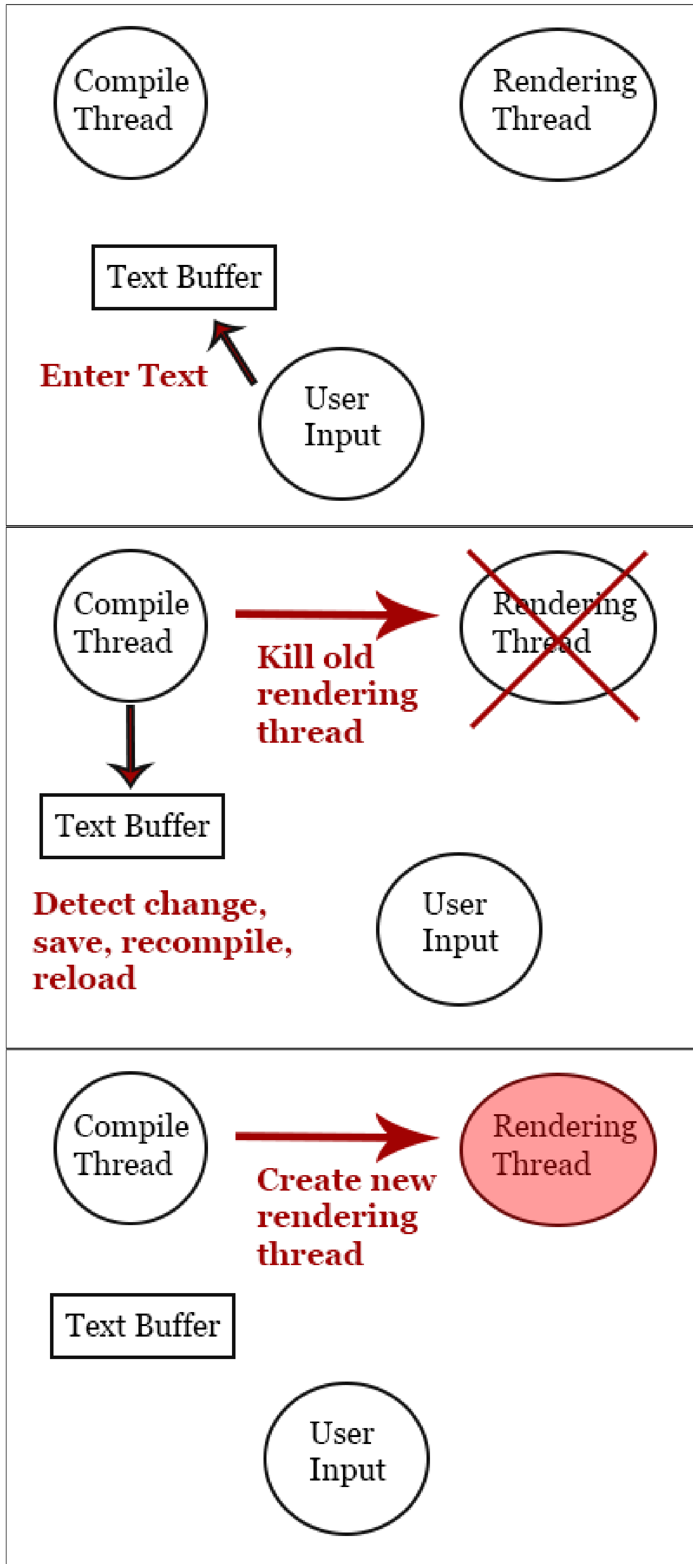


Figure 3.4: Interaction between threads upon the user changing the source code.

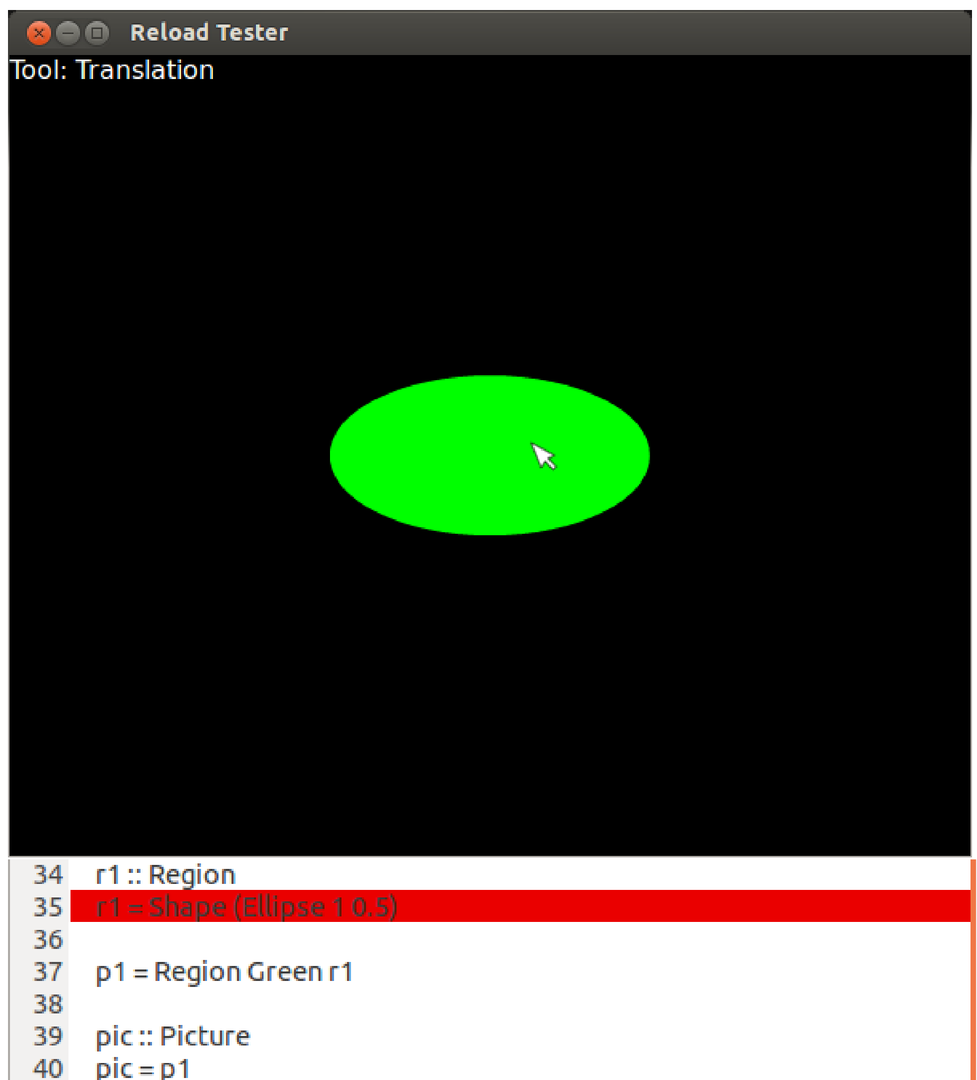


Figure 3.5: User hovers over Ellipse on screen. System uses tracking information to highlight the line on which it was created

system can inspect the data, extract the line number in the source where that region was created, and use this information to highlight that line in the text editor.

Note that the definition of which line is "responsible" for creating a particular part of the graphic is subject to debate. Instead of defining the point at which a Shape is turned into a Region as the line "responsible", I could have defined it as the point at which a Shape primitive is first mentioned ("Ellipse 2 3" for example). I could have equally defined the point of responsibility as the point at which a Region is locked down into a Picture, or the point at which the Picture is finally drawn to the screen. There were two reasons why Region construction was chosen as the point of responsibility. Firstly it is a point at which there is some stability - we know the actual geometry of the shape and its dimensions. Secondly and most importantly, Regions are the point at which transformations are applied, so tracking this point in particular makes updating the source code easier when the user manipulates the image.

3.4 Updating the Source by Manipulating the Image

In order to have *two-way* immediate feedback, the source code must change when the user manipulates the rendered image. Simple regular expression pattern matching and the tracking information explained in the previous section helps accomplish this.

As an example, if the user clicks an image of a red circle and drags it three units to the left, the following will happen. First, the image will change in response to the user moving the circle. Next, the system will extract the line number information from the corresponding tracked shape, then try to find a match on this line for the "Shape" keyword. It will then replace that line with a new one which reflects the user's change. If the original code looked like this:

```
exampleRegion :: Region
exampleRegion = Shape (Ellipse 1 1)

examplePicture :: Picture
examplePicture = Region Red exampleRegion
```

Then after the user has dragged the circle 3 units to the left, the code will be updated to look like this:

```
exampleRegion :: Region
exampleRegion = Translate (-3, 0) $ Shape (Ellipse 1 1)

examplePicture :: Picture
examplePicture = Region Red exampleRegion
```

Additionally, the system will attempt to compress multiple similar transformations into one, so if the user were to then move the circle one unit to the right, then 2 units up, the altered line would then be updated to:

```
exampleRegion = Translate (-2, 2) $ Shape (Ellipse 1 1)
```

Again, there are multiple alternative ways in which the source code could have been updated to reflect changes made by the user. For example, instead of having Translation "wrapper" operations, we could have instead encoded information about the Region's position directly into the Shape, then had the user's transformations directly overwrite these parameters. For example, to specify a circle at the centre of the screen (0,0), we could have had some notation like:

```
exampleRegion = Shape (Ellipse 0 0 4 4)
```

which our previous manipulations would have changed to:

```
exampleRegion = Shape (Ellipse -2 2 4 4)
```

The problem with this approach is that if the user had specified the value of such a parameter to be the result of applying a complicated function, then it is again unclear what to do.

```
exampleRegion = Shape (Ellipse (strangeCalculation 4 2) 0 4 4)
```

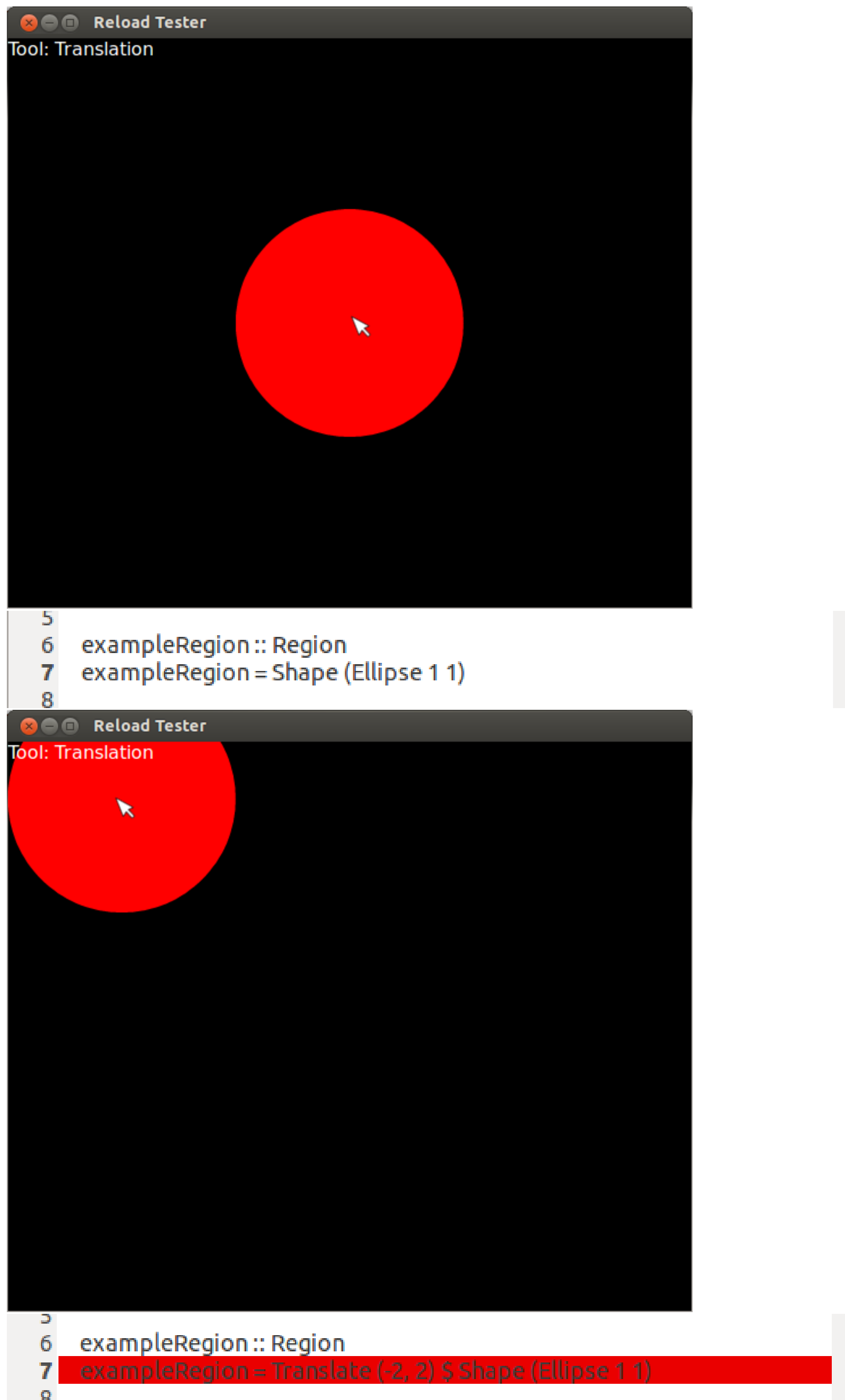


Figure 3.6: Direct manipulation example. Circle is originally at position (0,0). User drags circle 2 units left and 2 units up, and source code updates in response to this

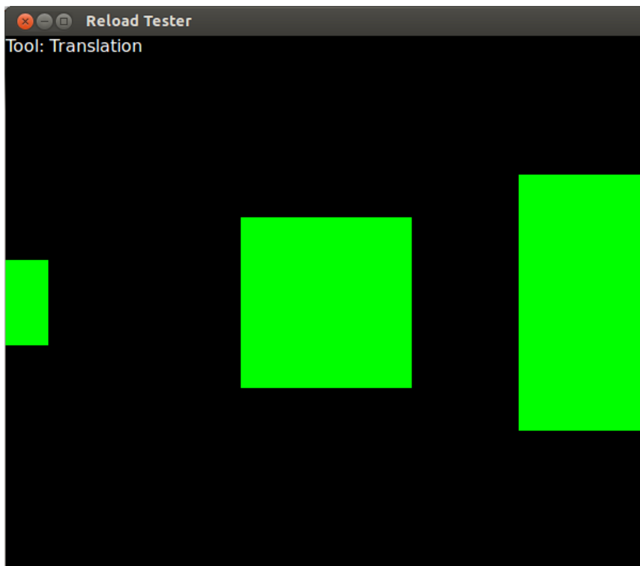


Figure 3.7: The “makeSquare” example. If user drags middle square down, the correct place to apply an update in the user’s source code is unclear

Should we now attempt to change the parameters to this function to get the desired result, or just replace this function application with a literal? The original method at least makes it explicit that we are simply adding a translation wrapper on to what the user has already written.

3.5 Limitations of Updating

The above method of updating the source code works correctly for the majority of cases. However, for a range of more complicated expressions, the result might be different from what the user expects. In these cases, the correct behaviour is not so clear.

The prime example of this is when a Region is created as the result of another function. Take the following code for example, which places three green squares of different sizes in a row:

```
p1, p2, p3 :: Picture
p1 = Region Green (Translate (-3, 0) (makeSquare 0.8))
p2 = Region Green (makeSquare 1.6)
p3 = Region Green (Translate (3, 0) (makeSquare 2.4))

finalPicture :: Picture
finalPicture = p1 `Over` p2 `Over` p3

makeSquare :: Float -> Region
makeSquare n = Shape (Rectangle n n)
```

If the user were to click on the middle square (p2) and drag it 2 units down, the user would probably want the source code to update just the position of the second square.

However, the actual change made is:

```
p1, p2, p3 :: Picture
p1 = Region Green (Translate (-3, 0) (makeSquare 2))
p2 = Region Green ((makeSquare 3))
p3 = Region Green (Translate (3, 0) (makeSquare 4))

finalPicture :: Picture
finalPicture = p1 `Over` p2 `Over` p3

makeSquare :: Int -> Region
makeSquare n = Translate (0, -2) $ Shape (Rectangle n n)
```

The change is made at the point where the region is constructed - in the “makeSquare” function. This has the effect of moving every square made using the “makeSquare” function. One could argue that, given the choice to build p1, p2, and p3 using the common function “makeSquare”, that this is actually the correct result. However, it is more likely that the user intended something like this:

```
p1, p2, p3 :: Picture
p1 = Region Green (Translate (-3, 0) (makeSquare 0.8))
p2 = Region Green (Translate (0, -2) $ (makeSquare 1.6))
p3 = Region Green (Translate (3, 0) (makeSquare 2.4))

finalPicture :: Picture
finalPicture = p1 `Over` p2 `Over` p3

makeSquare :: Float -> Region
makeSquare n = Shape (Rectangle n n)
```

For more complicated expressions involving higher-order functions, the correct update to apply is even less clear. Take the following code for example:

```
higherOrderPic :: Picture
higherOrderPic = foldr Over EmptyPic $
                   map (\x -> (Region Blue).Shape $ x)
                     [Rectangle x x | x <- [1,2,3]]
```

How should the following code be altered if the user attempts to move the second blue rectangle down two units? The system updates the code in the following way:

```
higherOrderPic :: Picture
higherOrderPic = foldr Over EmptyPic $
                   map (\x -> (Region Blue).Translate (0, -2) $
                      Shape $ x)
                     [Rectangle x x | x <- [1,2,3]]
```

However, it is more likely that the following more accurately represents the user’s intentions:

```
partialHigherOrderPic :: Picture
partialHigherOrderPic = foldr Over EmptyPic $
                   map (\x -> (Region Blue).Translate (0, -2) $
                      Shape $ x)
                     [Rectangle x x | x <- [1,2]]
```

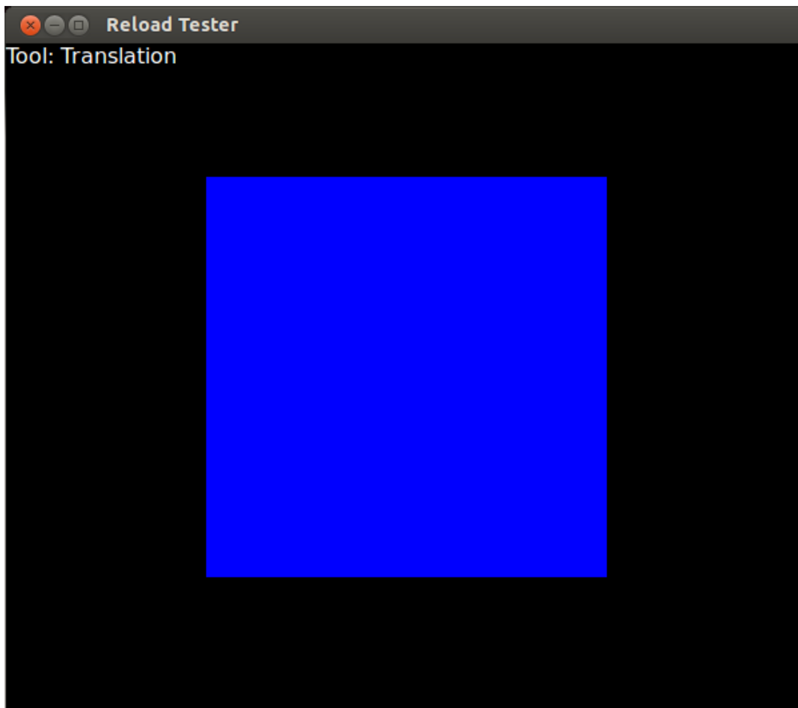


Figure 3.8: The "higherOrderPic" example. If the user directly manipulates a this image, it is unclear in general what update should be applied to the source code

```
alteredRectangle :: Picture
alteredRectangle = Region Blue (Translate (0, -2) (Shape (Rectangle
    3
3)))

higherOrderPic :: Picture
higherOrderPic = partialHigherOrderPic `Over` alteredRectangle
```

Even for this relatively small expression involving higher order functions, a large amount of code restructuring is required in order to accurately reflect the user's changes. In practice, such higher order functions could be composed in an arbitrary number of ways. The point is that the "correct" way to perform source code updates in the general case is unclear.

For the second year of the project, the behaviour of the source-code update mechanism was not improved upon. I chose instead to focus on other features such as extending the system to support animation. For future work, there were several potential techniques and research areas which might be fruitful if someone were attempting to improve the way in which the system decides which parts of the code are responsible for a particular part of the image:

- **Use type information**: More accurate insights about where updates should be placed might be gleaned if Haskell's extensive type information were used. Images in this project are built up in layers, each of which has a different type (Shapes graduate to become Regions which then eventually become Pictures). It is possible that this information could be leveraged so that transformation wrap-

pers could be placed at the exact point at which they are required.

- **Bi-directional transformations** [14]: As mentioned in the related work section, these relate two views of the same information and specify how a change in one would effect the other. If such a mapping could be discovered between a subset of Haskell source and their resulting images, then one would have a foundation on which to built robust two-way interactions.

- **Program slicing**: Research in this area deals with finding techniques for isolating parts of a program which affect specific values. This could be useful for finding out which parts of the source code are likely to be responsible for affecting particular parts of the overall image. While most of the research on program slicing focuses on imperative languages, there has been a small amount of research into using such techniques for functional languages [18]

Chapter 4

This Year's Implementation Work

At the beginning of the second year, most of the core functionality for a simple programming environment for *immediate*, *two-way* graphical interaction for 2D images was in place. The focus of the second year was on honing the existing implementation into a more polished product, on extending its functionality into other areas, and on evaluation of the finished product.

My key contributions in the second year of the project were as follows:

- Improvements to the architecture of the system delivered in the first year to increase robustness and speed.
- Providing a wider range of operations which support both *two-way* and *immediate feedback* interaction. In the first year, only Translation was fully supported, but in the final implementation Translation, Scaling, Rotation, and combinator operations such as Xor are supported. This required doing more sophisticated parsing so that the way source-code was updated could be improved.
- Expansion of the scope of the system to support not only graphics, but also simple animations.
- A 12 person user-evaluation to assess the effectiveness of the system.

The largest and most significant of these contributions was the expansion of the system to support simple animations. This was done in iterations which delivered incrementally more useful features to support animation.

The following sections explain the implementation of each of the contributions above in detail, highlighting the key decisions made, the benefits and limitations of the chosen approach, and some alternative implementation techniques that were considered.

4.1 Architecture Improvements

The two main improvements made to the overall system structure were:

- Changing the interaction between the compilation and rendering threads in order to tackle the issues of speed and maintainability raised by the original implementation.
- Offering feedback to the user both when their source code successfully compiles *and* when it fails to compile

4.1.1 Compilation and Rendering Threads

As explained previously, the structure of the implementation at the end of the first year suffered from a few problems. The rendering thread was destroyed and re-created every time the system noticed a change in the user's source code. This had the effect of creating enough overhead that the time between the user making a change and observing the result was a few seconds - longer than what could really be considered "immediate" feedback. It also meant that, since the rendering thread could be interrupted and destroyed at any moment, the code for the rendering thread had to be carefully restricted to avoid undefined behaviour.

Since the setup and teardown of the rendering thread upon every recompile was causing some slowdown and maintenance problems, the immediate solution appeared to be to find a way to keep all three threads running uninterrupted. Implementing such a solution was simple upon realising that the only part of the rendering thread that was actually changing was the picture it rendered.

Instead of the compile thread setting up an entirely new rendering thread upon recompiling and reloading the user's code, the system now uses the *hsPlugins* [4] library's "load" function to dynamically load *just* the final picture value "pic" from the compiled object code of the user's source code:

```
getModulePic :: IO (Either [String] (Module, Picture))
getModulePic = do
               mv <- load "UserImageCode.o" ["."] [] "pic"
               case mv of
                    LoadFailure messages -> return (Left messages)
                    LoadSuccess x y -> return (Right (x, y))
```

If the load is successful (i.e there exists a value called "pic" in the user's source code and it is of type Picture) then this value is passed into a shared data structure which can be read atomically by both the compile loop thread, and the rendering thread. The rendering thread is free to run without interruption, and simply reads new picture values to render as they become available.

With this setup, the time from the user making a change to their source code to the result appearing on screen is usually a fraction of a second, and the code for the rendering thread is now much simpler to write.

One alternative fix that I had considered was that perhaps the problem of the user's result not changing fast enough in response to their changes was simply because I was not checking for changes fast enough. This turned out to be false:

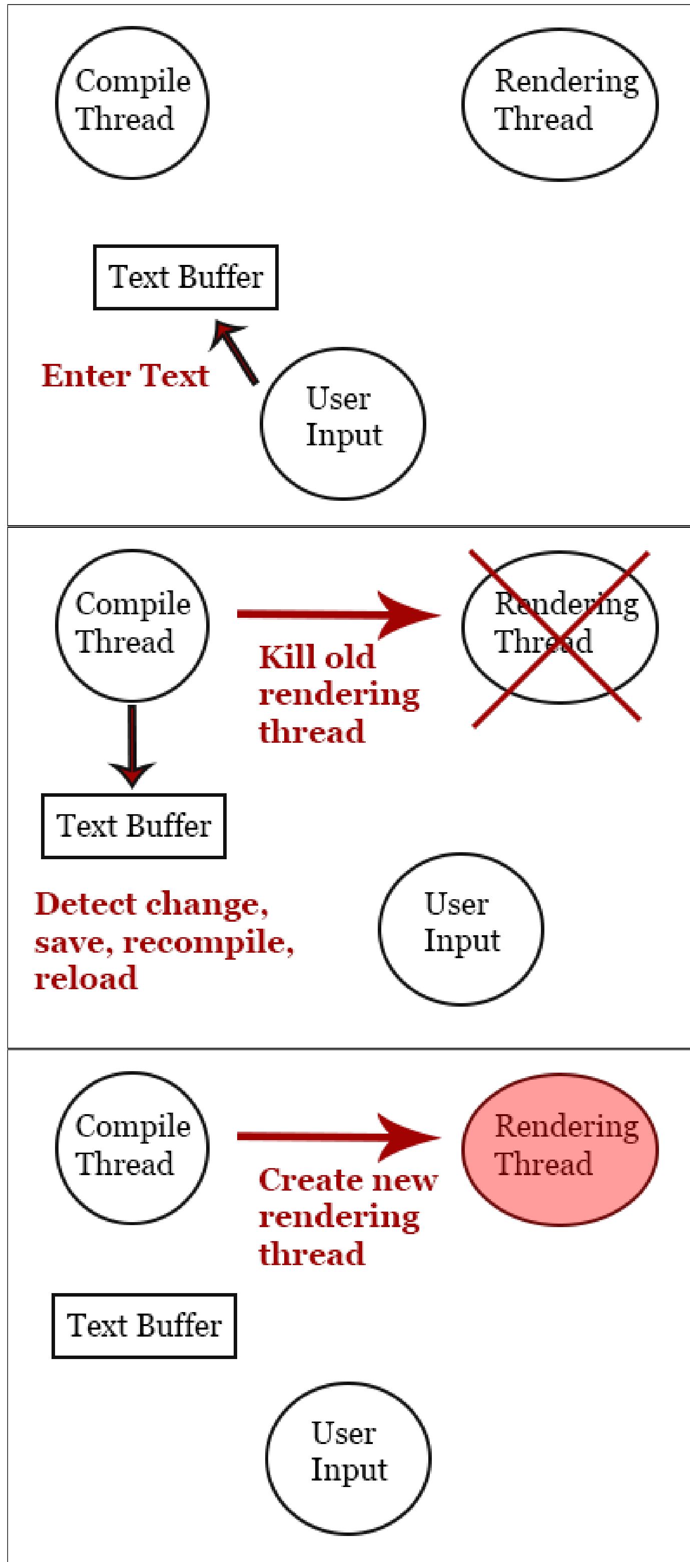


Figure 4.1: Old way in which threads interacted upon source code change

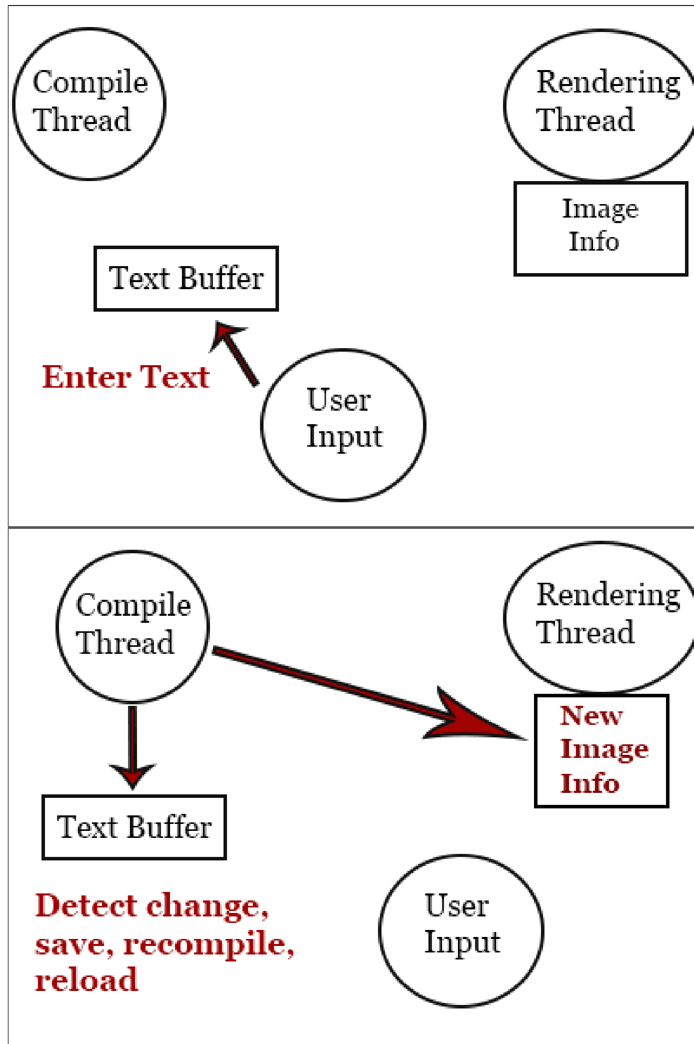


Figure 4.2: New way in which threads interact

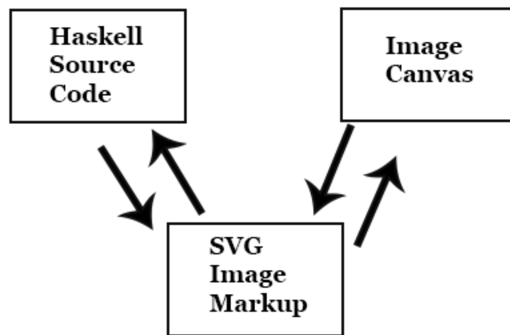


Figure 4.3: Alternative system architecture

Originally, the compilation thread checked for changes to the user's code every 30 milliseconds. I tried two alternatives. The first was to cut this interval down to 10 milliseconds, and the second was to have the compilation thread not check the text buffer at all, but instead trigger a recompile event every time the user entered a keystroke. Both of these methods made no noticeable difference to the result update times but had the side effect of starving the user interface thread. This meant that typing in the system became sluggish and unresponsive for no tangible benefit.

Other completely different architectures for the system were considered early in development. One example of this was a structure in which, instead of the user's code being directly rendered, the code was used to generate an intermediate textual representation of the graphic such as an SVG [19](XML markup for graphics) file. In such a system, the source code and displayed graphic would not interact with each other directly, but rather by reading and manipulating some intermediate structured text file.

As this approach was not implemented, it is unclear if it would help with the issues outlined previously in this report, but a clear disadvantage of this approach is that there is no longer a *direct* connection between the user's changes to their source code and the result (and vice versa). Instead, there is now some middle representation whose mapping to the original source and result may be somewhat unclear.

4.1.2 Offering feedback on failure

While offering immediate feedback on *correct* code, the implementation of the system from first year would fail silently if the user wrote code which caused a compile or load error. Since the user perceived no change to the resulting image upon writing incorrect code, and was given no further information, it was difficult for them to identify what mistake they had made, or if they had even made a mistake at all.

Since the dynamic compiler within the system generates diagnostic information about compiler and load errors anyway, an information box was implemented in the user's editor which would display this information immediately upon noticing their mistake. This way, the user could get immediate feedback on incorrect code they had written,

```
19  p5 = (Region Red r5) `Over` (Region Red r6)
20
21  pic :: Picture
22  pic = foldl Over EmptyPic [p1,p2,p3,p4]
23
24  a1, a2 :: Animation Picture
25  a1 t
26    | (t `mod'` 2) > 1 = Region Red (Rotate t $ Shape $ Rectangle 1 1)
27    | otherwise = Region Blue (Rotate t $ Shape $ Rectangle 1 1)
28
29  a2 t = Region Green (Translate ((cos t), 0) $ Shape (Ellipse 0.2 0.2))
 `Over` p5
30
31  anim :: Animation Picture
32  anim = a1
33
34  makeGreenSquare :: Float -> Picture
35  makeGreenSquare n = Region Green $ (Shape (Rectangle n n))
36
Test.hs:18:27: Not in scope: `loadOfNonsense'
```

Figure 4.4: Newly added error output at bottom of editor gives immediate feedback on source code errors

making it easier to correct mistakes.

4.2 Wider range of transformations

A number of improvements were made to the range of *two-way* interactions available in the implementation:

- Expanding the ways in which a user could directly manipulate an image by allowing them to Scale and Rotate regions as well as Translating them.
- Increasing the amount of tracking information added to regions in the pre-processing step to allow for more accurate updates.
- Providing accurate result-to-source-code updates on *Combinations* of regions (Xor, Union, Intersection, etc).
- Improving the updates the system does to the user's source code to prevent important information from being overwritten.

4.2.1 Rotation and Scaling Operations

While the SOE graphics library used as part of this project supported a "Scale" operation, the user had to write it explicitly themselves to make use of it. There was

no functionality in place to apply a Scale operation to a Region in response to direct manipulation of the image.

I implemented this in the system by creating a function which, when the user clicked and dragged a region on screen, computed the difference between the original mouse position and the current dragged mouse position, and used this to calculate the correct scaling factor to apply to the source code.

For rotation, the SOE graphics library did not even start off with a concept of what it meant to rotate a shape. Before I could even start providing a way for the user to rotate an image by directly manipulating regions with the mouse, I had to first add a "Rotate" operation into the source code of the underlying graphics library. This involved creating a "Rotate" constructor in a similar vein to the Translate and Scale operations:

```
Rotate <angle-in-radians> <region-to-rotate>
```

then telling the underlying library to perform the appropriate matrix operations for rotation when it encountered the Rotate constructor upon rendering.

I was then able to provide the user with a way to Rotate regions directly. I was able to do this by providing a function which, upon clicking and dragging a region, would calculate the angle between two vectors - the vector from the centre of the Region to the original mouse position, and the vector from the centre of the Region to the current mouse position - and use this to update the source code with the appropriate Rotation constructor.

The system now has three ways in which you can directly manipulate regions: Translation, Scaling, and Rotation. In order to prevent these operations from stacking up in the source code, the system attempts to compress multiple similar operations. For example, if you had a rotated square produced by writing the following code:

```
rotatedMovedSquare :: Region
rotatedMovedSquare = Translate (3,1) $ Rotate 0.6 (Shape (Rectangle
    2 2))
```

and then attempted to rotate the region by a further 0.5 radians, the system would not add in another Rotation wrapper, but would instead update the existing rotation:

```
rotatedMovedSquare :: Region
rotatedMovedSquare = Translate (3,1) $ (Rotate 1.1 (Shape (Rectangle
    2 2))
```

4.2.2 Providing more tracking information

In the first year system, the only tracking information that was added to a region in the pre-processing step was the line number upon which it was constructed. This has the obvious problem that if there were ever more than one "Shape" constructor mentioned on the same line, the functions which update the user's source code would not be able to differentiate between them.

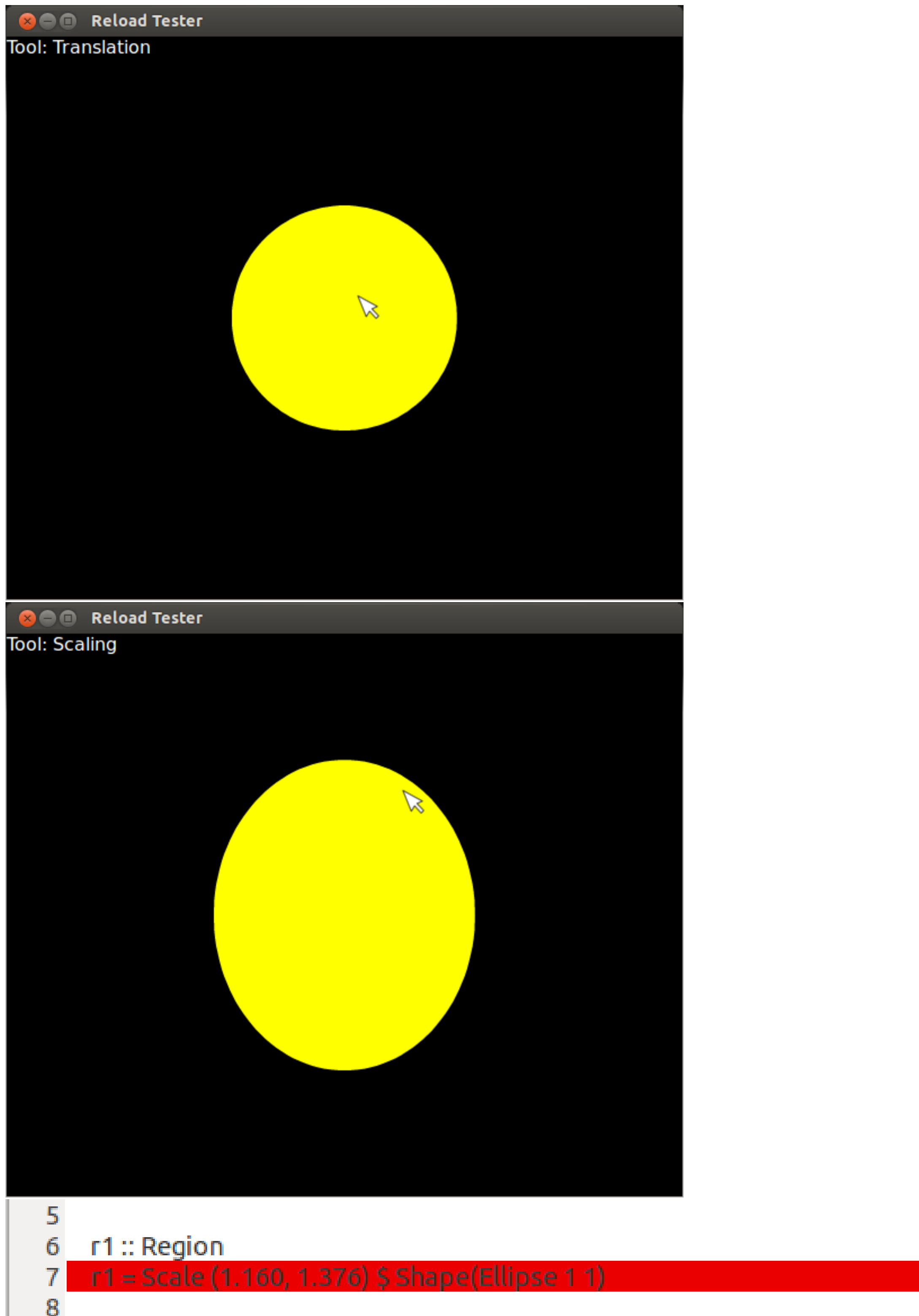


Figure 4.5: User clicks and drags to scale the circle on screen. The appropriate line in the source code updates to reflect this change

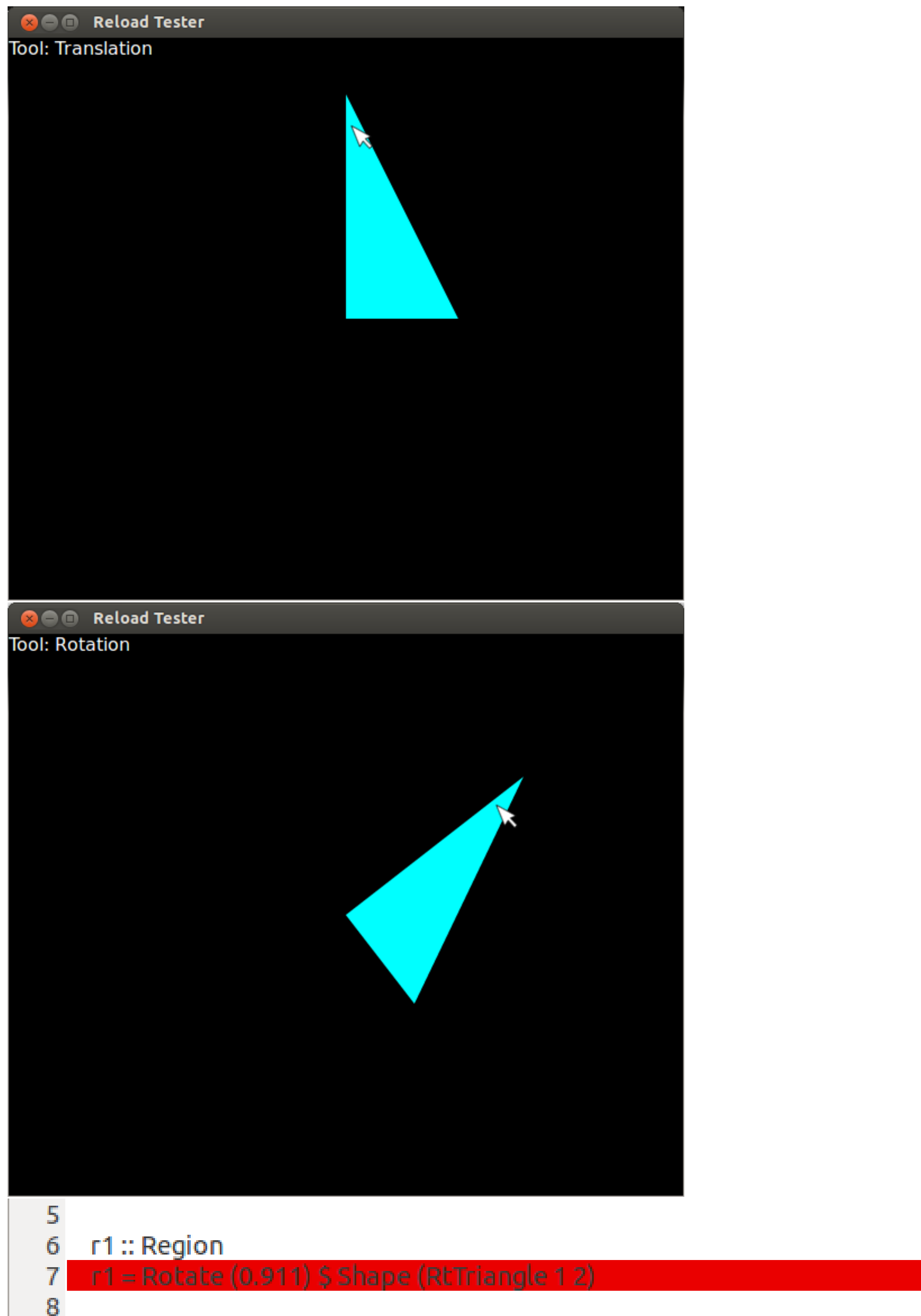


Figure 4.6: User clicks and drags to rotate the triangle on screen. The appropriate line in the source code updates to reflect this change

The new system not only tracks the line number on which a Region constructor appears, but also the order in which it appears on that line. For example, in the following code (assuming this is all written on the same line, 34):

```
OneLinePicture = (Region Blue (Shape (Ellipse 3 3))) 'Over' (Region
Red (Shape (RtTriangle 5 1)))
```

The pre-processor will convert this to:

```
OneLinePicture = (Region Blue (TrackedShape (34,1) (Ellipse 3 3)))
'Over' (Region Red (TrackedShape (34,2) (RtTriangle 5 1)))
```

Using this extra information, the system is able to update the user's source code in a more sophisticated way. Instead of just doing a blanket "find and replace" on the appopriate line, the System now uses a custom "Replace Nth occurrence" function. This function splits the line into a list of strings separated by occurrences of the "Shape" constructor. It then applies its update *only* to the element in the list which corresponds to the Region being manipulated. Once the update has been applied, the list of strings is then rejoined to form the correctly updated new line.

Now, if the user were to drag the blue circle in the above example 2 units to the left, we would get the following correct update:

```
OneLinePicture = (Region Blue (Translate (-2,0) $ Shape (Ellipse 3
3))) 'Over' (Region Red (Shape (RtTriangle 5 1)))
```

4.2.3 Updating Combined Regions

Combination operations are the ones which take two existing regions and combine them in some way to form a new region. Examples of combination operations are: "Xor", "Union", and "Intersection".

When a user attempts to manipulate a region which is part of a combined region, it is likely that they want to move the entire combination as if it were a single region. For example, if the user attempted to drag the following region two units down:

```
squareWithHole :: Region
squareWithHole = (Shape (Rectangle 1 1)) 'XOr' (Shape (Ellipse 0.25
   0.25))
```

the user would expect *both* the square and the circle which make it up to be translated two units down.

In order to do this, the system makes sure to apply the required transformation to both regions involved in the combination. However, either of these regions might be made up of a *further* combination of another two regions, so these changes have to be applied recursively.

Here is a snippet from the function which applies a rotation update which illustrates this:

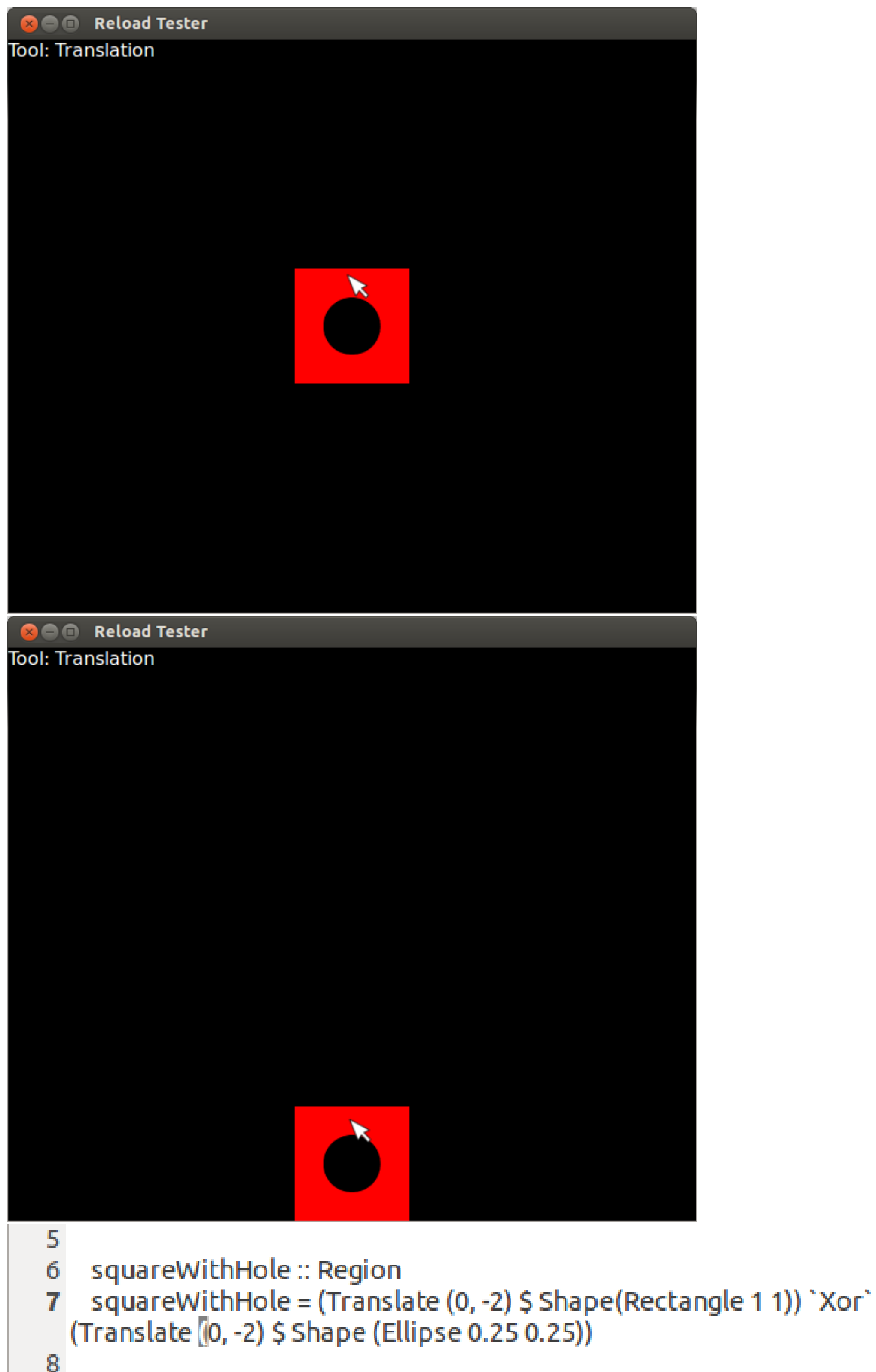


Figure 4.7: The “squareWithHole” example. User drags the region two units down and the system updates the source code to reflect this change

```
{- Code for handling the various other cases -}

-- Case which matches `Xor`
        (reg1 `Xor` reg2) -> do
            modifiedRegion1 <- rotateTrackedShape originalSrc sView
               angle reg1
            modifiedRegion2 <- rotateTrackedShape originalSrc sView
               angle reg2
            return (modifiedRegion1 `Xor` modifiedRegion2)
```

It is interesting to note here that if we did not have the extra tracking information to differentiate between different regions on the same line, then the updates applied to region combinations would be applied at least twice as many times as they needed to be.

4.2.4 Avoiding Overwriting Important Information

If the user has written a value in their source code which is the result of a complicated function call, then it is likely that they do not want subsequent direct manipulations to completely overwrite that code. Consider the following example code:

```
exampleSquare :: Region
exampleSquare = Translate ((sin (pi / 4)), (customFunction 5 3)) $
    Shape (Rectangle 2 2)
```

It is likely that the user's use of the sin function and their own custom function have some important meaning in the context of the program. If the user then drags the square 2 units to the right, it is more likely that this is meant as an *adjustment* to the information that they already have rather than a decision to completely re-position it. In other words, it would be risky for the system to completely destroy the information when it updates the source code, as this information may have been of value to the user:

```
exampleSquare :: Region
exampleSquare = Translate (2.707, 9) $ Shape (Rectangle 2 2)
```

Instead, the system now parses the arguments to the Transformation operations already attached to the Region before deciding how to update. If the arguments look like floating point numbers or ints, then the system will overwrite this information as normal. In other cases such as the one above, the system will leave this potentially important information alone and just add another Translation operation:

```
exampleSquare :: Region
exampleSquare = Translate ((sin (pi /4)), (customFunction 5 3)) $
Translate (2, 0) $ Shape (Rectangle 2 2)
```

Figure 4.8: Two pictures are identical except for having different Z-Orderings

4.2.5 Unsupported Operations

A two directional transformation which would be useful to have but which is not supported by the current system implementation is the reordering of the way in which pictures are layered on top of each other: in other words, changing the Z-Ordering of Pictures.

The reason this operation is not currently supported is that it would require more complicated and destructive updates to the source code than, for example, simply translating a region. To illustrate this difficulty, take the following example:

```
{-
  Definitions for regions r1..r4
-}

-- Picture definitions
  p1,p2,p3,p4,p5,p6, fullImage :: Picture
  p1 = Region Blue r1
  p2 = Region Red r2
  p3 = Region Green r3
  p4 = Region Yellow r4

  p5 = p1 `Over` p2
  p6 = p3 `Over` p4

  fullImage = p6 `Over` p5
```

Even for this relatively simple case, if we wanted to move p1 to be above p2 and p3 but below p4, then we really have to know how all the other Pictures relate to each other in order to rewrite the various ‘Over‘ operations.

One way to tackle this problem in future implementations might be to directly encode information about the Z-ordering of Pictures directly into the data as a number. For example, similar code written using this technique might look like this:

```
p1,p2,p3,p4, fullImage :: Picture
p1 = Region Blue 4 r1
p2 = Region Red 3 r2
p3 = Region Green 2 r3
p4 = Region Yellow 1 r4

fullImage = combineUsingZOrder [p1,p2,p3,p4]
```

If a user then wants to manipulate part of the image by moving a particular Picture forward, then the only update the system has to apply is to increment that Picture's Z-index.

4.3 Animation

The most significant extension of the system in the second year was to include support for simple animations. Implementation of this expansion was done in several phases, each of which included increasingly useful functionality:

1. Altering the system to render basic animations
2. Allowing the user to get immediate feedback in response to their changes to the animation code. This means showing the updated animation at the same point in time as immediately before the change was made.
3. Providing a way to view the entire trajectory of an animation in a single image (Onion skinning).
4. Providing a way for the user to directly interact with the result of the animation by manipulating time.

4.3.1 Support for Basic Animation

The first main implementation challenge was finding an appropriate way to represent animations in the system. In traditional imperative languages, you first create your renderable objects at the start of the program, then create an update function which runs every frame and updates the state of each object in accordance with how you want the animation to play out. For instance, the following code represents an animation of a circle moving to the right of the screen in an imperative language:

```
public void init(){
  circle = new Circle(0, 0, 2, 2, Color.RED);
}

--This runs every frame
public void update(){
 circle.x += 2;
}
```

In a functional language such as Haskell however, we do not have mutable state. We can simplify the above imperative code using the idea from "Haskell School of Expression" of representing animations as a function which takes in a particular value for time and outputs a corresponding image. With this in mind, we can create a data type:

```
Animation Picture
```

which is really just a synonym for

```
Time -> Picture
```

where "Time" is a type synonym for "Float". The previous example for moving a circle then becomes:

```
circleAnim :: Animation Picture
circleAnim t = Region Red (Translate (2 * t, 0) $ Shape (Ellipse 2
2))
```

The other main obstacle to implementing immediate feedback animations within the system was that the compilation thread was previously looking for a value named "pic" and loading it with type "Picture". This was changed so that the system could also look for a value named "anim" and if found load it as an "Animation Picture".

When the rendering thread reads the newly changed value, it does not know whether it is going to receive a Picture or an Animation, so the compiler passes it a special data type which could be either:

```
data UpdatableImage =
  StaticImage Picture |
  DynamicImage (Animation Picture)
```

Now all the functions in the render thread can simply pattern match against the constructor of this data type and take the appropriate action depending on whether it matches to a static image or an animation.

4.3.2 Immediate Updates at the Correct Point in Time

For animations, the feedback for the changes a user makes to their code can only really be considered "immediate" if the resulting feedback is shown at the point in time in the animation at which the change was made.

As an example, assume the user has written code in which a red circle moves from the left side of the screen to the right. If, when the ball has reached the middle of the screen, the user decides to change the colour of the ball to blue, it would be preferable if the animation continued from this middle point instead of the animation resetting to the left upon each change.

The previous decision to treat Animations as pure functions from time to Pictures makes the implementation of this concept easy. Haskell allows *higher-order functions*, meaning that functions can be passed as an argument to the rendering thread. In other

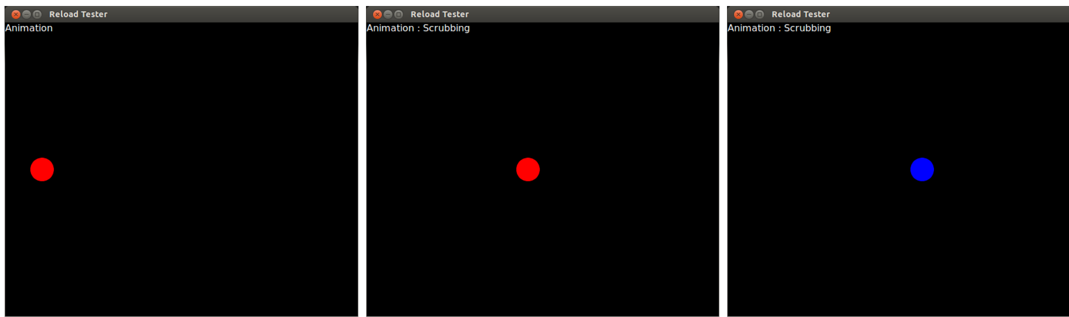


Figure 4.9: User changes colour of ball in source code half way through progression of the animation. Instead of resetting, the animation continues playing from where it currently was, but with the new colour change immediately displayed

words, we can simply swap out the old animation function for the new one when a change is made, but continue to feed in the current time parameter.

The system also provides the ability to pause animations. In order to do this, the rendering thread simply renders the last picture that was produced as a result of calling the animation function, and instead of producing new pictures on subsequent loops, it keeps track of the time it has been paused as an offset. When the user decides to unpause again, the new value fed to the animation function is the current time minus the offset for however long the system remained paused for.

4.3.3 Onion Skinning

It could be argued that so far the system does not provide "immediate" feedback for animations by some definitions. This is because if the user makes a significant change to the overall behaviour or trajectory of their animation, then to actually get feedback for whether that change was what they wanted they may have to watch the entire animation through to observe the behavioural change.

Take for example the following source code which describes a green circle moving in a figure-eight pattern between two red rectangles:

```
greenBallPos :: Float -> (Float, Float)
greenBallPos t = (x, y)
where
x = 2.5 * cos t
y = sin (2 * t)

greenBall :: Animation Picture
greenBall t = Region Green $ Translate (greenBallPos t) $ Shape (
    Ellipse 0.2 0.2)

redBlocks :: Picture
redBlocks = (Region Red leftBlock) `Over` (Region Red rightBlock)
where leftBlock = Translate (-1.5, 0) $ Shape (Rectangle 1 0.3)
rightBlock = Translate (1.5, 0) $ Shape (Rectangle 1 0.3)

figureEight :: Animation Picture
```

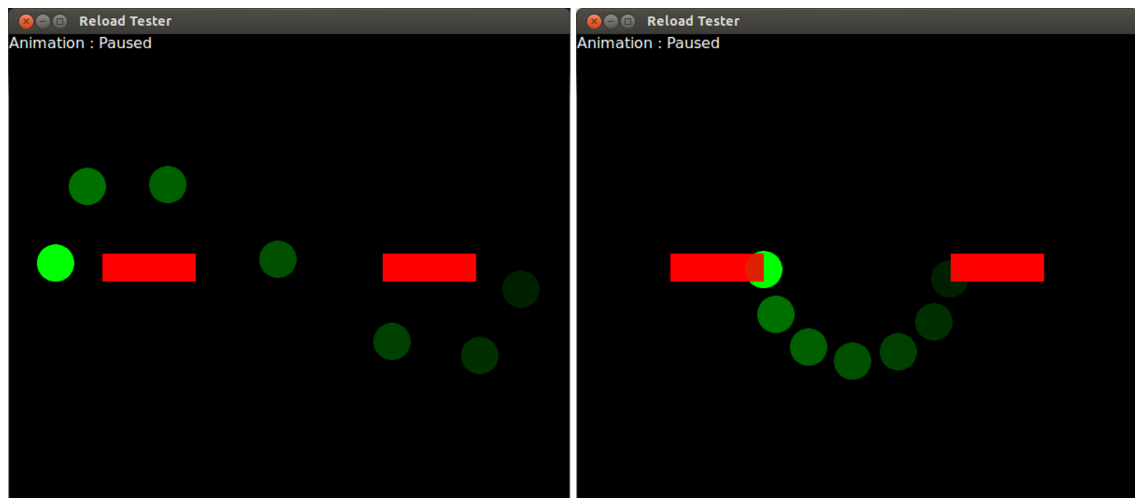


Figure 4.10: The first picture shows the “figureEight’ animation with onion skinning. The second shows how the future trajectory changes in response to the user changing the x parameter to 1 * (cos t) and the y parameter to (sin t)

```
  figureEight t = greenBall t `Over` redBlocks
```

If the user wants to make a change to the green ball’s trajectory by, for example, changing the (sin (2 t)) to (sin t), or changing the 2.5 * (cos t) to 1 * (cos t), they will then have to watch the animation through for a few seconds before they get a feel for how these changes have affected the movement pattern of the green ball. In a system which offered *truly* “immediate” feedback, there would be some way to see the entire trajectory of the animation at once.

The solution to this problem which has been implemented in this system is a technique called “Onion Skinning”.

“Onion Skinning” is a way of showing the future path of an animation in a single frame. In an onion skinned image, the state of the animation at future time intervals is displayed on screen at the same time as the current state of the animation. The further away the future interval is from the current time, the more transparent its image will be rendered.

The way this is implemented in the system is by providing the rendering system with a *list* of offsets from the current time instead of a single time value. Each of these offset times are then passed to the animation function to produce a picture, then each picture is rendered with a transparency which is inversely proportional to the size of the offset from the current time:

```
do
  let offsetTimes = [0.5, 1 .. 3]
  onionSkinnedGraphic = foldr overGraphic (picToGraphic $ anim
  currentAnimTime) [picToTransparentGraphic (float2Double $ 0.5 * (1
       - (abs offset) / (1 + maximum offsetTimes))) (anim $
     currentAnimTime + offset) | offset <- offsetTimes]

  setGraphic window onionSkinnedGraphic
```

```
drawInWindow window (text (0,0) "Animation : Paused")
```

With this feature, the user can now immediately see the current and future effect of their change to the source code in a single image.

4.3.4 Time Manipulation

The unique characteristic of animations, in contrast to static graphics, is that they are affected by the passage of time. When implementing ways for the user to directly interact with animations, the focus was therefore placed on this time aspect rather than on the transformations described previously for static graphics.

If the user clicks anywhere on the animation and drags the mouse left or right, they will be able to scrub backwards and forwards through time in the animation like a video player. Dragging the mouse left rewinds time, while dragging it right fast-forwards it.

Again, the implementation of this feature was made easy by the functional nature of the animations. When the user drags the mouse, the system calculates the offset between the mouse's original position and the current dragged position, then uses this offset as an input to the animation function to calculate the state of the picture at that scrubbed point in time. When the user releases the mouse, the animation then continues to play from the point in time which it was dragged back to.

This simple implementation of this feature would be significantly more complicated in the imperative style of animations. In the imperative style, the state of the animation is altered slightly on each loop, so in order to rewind time, one would have to figure out how to reverse each of those incremental changes made on each loop.

Chapter 5

Evaluation

As stated in the “Project Goals” section of the “Introductory Synopsis” chapter, there are two main ways in which I evaluated this project’s system.

The first way in which the system was evaluated was to treat the list of features mentioned in Bret Victor’s original demonstration “Inventing on Principle” as a “Gold Standard”, and to assess the scope and feature completeness of the implemented system against the “Ideal” system. This is done in the section “Comparison against the Gold Standard” and constitutes a small part of the overall evaluation.

The second and more substantial way in which the system was evaluated was to conduct an in-depth 12 person user evaluation. As part of this evaluation, users had to complete a series of simple tasks using the system and answer a series of questions afterwards to gauge their experience with the system. The results of these user evaluations were used to assess the system against the *evaluation criteria* outlined in the introduction:

Evaluation Criteria

- How *“immediate”* is the feedback (i.e How fast is it?)
- Does the implemented solution provide robust / accurate results? (Are users able to break the system through normal / antagonistic use)?
- Is the feature *useful*? In other words, are tasks completed faster with the system than without? Does the system provide a deeper understanding of the source code? Does the system encourage experimental creation?
- Is the system easy to use?

Discussion of this user evaluation takes up the bulk of this chapter.

In addition to these main methods of evaluation, the previous sections which dealt with the implementation of the system evaluated the quality of the implementation in terms of the reasons for particular technical decisions being made, the limitations of the system, and alternative techniques that were considered. The “Related Work” section also discussed the differences between this project’s implemented system and

other related systems by other authors. These areas are not elaborated on further in this chapter.

5.1 Comparison against the "Gold Standard"

In the "Main Inspiration" section, I outlined the key features present in Bret Victor's demonstrations which outline a "Gold Standard" for an immediate feedback, two-way graphics and animation programming environment:

1. Changes made to the program code are immediately reflected in the graphical result of a program
2. There is a direct connection between the source code and the graphical result which it produces - hovering over a shape in the graphic will highlight the line in the source code which is responsible for the creation of that shape and vice versa
3. Source code can be changed in a variety of ways which encourage experimentation with values. For example, numeric values in the code may be associated with slider widgets which allow the user to sweep across a range of values, or colour values may be associated with a colour palette.
4. Features which current Integrated Development Environments (IDEs) provide should be augmented to provide immediate feedback. For example, auto-complete suggestions for functions should show a preview of what the graphical result would look like if the given suggestion were applied.
5. The user is able to directly manipulate the graphical result in a number of ways. This includes translation, scaling, rotation, changing the colour or texture of a shape, re-arranging the z-ordering of shapes etc.
6. Changes made via direct manipulation of the graphical result should be immediately reflected in the source code. In other words, the source code should update to reflect the new state of the graphical result.
7. Changes to the source code should be immediately reflected in the resulting animation, but with the additional constraint that the animation should be able to continue running without interruption from the point at which the change to the source code was made.
8. The user should have the ability to see how their changes affect the entire span of the animation immediately. One way of accomplishing this may be to have a single image view of the whole animation which displays the trajectory the objects will take throughout the duration of the animation.
9. The user should be able to directly manipulate the animation. The unique component which is present in animation as compared to a static graphic is the passage of time, so such manipulations should revolve around changing, reversing and pausing time.

The majority of these features were implemented in the final version of this project's system to some degree. However, due to time and technical constraints, a few were only partially implemented or were left out of the final system.

The first feature, that users receive immediate feedback for changes to their source code, is fully implemented. The compiler instantly picks up changes to the user's source code and updates the result accordingly, as discussed in previous chapters. The second feature - that there is a direct connection established between the images on screen and the code responsible for them - is also implemented. Hovering over a region in the final image causes the system to highlight in red the line it thinks was responsible for creating that region. Due to time constraints, this was not implemented in the reverse direction (hovering over a line causing a region in the final picture to be highlighted in red), but this would be relatively easy to implement given the tracking information already available to the system.

The third and fourth features deal with providing a multitude of alternative ways for the user to interact with the source code other than just typing in information directly. These were not implemented. The reason for this is that after the first year of development, the focus of the project shifted more heavily towards providing support for animation, and adding features like slider widgets for numeric values were given a lower priority since they were seen as nice but not necessarily vital features for the system to have.

The fifth feature suggests a number of ways in which the user could interact directly with the resulting image. As detailed in the implementation chapters, Translation, Scaling and Rotation are the operations which are supported in the final system. Operations such as rearranging the Z-ordering of shapes on top of each other proved to be more complicated than initially thought, and were not implemented. Details of why can be found in section 4.2.5- "Unsupported Operations". Additionally, tools which would have allowed the user to directly manipulate the colour and texture of shapes were not implemented, as again they were seen as low priority features and were dropped due to time constraints.

The sixth feature states that the source code should be updated in response to the user's direct manipulation of the resulting graphic. As discussed in section 3.5 "Limitations of Updating", the system is able to update the source code in many cases, but it is unclear what the correct update should be in source code which involves more complicated expressions, with higher-order functions presenting particular challenges.

Features 7, 8 and 9 deal with extensions of the system to support animation, and were all fully implemented.

5.2 User Evaluation

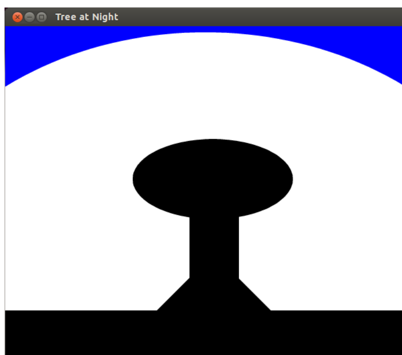
5.2.1 Task Description

The user evaluation consisted of a series of simple tasks which involved reproducing and manipulating graphics and animations using the SOE Graphics library. There were 12 users in total divided at random into a "System" group of 9 users and a "Control" group of 3 users. The "System" group used the immediate feedback tool developed as part of this project to complete the tasks, while the "Control" group used any editor or IDE of their choice along with the Haskell GHC compiler to complete the tasks. The reason for separating the users into a "System" and "Control" group is so that we can compare their relative performance on the tasks.

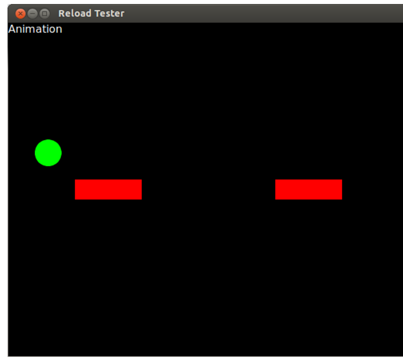
The tasks given to the users were as follows:

1. **Tree Exercise A:** Given the template file TreeExercise.hs (which contains source code for rendering the ground and some suggestions for other variable names the user might want to use), reproduce the following picture of a Tree scene:

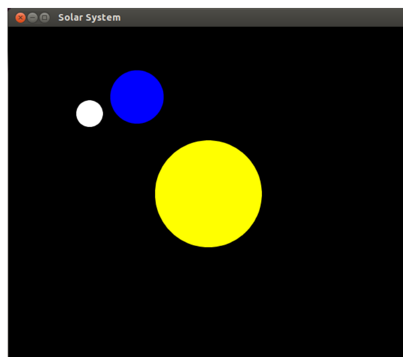
2. **Tree Exercise B:** Using your answer to the previous exercise, alter it to produce the following night time scene of the same tree:



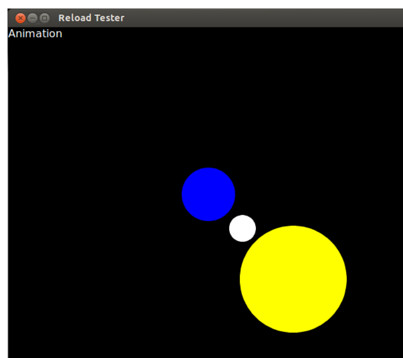
3. **Figure 8 Exercise A:** Given the template file Figure8Exercise.hs (which contains source code to render a green ball and two rectangles, but without the correct movement behaviour), reproduce the following animation of a green circle moving in a figure-8 pattern around two stationary red blocks.



4. **Solar System Exercise A:** Given the template file SolarSystemExercise.hs (which contains only suggestions for names of functions and values the user might want to use), reproduce the following animation of the earth, sun and moon.



5. **Solar System Exercise B:** Alter the answer to the previous question such the the sun and moon both revolve around the earth.



To ensure these tasks were consistent from user to user, the order of tasks was not varied and instructions for each task were read to the user from a pre-written script. The script lays out the description of each task in a standard way and encourages the users to "think aloud" while they are doing each exercise so that as much information about their experience of the tasks was available as possible. Users were also given a standard "Cheat Sheet" which outlined the basic data types and functions available in the system's graphics library, and provided instructions on how to create basic images. Full details of the cheat sheet, exercise script, and template files are available in the Appendix.

The time each user took on each exercise was measured with a stopwatch. This was so

that the relative times on each exercise for users who used the project's system could be compared with the times for users who used their own editor.

Additionally, users who used the system were given a questionnaire after they completed the exercises. This was to ensure that the feedback received from the user's at least touched in some way on each of the criteria listed at the beginning of this chapter.

The questionnaire consisted of 14 questions. The majority of questions asked the user to say how much they agreed with a particular statement by answering "Strongly Agree", "Agree", "Disagree", or "Strongly Disagree". Each question also had space for the user to add in their own additional comments relating to the question.

The 14 questions were as follows:

1. When making changes to the source code, the graphical result updated fast enough that I would consider the feedback "Immediate".
2. When directly manipulating the graphics, the source code updated fast enough that I would consider the feedback "Immediate".
3. It felt easier to figure out problems using the project's system than in my normal editor environment.
4. I felt more comfortable experimenting with and changing my source code than I usually would in my normal editor environment.
5. The features provided by the project's system allowed me to gain a more thorough understanding of what my source code was doing.
6. I found the system easy to use.
7. The system crashed or completely broke while I was using it.
8. The system behaved or made updates in a way which I did not expect.
9. What were features you would have liked the system to have which were not provided?
10. Highlighting the line of source code which was "responsible" creating the current region I was hovering over was a useful feature.
11. The ability to sweep backwards and forwards through time in an animation was helpful in completing tasks.
12. Pausing the animation to see an "Onion Skin" of the image was a useful visualisation of the trajectory of my animation.
13. I would consider using an immediate feedback system such as this for future projects.
14. Any additional comments.

The target users for these evaluation tasks were those who had a beginner to intermediate understanding of the Haskell programming language - i.e they had taken the

University of Edinburgh course “Informatics 1 - Functional Programming” or equivalent.

5.2.2 Results and Analysis

5.2.2.1 Time taken on Exercises

Exercises (System)	Times to Complete (Mins : Secs)	Average	Max	Minimum
Tree A	20:00, 16:53, 16:13, 17:53, 18:32, 17:31, 14:01, 16:57, 12:23	16:42	20:00	12:23
Tree B	01:39, 02:06, 02:10, 02:53, 01:15, 02:00, 01:44, 01:24, 01:13	01:49	02:53	01:13
Figure-8	05:38, 06:12, 07:42, 15:17, 01:50, 13:00, 03:55, 02:46, 02:19	06:31	15:17	01:50
Solar System A	11:01, 09:25, 11:58, 17:59, 12:46, 11:30, 07:56, 07:40, 08:04	10:55	17:59	07:40
Solar System B	00:49, 01:56, 01:30, 01:20, 01:14, 01:39, 02:09, 04:45, 00:45	01:47	04:45	00:45

Exercises (Control)	Times to Complete (Mins : Secs)	Average	Max	Minimum
Tree A	36:10, 44:42, 88:12	56:21	88:12	36:10
Tree B	02:45, 03:43, 03:59	03:29	03:59	02:45
Figure-8	04:02, 13:34, 11:23	09:39	13:34	04:02
Solar System A	15:45, 31:45, 29:54	25:48	31:45	15:45
Solar System B	05:39, 02:44, 05:32	04:38	05:39	02:44

These results show that, for every task, users who used the immediate feedback system were able to complete the task faster on average than those in the control group. There was variation in the size of the time saving depending on the task however. This was expected, and largely due to the structure of each of the tasks.

The task “Tree A” was the one in which there were the most significant differences in times between the system and control group. This was likely because this task involved reproducing an entire scene from scratch, and would usually involve a lot of trial and error in placing objects in the correct places.

For tasks “Tree B” and “Solar System B” there was still an improvement for the group using the system, but the difference was less significant. This was likely because these tasks only involved making a small, obvious change to an existing solution, meaning there was only a small amount to be gained by having immediate feedback to your changes.

The “Figure-8” task was the most interesting, as although the average time taken was lower for those using the system, it was not significantly lower, and indeed there were both very high and very low times in both the system and the control group. This result was due to the solution to this task relying on the user having some knowledge of trigonometry (i.e how the sin and cos functions behave). Some users figured out the solution almost immediately, while others got lost or confused. However, what is interesting is that many of the users in the “System” group who were initially stuck on the problem reported that the system’s ability to provide immediate feedback to their changes meant that they were able to gradually learn how different values affected the shape of the sin and cos waves by experimenting with applying different parameter values.

The above results show that the immediate feedback system does have a positive effect on the speed at which users complete basic tasks, but that the amount of benefit it gives can vary depending on the nature of the task.

5.2.2.2 Questionnaire Results

What follows is a presentation of the key findings from the questions presented in the questionnaire, and a brief analysis of what some of those results imply.

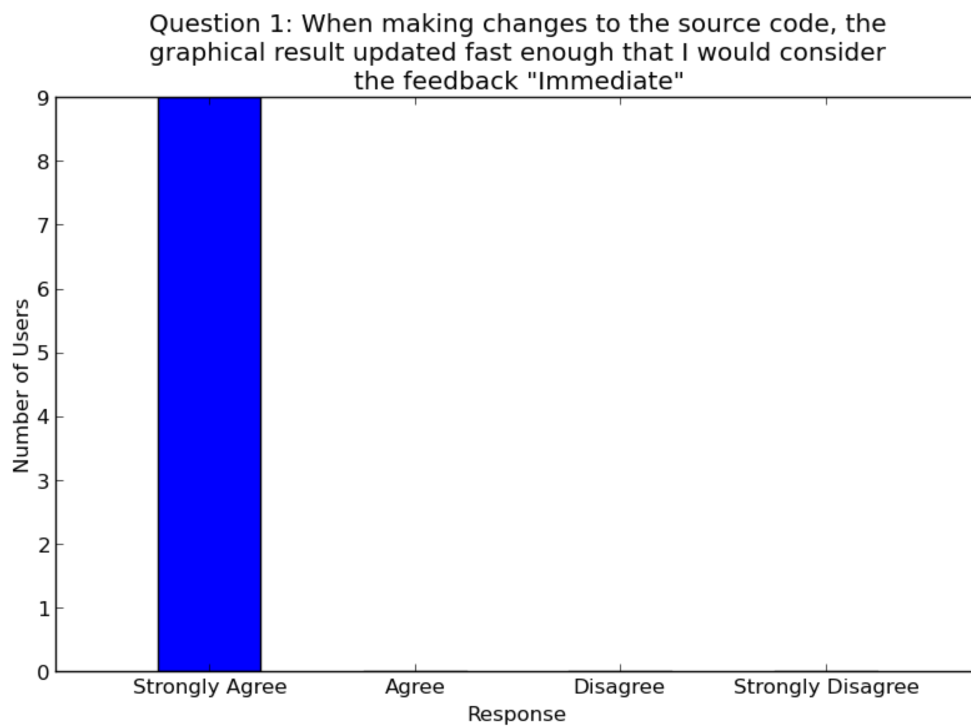


Figure 5.1: Question 1 results

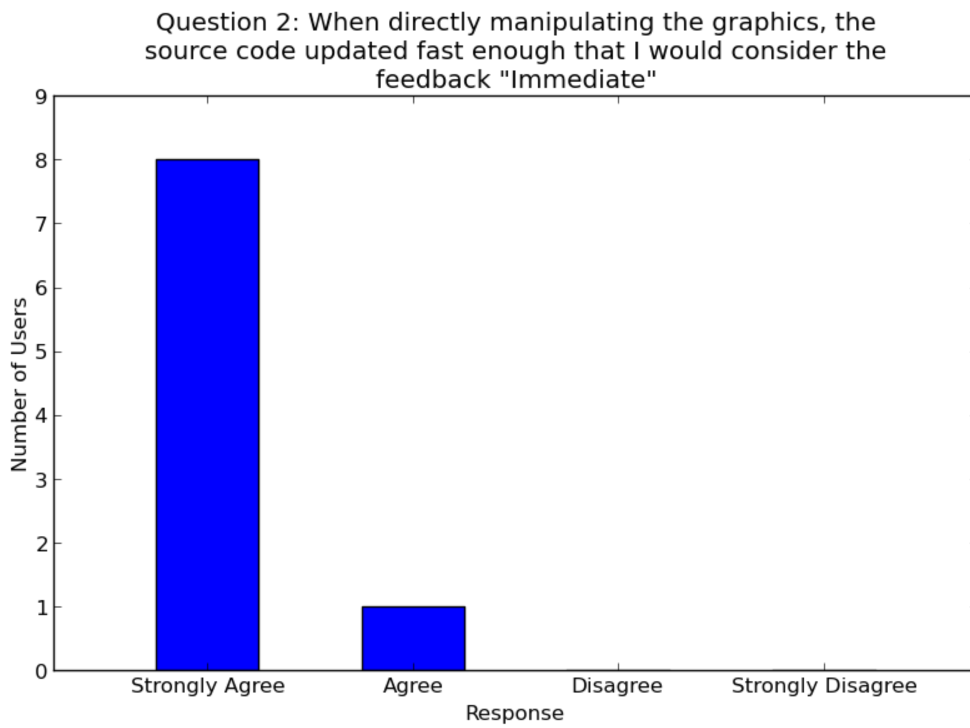


Figure 5.2: Question 2 results

The first two questions dealt with how fast the users perceived the “Immediate” feedback to be both in the direction of source code to result, and result to source code. The result here was overwhelmingly positive, in that all users agreed that the feedback was fast enough to be considered “Immediate”.

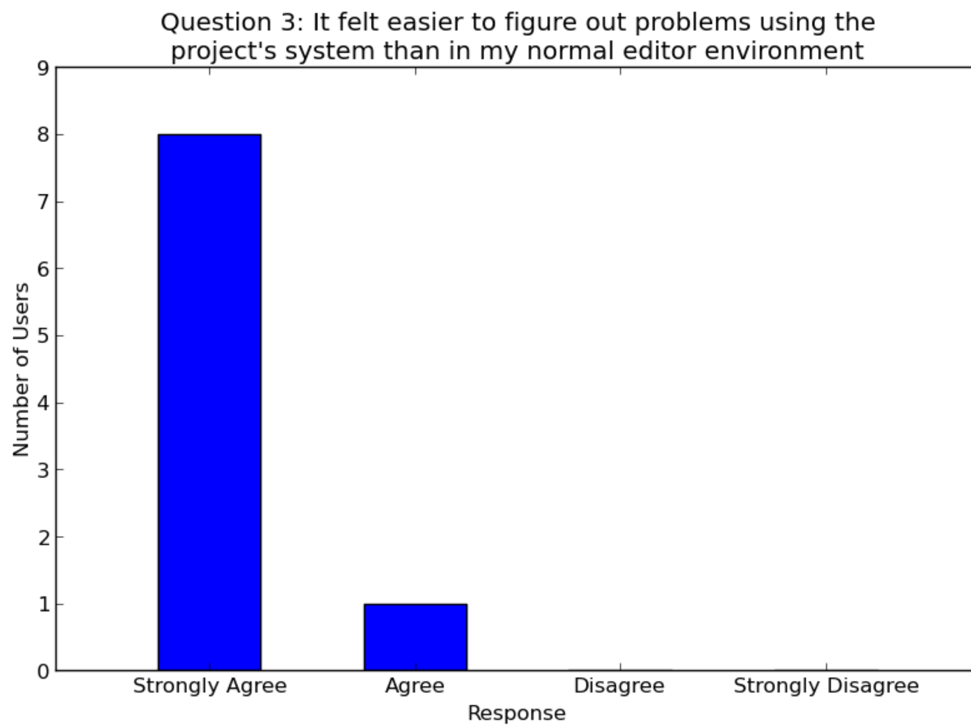


Figure 5.3: Question 3 results

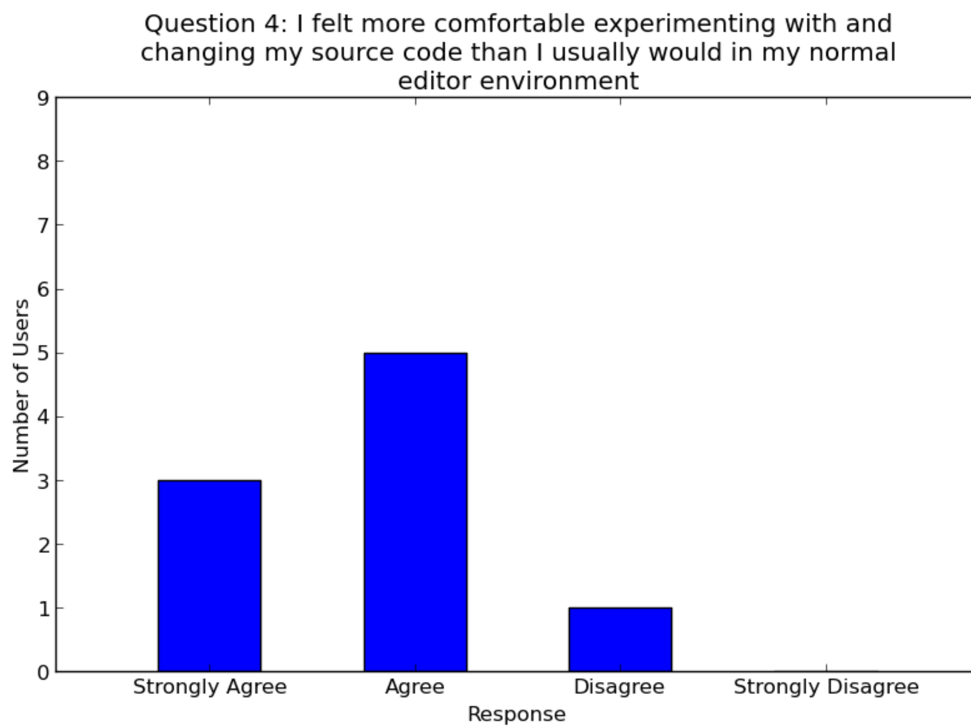


Figure 5.4: Question 4 results

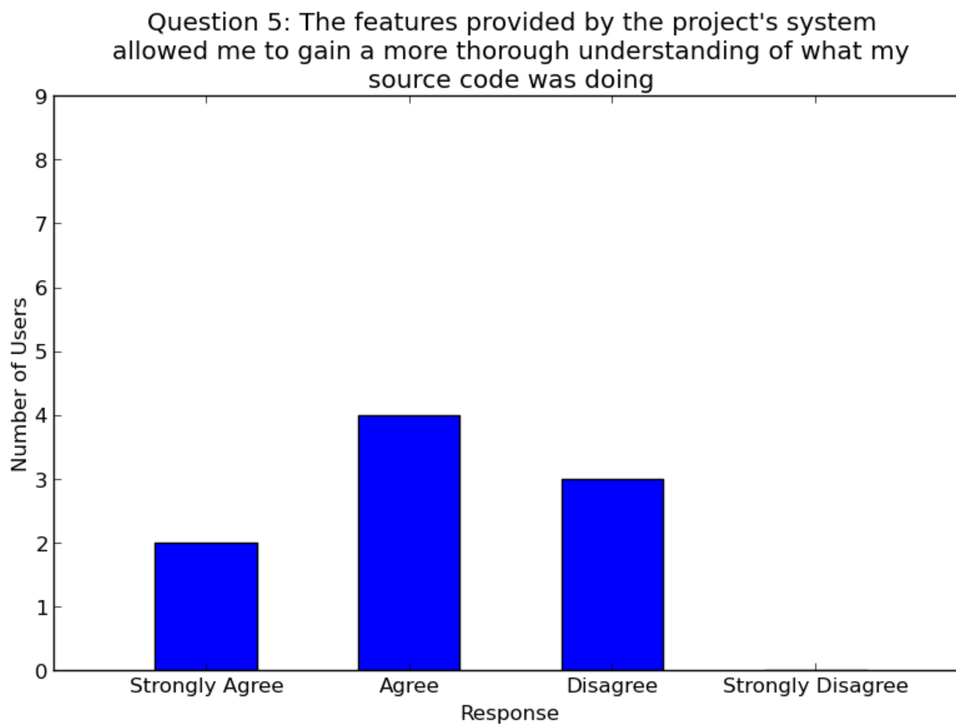


Figure 5.5: Question 5 results

Questions 3, 4 and 5 present three distinct ways in which the system could be considered useful. The questions gauged whether the system made *problem solving* easier, whether it encouraged *experimentation* with the source code, and whether it deepened the user's *understanding* of their source code respectively.

The majority of users agreed both that the system made it easier to solve problems than their usual development environment and that the system encouraged them to experiment with changing parts of their source code. Many users said that the two aspects went hand in hand, in that because there was such a small overhead to making small changes to their source code, the solution became immediately apparent after a few seconds of trial and error. The one user who disagreed that the system made him more comfortable making changes to his source code clarified that this was simply because he already felt very comfortable making changes to his source code in his current editor.

There was less of a consensus on whether the system helped deepen a user's understanding of the source code however. Because the tasks given to the users in this evaluation were not particularly complex, many users only hesitantly agreed with the statement, as they did not feel there was much that needed to be understood about their source code in the first place. Those who outright disagreed with the statement said that while they agreed the system allowed them to develop faster, it did not necessarily follow that this meant they were gaining some deeper understanding of their code.

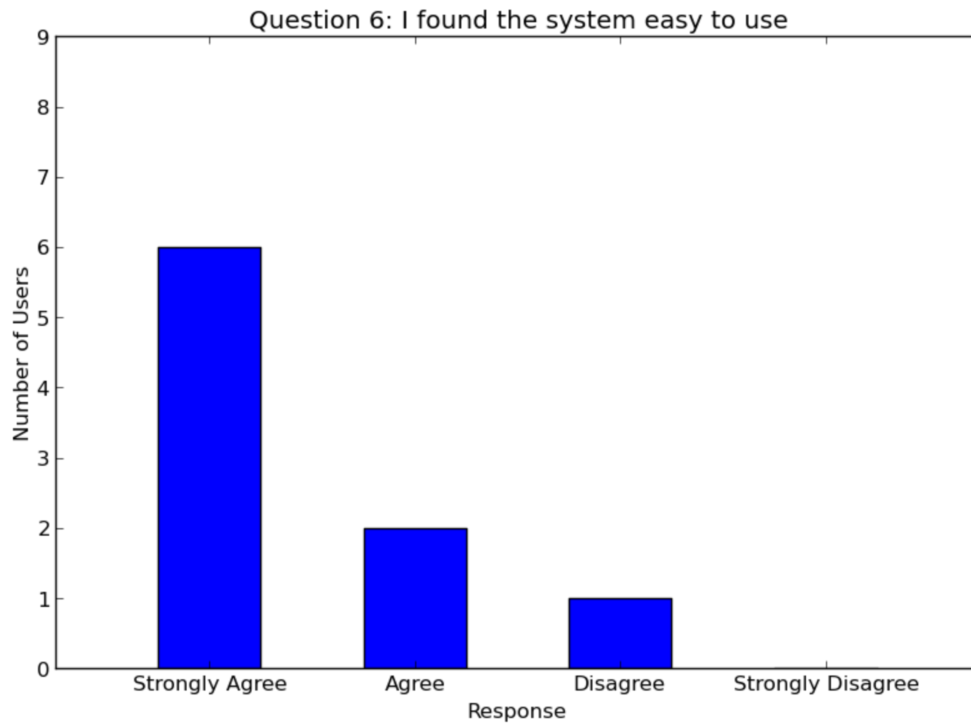


Figure 5.6: Question 6 results

Question 6 asks users to assess the usability of the system as a whole. While the majority of users agreed that the system was simple and intuitive to use, a few voiced concerns that the user interface lacked polish, and that certain manipulation tools - namely the “Scale” tool - felt “clunky” to use.

The main result here appears to be that while the core concepts were intuitive to understand, the usability could be greatly improved by spending more time on details of the user interface.

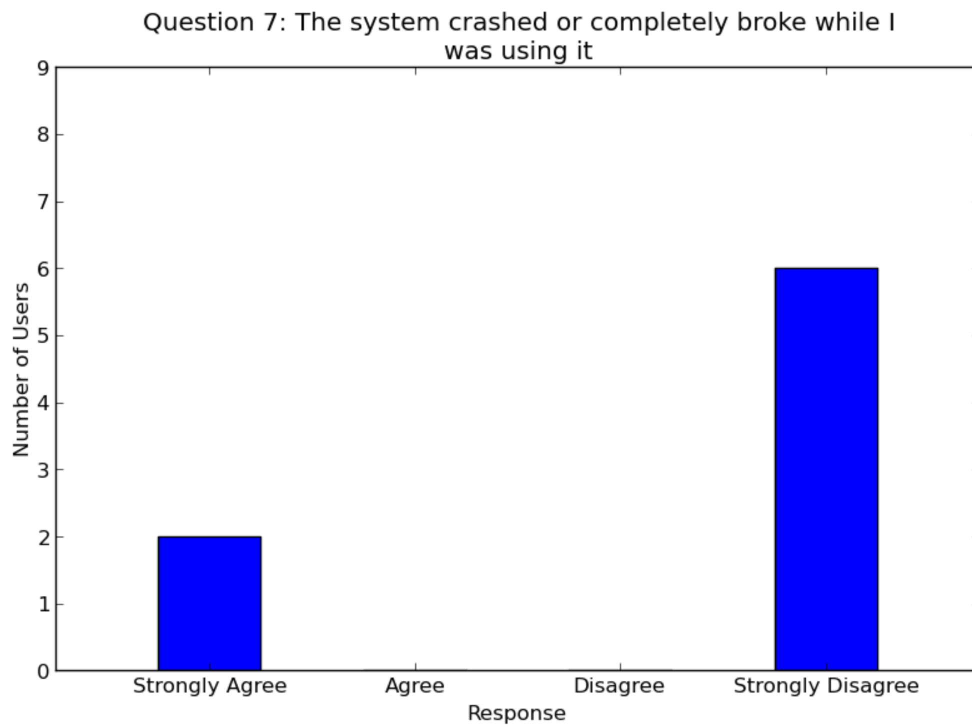


Figure 5.7: Question 7 results

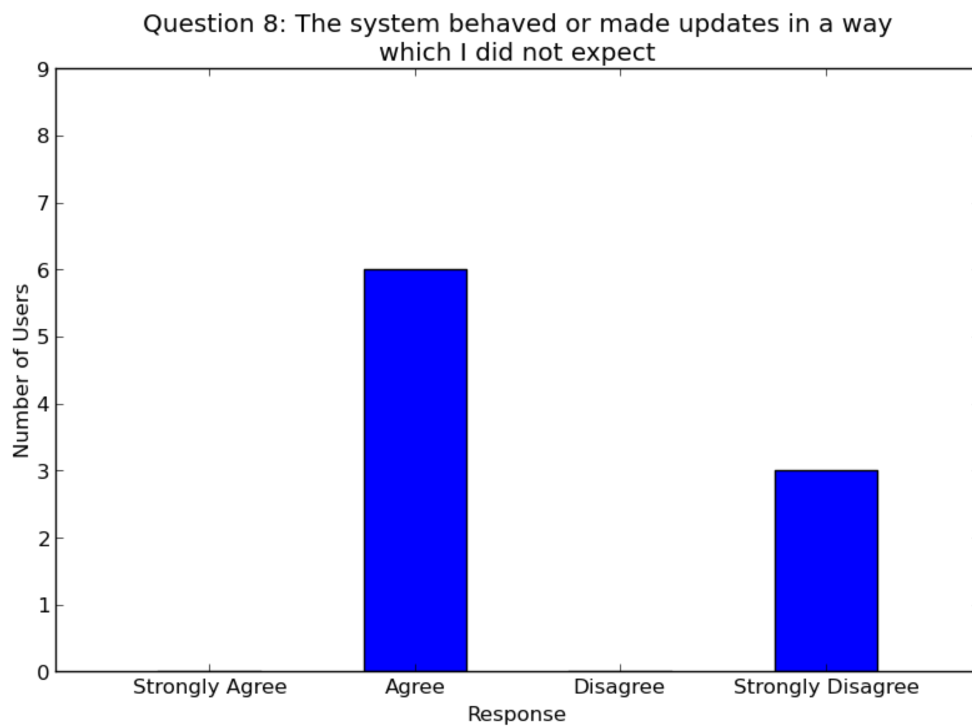


Figure 5.8: Question 8 results

Questions 7 and 8 assess the robustness and reliability of the system.

Only 2 users encountered a situation where the system hung, then crashed completely. In both cases, this was because the users had written a non-terminating function which caused the system to hang indefinitely upon evaluating it. Preventing users from making these errors seems difficult or impossible, since arbitrary Haskell programs can be written, although it would perhaps be possible to include a timeout and recover more gracefully. Since no other system-crashing errors were unearthed, I would consider the system to be reasonably reliable.

In terms of robustness to user interaction, there were a few issues that almost all users ran into in which the system behaved in a way that they were not expecting. The main issue users faced was that when they attempted to move a region using the translate tool, the region in question would pop to the front of the screen unexpectedly, and did not move back to its original z-position until the user made another change to the image. Another issue was that many users expected the scaling tool to scale the image proportionally - that dragging the image to make it larger would preserve the aspect ratio between width and height. When the tool instead scaled the two independently, some users were confused. Both these issues are minor quirks which could be easily fixed in a future version of the system.

Two users however ran into a more significant problem. When specifying functions which would create regions as the result of a function call, they would find that if they directly manipulated one of these regions, *all* regions created via this function call would be affected. This problem is discussed in section 3.5 “Limitations of Updating”, and is significantly more difficult to address. Since only two users encountered this problem however, it would appear that the system is relatively robust to the large majority of user input.

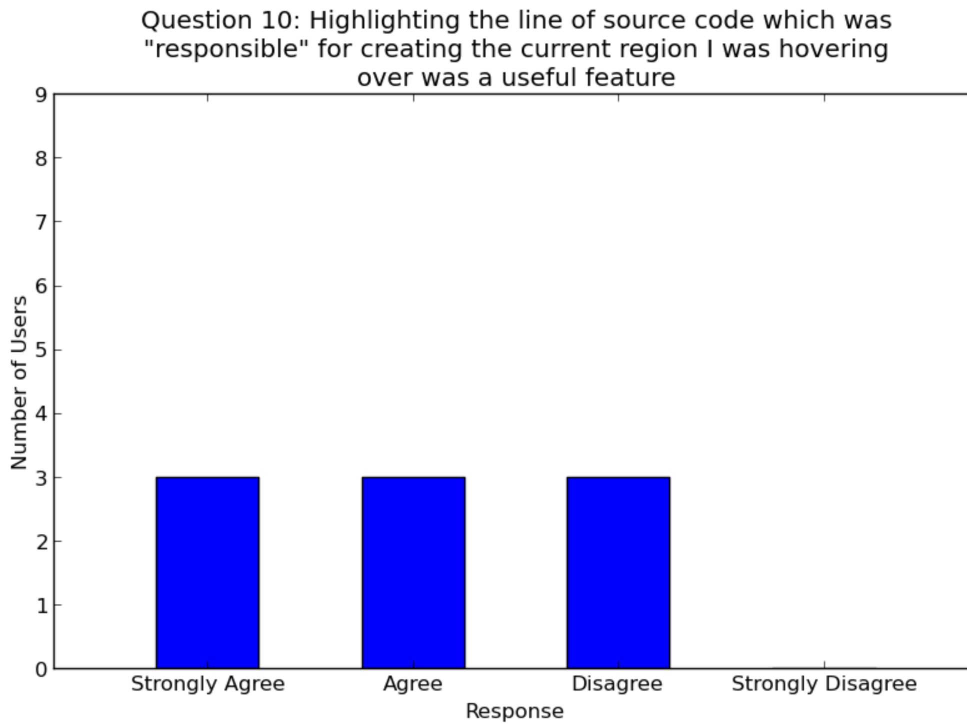


Figure 5.9: Question 10 results

Question 10 assesses the usefulness of the line highlighting feature, which is meant to provide a direct connection between the source code the user writes and the various parts of the resulting image. Responses to this feature were mixed. While some felt it had some utility in theory, many users said the examples given in the evaluation tasks were so simple that it was easier just to look at the source code themselves to find out which line was responsible for creating a particular region. Perhaps for more complicated scenes, in which you were creating a picture of an entire forest of trees for example, then the ability to find the code responsible for a particular part of a particular tree would be more important.

An opposite criticism was that this feature was not taken *far enough*. One user stated that they would have preferred the system to highlight *all* uses of a particular region, not just where it was created but also where it was passed as an argument, where it was used to create a Picture and so on.

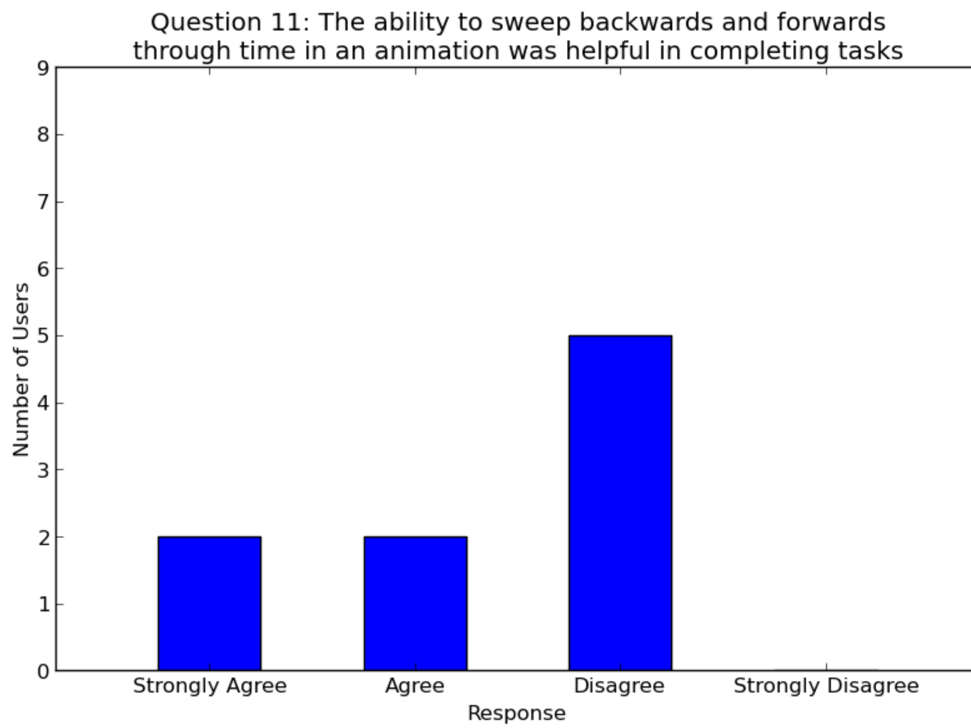


Figure 5.10: Question 11 results

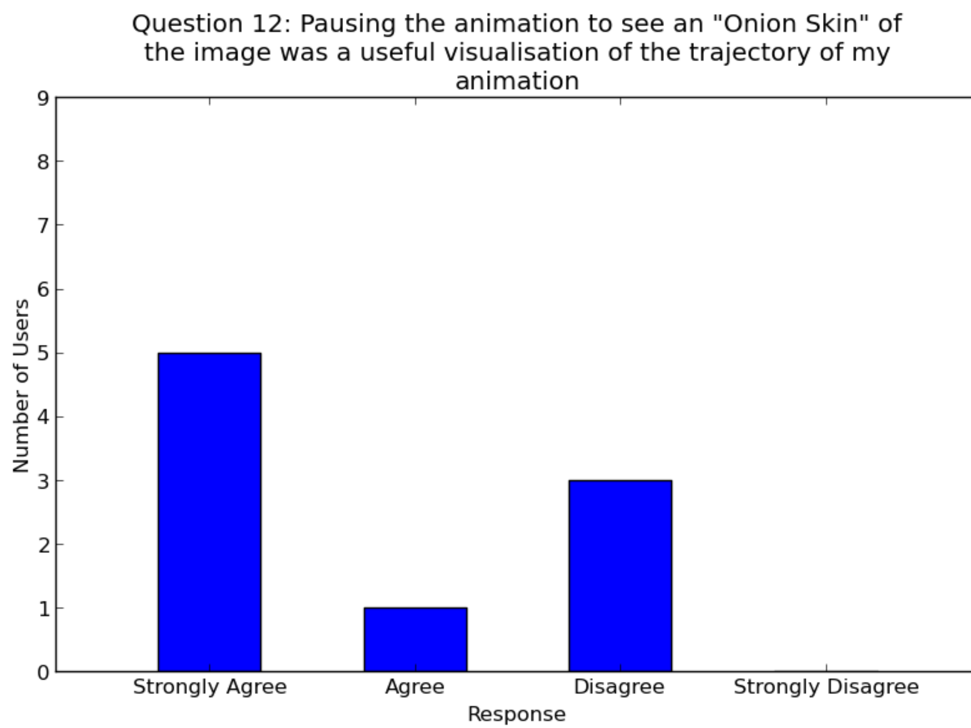


Figure 5.11: Question 12 results

Questions 11 and 12 assessed the usefulness of the features implemented to improve development of animations within the system - namely time manipulation and onion skinning. The interesting thing here was that users fell into groups of those who used the time manipulation controls and those who used the onion skinning feature. Those that preferred to use onion skinning said they felt that the time manipulation mechanics were redundant, as they had a view which was able to show them the entire trajectory of the animation at once. Those who favoured the time manipulation mechanics said they felt that the onion skinning feature was useless because they could simply sweep the animation to the point in time that they were interested in.

The conclusion I have drawn for this is that it is beneficial to provide the user with a multitude of options for viewing animations, as it means they are more likely to find a work flow that suits their needs.

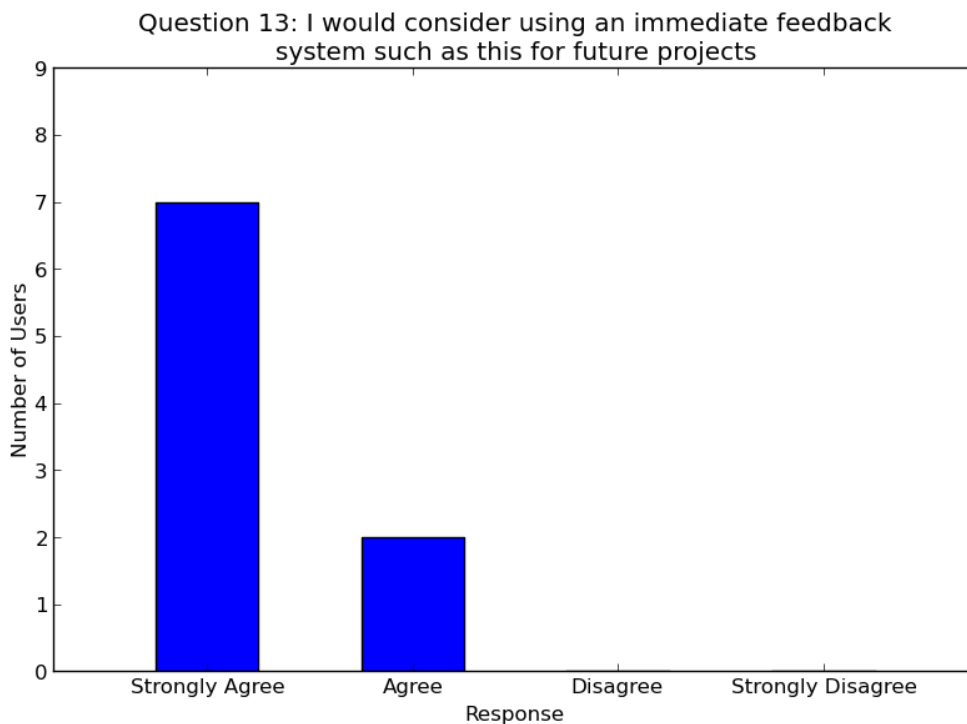


Figure 5.12: Question 13 results

The final statement asked whether they could see themselves using such an immediate, two-way feedback system for other applications. The result was positive, with many users wishing to use such techniques to help with their own academic projects, GUI programming, web development, and mobile applications. There appears to be a wide scope in terms of the perceived usefulness of such techniques in other domains.

5.2.2.3 User Suggestions for Improvement

As well as being timed and completing a questionnaire, users were also encouraged to think out loud during the exercises. Question 14 also asked the users to give any

additional comments they had. Many users took these as opportunities to suggest improvements to the system. Some of the most frequent suggestions and requests were:

- The direct manipulation tools should be altered to be more like the tools available in Photoshop, as many of the transformation tools in this system (especially the Scale tool) feel “clunky”.
- Provide a wider range of ways to interact with animations. Common requests were: an ability to directly manipulate the trajectory shown by the onion skinned image, an ability to apply transformations to the animation as a whole, and a mode which shows the onion skin of an animation while the animation is still running.
- Instead of separating the system into two separate windows - one for the user’s source code and one for resulting image - make both panels part of a single application window.
- Extend the system’s editor to have features present in a full integrated development environment (IDE). For example, provide auto-complete, module management, saving and loading of projects etc. A similar suggestion was that the system be made available as a plugin for an existing IDE such as Eclipse.

The nature of these suggestions is a positive sign, as most of them revolve around the idea of extending the system with additional features to form a more complete application. Few of the suggestions take issue with the core ideas of immediate, two-way feedback, or with the performance of the current implementation.

5.2.3 User Evaluation Conclusions

The user evaluation provided various insights into the success of the system in terms of usefulness, speed, usability, reliability and robustness.

The majority of users felt that the system was a success in terms of the immediacy of feedback, the freedom it gave them to quickly make changes to code, and the speed with which they could complete tasks. Comparison of the completion times between the system and control groups provides evidence that the system group’s impressions of improved speed were indeed correct.

While most users were able to use the system with relative ease and without any unexpected behaviour, a few users reported crashes due to infinitely looping code, while others encountered unexpected behaviour when the system attempted to update their source code. As discussed previously, these problems are non-trivial to fix.

Overall, users were enthusiastic about their experience with the system, and thought that the features presented within the current system had good potential to be adapted to other domains and applications.

Chapter 6

Conclusion

6.1 Project Summary

This report has explored the use of techniques for creating *two-way, immediate feedback* tools for graphics and animation in Haskell. The main contribution of this project was to implement a Haskell system which embodies these properties.

The system provides the ability to create simple graphics and animation programs. When the source code of these graphical programs is changed within the system, the results are instantaneously reflected in the resulting image.

Additionally, the system allows the user to directly manipulate static images by translating, rotating and scaling its constituent parts. The effects of these changes are reflected automatically and instantaneously as updates to the user's original source code.

The system also attempts to provide a direct connection between the user's source code and the resulting image by tracking which lines are responsible for the creation of each part of the final image.

For animation, the system provides immediate feedback to changes in the source code while keeping the animation running. It also provides a variety of ways to interact with and visualise the effect of changes to the animation via time manipulation and onion skinning.

An extensive user evaluation was also conducted as part of the project, which validated the success of the system along the criteria of speed, usefulness, flexibility, reliability and robustness. Users also identified areas in which the system could be extended in the future. These mainly involved extending the system to support a wider range of animation interactions, and the addition of more IDE-like features into the system's editor.

The system has some limitations, which mainly revolve around the system's ability to apply the correct updates to source code in response to the user's direct interactions with the resulting image in certain complex cases.

The plan for this project in the near future is to publish the system's source code online so that the Haskell community can use it and further extend it with new functionality.

6.2 Ideas for Future Work

The following is a list of ideas for extending the current system in future work:

1. For static graphics, the system offers "two-way" feedback by allowing the user to directly manipulate the graphics and having the source-code update to reflect these manipulations. It is not clear how interactions such as "Translating a shape" would work in the context of a dynamic animation, but if such ambiguities were resolved, this would make an interesting extension.

2. At the moment, the only ways to manipulate time are to move the mouse to drag time forwards or backwards. Other time manipulation mechanics to experiment with might include speeding up or slowing down time, or attempting to represent changes in time as updates to the source code.

3. Many users of the system, upon seeing the onion skinning example, remarked that an interesting and useful way to interact with an animation would be if they could manipulate the implied line of movement given by the onion skinned image, and have these manipulations alter the behaviour of the animation as a whole.

4. At the moment, animations are represented as a function from time to pictures. There is no reason why the passage of time need be the only event which affects the course of an animation. A possible extension would be to represent animations as a function which takes stream of events as input and produces a picture as an output. This is the approach taken by the field of *Functional Reactive Programming* [12].

5. Many users were excited by the features present in this system, but would be reluctant to give up their current development environment in order to use those features. If the system were adapted for use as a plugin to a popular IDE (Eclipse for example), then users would have the best of both worlds.

6. One of the original goals for the project was to have a variety of ways to interact with the *source code*. For example, having a slider widget appear when the user hovers over a numeric value which allows them to sweep through a range of values and immediately observe the results of this. Such additions would provide the user with a more expressive range of ways to experiment with their code.

7. In certain complex cases, the correct update for the system to apply to the source code in response to manipulation of the image is ambiguous. By exploring one of the areas mentioned in section 3.5 "Limitations of Updating" (e.g bi-directional transformations), it might be possible to resolve some of those ambiguities.

Chapter 7

Appendix

7.1 User Evaluation Task Script

The following text is the script which was read to users taking part in the user evaluation. This was to ensure that the information each user was given upon doing the tasks was consistent from user to user

User Evaluation Exercise Script

Craig Innes S0929508

February 16, 2014

Introduction

This set of tasks is designed to evaluate the effectiveness of the immediate feedback programming system developed as part of the "Programming as an Interactive Experience" thesis project. It consists of 3 exercises involving the recreation of various simple pictures and animations.

To be clear, it is the system which is being evaluated, not the user's performance on the tasks, so dont worry if you are struggling on a particular task - This is useful information!

You are encouraged to think aloud while going through these tasks. Are you stuck? Did the system do something that you werent expecting? Did something the system did for you make you happy? Both positive and negative feedback are equally useful in assessing the system. After completing these tasks, you will be given a short questionairre to ask your opinion on some aspects of the system.

Exercise One: Tree Picture

Task A

In the folder userEvaluation there is a file called TreeSolution. Run this file using the command:

```
./TreeSolution
```

Your task is to recreate this image in the system using the skeleton file TreeExercise.hs. This can be done by going into the workingPrototype folder and running the interactive programming system textEdit with the TreeExercise.hs file as an argument:

```
./textEdit TreeExercise.hs
```

Task B

In the folder userEvaluation there is a file called TreeNightSolution which shows the same scene at night when the moon is just rising. Alter your solution to Task A to recreate this night time scene.

Exercise Two: Figure Eight Animation

In the userEvaluation folder, run the file Figure8Solution:

```
./Figure8Solution
```

Using the skeleton file Figure8Exercise.hs, recreate this animation in the interactive system. Again, this can be done by running:

```
./textEdit Figure8Exercise.hs
```

Exercise Three: Solar System Animation

Task A

In the userEvaluation folder, run the file SolarSystemSolution:

```
./SolarSystemSolution
```

Using the skeleton file SolarSystemExercise.hs, recreate this animation in the interactive system. Again, this can be done by running:

```
./textEdit SolarSystemExercise.hs
```

Task B

Many hundreds of years ago, people believed that both the moon and sun re- volved around the earth. Can you alter your solution from Task A to reflect this belief?

7.2 User Evaluation "Cheat Sheet"

What follows is the text from an instruction sheet given to users taking part in the user evaluation. This was used to familiarise them with the SOEGraphics library used as part of this project

Creating an image in the Haskell SOEGraphics library consists of three stages:

1. Create a **Shape** using one of the primitive shape types.
2. Create a **Region** by taking a Shape and transforming and combining it.
3. Create a **Picture** by taking a Region and assigning it a colour.

Shapes

Shapes are constructed using the name of a shape followed by its dimensions:

- Rectangle w h
- RtTriangle w h
- Ellipse w h
- Polygon [(x,y)]

Regions

Construct a Region by using the Shape keyword and passing a value of type Shape as an argument.

Example 1

Shape (Rectangle 3 2) Creates a region out of a Rectangle Shape value.

Regions can be transformed using the three basic affine transformation operations.

- Translate (x,y)
- Scale (x,y)
- Rotate angle

Example 2

Translate (3,2) r1 – Moves the region r1 right 3 units and up 2 units

Multiple regions can also be combined into a single new region using the following commands:

- Union r1 r2 – Combination of area of region 1 and region 2.
- Xor r1 r2 – Union of r1 r2 minus the pixels shared by both r1 and r2.
- Intersect r1 r2 – Creates a new region containing only the intersection of regions r1 and r2.
- Complement r1 – Creates a new region which is the inversion of r1

Pictures

Pictures are the values which are actually displayed on screen. They are constructed using the “Region” constructor combined with a colour value and a region:

The currently supported colour values are:

- Black

- Blue
- Green
- Cyan
- Red
- Magenta
- Yellow
- White
- Brown

Example 3

Region Red (Shape Rectangle 3 3) – Creates a red square Picture

By layering these pictures over each other using the “Over” function, you can build up your final image:

Example 4

FinalPicture :: Picture FinalPicture = p1 ‘Over‘ p2 ‘Over p3

Animations

Animation in Haskell is just a synonym for a function which takes the current time to some other data type. For example, the type:

Animation Picture

is equivalent to:

```
Float -> Picture
```

Example 5

greenBallMovingRight :: Animation Picture greenBallMovingRight t = Region Green $ Translate (t, 0) $ Shape $ Ellipse 1 1

Displaying your final result

The system will always look for the final image to display under the name “pic” and the final animation to display under the name “anim”.

To switch whether the system is looking for an animation or an image, click the button at the bottom of the program.

7.3 User Evaluation Template Files

7.3.1 TreeExercise.hs

```
--Tree
foliageReg,trunkReg,rootLeftReg,rootRightReg :: Region
foliageReg = Empty
trunkReg = Empty
rootLeftReg = Empty
rootRightReg = Empty

--Background
skyReg, sunReg, groundReg :: Region
skyReg = Empty
sunReg = Empty
groundReg = Translate (0, -2.75) $ Shape(Rectangle 6 2)

foliage,trunk,roots,sky,sun,ground :: Picture

foliage = EmptyPic
trunk = EmptyPic
roots = EmptyPic
sky = EmptyPic
sun = EmptyPic
ground = Region Magenta groundReg

pic :: Picture
pic = foldl Over EmptyPic [ground]
```

7.3.2 Figure8Exercise.hs

```
greenBallPos :: Float -> (Float, Float)
greenBallPos t = (x, y)
  where
        x = cos t
        y = 0

greenBall :: Animation Picture
greenBall t = Region Green $ Translate (greenBallPos t) $ Shape (
   Ellipse 0.2 0.2)

redBlocks :: Picture
redBlocks = (Region Red leftBlock) `Over` (Region Red rightBlock)
  where leftBlock = Translate (-1.5, 0) $ Shape (Rectangle 1 0.3)
        rightBlock = Translate (1.5, 0) $ Shape (Rectangle 1 0.3)

figureEight :: Animation Picture
figureEight t = greenBall t `Over` redBlocks

anim :: Animation Picture
anim = figureEight
```

7.3.3 SolarSystemExercise.hs

```
sunPos,earthPos,moonPos :: Float -> (Float,Float)
sunPos t = (0,0)
earthPos t = (x,y)
  where
    x = (fst $ sunPos t)
    y = (snd $ sunPos t)
moonPos t = (x,y)
  where
    x = (fst $ earthPos t)
    y = (snd $ earthPos t)

sun, earth, moon :: Animation Picture
sun t = EmptyPic
earth t = EmptyPic
moon t = EmptyPic

solarSystem :: Animation Picture
solarSystem t = foldl Over EmptyPic []

anim :: Animation Picture
anim = solarSystem
```

Bibliography

[1] Bret Victor. Inventing on Principle. http://vimeo.com/36579366 , accessed 28 Mar 2014

[2] Bret Victor. Official Website. http://worrydream.com , accessed 28 Mar 2014

[3] Andr Pang, Don Stewart, Sean Seefried, and Manuel M. T. Chakravarty. Plugging Haskell In. In Proceedings of the ACM SIGPLAN Workshop on Haskell, pages 10-21. ACM Press, 2004

[4] Don Stewart. Hs-Plugins Library. http://hackage.haskell.org/package/plugins-1.4.1. Accessed 28 Mar 2014

[5] Don Kirkby. LivePy. https://github.com/donkirkby/live-py-plugin , accessed 28 Mar 2014

[6] Python Software Foundation. Turtle Drawing Library for Python. http://docs.python.org/2/library/turtle.html , accessed 28 Mar 2014

[7] Gabriel Florit. JavaScript Live Coding Tool. http://livecoding.io/ , accessed 28 Mar 2014

[8] TOPLAP organisation. TOPLAP Manifesto. http://toplap.org/wiki/ManifestoDraft , accessed 28 Mar 2014

[9] David Grifftiths, Fluxus Environment. http://www.pawfal.org/fluxus/ , accessed 28 Mar 2014

[10] Overtone Group. Overtone. http://overtone.github.io/ , accessed 28 Mar 2014

[11] Ge Wang. ChucK Programming Language. http://chuck.cs.princeton.edu/ , accessed 28 Mar 2014

[12] Paul Hudak. Haskell School of Expression 4th Edition. Cambridge Press, 2000.

[13] Paul Hudak. SOE Graphics Library. http://hackage.haskell.org/package/HGL-3.2.0.0/docs/Graphics-SOE.html , accessed 28 Mar 2014

[14] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. ACM Transactions on Programming Languages and Systems, 29(3):17, May 2007

[15] Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt. Boomerang: Resourceful Lenses for String Data. ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), January 2008

[16] Axel Simon, Duncan Coutts. Haskell Cairo Graphics Library. http://hackage.haskell.org/package/cairo , accessed 28 Mar 2014

[17] Diagrams Community. Diagrams Vector Graphics Library. http://projects.haskell.org/diagrams/ , accessed 28 Mar 2014

[18] J. Ahn and T. Han. Static Slicing of a First-Order Functional Language based on Operational Semantics. Korea Advanced Institute of Science & Technology (KAIST) Technical Report CS/TR-99-144, Dec 1999

[19] W3C. Scalable Vector Graphics (SVG) Standard. http://www.w3.org/TR/SVG/ , accessed 28 Mar 2014