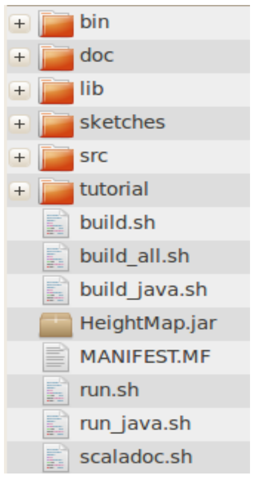
Loom

Overview

The latest incarnation of the graphics engine is called Loom (as in Jacquard Loom). Here are the key new features:

- revised directory structure
- xml configuration files store program parameters
- sound – playback of simple sound files, with distance based volume control.
- serial input – both continuous sensor readings and discontinuous readings (RFID codes and the like)
- lib directory with libraries and native code for serial reading. Have also included the scala-library.jar so that projects can run on machines that are not running scala
- range of build and run options (both Scala and Java)

Here is what the directory structure looks like:

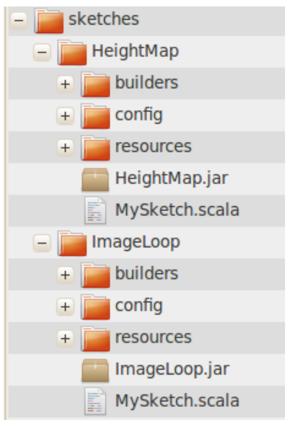


```
+ bin
+ doc
+ lib
+ sketches
+ src
+ tutorial
  build.sh
  build_all.sh
  build_java.sh
  HeightMap.jar
  MANIFEST.MF
  run.sh
  run_java.sh
  scaladoc.sh
```

bin	Compiled classes.
doc	API documentation
lib	Serial libraries and the scala-library.jar
sketches	Examples and user sketches
src	The source code for underlying engine
tutorial	PDF tutorials
build.sh	A range of build files, including one for scaladoc
.jar	Java archive file
manifest.mf	For building the Java archive
run.sh	Run files for Scala and Java

You don't need to worry about most of the directories. You will obviously need to consult the material in the tutorial and doc directories, but the main place that you will be dealing with is the sketches directory. This is where your sketches go. There are two examples sketches in the sketches directory. They serve as a good basis for developing your own sketches (just copy one of the sketches and rename it). Notice that each sketch no longer contains all the source code. The source code is now kept separate from individual sketches. This makes the project easier to manage, maintain and update and also tends to make compilation of your MySketch.scala file much faster.

Here is the overall structure of the sketches directory:



```
– sketches
    – HeightMap
        + builders
        + config
        + resources
          HeightMap.jar
          MySketch.scala
    – ImageLoop
        + builders
        + config
        + resources
          ImageLoop.jar
          MySketch.scala
```

Each of the two examples sketches (HeightMap and ImageLoop) contain the following directories:

- a builders directory which contains copies of the build and run files needed to build and run that project. Each directory actually contains 3 directories of these files – one for default (non serial input running), one for photo-resistor input and one for RFID input.
- A directory of configuration files. These are XML files that contain vital program parameters. More on this soon.
- A resources directory which contains directories for still images, image sequences and (wav) sounds.
- A copy of the compiled Java archive file.
- And most importantly the MySketch.scala file where all of your coding happens. This file is very similar to the file that you are used to working with in earlier versions of the engine.

Configuration Files

Previously we passed program parameters in the build.sh file. Now we have too many parameters to make this practical, so they have been placed in xml configuration files. You edit configuration files to modify aspects of program operation. Here are the parameters:

- **name**: the name of your sketch (this appears as the window title when running in windowed mode)
- **width**: set the width of your sketch (for full screen applications often good to set to width of screen)
- **height**: set the height of your sketch (for full screen applications often good to set to height of screen)
- **animating**: if true then the sketch animates, otherwise it remains static
- **fullscreen**: if true then full screen mode, otherwise windowed
- **bordercolor**: in fullscreen mode, denotes RGB values for border region surrounding the sketch
- **serial**: true for serial communication (either arrays of bytes (from sensors) or RFID code tags (ascii strings)
- **port**: the name of the serial port (may well need to adjust depending upon your system)
- **mode**: serial mode - bytes (arrays of bytes - sensor readings) or ascii (RFID code string)
- **quantity**: number of bytes (or ascii characters). Enter 4 for an array of 3 photo-resistor readings (the first byte is set to -5 as a start byte). 44 for RFID codes.

Build Files

These are the files that you use to compile (and run) your program. These are different for different sketches, hence worth keeping copies in your sketch folder so that you are not endlessly editing build files. Just replace the current files with the ones that you need from your sketch folder.

The **build.sh** file is for compiling just your MySketch.scala file in your sketch directory. It assumes that all the other code for the project has already been compiled. The **build_all.sh** file compiles all the code in the src folder and can take a longish time. You do not need to run this command unless you have modified code in the src directory. The **build_java.sh** file creates the sketch jar file (for Java based running of the program). Once you have built the jar you may want to copy it to your sketch folder.

You run the various build files by navigating to the Loom directory in terminal and then calling the files:

```
sh build.sh
```

Customising the code for your own sketches is straightforward. The bold text in the following build files is what you need to adjust – adding in your own sketch names and relevant configuration files.

build.sh:

```
#!/bin/bash
cd bin/org/loom/mysketch
rm *.class
cd ../../../../
scalac -deprecation -d bin -sourcepath sketches -classpath bin sketches/HeightMap/MySketch.scala
```

build_all.sh:

```
#!/bin/bash
cd bin
rm -r org
cd ../
scalac -deprecation -d bin -sourcepath src -classpath .:lib/RXTXcomm.jar -Djava.library.path=lib src/org/loom/scaffold/*.scala
scalac -deprecation -d bin -sourcepath src -classpath .:lib/RXTXcomm.jar -Djava.library.path=lib src/org/loom/geometry/*.scala
scalac -deprecation -d bin -sourcepath src -classpath .:lib/RXTXcomm.jar -Djava.library.path=lib src/org/loom/interaction/*.scala
scalac -deprecation -d bin -sourcepath src -classpath .:lib/RXTXcomm.jar -Djava.library.path=lib src/org/loom/media/*.scala
scalac -deprecation -d bin -sourcepath src -classpath .:lib/RXTXcomm.jar -Djava.library.path=lib src/org/loom/scene/*.scala
scalac -deprecation -d bin -sourcepath src -classpath .:lib/RXTXcomm.jar -Djava.library.path=lib src/org/loom/utility/*.scala
cd bin
scala -classpath .:../lib/RXTXcomm.jar -Djava.libary.path=../lib org.loom.scaffold.Main HeightMap config_default.xml
```

build_java.sh:

```
#!/bin/bash
cd bin
jar -cfm ../HeightMap.jar ../MANIFEST.MF *
cd ../
java -Djava.library.path=lib -jar HeightMap.jar HeightMap config_default.xml
```

scaladoc.sh:

This builds the API docs in the doc directory. No need to do unless you modify the source code. Here is the command:

```
#!/bin/bash
scaladoc -d doc -sourcepath src -classpath .:lib/RXTXcomm.jar src/org/loom/scaffold/*.scala
```

Run Files

There are two run files, one (run.sh) for Scala based running and one (run_java.sh) for Java based running. Both depend upon you already having compiled your sketch. Java running also obviously depends up you having created the sketch jar file with the build_java.sh.

You run the run files by navigating to the Loom directory and issuing one of the following two commands:

```
sh run.sh
sh run_java.sh
```

The bold sections in the text of the two run files indicates the areas that need to be adjusted so that they work with your own sketches:

run.sh:

```
#!/bin/bash
cd bin
scala -classpath .:../lib/RXTXcomm.jar -Djava.libary.path=../lib org.loom.scaffold.Main HeightMap config_default.xml
```

run_java.sh:

```
#!/bin/bash
java -Djava.library.path=lib -jar HeightMap.jar HeightMap config_default.xml
```

Changes to the Source (src) Directory

We used to just have three code files, all arranged in a massive flat directory. Now the code has been broken up into relevant subdirectories ('packages' in Java parlance). I'm not going to go into any of this in detail (do a google search on Java or Scala packages if you are interested), but the overall aim has been to make the project more structured. One important consequence is that the API docs are now a bit more legible – all the logically related code is joined together in specific packages. We have geometry, interaction, media, scaffold, scene and utility packages.

Changes to MySketch.scala

Remember, MySketch.scala is not in the src directory but in your sketch in the sketches directory.

No huge changes, but five additions:

1. Added **package** declaration and a range of relevant **class imports** at the top of the sketch:

```
//package declaration
package org.loom.scaffold

//import java library classes
import java.awt.{Color, Graphics2D, BasicStroke, Image}
import java.awt.image.BufferedImage
import java.awt.geom._
import java.awt.Polygon
import java.io.File
import javax.sound.sampled._

//import loom classes
import org.loom.geometry._
import org.loom.interaction._
import org.loom.media._
import org.loom.scene._
import org.loom.utility._
```

2. Added **serial communication** code. This code can either monitor continuous serial signals (the kind of thing you get from sensors) or discontinuous, event-based serial readings (RFID for instance). The serialByteReadings variable stores an array of current continuous sensor reading values – if there are any. This is an array of byte restricted values between 0 and 255. You can access this in the update method to change program parameters. Here, for instance, is an example that uses an array of three photo-resistor values to toggle the renderer display mode. We look for the index of the lowest value photo-resistor reading (via a call to a method in the Formulas object) and then vary the renderer mode on the basis of the lowest index.

```
    //changing renderer mode on the basis or sensor readings
    val least: Int = Formulas.getLeastValueIndex(serialByteReadings)
    //println("least: " + least)
    least match {
      case 0 => groundPlane.renderer.mode = Renderer.POINTS
      case 1 => groundPlane.renderer.mode = Renderer.LINES
      case 2 => groundPlane.renderer.mode = Renderer.FILLED
    }
```

We have also added a serialEventNotify method to capture discontinuous serial events. Here is some sample code that does pretty much what the code above does but with RFID readings. RFID readings come in as a large array of ascii characters (44 in all), but we only need to the last nine to identify the unique identification code for the card. We store this reading then not as a set of bytes but as a String in the serialStringReading variable. The example below shows how RFID style communication can be managed in the serialEventNotify method:

```
  /**
  serialEventNotify is for discontinuous serial messages - like RFID codes.
  Receives messages sent from Sketch via interaction manager and serial port (in Main)
  */
  override def serialEventNotify(): Unit = {
    println("new rfid reading: " + serialStringReading)

    serialStringReading match {

      case "1500D001B" => imageSequence = new ImageSequence(17, 1, 0, "random", "ran_", ".jpg", new Vector2D(0,0), new
Vector2D(1246, 796), true)
      case "1500CFF3B" => imageSequence = new ImageSequence(15, 1, 0, "square", "sq_", ".jpg", new Vector2D(0,0), new
Vector2D(1246, 796), true)
      case "1500D0025" => imageSequence = new ImageSequence(13, 1, 0, "swirl", "sw_", ".jpg", new Vector2D(0,0), new
Vector2D(1246, 796), true)

    }

  }
```

There are any number of ways in which the serial information that you capture can affect program parameters – it can vary aspects of the rendering, call up a different image sequence, change the sprite, breed new sprites, modify the sound, etc. Up to you to consider the creative possibilities.

3. **Sound**

Basic playback of sound files is now fairly easy:

```
  //load a looping sound file
  val soundManager: SoundManager = new SoundManager()
  val soundFilePath: String = ProjectFilePath.getResourceFilePath(ProjectFilePath.SOUND, "space.wav")
  soundManager.addSound(soundFilePath, true, Clip.LOOP_CONTINUOUSLY, 0)
```

Place wav sound file in sounds directory in the resources folder. If you don't want to loop continuously then you can replace the relevant parameter with a integer number of repeats (1 to play once, 2 for play twice, etc.).

In 3D scenes you can modify the volume by a depth factor:

```
    //adjust the volume of sound 0 - the nearer the louder, the further way the softer
    val inversePercent = getPercentSoundDistance(groundPlane.location.z)
    //println("inversePercent: " + inversePercent)
    soundManager.setVolume(0, inversePercent)
```

To make this work you need to define a distance away in which the sound source becomes silent (silentDistanceZ). Need to experiment this value to get it sounding right. Check HeightMap for relevant code examples.

I should note that you can also pan sound.

There are only 8 concurrent sounds maximum for now. I know it would be better if sound was linked to sprites...but not yet!

4. **ImageSequences**

A small change. You can now make image sequences either loop or play once. The latter may be useful for sensor or RFID based playback control.

5. **Image transparency**

See the top portion and update method of HeightMap MySketch.scala for how the transparency of an image can be controlled dynamically.