Plan/Proof-of-Concept: Game Development in Factor

Reggie Gillett
27133123
w6k8
reggie.gillett@gmail.com

Graham St-Laurent
23310121
i5l8
gstlaurent@gmail.com

Lynsey Haynes
12686119
a4h8
lynseyahaynes@gmail.com

Brittany Roesch
22015119
r1d8
brroesch@gmail.com

Gord Minaker
34615112
u4s8
gordonminaker@hotmail.com

ABSTRACT

This paper outlines the implementation of components and tools necessary to create a substantial program in Factor. In the context of this paper, discussions are based around the development of an interactive game. The authors discuss the implementation behind graphical user interfaces components including user input and output. In addition, we discuss some components of the Factor language that are integral to development including the object system, unit testing and debugging.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Languages

Keywords

Factor, Game, stack-based languages, concatenative languages

1. INTRODUCTION

For our end-of-term project, we have outlined a plan to create a substantial program in a stack-based language called Factor. Previously we have researched the tools and features in the language, and discussed the implications of programming in a concatenative language. The current paper expands on our previous investigations into the language and discusses how we plan to use Factor to accomplish the development of our game. Along the way we will highlight the components required for our implementation, provide example code behind our implementation, and provide screenshots of our game output.

2. GAME OVERVIEW

2.1 Game Design

In our game, the player will be presented with a collection of hamburger ingredients, a collection of actions, and a goal hamburger. It is the player's task to apply the various actions to the ingredients in an appropriate order to produce the goal hamburger, and there can be more than one way of doing it. The catch: the ingredients are presented in a regular Last-In-First-Out stack, and the actions can, correspondingly, only be applied to the topmost elements of the stack. Initially, our game will only consist of one level, with a small collection of ingredients and a small selection of actions, but the complete game would consist of increasingly complex levels gradually introducing a large variety of actions and ingredients, not all of which would necessarily be available in each level. To tie it all together and provide further enjoyment for the player, the game will provide context for each level with an overarching story about working in a (somewhat surreal) restaurant.

2.2 Game Layout and Example

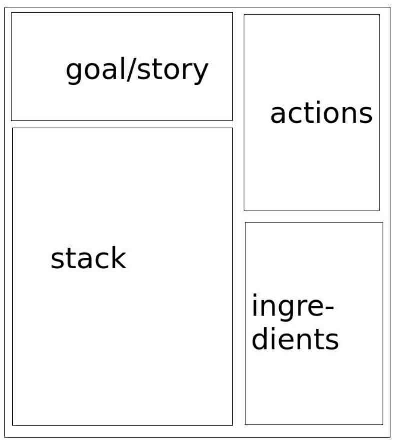


Figure 1: The window layout.

Figure 1 shows the layout for a typical game level. Each component is described below:

- **goal/story**: some text describing the current scenario and the player's current goal.
  Example: The customer wants a plain hamburger. Build one and send it.
- **stack**: a graphical representation of the current state of the stack.
  Example: From the bottom up, images of the bottom half of a bun, the top half of a bun, and a raw patty.
- **actions**: a group of buttons representing the available actions. Each button will have a name (the name of

the action), and a description of its effect.
Example: Swap (switches order of the top two items), Cook (remove the top element of the stack and replace it with a cooked version of it), Squish (takes all elements in the stack and compresses them together into a compact sandwich), Send to Customer (Removes the top element from the stack and submits it to the customer).

- **ingredients**: In levels where the player is able to push and pop elements to and from the stack, this area represents a bag of unordered ingredients. Each ingredient is a button that, when clicked, will push its ingredient to the stack.
Example: In a level with optional pickles, there could be a pickle here. In our first level, there will be no extra ingredients.

Taking as a starting point the level described in the above examples, the player could complete it by pressing, in order Cook, Swap, Squish, and Send to Customer. If the player presses Cook while there is a bun on the stack, though, it will be burnt, the customer will not be satisfied, and the player will lose. If the player presses Squish without the ingredients in the proper order, something disgusting is the result, and once again, the customer will not be satisfied and the player will lose. On the other hand, if the player follows the solution sequence but with a couple of extra swaps (Cook, Swap, Swap, Swap, Squish, Send), this will still be a valid solution and the player will win. If this swap-happy player decides to press swap one more time before Send, though, we will get a stack underflow. In order to replicate the misery of this situaion in our game without forcing an immediate loss, any stack underflow does nothing but provide an insulting message for the player.

3. OBJECT ORIENTEDNESS

Our game will be using Factor‘s implementation of objects to represent different elements that can be pushed onto the stack. An understanding of Factor’s OO syntax and behaviour will be beneficial moving forward with implmentation of our game. Although technically all variables are objects in factor, we are looking at specifically those with multiple slots, called Tuples[8]. Declaring a tuple with slots in Factor is similar to creating a new class in Java with fields. In the example below, an object called ‘element‘ with slots x, y, proper, and object would look like:

```
TUPLE: burger x y proper object ;
```

A basic constructor of an element could look like:

```
: <burger> ( -- burger )
burger new ;
```

This constructor creates a new element with empty fields. Fields can be set and read by the set and get methods. Setting the field x would look like:

```
>>x
```

while getting the value of field x would look like

```
x>>
```

A more complicated constructor might use a combination of the set methods above, or use the BOA[3] (By Order of Arguments) constructor. It creates the specified tuple by grabbing its required number of fields off of the data stack. A BOA constructor that we might use in our game:

```
: <burger> ( x y object -- burger )
 -1 swap burger boa ;
```

The tuple constructed then looks like :

```
T{burger {x 0} {y 0} {proper -1} {object [+]}}
```

4. USER INTERFACE

Central to any game is some form of user interface. The player should have some graphical interface presented to them that they can interact with. A trivial implementation of this would be a simple command-line interface. However, for our game we want a more of a challenge and in turn, a better learning opportunity. We will create windows which respond to user input and display graphics.

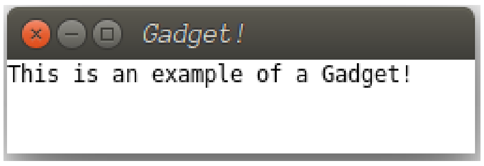
UI elements in factor are implemented using *gadgets*. Factor’s documentation describes them as: “graphical elements which respond to user input” [1]. It goes on to give a more technical description, “gadgets are tuples which (directly or indirectly) inherit from the class `gadget`”[1]. Factor’s creators provide us will a multitude of pre-made gadgets, with the ability to extend or create or own. Gadgets are key to displaying windows within a user’s operating system and we will be using them extensively in our discussion of creating a user interface.

```
: make-window ( -- )
    <pane>
    "Gadget!"
    open-window
```

The above is a word definition which creates a trivial pane. A *pane* is one of the pre-made gadgets factor provides for displaying formatted text [4]. Factor provides a word, textttopen-window, that places a gadget in a new window. It takes a gadget and a string as its arguments, the string functioning as the window’s title. We can see in the code above, we call the word that constructs a new pane: `<pane>` (this creates a pane and puts it on the stack). We then put the string “Gadget!” on the stack. We subsequently call open-window which consumes these two elements and produces a window with an empty pane and the title “Gadget!”. Essentially, this gets us a titled window with no formatting and no content. For our implementation of our game, we need to be able to make and change the content of pane and preferably we should be able to set some parameters of the window, particularly its size. We can do both of these things by extending our previous code as follows:

```
: make-window ( -- )
    <pane>
    dup
    T{ world-attributes { title "Gadget!" }
        { pref-dim { 250 50 } } }
    open-window
    [  "This is an example of a Gadget." print ]
    with-pane
```

Instead of simply passing a string to open-window we now pass a `world-attributes` tuple. This controls the attributes of the window that is being opened. This allows us to set the window's initial dimensions, title and a slew of other properties [10]. For the sake of simplicity, in this case we have simply designated a title and preferred dimensions. You will notice that we have also duplicated our pane. This will be handy when we want to add some content to it.



This is an example of a Gadget!

Figure 2: A simple window.

Each pane has a pane-stream which handles output and displays it on the associated pane rather than the standard output [4]. Factor provides several ways of manipulating this stream. The one we use above is `with-pane`. This word takes a pane and a quotation, clears the pane and calls the quotation replacing its output stream with that of the pane [9]. In our example this quotation simply prints the phrase "This is an example of a gadget". Calling this word gives us the window illustrated in Figure 2.

Gadgets can have multiple child gadgets which are displayed within them. The window itself is in fact a *World*, which is a type of gadget. The World acts as the root of a tree which in our example only contains one child, our pane. This tree structure allows for a heirarchical scheme that facilitates more complex window arrangements.

4.1 Key Commands

At some point, our game will have to receive and respond to input from the player. In most games, including ours, this takes the form of responding to key strokes or mouse clicks. In Factor terminology, user actions, including mouse clicks and keystrokes, are referred to as *gestures*. Factor conveniently provides us with a fairly simple word to implement responses to gestures in the Gadget environment: `set-gestures`. This takes a class and a hashtable as arguments. The hashtable maps gestures to quotations with a stack effect in the context of the given class[6]. The gestures are represented as tuples. We can add this to our previously created window as follows:

```
: make-window ( -- )
    <pane>
    dup
    T{ world-attributes { title "Gadget!" }
        { pref-dim { 250 50 } } }
    open-window*
    swap
    [  "This is an example of a Gadget." print ]
    with-pane
    class-of
    H{
     { T{ key-down f f "UP" }
       [ gadget-child [ "Up" print ] with-pane ] }
       T{ key-down f f "DOWN" }
       [ gadget-child [ "Down" print ] with-pane ] }
       T{ key-down f f "LEFT" }
       [ gadget-child [ "Left" print ] with-pane ] }
       T{ key-down f f " " }
       [ gadget-child [ "Right" print ] with-pane ] }
 }
    set-gestures
```

You'll notice that the call to `open-window` is now suffixed with an asterisk. This is an alternate word with the same functionality that returns the window that is created to the stack. We use it here to allow for more concise code. We then get its class using `class-of` and pass this and a hashtable containing several mappings for the direction keys. The details of the quotation are unimportant to our discussion, however the effect is printing the name of the key that was pressed in the window we have created.

4.2 Buttons

Buttons are another method of handling user input. In Factor, buttons are gadgets. Therefore they can be added to any existing gadget. The following code adds a button to our previously created interface (the gestures and text we added in the previous section have been removed for clarity):

```
: make-window ( -- )
    <pane>
    dup
    T{ world-attributes { title "Gadget!" }
        { pref-dim { 250 50 } } }
    open-window
    <pane-stream>
    "Click me!" [ "Clicked!" print ] <border-button>
    swap
    write-gadget
```

There are a few things that we do slightly differently in order to add a button. In this case we duplicate the pane we created initially. This is solely so that later in the word's execution we can attain an explicit reference to its output stream: the call to `<pane-stream>`. You will notice that we have also reverted to `open-window` without the asterisk, as in this illustrative example we no longer need the window on the stack. We create a button using the constructor `<border-button>`, which takes a string and a quotation as arguments. Factor provides several styles of button with associated constructors, for this example we use `<border-button>` which produces a standard button with a border [2]. We then do a swap-a-roo and call `write-gadget` which places a gadget in an output stream, in this case, the pane-stream we stored on the stack earlier.

4.3 Image Rendering

The final component of our UI will be the actual visual environment: images. Factor provides several different methods to do this. One way is to use the built-in OpenGL bindings to dynamically generate images. This is done by creating a new gadget to use as the canvas, and then using words from the `opengl` vocabulary to render the image. For instance, here is some code that will generate a red rectangle (rendered in Figure 3):

```
    20 20 1 glScalef     ! set 1:20 scale for XY
    COLOR: red gl-color ! current colour = red
    { 3 4 }  { 1 2 }  gl-fill-rect
       ! rectangle at pos ( 3 4 ) of size ( 1 2 )
```

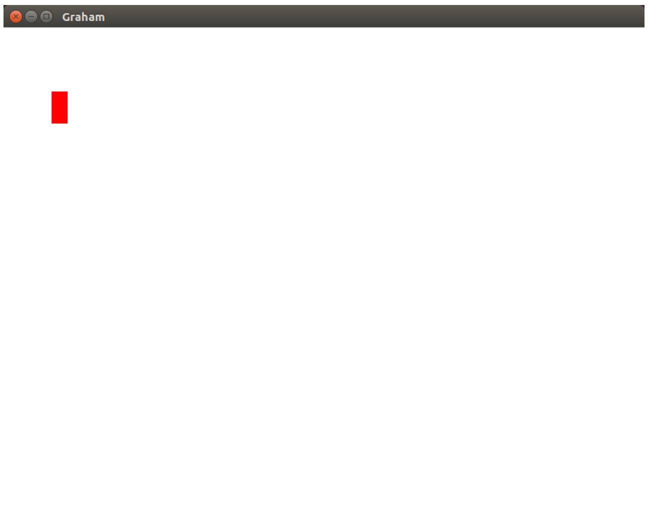
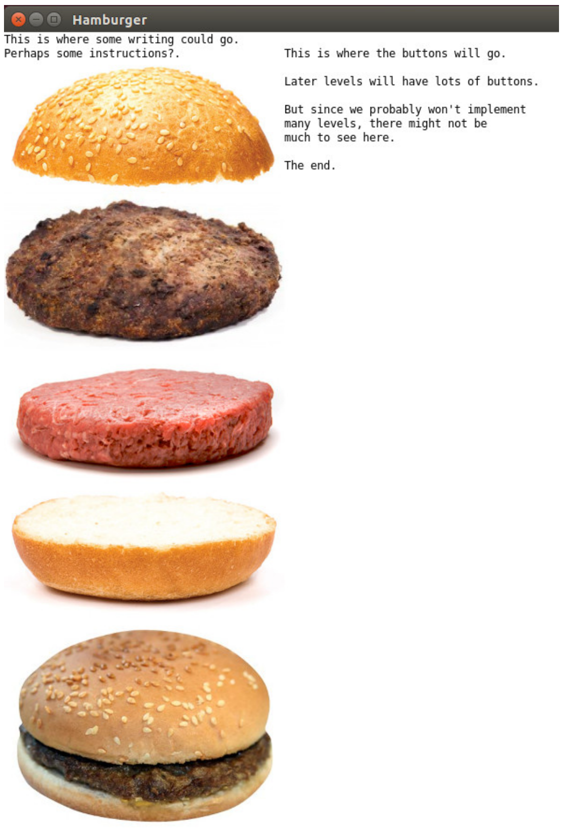


Figure 3: A red rectangle.

This can be quite laborious, but luckily, there is an easier way: simply read and render image files. Factor provides support for creating, manipulating, and displaying images in dozens of different formats. When working with ready image files, this is most easily accomplished by using the `<image-gadget>` constructor, which takes a string with an image file pathname and produces an `image-gadget`. As its name suggests, an image gadget is simply a gadget that displays an image. Here's an example that stacks a series of hamburger-related images on top of each other using a `pile` layout gadget. Then it, alongside some text (`pane` gadgets) is added to the top-most `super-gadget`, which is initialized as a 600 by 900 pixel window. The pane gadgets in the example serve as placeholders for where other components could be be a more sophisticated program (Figure 4).

```
USING: kernel accessors ui ui.gadgets ui.render
ui.gadgets.editors images.viewer combinators sequences
ui.gadgets.packs ui.gadgets.panes ;
IN: cs311

TUPLE: super-gadget < gadget ;
: <super-gadget> ( -- gg ) super-gadget new ;

M: super-gadget pref-dim* ( gadget -- {xy} )
    drop { 600 900 } ;


: burgerpic ( -- )
    <super-gadget>
    <shelf>

    <pile>
    <pane> dup { 200 200 } >>pref-dim
    [ "This is where some writing could go.
       Perhaps some instructions?." write ] with-pane
    add-gadget

    { "~/coolimages/topbun.png"
      "~/coolimages/grilledpatty.png"
      "~/coolimages/rawpatty.png"
      "~/coolimages/bottombun.png"
      "~/coolimages/burger.png" }
    [ <image-gadget> add-gadget ] each
    add-gadget

    <pane> dup { 200 400 } >>pref-dim
    [ """
This is where the buttons will go.

Later levels will have lots of buttons.

But since we probably won't implement
many levels, there might not be
much to see here.

The end.
""" write ] with-pane
    add-gadget
    add-gadget "Hamburger" open-window ;
```



This is where some writing could go.
Perhaps some instructions?.

This is where the buttons will go.

Later levels will have lots of buttons.

But since we probably won't implement
many levels, there might not be
much to see here.

The end.

Figure 4: A gadget full of images.

5. TESTING

Unit testing is important to consider in the small, initial stages of development so as not to be in a continual state of 'test deficits'. Here we quickly overview the test syntax and function of Factor. The following example is a simple addition function which takes two number values off of the data stack and returns the resulting number.

```
: add ( x x -- x ) + ;
```

In order to test this function, Factor has unit tests which are defined with a unit-test word.

The next example demonstrates a succeeding test. The unit-test word will start this evaluation with an empty stack. Next, it adds the values 3 and 7 to the stack and then will run the 'add' quotation. Finally, it compares the resulting stack with the expected output value given, which the user defines on the left hand side of the test.

```
[ 10 ] [ 3 7 add ] unit-test
```

You can also test for failing tests by using the must-fail word. This particular test will fail with a 'data stack underflow error' since there is an empty stack when the add quotation is executed.

```
[ add ] must-fail
```

If you want to be more specific with your failures, you can run a must-fail-with test which indicates what error the test will fail with. In this test, a 'no-math-method error' is thrown as a result of the 'add' word expecting only numbers to evaluate.

```
[ "three" "seven" add ] [ no-math-method? ] must-fail-with
```

Since tests in Factor are compiled to a vocabulary, it makes automated testing extremely simple. For example, if the tests above were defined as the 'addition' vocabulary, you can run them anytime by typing:

```
"addition" run
```

into the interactive listener.

5.1 Debugging Tools

Here we discuss some simple and effective debugging tools that have been useful when programming and debugging our game implementation.

Factor has a set of 'prettyprinting'' tools that allow for almost any object to be printed to the output stream[5]. It can also print the current data stack, function call stack, and retain stack. Its unparse method will turn any object into a string. This prettyprinting vocabulary can quickly and easily be used in debugging.

Factor also provides a walker which can be executed through the interactive listener. The walker will single-step through quotations. In addition to invoking the walker explicitly through the UI, it is possible to set breakpoints on words using words in the tools.walker vocabulary.[7]

6. SUMMARY

Here we have provided proof-of-concept by presenting code samples and screenshots of our implementation of all necessary UI components including keystrokes, buttons, and image rendering. We outlined some code examples that play an integral part in development, namely object orientation and unit testing. In addition, we discussed some debugging tools we have found handy and will presumably continue to do so. The development of an interactive game appears to be not only possible, but likely one of the benefits of coding in Factor. Although we have experienced several hurdles, most of which involve steep learning curves, the advantages (and disadvantages) are becoming apparent and development appears promising.

References

[1] Building user interfaces - factor documentation. `http://docs.factorcode.org/content/article-building-ui.html`.

[2] Button gadgets - factor documentation. `http://docs.factorcode.org/content/article-ui.gadgets.buttons.html`.

[3] By order of arguments (boa) constructor. `http://docs.factorcode.org/content/word-boa,kernel.html`.

[4] Pane gadgets - factor documentation. `http://docs.factorcode.org/content/article-ui.gadgets.panes.html`.

[5] The prettyprinter - factor documentation. `http://docs.factorcode.org/content/article-prettyprint.html`.

[6] Set-gestures - factor documentation. `http://docs.factorcode.org/content/word-set-gestures%2Cui.gestures.html`.

[7] Setting breakpoints - factor documentation. `http://docs.factorcode.org/content/article-breakpoints.html`.

[8] Tuples in objects. `http://docs.factorcode.org/content/article-tuples.html`.

[9] with-pane - factor documentation. `http://docs.factorcode.org/content/word-with-pane%2Cui.gadgets.panes.html`.

[10] world-attributes - factor documentation. `http://docs.factorcode.org/content/word-world-attributes%2Cui.gadgets.worlds.html`.