GSOC-13 Project Ideas for Kojo

[Kojo homepage: http://www.kogics.net/kojo]

1. Script Tracing in Kojo

Motivation/Benefit

This feature will allow users to trace programs within Kojo - to get a better understanding of what/how/when the programs do.

Background

http://gbracha.blogspot.in/2012/11/debug-mode-is-only-mode.html

High Level Specification

- There are two new UI elements - a trace button and a trace window.
- You click on the trace button to trace a program. Trace output goes into the trace window in the form of a trace history.
- When a trace run is over (this happens when a program finishes or you stop it), you can go over to the trace history to see the different trace points in the program. These are things like the entry to and exit from every procedure (command or function).
- When you click on a trace point, the corresponding source code line is highlighted in the editor, the canvas reverts to its state at that point, and you can view the value of all the visible variables/values at that point in time.
- You can move back and forth across the trace history, and jump to any point in the history.

Milestones

- Target 1 - trace a couple of non-graphics programs

  Example 1
  ```
  def factorial(n: Int): Int = if (n == 0) 1 else n * factorial(n-1)
  factorial(5)
  ```
  We should be able to see a trace of all the calls to factorial, with the different input values of n and the different return values.

  Example 2
  ```
  var counter = 0
  def incr(n: Int) {
      repeat(n) {
          counter += 1
      }
  }
  incr(3)
  ```
  We should be able to see a trace of all the calls to incr, with the different input values of n. We should also be able to see the value of counter before/after each invocation of incr.

- Target 2 - trace a small subset of the Turtle API
- Target 3 - trace the full Turtle API
- Target 4 - trace the Pictures API

The basic goal is to meet Targets 1 and 2. Anything beyond that is a plus.

Initial implementation thoughts:

- Launch a trace (headless) version of Kojo in JPDA mode.
- As the traced script runs within headless Kojo, collect its JPDA trace, and service its Canvas requests.
    - This requires the availability of a remote API within Kojo to provide Canvas services for headless Kojo. The JPDA Event mechanism might well serve as the required remote API. If not, a remote API will have to be developed within Kojo.
- Match up with the Canvas state with the JPDA trace, and put everything within trace history.
- Provide UI for trace history navigation.

Notes for Initial Experimentation

Kojo runs the code within the Script Editor inside a Wrapper object. The two Examples in Milestone 1 would run in Wrappers similar to the following:

Example 1

```
object Wrapper {
  def main(args: Array[String]) {
    // noop
  }

  def factorial(n: Int): Int = if (n == 0) 1 else n * factorial(n-1)
  factorial(5)
}
```

Example 2

```
object Wrapper {
  def main(args: Array[String]) {
    // noop
  }

  var counter = 0

  def repeat(n: Int) (fn: => Unit) {
    var i = 0
    while(i < n) {
      fn
      i += 1
    }
  }

  def incr(n: Int) {
      repeat(n) {
          counter += 1
      }
  }

  incr(3)
}
```

We can play with these Wrappers using JPDA in the following manner:

- Compile the program to be traced:

```
scalac Wrapper.scala
```

- Launch the program to be traced:

```
java -Xdebug -Xrunjdwp:transport=dt_socket,address=8000,server=y,suspend=y -cp ".:scala-library.jar" Wrapper
```

- Connect to the traced program using jdb:

```
jdb -connect com.sun.jdi.SocketAttach:hostname=localhost,port=8000
```

- Use jdb to trace method entry/exit, and local variables and fields:

```
> trace methods
> cont
> locals
> print Wrapper$.MODULE$.counter
```

Based on this, we can write code similar to jdb's code to trace methods, and determine the values of locals and fields.

Required Skills

- Good working knowledge of Scala.
- Familiarity with Java and the JVM.
- Familiarity with Swing (the Java GUI framework).
- Familiarity with 2D graphics.

Bonus Skills

- Experience with JPDA.
- Experience with Debugger Implementation.

2. Sprite Enhancements in Kojo

Motivation/Benefit
This set of features will make it easier for users to write sprite based games, animations, and cartoons within Kojo.

Background
Kojo has historically supported drawing and animation primarily through Vector graphics. This has included the command oriented Turtle graphics and the functional Picture graphics.

In recent times, Kojo has acquired good support for sprite based animation. This has involved:

- Refinements related to multiple turtles/sprites.
- Support for specifying sprite costumes.
- Support for cycling through a sequence of sprite costumes during an animation.
- Refinements to the translation, rotation, and scaling of sprites.

High Level Specification
The next set of required sprite features, which are the goal of this GSOC project, include:

- Sprite Collision Detection. This involves:
    - Implementing image edge detection for sprite images (to obtain a vector representation of the sprite image boundary)
    - Leveraging the vector collision detection support in Kojo to do collision detection for sprites.
    - Making this feature available via the Kojo API, so that it is available in games/animations.
- Sprite Messaging - to allow sprites to communicate with each other.
- Sprite 'Speaking' and 'Thinking' - to enable the display of speaking/thinking bubbles next to sprites for specified durations.

The Scratch (scratch.mit.edu) project is a good source of ideas in this area.

Required Skills

- Good working knowledge of Scala.
- Good working knowledge of 2D graphics.
- Familiarity with Java and the JVM.

3. Friendlier Error Messages in Kojo

Motivation/Benefit
This feature will make it easier for users to understand and recover from syntax errors in their scripts.

Background
One of the areas that needs improvement within Kojo relates to the complexity of the error messages that show up for scripts with syntax errors.

Project Description
There are two approaches to trying to solve this problem:

- Write custom Scala parsers for different subsets of Scala (Level 1, Level2, etc) that output suitable error messages for Kojo.
- Annotate the current error messages with helpful (in the context of Kojo) text. This approach can potentially be very powerful; it can analyze the current code to determine what the user is trying to do, the error location to determine the context of the error, and the error message to determine what went wrong. It can then combine these three elements to help the user understand and fix the error.

Required Skills

- Good working knowledge of Scala.
- Good working knowledge of Parser writing.
- Familiarity with Swing (the Java GUI framework).