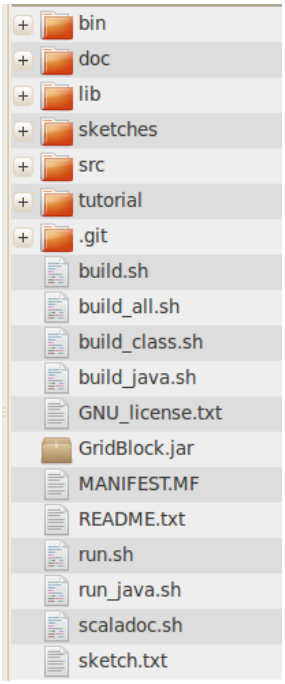
Loom

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)

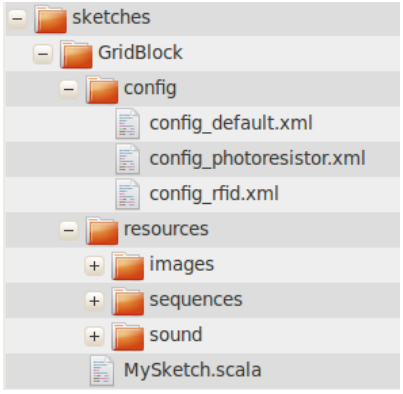
Here is what the directory structure looks like:



bin	Compiled classes.
doc	API documentation
lib	Serial libraries and the scala-library.jar
sketches	Example and user sketches
src	The source code for underlying engine
tutorial	PDF tutorials
.git	For git-hub content versioning system
build.sh	A range of build files, including one for scaladoc
GNU_license	The standard GNU license
*.jar	Java archive file
MANIFEST.MF	For building the Java archive
run.sh	Run files for Scala and Java
scaladoc.sh	Builds the scala docs
Sketch.txt	Indicates current sketch and config file

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:



Each sketch contains the following:

- a config directory that holds a set of configuration files. These are XML files that store program parameters. More on this soon.
- A resources directory that contains directories for still images, image sequences and (wav) sounds.
- 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

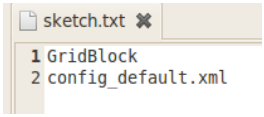
Previously we passed program parameters in the build.sh file. Now we have too many parameters to make this practical, so they have been placed in xml configuration files. You edit configuration files to modify aspects of program operation. Here are the parameters:

- **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), char or rfid (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), 1 for a single char and 44 for RFID codes.

Bash Shell Scripts (build, run, etc.)

The main loom directory contains a whole set of bash shell scripts for building (compiling) and running your sketches.

You begin by specifying which sketch you want to deal with. You do this in the sketch.txt file.



The first line indicates the name of the sketch, the second the name of the relevant config file.

You are then ready to begin building and running your sketch. Here is a description of the different build, run, etc. files:

build.sh	This compiles the MySketch.scala file in your sketch directory to the bin directory. It gets the sketch and config information from sketch.txt, as do all the rest of the relevant build/run files.
build_all.sh	This compiles all the code in the src directory. This is very slow and not necessary if you are just editing MySketch or making changes to a single class in src. Once you have compiled everything you still need to run build.sh to actually compile the MySketch.scala file in your sketch directory. This is because there is a default MySketch in the src directory that needs to be overwritten at the compile stage by your own sketch.
build_class.sh	This is for compiling just a single class in the src directory. Need to update the code to point to your particular class in the relevant package.
build_java.sh	Once you have built your sketch then you can run this command to produce a java archive file of your sketch. This is useful for running your sketch outside the Scala environment (in Java).
run.sh	This runs your compiled code.
run_java.sh	This runs the Java archive file (.jar) in Java.
run_java_mac.sh	This is necessary to run serial input files on the Mac. You cannot run them in Scala because we can't figure out how to include the -d32 parameter which forces the Mac to run the serial library in 32 bit mode. So to build and run on mac: you need to first build_all if you have not done so, then build, then build_java. These stages will throw library exception errors – don't worry. The do run_java_mac to actually run your final jar file (it should work!!!). Thanks to Etienne for this fix.
scaladoc.sh	This builds the API docs for Loom.

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 six additions:

1. MySketch inherits from Sketch (org.loom.scaffold.Sketch). I have moved as much code as possible out of MySketch and put it into the parent class (Sketch).

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.