The Guide to Psychophysics Programming

with JAVA

Huseyin Boyaci

September 28, 2006

Contents

1 Introduction

Today there are countless number of labs in the world where researchers investigate behavioral and neuronal responses of human observers to visual and other stimulation. In many of those labs computer controlled stimulus presentation techniques are extensively used. Numerous options lie before the computational psychophysicist, which range from easier to use and less flexible "blackbox" programs to harder to program but more flexible compiler languages. Yet most of those tools lack proper documentation, or the documentation isn't written exclusively with psychophysical programming in mind. In other words, what a young psychophysicist doesn't have as an option is abundance of resources that will guide him or her while learning psychophysical programming. This guide aims to contribute to community by providing a detailed overview of computer based stimulus presentation techniques. The guide is not a manual for a certain toolbox, it is a programming book for the psychophysicist. In this Guide, concepts and problems are introduced, and their solutions are implemented using Java platform, but the same principles should apply whatever programming language you use. Then why Java? Why not a guide on Psychtoolbox, for example?

1.1 Why Java?

On one hand there are "easy to use tools", such Presentation, which become painfully complicated once you try to implement slightly more interactive experiments. On the other hand, C or C++ with native graphics libraries let you get "closest to the metal" and virtually control everything as you like, but understanding their syntax is hard and they leave too much room for errors while coding. In that regard Psychtoolbox takes its admirable position in the center of the spectrum allowing both flexibility and ease of use. But nothing is perfect and it has its own share of shortcomings. Psychophysics programming with Java offers many serious benfits over any other option the researcher has. The biggest motivation in the beginning for me to switch to Java was "communication": I needed a platform which allowed me share my experimental designs with my collegues, I thought that I should be able to e-mail them a single file and it should run with just a mouse click whatever operating system, whatever computer architecture they are using. Moreover, I needed to be able to communicate with myself: I use a Mac in my lab and office, but Linux at home and on my laptop for fMRI experiments. I found that lack of a solution to this simple need was unacceptable. Do I have to prepare two versions of each experiment if I want to work on it both at home and in the lab? And a third one to send to my collegue, which runs on MS Windows using DirectX libraries? What about sharing it with others who visit my web page? After switching to Java, my collegues and I found solutions to problems in our experimental designs literally over night, over a few e-mails. I can now convert my experiments into Java applets by changing just a few lines in my code and place them on web page. Advantages of Java are not limited by ease of communication. Due to its object oriented design and technologies, Java provides a very effective platform of programming and development. It boosts the *productivity* in several other ways as I will explain in more detail below.

1.2 What Java offers to the Psychophysicist

First, the key technologies that concern the computational psychophysicist.

Full Screen Exclusive Mode and new Java2D API

Full Screen Exclusive Mode (FSEM) is introduced to core Java platform with version 1.4. Sun probably had the gaming industry in mind when they introduced it, nevertheless this offered a marvelous opportunity for psychophysicists. This mode and the new Java2D API, allow capturing the entire screen, provides tools for double buffering and proper vertical synchronization, all of which are essential for psychophysics programming. Depending on the OS/architecture, Java may use different strategies in FSEM. For instance on MS Windows it will most probably use the DirectX routines, on Mac OS X it uses native OpenGL libraries, on Linux/Unix it uses X11 libraries but can be instructed to use OpenGL libraries instead.

Architecture specific details are hidden while low level access is still possible

Engineers at Sun do the hard work of hiding the layer between you and the low level interaction with the hardware. This way your code behaves in the exact same way on different platforms even though the Java interpreter may do completely different jobs. On the other hand Java doens't lack ways for low level hardware access for visual displays.

3D graphics

For 3D graphics there are a few options. Among the most popular ones are Java-3D, which provides a high level scene graphics API, and JOGL, which stands for Java bindings for OpenGL. These two options offer different levels of flexibility and level of hardware access. There is a strong community support behind those open source projects.

Converting your experiments into Java Applets

It is a matter of changing a few lines to convert your experiments into Java applets, or web applications (called as WebStart technology).

Pack and send, share your work with collegues

You can package your experiment in a so called jar file and send to your collegues. Whatever platform they are on, they can run the experiment with a mouse click.

Networked experiments

Java platform has excellent support for network programming, providing all the necessary tools for networked experiments.

Multithreaded programming

In psychophysics or neuroimaging experiments, particularly with animations and simultaneusly running task trials, it is desirable to have a program which can run multiple tasks concurrently. Core Java platform provides an extensive collection of tools for multitasking programming, also known as multithreaded programming, or parallel programming.

Matlab and Mathematica interface

It is a fact that many computational psychophysicists rely on Matlab or Mathematica for programming. Those who like to continue using their preferred environment can do so, because both Matlab and Mathematica offer allow interaction with Java objects.

Next, more general advantages of Java platform for improving productivity.

Cross platform portability

Even though it is not as perfect as it was promised, Java *is* platform independent (OS and architecture) to a great extent. The behavior of my experimental programs were always identical on all three systems I used (Linux, Mac OS X, MS Windows). This improves productivity and communication:

- improved productivity at programming phase - easy team work: Researchers working on their preferred OS can still work on the same piece of code together.
- Improved productivity at application phase: Researchers can work on their preferred environments and can still run their experiments on different systems in their labs. For example, they can prepare an experimental program to measure the behavioral response to a visual stimulus in their psychophysics lab on Mac OS X and use the same code on a PC to determine the behevioral effect to the same stimulus inside the scanner before an fMRI experiment.
- Code sharability: Once written a class (objects belong to classes), you can use the same class again and again (see also Object Oriented Programming below), moreover anybody else can use the same code on their own platform. This is actually a great asset because it means that you can get advantage of many existing classes and build upon them new tools.

Object Oriented Programming advantages

- Extendibility and reusability: OOP by design targets re-using and/or extending existing classes. Mind you that this is not just re-use of your functions in a more traditional language. You will find many examples where we take full advantage of OOP in this guide.
- Hide the implementation: By hiding the implementation you can write code which does not break the client's program with every new version.
- Strict type checking: Once it compiles a Java program usually runs without any problem, you will not get memory errors as you would in C. This is because Java compiler is extremely strict in variable typess. Java won't do the conversions for you, it wants *you* to do the "type casting" to see that you know what you are doing.
- Exception handling: In languages like C, and Fortran there is no real exception handling. You invoke a method and if it fails your program crashes. Or they may return a funny result which indicates that the method failed to accomplish what you asked from it. Java provides a much more advanced approach to handling such errors through its Exception class. This way you can put the mechanisms in your code which will fix the problems during its execution. Even if it is unfixible, you can instruct your code to gracefully exit rather than crash.
- Automatic memory management: Java has an automatic "garbage collector". You usually do not have to worry about memory allocations and deallocations. In case you have special needs you can still manually manage your memory.

Easier to code

- Extensive API (collection of thousands of classes) in the core Java platform. This eliminates to re-write many classes and their methods, because as discussed above using existing classes is the nature of OOP.
- Excellent documentation of the core classes. You can easily reach the most current documentation online at java.sun.com. The documentation is superior to even older compiler languages such as C.
- High level: One line of code does a lot for you when you use existing classes as building boxes. This makes your code less complicated, easier to read, and less prone to errors.
- New generation of Integrated Development Environments (IDEs) provide very easy and effective programming environment. One commonly used IDE is Eclipse. The editor of Eclipse flags the mistakes as you are typing your code and suggest possible options to fix those mistakes. You do not need to remember which libraries to import, because Eclipse will do that for you automatically, and suggest automatic code completions.
- Apart from the IDEs, the faithful user can still use Matlab or Mathemetica as interface to Java objects.

1.3 Comparison to other tools, particularly to Psychtoolbox

Java is a programming platform, which is as complete as possible for the modern programmer, covering a whole range of areas, from parallel programming to network tools. It is not fair to compare Matlab to Java in those areas, because Matlab is not a real programming platform, it is an advanced tool for numerical analysis of small technical problems. It is quicker to learn, quicker to implement small programs in Matlab. But it does not provide the same flexibility and variety of tools as Java. Psychtoolbox was motivated by the fact that it was hard to learn the syntax of C and graphics libraries in the absance of development environments as we have today. It is also hard to code in C because it leaves too much room for errors. Therefore there emerged a need to place a layer between C and the novice programmer. Psychtoolbox has been that layer for many years. But now Java opens another door eliminating the need to that extra layer, because it offers not only a much advanced programming platform but also a programming language, which is easy to code in.

1.3.1 Advantages

- Java is platform independent, Psychtoolbox is not: Matlab runs on multiple platforms, but Psychtoolbox does not. Even on the platforms that Psychtoolbox runs you still have to make some modifications in your code if you want to port it to another platform (actually the platforms supported by the latest version are limited by two, MS Windows and Mac OS X).
- You can convert your experiments into Java applets, there is no such option with Psychtoolbox
- You can prepare a single executable file of your experiment and send to collegues, they can then run it with a mouse click. There is no such possibility with Psychtoolbox.
- Psychtoolbox doesn't provide tools for 3D rendering. Java offers multiple options, at multiple levels of ease and levels of programming.
- Java has extensive coverage in threaded programming and network programming areas, Psychtoolbox doesn't.

- Object oriented programming, and other advanced programming techniques inrease productivity. Matlab/Psychtoolbox doesn't provide any such advanced programming techniques.
- New generation IDEs offer much superior programming environments compared to Matlab's editor. One such IDE is an open source project called Eclipse.
- The underlying graphical details of Psychtoolbox are hidden away from the researcher. One can in principle download and inspect the C source code, but it is not such an easy task to fully understand what is going on. You would also need to understand the native graphics libraries on the two OSs that it runs. This is a discomfarting solution for a researcher. Sooner or later the researcher will need more independence then that.
- Java is freely available (though its source is not completly open.) Matlab needs a licence to run. You either purchase a standalone licence, or use a network licence if your institute has one. The first option is more reliable but not cheap. The second one is free for you, but frequently suffers from number of user limitations and network distruptions. A lab under tight budget could reduce the costs considerably by using a Java system (one could further reduce the costs if the system is built on Linux).
- Backward portability: It is unlikely that a program written for an older version of Java platform breaks when run with a newer version. You can upgrade your Java platform to the latest version without the fear of breaking older code. Matlab/Psychtoolbox frequently breaks older code.

1.3.2 Disadvantages (mispercieved and real)

It is easier to program in Matlab, OOP and Java are difficult

It is definetaly simpler to program small projects in Matlab as a starter. But this simplicity doesn't scale to larger projects and it does not provide any of the productivity benefits that OOP offers in the longer run. Yes, if it is a highschool intern in your lab one may then argue that learning Java is going to waste his or her time. But you can provide a framework for experiments in a specialized class and let your student use that class, even create objects of those classes from within Matlab or Mathematica if they prefer that option. I am building such a framework in this guide and it is successfully used by inexperienced students easily. As you will see starting with Chapter 2, using such a framework can be as simple as using Psychtoolbox, if not easier! Moreover, Java and OOP are taught in courses in many Computer Science programs both at undergraduate and graduate levels. New generation of students do come to our psychophysics labs with that knowledge and are usually more keen on using Java rather then Matlab.

Community support: Many Psychophysicists use Psychtoolbox

Psyctoolbox/Matlab option is extensively used by our community. This naturally leads to a strong community support. For the moment there is no such community for Java. Your source of support will probably be the Java gaming community for now.

Integration with analysis programs: I run my experiments with Psychtoolbox, analyze them with Matlab

If you are doing your analysis using Matlab, it may seem like it is a better idea to limit yourself only to that program and use Psychtoolbox. But you can use Java from within Matlab/Mathematica. As for the pure Java programmers, I think soon there will be enough numerical computation tools, if not already are, available to perform any important statistical analysis with Java. The number of such methods may exceed the number that exists in Matlab. The widely used gnu Scientific Library (by computational physicists for example) may soon be ported to Java. I will provide some numerical algorithms in the end of the Guide.

Performance

Java *is* slower than low level compiler languages such as C. However the difference is shrinking with introduction of new technologies, notably the Hot Spot Technology. Nowadays the speed of Java is about 2/3 of C. And certainly Java is not inferior to Matlab in terms of speed. I have not met a situation where my experimental code suffered from cpu or gpu speed. Only in numerical computations I found C superior to Java (less than one fold), Java superior to Matlab usually a few folds!

Hardware manufacturers provide interfaces with Psychtoolbox, not with Java

This is the only real disadvantage of using Java in my opinion. Many companies provide interfaces using Matlab/Psychtoolbox to use their devices. For example CRS provides Matlab functions for Bits++ which integrate with Psyctoolbox. But as you will see in Chapter A, it is not all that difficult to write your own interface if you know how to program independently and as the number of users increase there will be many classes available on the internet for such purposes. Since Java allows low level access it is usually possible to build your own interfaces. Besides one would expect the manufacturing companies should soon notice the advantage of building their tools on Java to reduce their costs, just like Matlab did.

1.3.3 So, which one to choose?

In summary, Psychtoolbox was a temprory solution. The C compiler language was hard to learn and hard to code with, there was a need for the community to find an easier solution. Psychtoolbox provided that solution using an unlikely ally Matlab, which is actually an advanced numerical analysis tool, not a fullscale programming environment. But the compilers constantly advance, the IDEs get better and better with large community support and the need for such temprorary solutions is eliminated.

All of the disadvantages that Java suffers from will soon change. As the number of psychophysicists using Java increases, a community will form. Besides, there is already a very strong community support behind Java2D, java3D, and JOGL. Even though they mostly have gaming, not psychophysics programming in mind I benefited a lot from them.

However, if you already have library of functions to run your experiments using Psychtoolbox and rely heavily on Matlab for analysis, it wouldn't make sense to port all those to Java now, but I would recommend implementing the new ones in Java. If you are newly creating your library or faced a situation where Psychtoolbox isn't sufficient any more, switch as soon as possible to pure Java programming.

Another option is using Java objects from within Matlab of Mathematica. Even if you like to continue using your preferred platform, Matlab or Mathematica, you can still create and use Java objects from within them. See Section 1.4.2 below. Using Java objects has one distinctive advantage: they are platform independent! Matlab and Mathematica both have multiple OS versions, for the end user they are platform independent. Whereas Psychtoolbox fails to be completely platform independent (and built only for Matlab). When you replace Psychtoolbox with Java objects you would virtually be completely platform independent even if you are using Matlab or Mathematica for development.

1.4 How to use the Guide

1.4.1 The Guide as a programming book

Throughout the Guide I introduce the tools of Java, which are useful for psychophysics programming. I start with "Hello Psychophysicist" chapter where I introduce the essentials of displaying visual stimulus on displays. In the following chapters I introduce further details for psychophysics programming, such as

multithreaded programming, event handling and accurate timing. While introducing the tools from core Java interface, I also build up custom classes, which are suitable for reuse to design real psychophysics experiments. I do not shy away from the details and go to greater lengths to explain them. I provide simple "test" programs, some of which are replicates of real experiments I actually used in my own research, some are small tests of the functionalities just introduced in the chapter. In Appendix B, I introduce some numerical analysis methods. These are simple but very useful classes. One of them provides methods for computing population statistics like mean, varience and others. Another class provides methods for function minimization.

The reader who wants to learn and program in Java platform should read the entire Guide. Chapters 2 to 9 are essential for the fundamentals. The remaining chapters can be read selectively. For instance if you have a multiple display system you should read Chapter 10, as well. Converting your experiments to Java Applets is discussed in Chapter 11. Appendix A explains how to utilize Bits++ using Java. There is no real prerequisete but some modest Java or object oriented programming knowledge would be helpful. I try to reference further readings if I use an advanced programming technique.

1.4.2 The Guide as a manual to the tools developed

In the end of the guide, in "Chapter C: Demos, installation instructions, and Manual pages", I provide a glossory of all the classes and methods I built up, and show how to use them (not how they work). I point to the chapters where the details are discussed in greater lengths. Consider this as manual pages for the classes. I present numerous demos in that Chapter. Those demos are based on the test programs from earlier Chapters. This way those who want to get more detailed information could turn to those chapters. I implement the same test programs in Matlab and Mathematica, as well.

1.4.3 List of classes and methods built in the Guide

FullScreen

Initiates FSEM window, and provides methods to present stimuli on the secreen and to collect observer response

- FullScreen() - initiates FSEM, inherits from Java core class JFrame hence all its mehods are also available
- displayImage() - diplays image
- displayText() - displays text
- blankScreen() - blanks the screen
- setNBuffers(), getNBuffers() - set/get the number of video buffers
- updateScreen() - updates the screen by moving the back buffer (if it exists) to front
- setBackground(), getBackground() - set/get the background color
- hideCursor(), showCursor() - hide/show cursor
- closeScreen() - terminates FSEM
- getKeyPressed(), getKeyTyped(), getKeyReleased() - obtain observer's key presses etc.

- getWhenKeyPressed(), getWhenKeyTyped(), getWhenKeyReleased() - obtain the time of key presses
- flushKeyPressed(), flushKeyTyped(), flushKeyReleased() - clear the que of key press events
- isFullScreenSupported() - check whether FSEM is possible on your system
- isDisplayChangeSupported() - check whether display mode change is supported on your system
- isDisplayModeAvailable() - check whether a particular display mode available
- setDisplayMode(), getDisplayMode() - set/get display mode (resolution etc.)
- getDisplayModes() - obtain all available display modes
- reportDisplayMode() - obtain a readable version of current display mode
- reportDisplayModes() - obtain a readable list of all available display modes

NormalWindow

Similar to FullScreen, except stripped of Full Screen Exclusive Mode specific methods. It opens a normal window instead of FSEM window. By replacing your FullScreen object with NormalWindow object you can get an application to put on your web page. (See Chapter 10.) It allows user to decide on various rendering strategies. (FSEM always uses active rendering, in normal window you can decide between passive and active rendering.)

- NormalWindow() - initiates a normal window, inherits from Java core class JPanel therefore it has all its methods available
- setPassiveRendering() - set passive rendering true or false (see Chapter 10)
- isPassiveRendering() - obtain the current state of passive rendering, true or false
- displayImage() - diplays image
- displayText() - displays text
- blankScreen() - blanks the screen
- updateScreen() - updates the screen by moving the back buffer (if it exists) to front
- setBackground(), getBackground() - set/get the background color
- hideCursor(), showCursor() - hide/show cursor
- getKeyPressed(), getKeyTyped(), getKeyReleased() - obtain observer's key presses etc.
- getWhenKeyPressed(), getWhenKeyTyped(), getWhenKeyReleased() - obtain the time of key presses
- flushKeyPressed(), flushKeyTyped(), flushKeyReleased() - clear the que of key press events

Clut

Provides methods for (inverse) color look up operations. See Chapter 8 and Chapter A.

- Clut() - creates a CLUT object, and sets the values in the look up table either by reading a file holding the table or from an array
- setClut() - sets a the values in the look up table
- lum2Pix() - finds the pixel that results in a luminance value closest to the desired luminance value
- pix2Lum() - returns the luminance that results from the given pixel
- getMaxLum() - get the maximum luminance value in the table

BitsPP

A class which has methods for using Bits++. It inherits from FullScreen so it has all its methods and some additional methods needed to communicate with Bits++ device. See Chapter A.

- BitsPP() - initiates a Full Screen Exclusive Mode window and initializes Bits++ look up table
- initClut() - initializes Bits++ look up table
- setClut() - set the Bits++ look up table
- displayImage() - diplays image by using Bits++ look up table
- displayText() - displays text by using Bits++ look up table
- blankScreen() - blanks the screen by using the Bits++ look up table
- closeScreen() - terminates FSEM and leaves the screen in a nice condition by setting a linear Bits++ look up table

BitsPPFake

Similar to BitsPP class, except it is fake. Designed mainly for development when away from your Bits++ device.

- BitsPPFake() - initiates a Full Screen Exclusive Mode window and initializes Bits++ look up table
- initClut() - initializes Bits++ look up table
- setClut() - set the Bits++ look up table
- displayImage() - diplays image by using Bits++ look up table but only its most significant 8 bits

Sample

This class provides methods to compute statistics of sample populations.

- More information Coming soon!.....

Distributions, FunctionMulti, FunctionSingle, Integration, Minimize, MLE_TwoAFC, SpecialFunctions etc.

Classes which provide numerical analysis methods.... More information Coming soon!....

1.5 Setting up the Java platform

The latest Java development kit (JDK) can be downloaded from `http://java.sun.com` (current latest stable version is 5.0. Note: you should get JDK not JRE!) That site hosts versions for various operating systems, including MS Windows and Linux. I recommend installing the documentation package, as well. There may also be an external link for the Mac OS X version. The examples in this guide requires at least version 5.0. Apart from the JDK, you may want to install a development environment. One such environment is Eclipse. I will provide more information on Eclipse below. Eclipse has versions for all three OSs and is an open source project, available freely.

1.5.1 MS Windows

The installation on MS Windows is usually straightforward. However, if you had an older version of JDK you may want to it after installing the newer version.

1.5.2 Mac OS X

The Mac OS X version of Java 5.0 can be found at http://www.apple.com/downloads/macosx/ `http://www.apple.com/downloads/macosx/` by searching for Java SE 5.0. Get the latest update. On Mac OS X (Tiger) the default JDK is still version 1.4 (as of September 2006, no version 5.0 for Panther is available). When the 5.0 version is installed the default Java compiler and virtual machine are not automatically updated. To fix that you should run the preferences utility under /Applications/Utilities/Java/J2SE 5.0/ directory. If you like to inspect further, Java Virtual Machine (VM) is located under /System/Library/Frameworks/JavaVM.framework.

1.5.3 Linux - Fedora Core 5 (FC5)

Linux installation on some distributions may need extra work. Please consult your distribution's documentation. Below I will discribe how to set up a Fedora Core 5 system for psychophysics experiments.

Installing Sun's JDK

Due to its closed source and its license Sun's Java Development Kit is not included in Fedora Core Distribution. Instead FC5 ships with gnu version of Java, which is completely compliant with Sun's version but currently lacks necessary graphical tools for psychophysics experiments. Therefore you first have to install Sun's JDK. There are various ways of installing Sun's JDK. One way is using the online FC5 repositories. These repositories provide pre-compiled binaries and places them in proper locations on your system and performs necessary 'alternatives' commands.

Other option is manual installation. First get the latest version from http://java.sun.com, get the beta of version 6.0, not 5.0 because 6.0 works better in FSEM on Linux systems. However, if you would like to use Matlab for development you should install version 5.0. Download the "Linux self extracting file" not the rpm package. After download, open a terminal and become root (su -), move the file to /opt directory

```
mv jdk-6-beta2-linux-i586.bin /opt
```

change directory to /opt

```
cd /opt
```

change the mod of the file to executable,

```
chmod +x jdk-6-beta2-linux-i586.bin
```

and then execute the program

```
./jdk-6-beta2-linux-i586.bin
```

this will install the platform. After install succeds, you should tell your system that the Sun's JDK/JRE is your primary java platform, not the gcc-java which is the default package in FC5. Create a symbolic link

```
ln -s jdk1.6.0 jdk1.6
```

install the new JDK as java alternatives

```
/usr/sbin/alternatives --install /usr/bin/java java /opt/jdk1.6/bin/java 2 \
--slave /usr/bin/javac javac /opt/jdk1.6/bin/javac --slave /usr/bin/javadoc \
javadoc /opt/jdk1.6/bin/javadoc --slave /usr/bin/jar jar /opt/jdk1.6/bin/jar \
--slave /usr/bin/javaws javaws /opt/jdk1.6/bin/javaws
```

configure the java alternatives

```
/usr/sbin/alternatives --config java
```

choose the newly installed JDK as the primary choice. Check the alternatives to java

```
/usr/sbin/alternatives --display java
```

Because of the symbolic link, you don't need to install new alternatives each time you install an update to JDK/JRE. For browser (firefox) plugins to work, do the following as root

```
ln -s /opt/jdk1.6/jre/plugin/i386/ns7/libjavaplugin_oji.so \
/usr/lib/mozilla/plugins/libjavaplugin_oji.so
```

Installing Eclipse

FC5 includes a natively built Eclipse package. You shouldn't download and install Eclipse from eclipse.org, instead use the "Add/Remove Software" under Applications menu. Click on Applications→Add/Remove Software. Click on Development and check the box next to Eclipse, then click apply.

1.6 Getting the source code for the Guide

The source code of the programs in this guide can be found following the links at `http://tc.umn.edu/~boyac003/Guide`. You can either download individual programs or download the compressed package. After downloading and putting them in a directory, follow the directions in the next section to compile and run the programs.

Those readers, who would like to first try the demos and install the final version of all tools in a package called PsychWithJava can find the instructions to do so in Chapter C: The Guide to Psychophysics programming with Java: Manual pages and demos.

1.7 Development environments

1.7.1 Using command-line tools

This is the harder way of development, nevertheless it is very useful to know in case your options are limited, for example on a colleague's computer who doesn't have an advanced IDE installed. As an example, I will show how to compile and run the sample code from Chapter 2. First download all the files from Chapter 2 into a directory. Open a terminal, and change directory to PsycWithJava/ch02/, where PsycWithJava is the directory where you downloaded the code, type

```
cd PsycWithJava/ch02
```

and press enter (if using MS Windows replace the slash with back slash). Then type

```
javac HelloPshychophysicist.java
```

and press enter. This will "compile" the source code. After this you will see couple of something.class files in your directory. If there were no errors during the compilation, type

```
java HelloPsychophysicist
```

and press enter. You should now be running the first example of the book detailed in Chapter 2. Note that, you also need FullScreen1.java file in the same directory. Even though you didn't compile FullScreen1.java, the compiler still compiled and produced FullScreen1.class file. Because HelloPsychophysicist uses FullScreen1 and the compiler automatically compiles other necessary files in your directory.

To edit your code you can use any of your favorite editors, for example vi under Linux/UNIX systems.

1.7.2 Using Eclipse, an Integrated Development Environment (IDE)

Integrated development environments (IDE) certainly make the development phase easier and boost productivity. One of many such environments is Eclipse, which is an open source project and available at `http://eclipse.org`. I will demonstrate how to use Eclipse here. First install the latest version (3.1 currently) for your OS (Fedora Core 5 users, see section 1.5.3 above for installing Eclipse). If you haven't yet downloaded the source code of The Guide, download it as explained above in Section 1.6.

Start Eclipse. (The first time it is invoked Eclipse offers a Welcome screen, it is possible to use tutorials linked from that welcome screen.)

Click on the File Menu. Click New →Project. A "New Project" window will appear. Choose "Java Project" and click "Next" (Fig. 1.1(a)). "New Java Project" dialog should appear (Figure 1.1(b))

In the "New Java Project" window, inspect the radio buttons under JDK Compliance (see Figure **??**). If the default compiler compliance isn't set to 5.0, click on "Configure default". "Properties" window should appear, using it set the compliance level. In case 5.0 is not listed among the available compilers, you may need to update your configuration as follows: Close the Properties window and New Java project window. Go to Window →Preferences, navigate to Java →Installed JREs using the list on the left column (Figure 1.2). If JRE 5.0 (or 1.5, they actually are the same thing, i.e. version 5.0 = version 1.5) is not among the installed JREs, click on Add button and add the directory where JRE 5.0 is installed. Make sure that the box next to JRE 5.0 is checked (FC5 users: choose JRE 1.6 as default, not 1.5). Finally go to Compiler item in the Preferences dialog and set the default compiler to 5.0 (Figure 1.3.)

If you had closed the "New Java Project" window open it as described above. In the "New Java Project" window type "Chapter 2" for Project name, click on the radio button next to "Create project from existing

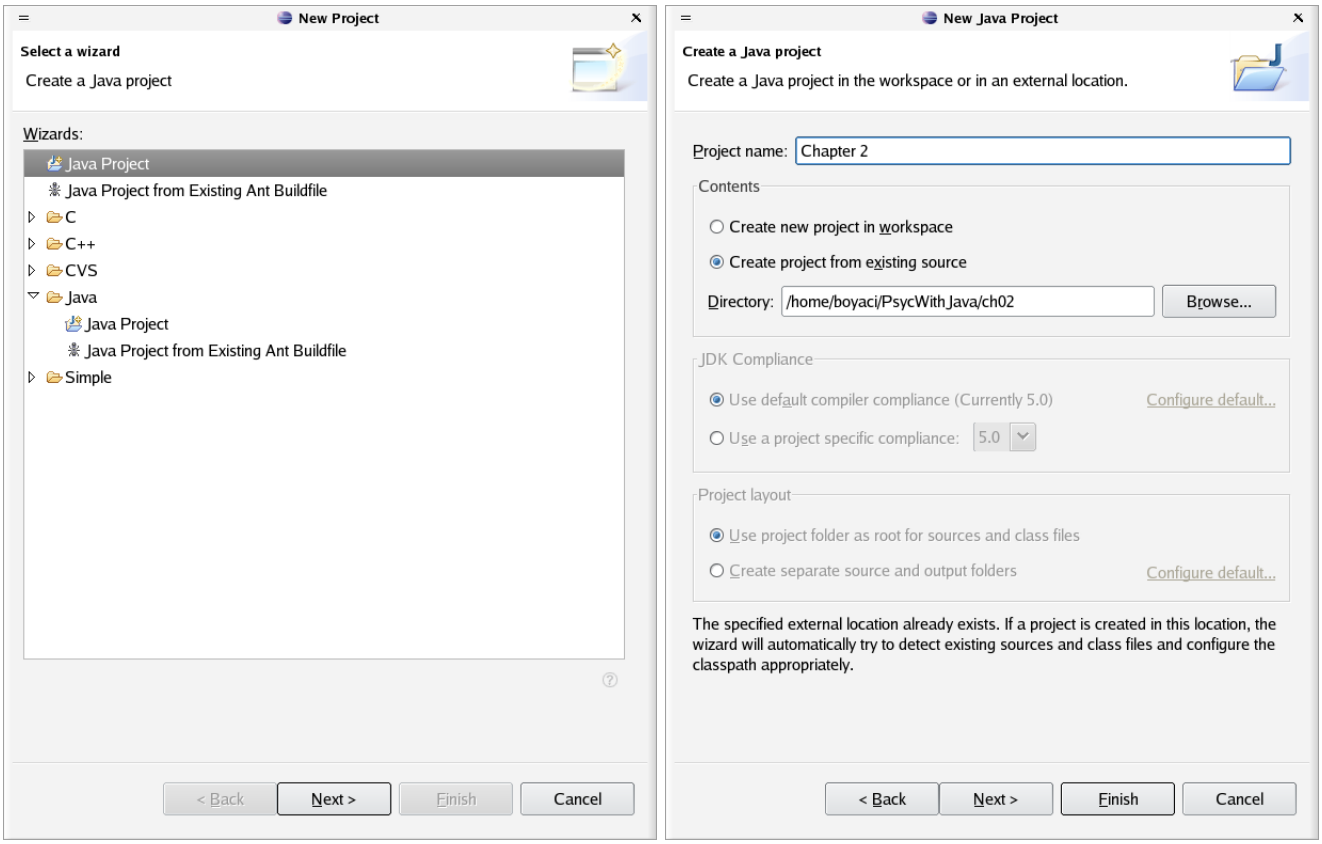

Figure 1.1: Left: New Project window. Right: New Java Project window

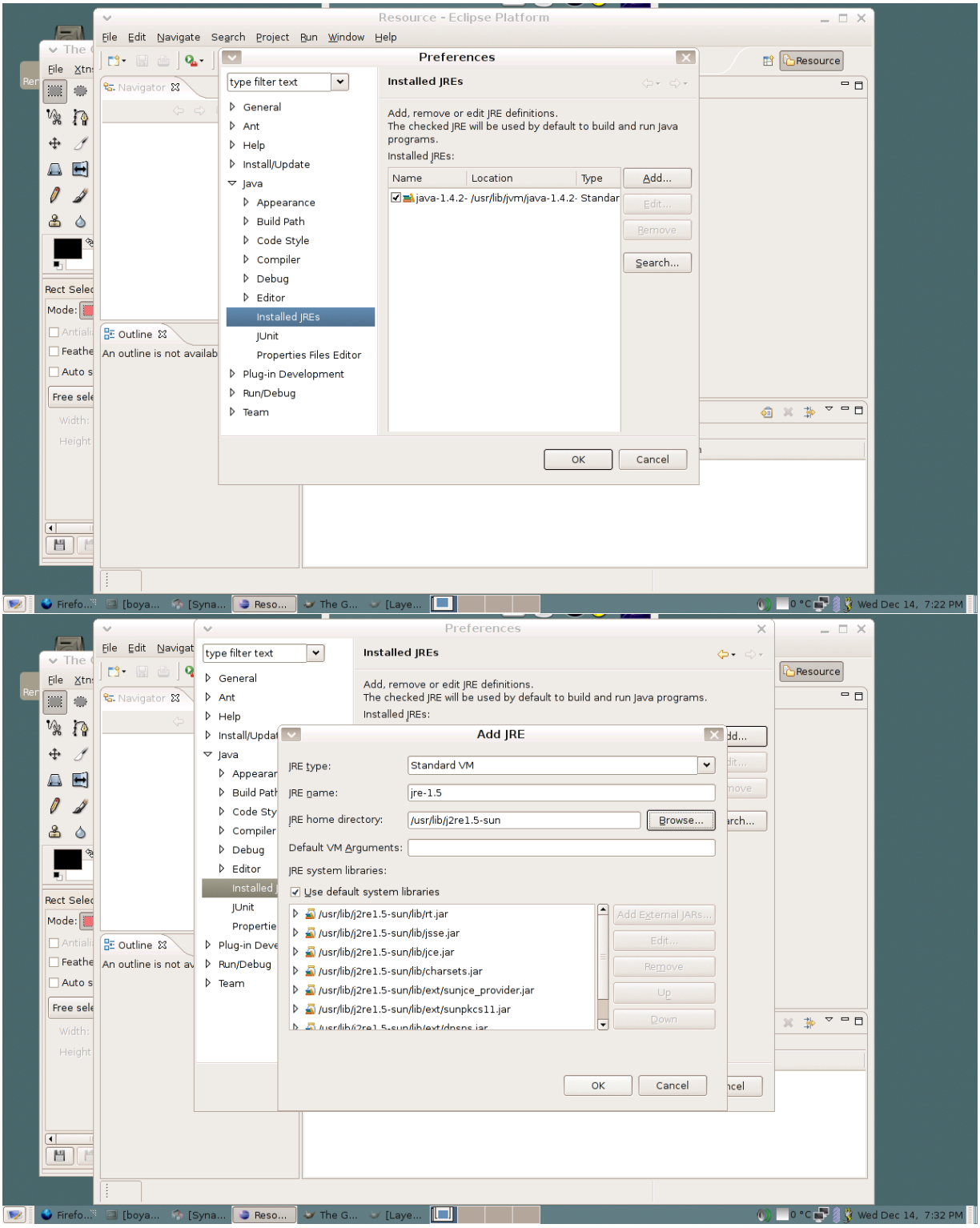

Figure 1.2: Setting up the Installed JREs.

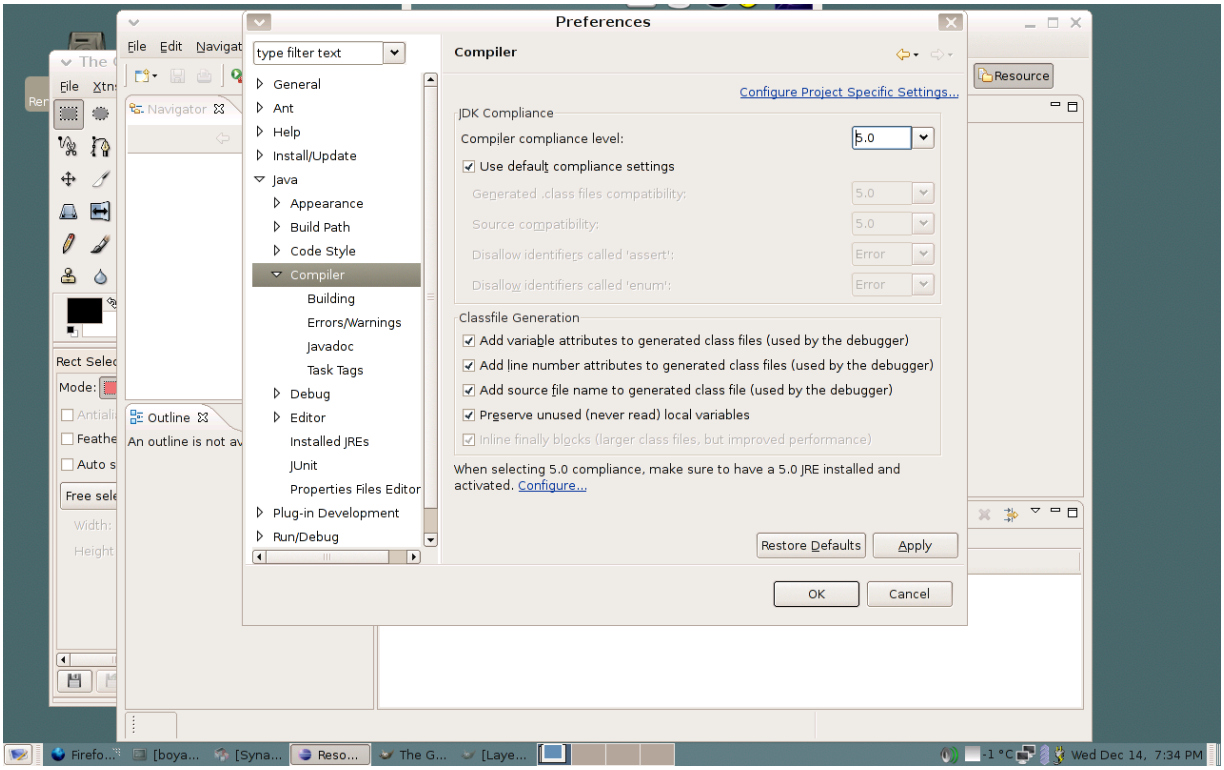

Figure 1.3: Setting the compiler compliance level.

source" and type in the full path to Guide/ch02 (this should be directory where you downloaded the source code), or alternatively click on "Browse" button to choose the directory. Click "Finish" (Fig. **??**).

Now the project "Chapter 2" is created. Click on the triangle next to Chapter 2 inside the "Package Explorer Pane", and then on the triangle next to "(default package)". Next double click on "HelloPsychophysicist.java". This will open the listing of the program in the editor pane. You can edit the code and save it by clicking on the "Save" button (the conventional floppy disk icon) or by choosing File→Save. To run the program, Right-Click (CTRL-Click on Mac OS X) on "HelloPsychophysicist.java" at the "Package Explorer Pane" and choose "run as"→"java application". You should now be running the first example of this guide. In case there are errors in the code, Eclipse flags them with "red" cards on the right side of the editor. Carefully inspect and fix them.

1.7.3 Using with Matlab

It is possible to create Java objects from within Matlab. You can also invoke methods of Java objects, which means that you can use the tools I will create in this guide from Matlab. I provide examples of doing this in Chapter C.

1.7.4 Using with Mathematica

Same as Matlab, it is possible to create Java objects and invoke their methods from Mathematica. Examples can be found in Chapter C.

1.8 Further readings

For Java platform in general I recommend the following books:

- Core Java, Volume 1 Fundamentals and Volume 2 Advanced Features, 7th edition (J2SE 5.0) by Horstmann and Cornell.
- Thinking in Java, by Bruce Eckel.

For Java2D and other graphics tools

- Killer Game Programming in Java, by Davison.

Other useful community sources

- Java2D forum: http://forums.java.net/jive/forum.jspa?forumID=69

2 Hello Psychophysicist

In this chapter

- Initiating the full screen exlusive mode (FSEM)
- Active vs. passive rendering; Double buffering
- Displaying text and image on the screen
- Pausing (sleeping) for a while
- Hiding and showing the cursor
- Terminating the FSEM

We start with a simple example, which displays the text “Hello Psychophysicist” and two images on an otherwise entirely blank screen (see Chapter 1, *Development environments* Section for information on compilation and execution)

```
/*
 * chapter 2: HelloPsychophysicist.java
 *
 * displays the text "Hello Psychophysicist" and two images
 * on an otherwise entirely blank screen
 *
 */
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;
import javax.imageio.ImageIO;

public class HelloPsychophysicist {

  public static void main(String[] args) {

    FullScreen1 fs = new FullScreen1();
    fs.setNBuffers(2);

    try {
      fs.displayText("Hello Psychophysicist");
      fs.updateScreen();
      Thread.sleep(2000);
```

```
        fs.blankScreen();
        BufferedImage bi1 = ImageIO.read(new File("psychophysik.png"));
        fs.displayImage(bi1);
        fs.updateScreen();
        fs.hideCursor();
        Thread.sleep(2000);
        fs.blankScreen();
        BufferedImage bi2 = ImageIO.read(new File("fechner.png"));
        fs.displayImage(0,0,bi2);
        fs.updateScreen();
        Thread.sleep(2000);
      } catch (IOException e) {
        System.err.println("File not found");
        e.printStackTrace();
      } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
      }
      finally {
        fs.closeScreen();
      }
    }
  }
```

Initiating Full Screen Exclusive Mode

The first line in the main() method

```
FullScreen1 fs = new FullScreen1();
```

creates an object of the FullScreen1 class. As soon as a FullScreen1 object is created it switches to full screen exclusive mode (FSEM). The visible effect is that the screen goes completely blank and all the OS and Window Manager related decorations disapear, such as a task bar.

Setting the number of video buffers

In the next line

```
fs.setNBuffers(2);
```

sets the total number of video buffers to 2, one is the actual display screen, the other is a back buffer. See Section 2.2 below for more on Double Buffering. If using 2 buffers doesn't work on your hardware/OS, try using 1 or 3. Single buffer (fs.setNBuffers(1)) almost always works, but this results in sacrificing double buffering and introduces tearing artifacts. See chapter XX for more on fine tuning the Buffer Strategies.

Displaying Text, Image, and blank screen

In the next line we start a try-catch-finally clause: it *tries* to perform the statements inside the try{...} part, if anything goes wrong, the execution jumps to the catch{...} part. If nothing goes wrong the catch{...} part is skipped. In either case finally{...} part is always executed. This is a better programming style and robust way

of dealing with errors at run time: If an error happens to ocur during execution, the program doesn't have to terminate abruptly, instead it can try to fix the problem and continue its execution. Even if everything fails, it may still terminate, but terminate in a cleaner way, in a way decided by the programmer a-priori. Moreover, placing short descriptive messages inside catch{...} will surely be helpful.

The lines inside the try clause

```
fs.displayText("Hello Psychophysicist");
fs.updateScreen();
Thread.sleep(2000);
fs.blankScreen();
BufferedImage bi1 = ImageIO.read(new File("psychophysik.png"));
fs.displayImage(bi1);
fs.updateScreen();
fs.hideCursor();
Thread.sleep(2000);
fs.blankScreen();
BufferedImage bi2 = ImageIO.read(new File("fechner.png"));
fs.displayImage(0,0,bi2);
fs.updateScreen();
Thread.sleep(2000);
```

do exactly what the names of the methods imply: display text, display image, blank the screen, hide the cursor, read in images, and sleep for a while. displayText(), displayImage and blankScreen() methods actually manipulate the back video buffer (given that there is a back buffer, recall that we turned on double buffering by setting the number of buffers to 2), but those changes are not actually displayed on the screen until the the updateScreen() method is invoked. Details of updateScreen() method is given below in Section XXX.

displayText(String message) displays the given text at the center of the screen. The overloaded version of this method **displayText(int x, int y, String message)** displays the given text at the location (x,y) relative to the upper left corner of the screen.

Next method **blankScreen()** blanks the entire screen with the default background color.

displayImage() method displays an image centered on the screen. This method accepts images of the type BufferedImage. To obtain a BufferedImage from the file we first need to create a File object

```
new File("psychophysik.png")
```

very roughly speaking, this is similar to opening the file for reading its contents. This File object is then passed to the ImageIO.read() method, which actually reads the file and creates a BufferedImage object

```
BufferedImage bi1 = ImageIO.read(new File("psychophysik.png"));
```

We then display this BufferedImage object

```
fs.displayImage(bi1);
```

(See Chapter XX for other ways of creating images.) ImageIO.read() also throws an Exception, namely IOException, signaling that an I/O problem of some sort has occurred. This time we warn the user about the exception

```
catch (IOException e) {
  System.err.println("File not found");
  e.printStackTrace();
}
```

printStackTrace() method reports further details about the Exception.

Pausing (sleeping) for a while

sleep(long msec) method of the Thread class (one of the classes in the standard Java Core API) pauses the execution of the program for given number of milliseconds, msec. But this method *throws* an Exception. That means, something may go wrong when we invoke this method and Java compiler forces us to be aware of this and take precautions. That's why we need to *catch* the *Exception* that sleep() *throws*

```
catch (InterruptedException e) {
    Thread.currentThread().interrupt();
}
```

In this example we don't have to worry too much because we have a single running thread (see Chapter XX for more on threads). We only invoke the interrupt() method of Thread class which clears the interrupted state of the current thread. In Chapter XX I will provide more details about *Exception* handling in Java.

Terminating FSEM

In finally{...} we cleanly close the FSEM and let the program terminate normally

```
finally {
  fs.closeScreen();
}
```

Summary

Here is a quick summary for displaying visual stimulus using FullScreen (or FullScreen1) class

1. To Initiate the Full Screen Exclusive Mode create a FullScreen object, by invoking the constructor method **FullScreen()**.
2. (Optional) Adjust the number of video buffers by invoking **setNBuffers(int n)** method.
3. Put your stimulus display code inside the try{} portion of a try-catch-finally clause.
4. To display images, text and blank screen, use **displayImage()**, **displayText()**, **blankScreen()** methods, followed by **updateScreen()**, which actually applies and shows the changes on the screen.
5. (Optional) Hide the cursor during the animation using **hideCursor()** method.
6. ***Properly terminate FSEM***. Put the call to **closeScreen()** method inside the finally{} portion of the try-catch-finally clause. This way you can be sure that the FSEM will be terminated properly. Otherwise your system may hang.

In the sections below I will explain in further detail how the FullScreen1 object is created in FSEM and how its methods work.

2.1 Full Screen Exclusive Mode (FSEM)

The first method invocation in the main() method

```
FullScreen1 fs = new FullScreen1();
```

creates an object of the FullScreen1 class and as soon as it is created the windowing environment switches to full screen exclusive mode (FSEM). FSEM feature was first introduced with Java 2 Standard Edition (J2SE) 1.4 (current J2SE version is 1.5.) It allows programmers to suspend the windowing system of the underlying OS, and directly access the video card and draw on the screen. If FSEM is not supported a regular window is positioned at (0,0) and resized to fit the whole screen to imitate full screen exclusive mode (for further details see Full-Screen Exclusive Mode API tutorial at `http://java.sun.com/docs/books/tutorial/extra/fullscreen`.)

Here are the class decleration, global fields and the constructor of the FullScreen1 class

```
public class FullScreen1 extends JFrame {

  private static final GraphicsEnvironment gEnvironment =
      GraphicsEnvironment.getLocalGraphicsEnvironment();
  private static final GraphicsDevice gDevice =
      gEnvironment.getDefaultScreenDevice();
  private static final GraphicsConfiguration gConfiguration =
      gDevice.getDefaultConfiguration();

  private int nBuffers = 1;
  private Color bgColor = Color.BLACK;
  private Color fgColor = Color.LIGHT_GRAY;
  private Font defaultFont = new Font("SansSerif", Font.BOLD, 36);

  public FullScreen1() {

    super(gConfiguration);
    try {
      setUndecorated(true);
      setIgnoreRepaint(true);
      setResizable(false);
      setFont(defaultFont);
      setBackground(bgColor);
      setForeground(fgColor);
      gDevice.setFullScreenWindow(this);
      setNBuffers(nBuffers);
    }
    finally {}
  }
  //...
}
```

First note that FullScreen1 *extends* JFrame

```
public class FullScreen1 extends JFrame
```

this means that FullScreen1 *inherits* all the methods and fields of the JFrame class, which is one of the more than 3,000 classes in the core Java Application programming interface (API). In other words, FullScreen1 *is-a* JFrame *and more*. This is called *inheritance* in object oriented programming.

Tip: Have a look at the definition of fields and methods in JFrame at `http://java.sun.com/j2se/1.5.0/docs/api/javax/swing/JFrame.html`. You can apply all those methods to a FullScreen1 object.

Hint: Did you notice the private keyword in the field declerations? This keyword ensures that no one, other than the methods in FullScreen1 itself can access those fields. This is called *Data Hiding* and is indeed an extremely useful feature in object oriented programming. Why? we will see more later.

The method

```
public FullScreen1() {
  //...
}
```

constructs a FullScreen1 object, therefore it is called a *constructor*. Moreover, this is a *default constructor* because it accepts no arguments. First line

```
super(gConfiguration);
```

is a call to the *super* class JFrame, the class from which FullScreen1 is inherited. The argument is a GraphicsConfiguration object. This allows us to create a FullScreen1 object using the characteristics of our current graphics destinations such as our monitor.

The next methods

```
setUndecorated(true);
setIgnoreRepaint(true);
setResizable(false);
```

are all needed for a proper switch to FSEM: we set FullScreen1 undecorated because we want the screen completely blank, without any window borders; we set FullScreen1 ignore the repaint commands of the underlying OS, because we want to take the whole control of the display - this is called *Active Rendering* (see Section 2.2 below); we set FullScreen1 non-resizable, becuase we want it to always occupy the entire screen, we don't want anyone to resize it.

Next lines

```
setFont(defaultFont);
setBackground(bgColor);
setForeground(fgColor);
```

set the default foreground and background colors, and the default font. Note that text will be rendered with this font and with this foreground color.

The next method

```
gDevice.setFullScreenWindow(this);
```

initiaties the FSEM, now our program has the full control of the graphics device through this FullScreen1 object. Only after this we can adjust the number of buffers we want to use by invoking the method

```
setNBuffers(nBuffers);
```

By default the number of buffers is set to 1.

2.2 Active Rendering and Double Buffering

Whether in FSEM or not, we almost always want to use active rendering as opposed to passive rendering in our experiments - in passive rendering the underlying OS may intervene and send directives to the rendering program, whereas in active rendering the program itself is responsible of drawing and re-drawing the contents on the screen without the OS's intervention, that means more control over the behavior of the stimulus. To do this, we first draw the image on an *offscreen buffer*, often called as *back buffer.* Only after rendering onto back buffer is finished the image is brought to the screen, or *front buffer*. The critical variable is the mechanism of this process. There are several mechanisms to bring the back buffer to front. One way is rendering to a back buffer and then moving just the video pointer. In this way nothing is copied between different locations on the video memory, only a memory pointer is flipped internally: what was back buffer before becomes the front, what was front buffer becomes the back. This is called *page flipping.* The other possibility is actually copying the entire memory to the front buffer, this is called *blit buffering,* or *blitting* in short. This can happen in two ways: accelerated or unaccelerated. See Chapter XX for more details.

Here is how we create the appropriate BufferStrategy in the setNBuffers() method

```
public void setNBuffers(int n) {

  try {
    createBufferStrategy(n);
    nBuffers = n;
  } catch (IllegalArgumentException e) {
    System.err.println(
        "Exception in FullScreen.setNBuffers: "
        +"requested number of Buffers is illegal - falling back to default");
    createBufferStrategy(nBuffers);
  }
  try{
    Thread.sleep(200);
  } catch (InterruptedException e) {
    Thread.currentThread().interrupt();
  }
}
```

this method attempts to create a new strategy with n buffers (including the front buffer) and with the best available strategy: it tries page-flipping first, if not available tries blitting with accelerated buffers, if that is also not available it tries unaccelerated blitting. The BufferStrategy class represents the mechanism of how the memory on a particular Canvas or Window is organized. Note that we needed to include an arbitrary amount of sleep time (200 ms.) because unfortunately createBufferStrategy() method is asynchronous and it is necessary to implement this rather inelegant work around. However, this is done only once during the creation of the FullScreen1 object. See Chapter XX for more details on fine tuning your buffer strategies.

2.3 Displaying Image and Text

displayImage() method draws a BufferedImage on a back buffer (if available), but not directly on to the screen (it draws directly on the screen if the total number of buffers is 1, i.e. there is no back buffer). This method is *overloaded:* It is possible to invoke the displayImage() method in two different ways. If the

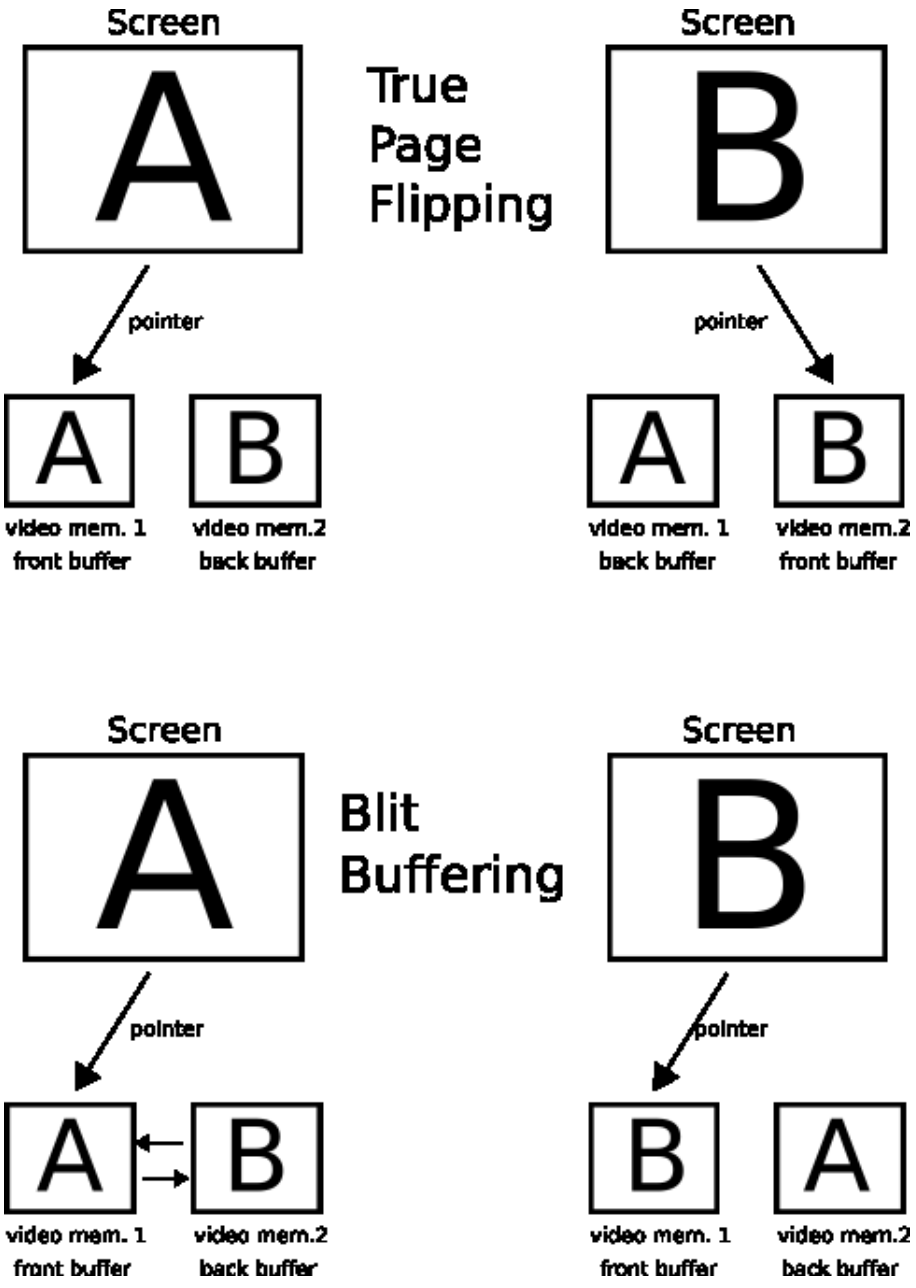

Figure 2.1: True page flipping and blit buffering. In true page flipping only a pointer changes. In Blit Buffering the video memories are sawpped.

argument is only a BufferedImage, the method positions the image at center. The second displayImage(int x, int y, BufferedImage bi) method places upper left corner of the image at the given coordinates, x and y.

The first method simply calculates the coordinates of the upper left corner of the image to position it at the center and then invokes the other version

```
public void displayImage(BufferedImage bi) {

  double x = (getWidth() - bi.getWidth()) / 2;
  double y = (getHeight() - bi.getHeight()) / 2;
  displayImage((int) x, (int) y, bi);
}
```

here is the second overloaded version

```
public void displayImage(int x, int y, BufferedImage bi) {

  Graphics2D g =  (Graphics2D)getBufferStrategy().getDrawGraphics();
  try {
    if(g!=null && bi!=null)
      g.drawImage(bi, x, y, null);
  }
  finally {
    g.dispose();
  }
}
```

First we obtain the Graphics2D object associated with the video buffer by invoking the getDrawGraphics() method.

```
Graphics g = getBufferStrategy().getDrawGraphics();
```

Graphics2D class is the fundamental class for rendering 2-dimensional shapes, text and images. Then we simply draw our image on this Graphics2D, effectively drawing the image on the video buffer. Finally we dispose the Graphics2D object that we obtained in the beginning.

The logic of displayText() methods are similar to displayImage() methods

```
public void displayText(String text) {

  Graphics2D g = (Graphics2D)getBufferStrategy().getDrawGraphics();
  if( g!=null && text!= null){
    Font font = getFont();
    g.setFont(font);
    FontRenderContext context = g.getFontRenderContext();
    Rectangle2D bounds = font.getStringBounds(text, context);
    double x = (getWidth() - bounds.getWidth()) / 2;
    double y = (getHeight() - bounds.getHeight()) / 2;
    double ascent = -bounds.getY();
    double baseY = y + ascent;
    displayText((int) x, (int) baseY, text);
```

```
    }
  }
  public void displayText(int x, int y, String text) {
    Graphics2D g = (Graphics2D)getBufferStrategy().getDrawGraphics();
    try {
      if(g!=null && text!=null){
        g.setFont(getFont());
        g.setColor(getForeground());
        g.drawString(text, x, y);
      }
    }
    finally {
      g.dispose();
    }
  }
```

The active rendering mechanism is exactly the same as in the displayImage() method. The difference arises from the complexity of finding the right coordinates to center the text on the screen. To get the necessary information we use a Graphics2D object rather then a Graphics object.

blankScreen() method clears and blanks the entire screen

```
  public void blankScreen() {

    Graphics2D g = (Graphics2D)getBufferStrategy().getDrawGraphics();
    try {
      if(g!=null){
        g.setColor(getBackground());
        g.fillRect(0, 0, getWidth(), getHeight());
      }
    }
    finally {
      g.dispose();
    }
  }
```

As you see, blankScreen() uses active rendering in a very similar way as the displayImage() and displayText() methods. First we obtain a Graphics object and on this object we draw a rectangle which covers the entire screen. This is achieved by a call to the fillRect() method of the Graphics class. I will also introduce two methods to get and set background color. Normally this shouldn't be necessary because JFrame has the two methods, getBackground() and setBackground(), to do that. But I found that those methods don't work reliably in FSEM so I will overide them

```
    public Color getBackground(){

      return bgColor;
    }

    public void setBackground(Color bg){
```

```
    bgColor = bg;
  }
```

We drawed the text or image on the video buffer, how are we going to actually display it on the screen? To display the video buffer on the screen, the program must invoke the updateScreen() method to apply the change to the actual display screen. This method draws the back buffer on the screen using the BufferStrategy created above

```
public void updateScreen() {
  if (getBufferStrategy().contentsLost())
    setNBuffers(nBuffers);
  getBufferStrategy().show();
}
```

We use contentLost() method of BufferStrategy class to check whether the video memory content is demaged. Sometimes the video memory may get lost, if this is the case we should re-allocate the memory, the best way to do this is to freshly create a new BufferStrategy object

```
if (getBufferStrategy().contentsLost())
  setNBuffers(nBuffers);
```

In the displayImage() or displayText() methods, the image is drawn on the back buffer but not yet on the display screen. The show() method takes the necessary step to make the back buffer visible on the display using the strategy previously created

```
  getBufferStrategy().show();
```

Note: show() method of BufferStrategy waits for a vertical-sync (VSync) signal from the screen, this means that the back buffer will be brought to the front in synchrony with the screen's vertical refresh frequency. This eliminates the much undesired tearing effect in psychophysics experiments. Nevertheless all this is true only in principle, you should still check and fine tune your buffer strategies for the desired performance. Moreover, vertical-sync may not be available on all platforms and with all video cards. I had both satisfactory results and failures with various OSs and video cards. Newer NVidia chip based cards under Windows XP and Mac OS X seem to usually work. Under Linux JSE 1.5 doesn't always get the synchronization right, however JSE 1.6 seems to work pretty well with the opengl pipeline enabled. See Chapter XX for more details on vertical-sync.

2.4 Hiding and showing the cursor

hideCursor() and showCursor() methods hide and show the cursor respectively

```
public void hideCursor() {

  Cursor noCursor = null;
  Toolkit tk = Toolkit.getDefaultToolkit();
  Dimension d = tk.getBestCursorSize(1,1);
  if((d.width|d.height)!=0)
    noCursor = tk.createCustomCursor(
        gConfiguration.createCompatibleImage(d.width, d.height),
```

```
        new Point(0, 0), "noCursor");
  setCursor(noCursor);
}

public void showCursor() {
  setCursor(new Cursor(Cursor.DEFAULT_CURSOR));
}
```

There isn't really any easy way to hide the cursor. We create a compatible BufferedImage (See Chapter XX for more on BufferedImage types) and use this image as the cursor without even initializing it. When the image passed to createCustomCursor() method is not initialized, Java does not display any cursor. **showCursor()** method displays the default cursor.

2.5 Terminating FSEM

In the end we want to cleanly close the FSEM and give the resources back. Here is how we implement the closeScreen() method

```
public void closeScreen() {
  gDevice.setFullScreenWindow(null);
  dispose();
}
```

We close the FSEM by invoking setFullScreenWindow(null), and finally release all the resources used by this FullScreen1 object and hand them back to the OS with a call to the dispose() method. In this current example we didn't change the display mode, but we could have done that (See Chapter XX on Managing Displays). Had we actually changed the display mode, then it would be a good idea to hand display back to the OS in its original mode.

Here is the complete listing of FullScreen1.java

```
/*
 * chapter 2: FullScreen1.java
 *
 * Provides methods to switch to FSEM and
 * display text and image on the screen
 *
 */
import java.awt.*;
import java.awt.font.FontRenderContext;
import java.awt.geom.Rectangle2D;
import java.awt.image.BufferedImage;
import javax.swing.JFrame;
public class FullScreen1 extends JFrame {
  private static final GraphicsEnvironment gEnvironment = GraphicsEnvironment
      .getLocalGraphicsEnvironment();
```

```
private static final GraphicsDevice gDevice = gEnvironment
    .getDefaultScreenDevice();
private static final GraphicsConfiguration gConfiguration = gDevice
    .getDefaultConfiguration();

private int nBuffers = 1;
private Color bgColor = Color.BLACK;
private Color fgColor = Color.LIGHT_GRAY;
private final Font defaultFont = new Font("SansSerif", Font.BOLD, 36);

public FullScreen1() {
  super(gConfiguration);
  try {
    setUndecorated(true);
    setIgnoreRepaint(true);
    setResizable(false);
    setFont(defaultFont);
    setBackground(bgColor);
    setForeground(fgColor);
    gDevice.setFullScreenWindow(this);
    setNBuffers(nBuffers);
  }
  finally {}
}
public void setNBuffers(int n) {
  try {
    createBufferStrategy(n);
    nBuffers = n;
  } catch (IllegalArgumentException e) {
    System.err.println(
        "Exception in FullScreen.setNBuffers(): "
        +"requested number of Buffers is illegal - falling back to default");
    createBufferStrategy(nBuffers);
  }
  try{
    Thread.sleep(200);
  } catch (InterruptedException e) {
    Thread.currentThread().interrupt();
  }
}
public int getNBuffers() {
  return nBuffers;
}
public void updateScreen() {

  if (getBufferStrategy().contentsLost())
    setNBuffers(nBuffers);
  getBufferStrategy().show();
```

```
}

public void displayImage(BufferedImage bi) {
  if( bi!=null){
    double x = (getWidth() - bi.getWidth()) / 2;
    double y = (getHeight() - bi.getHeight()) / 2;
    displayImage((int) x, (int) y, bi);
  }
}
public void displayImage(int x, int y, BufferedImage bi) {
  Graphics2D g =  (Graphics2D)getBufferStrategy().getDrawGraphics();
  try {
    if(g!=null && bi!=null)
      g.drawImage(bi, x, y, null);
  }
  finally {
    g.dispose();
  }
}
public void displayText(String text) {
  Graphics2D g = (Graphics2D)getBufferStrategy().getDrawGraphics();
  if( g!=null && text!= null){
    Font font = getFont();
    g.setFont(font);
    FontRenderContext context = g.getFontRenderContext();
    Rectangle2D bounds = font.getStringBounds(text, context);
    double x = (getWidth() - bounds.getWidth()) / 2;
    double y = (getHeight() - bounds.getHeight()) / 2;
    double ascent = -bounds.getY();
    double baseY = y + ascent;
    displayText((int) x, (int) baseY, text);
  }
  g.dispose();
}
public void displayText(int x, int y, String text) {
  Graphics2D g =  (Graphics2D)getBufferStrategy().getDrawGraphics();
  try {
    if(g!=null && text!=null){
      g.setFont(getFont());
      g.setColor(getForeground());
      g.drawString(text, x, y);
    }
  }
  finally {
    g.dispose();
  }
}
public void blankScreen() {
```

```
    Graphics2D g = (Graphics2D)getBufferStrategy().getDrawGraphics();
    try {
      if(g!=null){
        g.setColor(getBackground());
        g.fillRect(0, 0, getWidth(), getHeight());
      }
    }
    finally {
      g.dispose();
    }
  }

  public Color getBackground(){

    return bgColor;
  }

  public void setBackground(Color bg){

    bgColor = bg;
  }

  public void hideCursor() {
    Cursor noCursor = null;
    Toolkit tk = Toolkit.getDefaultToolkit();
    Dimension d = tk.getBestCursorSize(1,1);
    if((d.width|d.height)!=0)
      noCursor = tk.createCustomCursor(
          gConfiguration.createCompatibleImage(d.width, d.height),
          new Point(0, 0), "noCursor");
    setCursor(noCursor);
  }

  public void showCursor() {
    setCursor(new Cursor(Cursor.DEFAULT_CURSOR));
  }

  public void closeScreen() {
    gDevice.setFullScreenWindow(null);
    dispose();
  }
}
```

3 Introduction to threads

In this chapter

- What is multithreaded programming?
- Constructing concurrently running Threads

Modern computers have the ability to perform multiple tasks seemingly, i.e. running multiple programs or processes, simultaneously. Strictly speaking, a single CPU can perform a single computation at a time. But the high speeds of the processors and the ability of modern operating systems to schedule which computations are going to be performed at each moment make the *multitaksing* possible.

Multithreading, on the other hand, extends the idea of multitasking and allows a single program to perform multiple taks simultaneously (or strictly speaking, concurrently on a single CPU). Each of these tasks is called a *thread*. The programs which can run more than one task concurrently are called *multithreaded*. This technique of programming is often named as *parallel programming*.

Multithreading is extremely useful for a psychophysics experiment. Suppose that you want to present your observers an animation and ask them to adjust the luminance of a test patch in the scene. In order to allow a smooth flow of the animation, you can construct three threads which perform the following three tasks: One for displaying the frames of animation, a second one for getting the observer response, and the third one to re-render the stimulus and change the luminance of the test patch depending on the observer's response.

3.1 Constructing concurrently running threads

Although extremely useful, multithreaded programming can get very complex. In this chapter I only introduce how to construct a new thread and run an *experiment* in that new thread. Later in Chapter XX, we will work on a more complex example of multithreaded programming.

The creation of a thread is actually quite straightforward. You place the code that performs the task in the run() method of a class which *implements* the Runnable interface (see the references listed in Chapter XXX for more on object oriented programming and interfaces)

```
class RunnableTest implements Runnable {

  public void run(){

    // code to perform your task
  }
}
```

next you construct an object of that class, then construct a Thread with that object and finally start the Thread. Here are those three steps

```
Runnable test = new RunnableTest();
Thread experiment = new Thread(test);
experiment.start();
```

I will use the same example as in previous chapter to emphasize how to construct a new thread. Here is the multithreaded version of HelloPsychophysicist example from Chapter 2.

```
/*
 * chapter 3: HPThreaded.java
 *
 * Multithreaded version of HelloPsychophysicist of Chapter 2
 *
 * displays the text "Hello Psychophysicist (Threaded)"
 * and two images on an otherwise entirely blank screen
 *
 */
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;
import javax.imageio.ImageIO;
public class HPThreaded extends FullScreen1 implements Runnable {

  public static void main(String[] args) {

    HPThreaded fs = new HPThreaded();
    fs.setNBuffers(2);
    Thread experiment = new Thread(fs);
    experiment.start();
  }

  public void run() {

    try {

      displayText("Hello Psychophysicist (Threaded)");
      updateScreen();
      Thread.sleep(2000);
      blankScreen();
      hideCursor();
      BufferedImage bi1 = ImageIO.read(new File("psychophysik.png"));
      displayImage(bi1);
      updateScreen();
      Thread.sleep(2000);
      blankScreen();
      BufferedImage bi2 = ImageIO.read(new File("fechner.png"));
      displayImage(0,0,bi2);
      updateScreen();
      Thread.sleep(2000);
```

```
      } catch (IOException e) {
        System.err.println("File not found");
        e.printStackTrace();
      } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
      }
      finally {
        closeScreen();
      }
    }
  }
```

The first difference from the previous version is that HPThreaded inherits from the FullScreen1 class

```
public class HPThreaded extends FullScreen1
```

This means that HPThreaded itself is-a FullScreen1. We can construct an object of HPThreaded as we constructed an object of FullScreen1 in the previous chapter. All the methods of the FullScreen1 class will be available for the object of the class HPThreaded as well. This approach is going to save us quite a bit of bookkeeping and I will use it throughout this guide.

The other difference is that HPThreaded implements the Runnable interface

```
public class HPThreaded extends FullScreen1 implements Runnable
```

This way we can create a HPThreaded object with the convinience of having all the methods of FullScreen1 class accessible, moreover we can also create a new Thread using that object and put the experimental code inside its own run() method (*multiple inheritance*). This results in a clearer and a simpler code.

As explained above, we first create an object of a class which implements the Runnable interface

```
HPThreaded fs = new HPThreaded();
```

Note that the HPThreaded class must implement the run() method (see below). Because HPTHreaded is a Runnable, we can create a Thread object using a HPTHreaded object

```
Thread experiment = new Thread(fs);
```

At last, to initiate the execution of the run() method, we invoke the start() method of the Thread class

```
experiment.start();
```

Note that we don't directly invoke the run() method of the Runnable class, instead invoke the start() method of Thread class.

Next, let's inspect the run() method

```
  public void run() {

    try {

      displayText("Hello Psychophysicist (Threaded)");
      updateScreen();
```

```
      Thread.sleep(2000);
      blankScreen();
      hideCursor();
      BufferedImage bi1 = ImageIO.read(new File("psychophysik.png"));
      displayImage(bi1);
      updateScreen();
      Thread.sleep(2000);
      blankScreen();
      BufferedImage bi2 = ImageIO.read(new File("fechner.png"));
      displayImage(0,0,bi2);
      updateScreen();
      Thread.sleep(2000);
    } catch (IOException e) {
      System.err.println("File not found");
      e.printStackTrace();
    } catch (InterruptedException e) {
      Thread.currentThread().interrupt();
    }
    finally {
      closeScreen();
    }
  }
```

This portion of the code is the same as the corresponding portion of the HelloPsychophysicist class from the previous Chapter. With only one difference: notice that here we directly invoke the methods of FullScreen1 class, for example

```
displayText("Hello Psychophysicist (Threaded)");
updateScreen();
```

instead of

```
fs.displayText("Hello Psychophysicist");
fs.updateScreen();
```

We are able to this, because the run() method is in the HPThreaded class, which inherits from the FullScreen1. Just as the methods in the FullScreen1 class doesn't need an explicit object reference to invoke each other, the calls to FullScreen1 methods from within the run() method of HPThreaded also doesn't need an explicit object reference. (Nevertheless there is a special keyword **this,** which could be used to make an explicit reference to an object, for example **this.updateScreen();**)

3.2 Summary

Here are the steps to take to write a Threaded program

1. Prepare a class which *implements* the Runnable interface (say RunnableTest)
2. Place the code which performs the task in the run() method of RunnableTest class

3. Construct an object of RunnableTest class:

```
RunnableTest rt = new RunnableTest();
```

4. Construct a Thread with that RunnableTest object:

```
Thread experiment = new Thread(rt);
```

   This allocates a new Thread and the argument is the object whose run method is going to be invoked. In this case the argument is conveniently an HPThreaded object.

5. Finally start the Thread:

```
experiment.start();
```

This was a very brief introduction to multithreading, see Chapter XXX for more complex examples. In this Chapter we also established a more convinient coding style. This style saves us some bookkeeping and results in clearer code. In the remaining of the Guide I will follow this convention of style by performing the 5 steps mentioned above.

4 Accurate timing

Accurate timing is often essential in behavioral experiments for several reasons, including:

- The duration and timing of stimlus presentation may have to be accurate,
- The experimenter may need to know the time differences between events as accurate as possible, for example the time between the onset of a stimulus and a key press as a response to that stimulus.

However, there are a few sources of "inaccuracy" to consider

1. The precision (resolution or granularity) of the timer utility used may not be sufficient for the required accuracy,
2. Thread.sleep() method may introduce timing inaccuracy,
3. With double buffering enabled, the exact time of displaying each single stimulus frame depends on the refresh rate of the graphics hardware (because for a tearless, artifact free presentation the experimental program must wait for the vertical synchronization signal from the video card, see Chapter XX, section XXX.) This may introduce timing inaccuracies.
4. Other sources of inaccuracies (hardware or software), such as keyboard response time, particularly with usb connection.

In this chapter I focus on accurate timing in animations. Later, in Chapter XX, I will emphasize timing issues regarding observer responses.

Below, I will first demonstrate how timing inaccuracies can arise. I will then present a simple flickering grating example in which precautions are taken against timing inaccuracies.

4.1 Sources of timing inaccuracies

4.1.1 Precision (resolution) of timer utilities in core Java

First of all one needs a precise timer to measure time differences, without a precise timer there would not be an accurate time measurement. Java Core provides two very useful methods to measure time: System.currentMillis() and System.nanoTime(). As the names imply, the former method returns the current time in milliseconds (10^{-3} seconds), the later one returns, in nanoseconds (10^{-9} seconds), the time elapsed since some fixed but arbitrary time. System.nanoTime() utilizes the most precise timer available in the system.

The following short program tests the presision (resolution) of those two timer utilities.

```
/*
 * chapter 4: TimerPrecision.java
 *
 * Finds out the timer precision (resolution or granularity) emprically
```

```
 *
 */
public class TimerPrecision {

  public static void main(String[] args) {

    long start;
    long stop;
    long total = 0L;
    int nRepeat = 100;

    for (int i = 0; i < nRepeat; i++) {
      start = System.currentTimeMillis();
      stop = System.currentTimeMillis();
      while (stop == start)
        stop = System.currentTimeMillis();
      total += (stop - start);
    }
    System.out.printf("currentTimeMillis(): %1.5f ms.\n", total
        / (double) nRepeat);

    total = 0L;
    for (int i = 0; i < nRepeat; i++) {
      start = System.nanoTime();
      stop = System.nanoTime();
      while (stop == start)
        stop = System.nanoTime();
      total += (stop - start);
    }
    System.out.printf("nanoTime(): %1.5f msec.\n", total
        / (double) nRepeat / 1000000.0);
  }
}
```

Results: resolution of currentTimeMillis() is about 1 millisecond under my Linux (Fedora Core 4) box, XX under my Mac OS X Tiger desktop, XX under my Windows XP laptop. Resolution of nanoTime() is about 0.004 milliseconds under Linux, XX under Mac OS X Tiger, XX under Windows XP.
Conclusion: These results suggest that the resolution of currentTimeMillis() very well satisfy the usual needs of psychophysical testing, whereas the resolution of nanoTime() is probably far beyond those needs.

4.1.2 Thread.sleep() inaccuracies

In previous chapters we have used the Thread.sleep() method quite often. But how accurate is the sleep() method? The following program determines the accuracy of the sleep() method.

```
/*
 * chapter 4: SleepInaccuracy.java
 *
```

```
 * Demonstrates that Thread.sleep() may introduce inaccuracies
 *
 */

public class SleepInaccuracy {

  public static void main(String[] args) {

    long start;
    long stop;
    int nRepeat = 10;

    try{
      for(long duration = 100L; duration >= 1; duration /= 10){
        start = System.nanoTime();
        for(int i = 0; i< nRepeat; i++)
          Thread.sleep(duration);
        stop = System.nanoTime();
        long diff = stop - start;
        System.err.printf( "sleep(%d): slept %.3f msec.\n",duration,
            (double)diff / 1000000 / nRepeat);
      }
    }catch(InterruptedException e){
      Thread.currentThread().interrupt();
    }
  }
}
```

Results: Here are the results:
under Linux:

```
sleep(100): slept 103.988 msec.
sleep(10): slept 15.848 msec.
sleep(1): slept 7.948 msec.
```

under Mac OS X:

```
sleep(100): slept 103.988 msec.
sleep(10): slept 15.848 msec.
sleep(1): slept 7.948 msec.
```

and under Windows XP:

```
sleep(100): slept 103.988 msec.
sleep(10): slept 15.848 msec.
sleep(1): slept 7.948 msec.
```

Conclusion: Those results demonstrate that some degree of care must be taken while using the sleep() method especially if the sleep time is short. The problem is probably due to the complex nature of releasing

and acquiring the current Thread status. If the waiting times needed are too short, it would be advisable to use a different method rather than Thread.sleep.

What are other methods? Give a few examples.

4.1.3 Stimulus display time and display refresh synchronization

When double buffering is enabled, the tools we created in Chapter 2 display the frame upon receiving a vertical refresh signal from the video card. That means that even with no sleep() between them, two consecutive screen updates can occur only as fast as the graphics system's (hardware) refresh rate. Of course if double buffering is disabled, then the updates can take place much faster, however this would cause tearing articats. The following program examines the elapsed time between two consecutive updates.

```
/*
 * chapter 4: SleepInaccuracy.java
 *
 * Demonstrates that the time between two video frames depends on the refresh
 * rate of the graphics device
 *
 */
import java.awt.Graphics;
import java.awt.image.BufferedImage;
import static java.lang.Math.*;
public class VideoFrameSync extends FullScreen1 implements Runnable {
  public static void main(String[] args) {
    VideoFrameSync fs = new VideoFrameSync();
    fs.setNBuffers(2);
    Thread experiment = new Thread(fs);
    experiment.start();
  }
  public void run() {
    BufferedImage bi = (BufferedImage) createImage(80, getHeight() / 2);
    Graphics g = bi.getGraphics();
    g.fillRect(0, 0, bi.getWidth(), bi.getHeight());

    try {

      int y = (getHeight() - bi.getHeight()) / 2;
      long start;
      long dTime;
      long maximum = Long.MIN_VALUE;
      long minimum = Long.MAX_VALUE;
      long total = 0L;
      int nRepeat = 10;
      int counter = 0;

      for (int j = 0; j < nRepeat; j++) {

        for (int i = 0; i <= getWidth() - bi.getWidth(); i += bi.getWidth()) {
```

```
            start = System.nanoTime();
            blankScreen();
            displayImage(i, y, bi);
            updateScreen();
            dTime = System.nanoTime() -start;
            total += dTime;
            maximum = max(maximum, dTime);
            minimum = min(minimum, dTime);
            counter++;
          }
        }
        System.err.printf("Avarage per frame: %.3f msec.\n",
            (double)total / 1000000/ counter);
        System.err.printf("Maximum duration: %.3f msec.\n",
            (double)maximum / 1000000);
        System.err.printf("Minimum duration: %.3f msec.\n",
            (double)minimum / 1000000);
      }
      finally {
        closeScreen();
      }
    }
  }
```

Results: On a 60 Hz monitor (~16 msec. refresh rate):

```
Avarage per frame: 0.165 msec.
Maximum duration: 3.307 msec.
Minimum duration: 0.070 msec.
```

Conclusion: These results reveal two facts: 1) The time resolution of stimulus display is much cruder than that of timer utilies. Nevertheless, the refresh rate of modern computer monitors are usually fast enough for psychophysics experiments. 2) The render update may vary slightly.

4.1.4 Other factors

what other factors? Feedback wellcome. How to determine my keyboard's response time?

4.2 Accurate timing in animations

I will show how to determine the time of observer response in Chapter XX when I introduce getting observer responses. My focus in this section will be on precise timing of presenting visual stimulus on the display. In the example below I will introduce some strategies to mitigate the timing inaccuracies in animations introduced by the factors I showed above.

If you are writing your own code, first of all seperate the rendering and screen updating (they are seperated in the FullScreen class we have been working on.) In the following example, the stimuli consist of counter phase flickering sinusoidal gratings, I will explain the creation of sinusoidal gratings in the next section.

```
/*
 * chapter 4: Flicker.java
 *
 * Demonstrates accurate timing.
 *
 */
import java.awt.geom.AffineTransform;
import java.awt.image.AffineTransformOp;
import java.awt.image.BufferedImage;
import static java.lang.Math.*;
public class Flicker extends FullScreen1 implements Runnable {
  static final int flick = 64;
  static final int repeat = 1;
  static final int block = 1280;
  public static void main(String[] args) {
    Flicker fs = new Flicker();
    fs.setNBuffers(2);
    Thread experiment = new Thread(fs);
    experiment.start();
  }
  public void run() {
    try {

      long start = System.currentTimeMillis();
      BufferedImage[][] bi = new BufferedImage[2][2];
      bi[0][0] = aGrating(127, 0.015, 0.9, 0, 513, -3 * PI / 4);
      bi[0][1] = aGrating(127, 0.015, 0.9, PI, 513, -3 * PI / 4);
      bi[1][0] = aGrating(127, 0.015, 0.9, 0, 513, -PI / 4);
      bi[1][1] = aGrating(127, 0.015, 0.9, PI, 513, -PI / 4);
      Thread.sleep(Math
          .max(0, 500 - (System.currentTimeMillis() - start)));
      long total = 0;
      long overTime = 0;
      int adjust = 0;
      blankScreen();
      updateScreen();

      for (int k = 0; k < repeat; k++) {
        for (int i = 0; i < bi.length; i++) {
          start = System.currentTimeMillis();
          if (abs(overTime) > 2 * flick)
            adjust = (int) (overTime / (2 * flick));
          else
            adjust = 0;
          for (int f = 0; f < (int) (block / flick) / 2 - adjust; f++) {
            for (int j = 0; j < bi[i].length; j++) {
              long startFlick = System.currentTimeMillis();
              displayImage(bi[i][j]);
```

```
            updateScreen();
            Thread.sleep(Math.max(0, flick
                - (System.currentTimeMillis() - startFlick)));
          }
        }
        total += System.currentTimeMillis() - start;
        overTime += (System.currentTimeMillis() - start) - block;
      }
    }
  } catch (PixelOutOfRangeException e) {
    e.printStackTrace();
  } catch (InterruptedException e) {
    Thread.currentThread().interrupt();
    e.printStackTrace();
  }
  finally {
    closeScreen();
  }
}
/**
 * Prepares an achromatic grating
 */
public static BufferedImage aGrating(int mean, double cpp, double amp,
    double phase, int size, double orientation)
    throws PixelOutOfRangeException {
  int[] gPixel = new int[size * size];
  int sizeSquare = size * size / 4;
  for (int j = 0; j < size; j++) {
    int x = (j - size / 2);
    int val = (int) (amp * mean * cos(phase + x * cpp * 2 * PI));
    x = x * x;
    int iLow = (int) (size / 2 - sqrt(sizeSquare - x));
    int iHigh = (int) (size / 2 + sqrt(sizeSquare - x));
    for (int i = iLow; i <= iHigh; i++) {
      int k = i * size + j;
      if ((gPixel[k] = mean + val) > 255 || gPixel[k] <0)
        throw new PixelOutOfRangeException(
            "Exception in aGrating: calculated pixel value out of range [0,255]");
    }
  }
  BufferedImage bi = new BufferedImage(size, size,
      BufferedImage.TYPE_BYTE_GRAY);
  bi.getRaster().setPixels(0, 0, size, size, gPixel);
  if(orientation !=0 ){
    AffineTransformOp ato = new AffineTransformOp(AffineTransform
        .getRotateInstance(orientation, (double) size / 2, (double) size / 2),
        AffineTransformOp.TYPE_NEAREST_NEIGHBOR);
    BufferedImage bic = new BufferedImage(size, size,
```

```
            BufferedImage.TYPE_BYTE_GRAY);
        ato.filter(bi, bic);
        return bic;
      }
      else
        return bi;
    }
  }
  class PixelOutOfRangeException extends Exception {
    PixelOutOfRangeException(String s) {
      super(s);
    }
  }
```

Explain the code related to animation here. Explain the grating below. Also: dropping frames...

4.2.1 Achromatic grating

Creating the achromatic grating...

4.2.2 Exception Handling in Java

Exception is an object.

4.3 Other timing methods

4.4 Summary

here is what to do...

6 Getting observer response

In previous chapters we only displayed images and text on the screen. But in a psychophysics experiment you often want to get the observer's response to the stimuli presented, for instance through a key press. In this chapter I will show how to collect observer response using tools of Java. I will first explain the event handling mechanism in Java and demonstrate a sample implementation (an asymmetric matching experiment). This example emphasizes writing your own specialized event handling mechanism for the particular experiment you are implementing. Later I will show how to implement a thread safe mechanism to collect observer response into the FullScreen class. I will introduce generalized methods to conveniently collect observer response without writing a specialized event handler.

6.1 Event handling mechanism in Java

Events - such as key presses - are handeled by your Java program in the following manner: You first construct an *event source.* Event source can be a button, a menu or a window, including a FullScreen window. When an interesting activity occurs, for instance when a key is pressed or the mouse moves, the operating system informs your event source. On the other hand, the event source also keeps in touch with a set of *event listener objects* whose appropriate methods are invoked when an event occurs. Once an intresting event occurs, the event source creates an *event object,* which stores the information about the event. The event source then passes the event object to the appropriate method of the event listener object. This type of mechanism is often referred as *callback* mechanism.

Let's have a closer look at the components of this mechanism: First we need to construct an event source. Of course a FullScreen object, or any class inheriting from it, *is* an event source. So by creating an instance of FullScreen class you are automatically creating your event source. Next an event listener must be *added* to, i.e. associated with, the event source by invoking an appropriate method. For example suppose we want to record the observer's responses through the keyboard, then we must assign an *event listener* as follows

```
public class MyKeySource extends FullScreen1 {
  //...
  MyKeySource(){
    super();
    MyKeyListener mkl = new MyKeyListener();
    mks.addKeyListener(mkl);
  }
  //...
}
```

here mkl is the associated object that will listen to and trap the key events. The KeyListener object constitutes the other component of the event handling mechanism. MyKeyListener class should implement the KeyListener interface and *must* have the following three methods

```
class MyKeyListener implements KeyListener {
  public void keyPressed(KeyEvent ke) {
```

```
    // ..
  }
  public void keyReleased(KeyEvent ke) {
    // ..
  }
  public void keyTyped(KeyEvent ke) {
    // ..
  }
}
```

Whenever the observer hits a key, MyKeySource creates a KeyEvent object and invokes the proper method of MyKeyListener - one of the keyPressed(), keyReleased() or keyTyped() methods. You put the code determining the behavior of your program inside those methods.

This approach has one drawback: MyKeyListener is a class by itself and instances of it won't have access to fields of MyKeySource. One often needs the methods of the listener be able to access and modify properties of the source, for instance, typing in a letter could change the background color of the screen. There are several ways to mitigate this problem. One common approach is using an *annonymous inner class* (See Horstmann & Cornell, Chapter XXX)

```
public class MyKeySource extends FullScreen2 {
  public MyKeySource() {
    super();
    addKeyListener(new KeyListener(){
      public void keyPressed(KeyEvent ke) {
        // ..
      }
      public void keyReleased(KeyEvent ke) {
        // ..
      }
      public void keyTyped(KeyEvent ke) {
        // ..
      }
    });
  }
  //...
}
```

Now the methods of the KeyListener have access to the fields and methods of the outer class. This approach is practical, though it has its own drawbacks.

The approach I am going to adapt is different than the obove two. Recall that listener object that MyKeySource is in touch *must* implement the KeyListener interface, this is the only requirement. So we can make MyKeySource class implement KeyListener interface with its 3 necessary methods, then add *itself* as a listener, in other words, MyKeySource becomes both the source and listener of KeyEvents. Here is how that modified MyKeySourceAndListener class looks like

```
public class MyKeySourceAndListener extends FullScreen
    implements KeyListener {

  public MyKeySourceAndListener() {
```

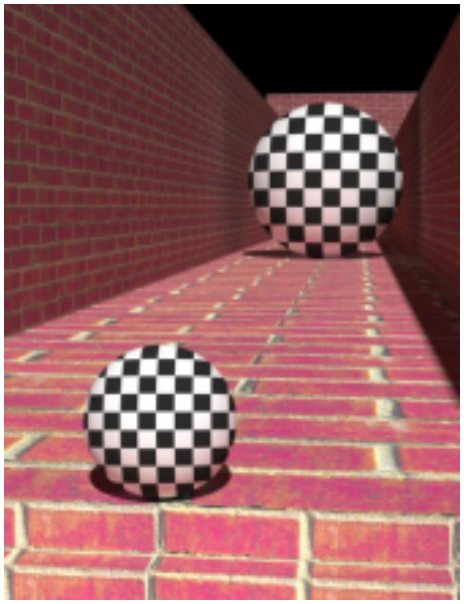

Figure 6.1: The further sphere seems to occupy a portion of the visual field which is larger than the closer one although the visual angle they subtend are exactly the same.

```
      super();
      // ...
      addKeyListener(this);
    }
    // ...
    public void keyPressed(KeyEvent ke) {
      // ..
    }
    public void keyReleased(KeyEvent ke) {
      // ..
    }
    public void keyTyped(KeyEvent ke) {
      // ..
    }
    //...
  }
```

In the next section I will show how to implement this approach.

6.2 Writing your own specialized event handler

An object seems to subtend a larger visual angle than an other one subtending identical visual if its perceived distance to the observer is larger than the later one. Using two dimensional depth cues (such as percpective) one can create interesting visual illusions. One such illusion is illustrated in Figure 6.1. The sample example of this section will test the magnitude of illusion behaviorly using an asymmatric matching procedure (See Murray, Boyaci, and Kersten, Nat. Neuro. 2006).

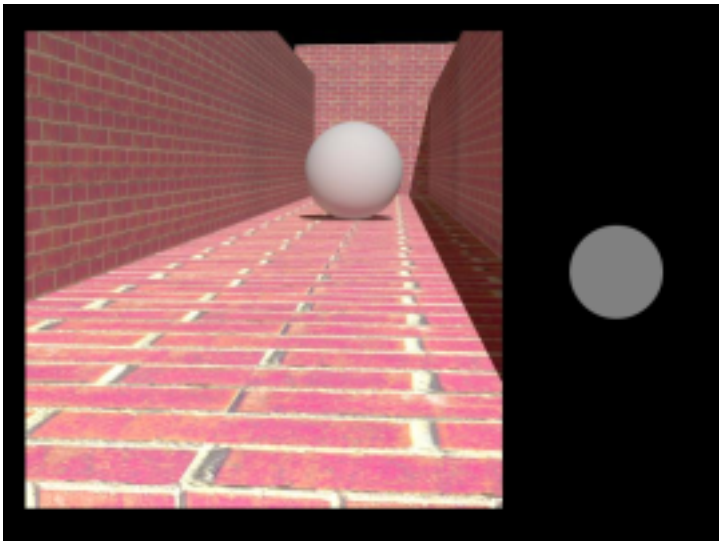

Figure 6.2: Screen shot from the Sizes experiment.

In this sample experiment observers task is adjusting the size of a two dimensional disk until it matches the image size of a three dimensional ball in context (see Figure 6.2). Observer will adjust the size of the disk using the arrow keys and finalize each trial by pressing the space bar.

As I indicated before, you have a class which inherits from FullScreen and implements KeyListener interface, and then adds itself as a KeyListener

```
public class SizesAsM extends FullScreen1 implements Runnable, KeyListener {
  SizesAsM(){

    super();
    addKeyListener(this);
  }
```

As we have always been doing, SizesAsM extends FullScreen1 and implements Runnable. The main() method of the class is not very different from previous examples. Inside the run() method the program first reads in the necessary image files and determines the coordinates to place them. The program also creates a MAX_DISK_SIZE by MAX_DISK_SIZE BufferedImage to draw the matching disk.

The program then displays the instruction text on the screen and tells the observer to press the 's' key

```
displayText(100, 800, "Now hit the \"s\" key to start");
```

and waits until the observer actually presses the 's' key

```
while(!start){}
```

where start is a global boolean variable which is set to false initially. The keyTyped() method (remember that keyTyped() is one of the methods of the KeyListener interface which is implemented by SizesAsM) is responsable to change the value of start to true

```
public void keyTyped(KeyEvent ke) {

  char keyTypedChar = ke.getKeyChar();
  if (keyTypedChar == 's')
      start = true;
}
```

The method first gets the typed key as a character and if it is 's' it sets the value of the global variable start to true, which allows the experiment start.

In an asymmetric matching experiment you usually want to randomize the order of the test stimulus presented. Here is how you can do it. You first create an ArrayList with an initial capacity of total number of test stimuli

```
List<Integer> trials = new ArrayList<Integer>(N_BACK+N_FRONT);
```

Next you populate this ArrayList with integers

```
for (int i = 0; i < N_BACK + N_FRONT; i++)
  trials.add(i);
```

each representing a trial that corresponds to a different stimulus. The static method shuffle() of Collections class randomizes the order of elements of a List

```
Collections.shuffle(trials);
```

Once the trials are randomized you can get the elements from the ArrayList efficiently. First convert the ArrayList into an Iteretor

```
Iterator<Integer> it = trials.listIterator();
```

Next get the next element from the iteretor until there are no more elements left, and set the next stimulus image, s, to present to the observer

```
while (it.hasNext()) {
  int nStim = it.next();
  s = stim[nStim];
  updateScene();

  // ....
}
```

s is a global BufferedImage variable and the method updateScene() takes care of putting the images on the screen

```
public void updateScene() {
  blankScreen();
  displayImage(bgX, bgY, stimBG);
  displayImage(x, y, s);
  displayImage(matchX, matchY, match);
  updateScreen();
}
```

After the stimulus and the matching disk is displayed on the screen, the program waits for observer response until the trial ends with observer's hitting the space bar

```
while(!trialDone){}
```

trilaDone is set to false at the beginning of each trial. While the trialDone is false the current thread waits idle, but the thread of the KeyListener goes on working in the backgorund. The keyPressed() method takes care of interpreting the observer's key strikes and updating the scene accordingly

```
public void keyPressed(KeyEvent ke) {
  int keyPressedCode = ke.getKeyCode();
```

this stores the code of the pressed key in an integer. First the program checks if the observer pressed the escape key

```
  if (keyPressedCode == KeyEvent.VK_ESCAPE) {
    closeScreen();
    System.exit(0);
  }
```

If the experiment has already started, the program checks whether any of the up/down/left/right arrow or space bar is pressed and either sets the size of the disk or sets the value of trialDone to true

```
  else if (start) {
    switch (keyPressedCode) {
      case LARGER:
        matchSize = min(MAX_DISK_SIZE, matchSize + 5);
      break;
      case SMALLER:
        matchSize = max(0, matchSize - 5);
      break;
      case fLARGER:
        matchSize = min(MAX_DISK_SIZE, matchSize + 20);
      break;
      case fSMALLER:
        matchSize = max(0, matchSize - 20);
      break;
      case COMPLETE:
        trialDone = true;
      break;
    }
```

LARGER, SMALLER etc. are global integer variables corresponding to the codes of arrow keys and the space bar, and they are set during the initialization of the SizesAsM class. At the end, if the pressed key was one of the arrow keys the program generates a new matching disk and updates the scene

```
    if (!trialDone) {
      generateMatch();
      updateScene();
    }
```

generateMatch() simply draws a gray disk cnetered on the BufferedImage match, then updateScene() method displays that new disk on the right hand side of the test image.

Here is the entire code of SizesAsM

```
/*
 * chapter 6: SizesAsM.java
 *
 * Shows how to write a specialized event handler for a
 * specific task.
 *
 */
import static java.lang.Math.*;
import java.awt.*;
import java.awt.event.*;
import java.awt.image.BufferedImage;
import java.io.*;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Iterator;
import java.util.List;
import javax.imageio.ImageIO;
public class SizesAsM extends FullScreen1 implements Runnable, KeyListener {
  final static int MAX_DISK_SIZE = 240;
  final static int N_BACK = 5;
  final static int N_FRONT = 5;
  final static int N_TRIALS = 10;
  final static RenderingHints HINTS = new RenderingHints(
      RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);
  BufferedImage match;
  Graphics2D gMatch;
  BufferedImage stimBG;
  int matchSize;
  int matchX;
  int matchY;
  int bgX;
  int bgY;
  int x;
  int y;
  BufferedImage s;
  boolean start = false;
  boolean trialDone = false;
  final static int LARGER = KeyEvent.VK_UP;
  final static int SMALLER = KeyEvent.VK_DOWN;
  final static int fLARGER = KeyEvent.VK_RIGHT;
  final static int fSMALLER = KeyEvent.VK_LEFT;
  final static int COMPLETE = KeyEvent.VK_SPACE;
  SizesAsM(){
```

```
  super();
  addKeyListener(this);
}

public void keyTyped(KeyEvent ke) {
  char keyTypedChar = ke.getKeyChar();
  if (keyTypedChar == 's')
    start = true;
}

public void keyPressed(KeyEvent ke) {
  int keyPressedCode = ke.getKeyCode();
  if (keyPressedCode == KeyEvent.VK_ESCAPE) {
    closeScreen();
    System.exit(0);
  }
  else if (start) {
    switch (keyPressedCode) {
      case LARGER:
        matchSize = min(MAX_DISK_SIZE, matchSize + 1);
      break;
      case SMALLER:
        matchSize = max(0, matchSize - 1);
      break;
      case fLARGER:
        matchSize = min(MAX_DISK_SIZE, matchSize + 20);
      break;
      case fSMALLER:
        matchSize = max(0, matchSize - 20);
      break;
      case COMPLETE:
        trialDone = true;
      break;
    }
    if (!trialDone) {
      generateMatch();
      updateScene();
    }
  }
}

public void keyReleased(KeyEvent ke) {

}

public static void main(String[] args) {

  SizesAsM sAM = new SizesAsM();
```

```
    sAM.setNBuffers(2);
    Thread experiment = new Thread(sAM);
    experiment.start();
  }
public void run() {
    try {
      stimBG = ImageIO.read(new File("bg.jpg"));
      bgX = (getWidth() / 2 - stimBG.getWidth()) / 2;
      bgY = (getHeight() - stimBG.getHeight()) / 2;
      BufferedImage[] stim = new BufferedImage[N_FRONT + N_BACK];
      for (int i = 0; i < N_FRONT; i++)
        stim[i] = ImageIO.read(new File("front_" + Integer.toString(i + 1)
            + ".jpg"));
      for (int i = 0; i < N_BACK; i++)
        stim[i + N_FRONT] = ImageIO.read(new File("back_"
            + Integer.toString(i + 1) + ".jpg"));
      int stimX_Back = bgX;
      int stimY_Back = bgY;
      int stimX_Front = bgX;
      int stimY_Front = bgY + (stimBG.getHeight() - stim[0].getHeight());
      match = (BufferedImage) createImage(MAX_DISK_SIZE, MAX_DISK_SIZE);
      matchX = getWidth() / 2 + (getWidth() / 2 - match.getWidth()) / 2;
      matchY = (getHeight() - match.getHeight()) / 2;
      List<Integer> trials = new ArrayList<Integer>(N_BACK + N_FRONT);
      for (int i = 0; i < N_BACK + N_FRONT; i++)
        trials.add(i);
      displayText(100, 200, "Task:");
      displayText(100, 350,
          "  Use the arrow keys to adjust the size of the disk on the right");
      displayText(100, 400,
          "  to match the image size of the ball in the scene");
      displayText(100, 450,
          "  (Up/Down for fine adjustment; Left/Right for fast adjustment)");
      displayText(100, 500, "  Press space bar to finalize trial");
      displayText(100, 600, "  Press ESC to quit");
      displayText(100, 800, "Now hit the \"s\" key to start");
      updateScreen();
      while (!start) {}
      for (int t = 0; t < N_TRIALS; t++) {
        Collections.shuffle(trials);
        Iterator<Integer> it = trials.listIterator();
        while (it.hasNext()) {
          trialDone = false;
          matchSize = (int) (random() * 150);
          generateMatch();
          int nStim = it.next();
          // for feedback: largest ball has 179 pixels, others are calculated
          // from their index
```

```
        String message = new String("Test Size = "
            + Integer.toString((nStim % 5 + 1) * 20 * 179 / 100) + ";");
        s = stim[nStim];
        if (nStim < N_FRONT) {
          x = stimX_Front;
          y = stimY_Front;
        }
        else {
          x = stimX_Back;
          y = stimY_Back;
        }
        updateScene();
        while (!trialDone) {}
        message = message
            + new String(" your response = " + matchSize + " (pixels)");
        blankScreen();
        // feedback message
        displayText(message);
        updateScreen();
        Thread.sleep(3000);
      }
    }
  } catch (IOException e) {
    System.err.println("File not found");
    e.printStackTrace();
  } catch (InterruptedException e) {
    Thread.currentThread().interrupt();
    e.printStackTrace();
  }
  finally {
    closeScreen();
  }
}

public void updateScene() {

  blankScreen();
  displayImage(bgX, bgY, stimBG);
  displayImage(x, y, s);
  displayImage(matchX, matchY, match);
  updateScreen();
}

public void generateMatch() {

  gMatch = match.createGraphics();
  gMatch.setRenderingHints(HINTS);
  gMatch.setColor(Color.BLACK);
```

```
      gMatch.fillRect(0, 0, MAX_DISK_SIZE, MAX_DISK_SIZE);
      int x = (int) floor((MAX_DISK_SIZE - matchSize) / 2);
      int y = x;
      gMatch.setColor(Color.GRAY);
      gMatch.fillOval(x, y, matchSize, matchSize);
      gMatch.dispose();
    }
  }
```

6.3 A built-in thread safe event handler for the FullScreen class

In the previous section I showed how to write a specific event handler for an experiment. Even though that approach is usually the right one, it is not always very pratical. In certain situations it becomes harder and harder to implement the experiment in that way. Suppose, for example, that in different stages of the experiment hitting a certain key should result in different behaviour. Implementing this with the above approach would be difficult, at least would result in ugly looking and confusing code. Below I will show how to implement a built-in event handler in the FullScreen class, which makes the event handling in your experiment much easier and much more flexible.

Sooner or later one needs to write a threaded program (see Chapter XXX for threaded programming). In a threaded program it is essential to implement a thread safe event handling mechanism. For example, you wouldn't want different two different threads, one writing the other trying to read, accessing the key presses simultaneously. In a sequential program you don't need to worry about the access to the data fields, one method modifies a field, another reads it. No one steps on someone else's toe. But when a program runs on multiple threads, a method may try to modify a field while another method is reading it.

Below I will develop a thread safe event handling mechanism built into the FullScreen class, which allows you to easily and effortlessly collect observer responses. Here is how it works in principle: everytime an event occurs the appropriate methods of FullScreen (for example keyPressed() method) adds a new element to a thread safe BlockingQueue object. I will then implement methods which allow your program to access this BlockingQueue object and read the "head" (the first event element put to the BlockingQueue object) or reset ("flush") the entire event list.

I implement two BlockingQueue lists, one for the code of the event, another for the time when the event occured (this approach, rather than storing the event itself in a single list improves flexibility)

```
public class FullScreen2 extends JFrame impliments KeyListener {
  private BlockingQueue<Integer> keyPressed;
  private BlockingQueue<Long> whenKeyPressed;
  // ...

  FullScreen2(){
    // ...
    keyPressed = new LinkedBlockingQueue<Integer>();
    whenKeyPressed = new LinkedBlockingQueue<Long>();
  }

  //...
}
```

We can now focus on handling the KeyEvents. Let's first decide on what to do in case the observer presses a key. We should, of course, determine which key is pressed, then the time of the key press

```
  public void keyPressed(KeyEvent ke) {

    if (ke.getKeyCode() == KeyEvent.VK_ESCAPE) {
      closeScreen();
      System.exit(0);
    }
    else {
      keyPressed.offer(ke.getKeyCode());
      whenKeyPressed.offer(ke.getWhen());
    }
  }
```

getKeyCode() method returns the *virtual key code* of the key pressed. For instance, if the up arrow key is pressed the returned value is VK_UP. getWhen() method returns the actual time when the event occured (in milliseconds). If the escape key is pressed the program terminates. The offer() method inserts the specified element into the LinkedBlockingQueue, returning true upon success, false if the capacity of the is exceeded. Similarly

```
public void keyReleased(KeyEvent ke) {
  try {
    keyReleased.offer(ke.getKeyCode());
    whenKeyReleased.offer(ke.getWhen());
  } catch (InterruptedException e) {
    Thread.currentThread().interrupt();
    e.printStackTrace();
  }
}
```

returns the information about key releases. In case, for example, the observer presses "A" - capital a - FullScreen2 captures it in the keyTyped() method

```
public void keyTyped(KeyEvent ke) {
  try {
    keyTyped.offer(String.valueOf(ke.getKeyChar()));
    whenKeyTyped.offer(ke.getWhen());
  } catch (InterruptedException e) {
    Thread.currentThread().interrupt();
    e.printStackTrace();
  }
}
```

here getKeyChar() method returns a char type data. Note that I convert the char into String because a BlockingQueue can hold only objects and char is not an object (it is a primitive and doesn't have an object wrapper as, for example, int does with Integer.)

Above three methods write into the BlockingQueue, next let's see the querry methods that your program should use to access the stored elements in those BlockingQueue

```
public Integer getKeyPressed(long ms){

  Integer c = null;
  try {
    if(ms < 0)
      c = keyPressed.take();
    else
      c = keyPressed.poll(ms, TimeUnit.MILLISECONDS);
  } catch (InterruptedException e) {
    Thread.currentThread().interrupt();
    e.printStackTrace();
  }
  return c;
}
```

poll() method retrieves the head of the BlockingQueue and also removes it from the queue. If invoked with a time variable, it waits for the specified amount of time until an element becomes available, if no element becomes available before the end of that time period it returns null. take() is similar to poll(), except it waits indefinetaly until an element becomes avialable. So if you invoke the getKeyPressed() with a negative argument, it will wait until your observer presses a key, on the other hand if you invoke it with a positive value (or zero), it will wait only for the amount of time (in milliseconds) that you specified for the observer response, if no element is available before the time limit it returns null. For convinience I overload the getKeyPressed() method

```
public Integer getKeyPressed(){

  return keyPressed.poll();
}
```

this overloaded version returns the head of the queue immediately, if the queue is empty it returns null.

```
public Long getWhenKeyPressed(){

  return whenKeyPressed.poll();
}
```

returns the time when the key press event occured and removes it from the queue. flushKeyPressed() method clears both keyPressed and whenKeyPressed queues

```
public void flushKeyPressed(){

  keyPressed.clear();
  whenKeyPressed.clear();
}
```

The entire listing of FullScreen2 class is given below

```
/*
 * chapter 6: FullScreen2.java
 *
 * Provides methods to get key presses
 *
 */
import java.awt.*;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;
import java.awt.font.FontRenderContext;
import java.awt.geom.Rectangle2D;
import java.awt.image.BufferedImage;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;
import java.util.concurrent.TimeUnit;
import javax.swing.JFrame;
public class FullScreen2 extends JFrame implements KeyListener{
  private static final GraphicsEnvironment gEnvironment = GraphicsEnvironment
      .getLocalGraphicsEnvironment();
  private static final GraphicsDevice gDevice = gEnvironment
      .getDefaultScreenDevice();
  private static final GraphicsConfiguration gConfiguration = gDevice
      .getDefaultConfiguration();

  private int nBuffers = 1;
  private Color bgColor = Color.BLACK;
  private Color fgColor = Color.LIGHT_GRAY;
  private final Font defaultFont = new Font("SansSerif", Font.BOLD, 36);

  private BlockingQueue<String> keyTyped;
  private BlockingQueue<Long> whenKeyTyped;
  private BlockingQueue<Integer> keyPressed;
  private BlockingQueue<Long> whenKeyPressed;
  private BlockingQueue<Integer> keyReleased;
  private BlockingQueue<Long> whenKeyReleased;

  public void keyTyped(KeyEvent ke) {
    keyTyped.offer(String.valueOf(ke.getKeyChar()));
    whenKeyTyped.offer(ke.getWhen());
  }
  public void keyReleased(KeyEvent ke) {

    keyReleased.offer(ke.getKeyCode());
    whenKeyReleased.offer(ke.getWhen());
  }

  public void keyPressed(KeyEvent ke) {
```

```
  if (ke.getKeyCode() == KeyEvent.VK_ESCAPE) {
    closeScreen();
    System.exit(0);
  }
  else {
    keyPressed.offer(ke.getKeyCode());
    whenKeyPressed.offer(ke.getWhen());
  }
}

public FullScreen2() {
  super(gConfiguration);
  try {
    setUndecorated(true);
    setIgnoreRepaint(true);
    setResizable(false);
    setFont(defaultFont);
    setBackground(bgColor);
    setForeground(fgColor);
    gDevice.setFullScreenWindow(this);

    keyTyped = new LinkedBlockingQueue<String>();
    whenKeyTyped = new LinkedBlockingQueue<Long>();
    keyPressed = new LinkedBlockingQueue<Integer>();
    whenKeyPressed = new LinkedBlockingQueue<Long>();
    keyReleased = new LinkedBlockingQueue<Integer>();
    whenKeyReleased = new LinkedBlockingQueue<Long>();
    addKeyListener(this);

    setNBuffers(nBuffers);
  }
  finally {}
}
public void setNBuffers(int n) {
  try {
    createBufferStrategy(n);
    nBuffers = n;
  } catch (IllegalArgumentException e) {
    System.err.println(
        "Exception in FullScreen.setNBuffers(): "
        +"requested number of Buffers is illegal - falling back to default");
    createBufferStrategy(nBuffers);
  }
  try{
    Thread.sleep(200);
  } catch (InterruptedException e) {
    Thread.currentThread().interrupt();
  }
```

```
}
public int getNBuffers() {
  return nBuffers;
}
public void updateScreen() {

  if (getBufferStrategy().contentsLost())
    setNBuffers(nBuffers);
  getBufferStrategy().show();
}

public void displayImage(BufferedImage bi) {
  if( bi!=null){
    double x = (getWidth() - bi.getWidth()) / 2;
    double y = (getHeight() - bi.getHeight()) / 2;
    displayImage((int) x, (int) y, bi);
  }
}
public void displayImage(int x, int y, BufferedImage bi) {
  Graphics2D g =  (Graphics2D)getBufferStrategy().getDrawGraphics();
  try {
    if(g!=null && bi!=null)
      g.drawImage(bi, x, y, null);
  }
  finally {
    g.dispose();
  }
}
public void displayText(String text) {
  Graphics2D g = (Graphics2D)getBufferStrategy().getDrawGraphics();
  if( g!=null && text!= null){
    Font font = getFont();
    g.setFont(font);
    FontRenderContext context = g.getFontRenderContext();
    Rectangle2D bounds = font.getStringBounds(text, context);
    double x = (getWidth() - bounds.getWidth()) / 2;
    double y = (getHeight() - bounds.getHeight()) / 2;
    double ascent = -bounds.getY();
    double baseY = y + ascent;
    displayText((int) x, (int) baseY, text);
  }
  g.dispose();
}
public void displayText(int x, int y, String text) {
  Graphics2D g =  (Graphics2D)getBufferStrategy().getDrawGraphics();
  try {
    if(g!=null && text!=null){
      g.setFont(getFont());
```

```
      g.setColor(getForeground());
      g.drawString(text, x, y);
    }
  }
  finally {
    g.dispose();
  }
}
public void blankScreen() {
  Graphics2D g = (Graphics2D)getBufferStrategy().getDrawGraphics();
  try {
    if(g!=null){
      g.setColor(getBackground());
      g.fillRect(0, 0, getWidth(), getHeight());
    }
  }
  finally {
    g.dispose();
  }
}

public Color getBackground(){

  return bgColor;
}

public void setBackground(Color bg){

  bgColor = bg;
}

public void hideCursor() {
  Cursor noCursor = null;
  Toolkit tk = Toolkit.getDefaultToolkit();
  Dimension d = tk.getBestCursorSize(1,1);
  if((d.width|d.height)!=0)
    noCursor = tk.createCustomCursor(
        gConfiguration.createCompatibleImage(d.width, d.height),
        new Point(0, 0), "noCursor");
  setCursor(noCursor);
}

public void showCursor() {
  setCursor(new Cursor(Cursor.DEFAULT_CURSOR));
}

public void closeScreen() {
  gDevice.setFullScreenWindow(null);
```

```
  dispose();
}

public String getKeyTyped(long ms){

  String c = null;
  try {
    if(ms < 0)
      c = keyTyped.take();
    else
      c = keyTyped.poll(ms, TimeUnit.MILLISECONDS);
  } catch (InterruptedException e) {
    Thread.currentThread().interrupt();
    e.printStackTrace();
  }
  return c;
}

public String getKeyTyped(){

  return keyTyped.poll();
}

public Long getWhenKeyTyped(){

  return whenKeyTyped.poll();
}

public void flushKeyTyped(){

  keyTyped.clear();
  whenKeyTyped.clear();
}

public Integer getKeyPressed(long ms){

  Integer c = null;
  try {
    if(ms < 0)
      c = keyPressed.take();
    else
      c = keyPressed.poll(ms, TimeUnit.MILLISECONDS);
  } catch (InterruptedException e) {
    Thread.currentThread().interrupt();
    e.printStackTrace();
  }
  return c;
}
```

```
  public Integer getKeyPressed(){

    return keyPressed.poll();
  }

  public Long getWhenKeyPressed(){

    return whenKeyPressed.poll();
  }

  public void flushKeyPressed(){

    keyPressed.clear();
    whenKeyPressed.clear();
  }

  public Integer getKeyReleased(long ms){

    Integer c = null;
    try {
      if(ms < 0 )
        c = keyReleased.take();
      else
        c = keyReleased.poll(ms, TimeUnit.MILLISECONDS);
    } catch (InterruptedException e) {
      Thread.currentThread().interrupt();
      e.printStackTrace();
    }
    return c;
  }

  public Integer getKeyReleased(){

    return keyReleased.poll();
  }

  public Long getWhenKeyReleased(){

    return whenKeyReleased.poll();
  }

  public void flushKeyReleased(){

    keyReleased.clear();
    whenKeyReleased.clear();
  }
}
```

6.3.1 Examples using built-in methods

Here is the new version of the “Sizes” experiment using the built-in methods of the new FullScreen2 class

```
/*
 * chapter 6: SizesAsM2.java
 *
 * Shows how to use built-in methods of FullScreen for collecting
 * observer response
 *
 */

import static java.lang.Math.*;
import java.awt.*;
import java.awt.event.*;
import java.awt.image.BufferedImage;
import java.io.*;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Iterator;
import java.util.List;
import javax.imageio.ImageIO;

public class SizesAsM2 extends FullScreen2 implements Runnable {

  final static int MAX_DISK_SIZE = 240;
  final static int N_BACK = 5;
  final static int N_FRONT = 5;
  final static int N_TRIALS = 10;

  final static RenderingHints HINTS = new RenderingHints(
      RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);
  BufferedImage match;
  Graphics2D gMatch;
  BufferedImage stimBG;

  int matchSize;
  int matchX;
  int matchY;
  int bgX;
  int bgY;

  int x;
  int y;
  BufferedImage s;

  final static int LARGER = KeyEvent.VK_UP;
  final static int SMALLER = KeyEvent.VK_DOWN;
  final static int fLARGER = KeyEvent.VK_RIGHT;
```

```
final static int fSMALLER = KeyEvent.VK_LEFT;
final static int COMPLETE = KeyEvent.VK_SPACE;

public static void main(String[] args) {

  SizesAsM2 sAM = new SizesAsM2();
  sAM.setNBuffers(2);
  Thread experiment = new Thread(sAM);
  experiment.start();
}

public void run() {

  try {
    stimBG = ImageIO.read(new File("bg.jpg"));
    bgX = (getWidth() / 2 - stimBG.getWidth()) / 2;
    bgY = (getHeight() - stimBG.getHeight()) / 2;
    BufferedImage[] stim = new BufferedImage[N_FRONT + N_BACK];
    for (int i = 0; i < N_FRONT; i++)
      stim[i] = ImageIO.read(new File("front_" + Integer.toString(i + 1)
          + ".jpg"));
    for (int i = 0; i < N_BACK; i++)
      stim[i + N_FRONT] = ImageIO.read(new File("back_"
          + Integer.toString(i + 1) + ".jpg"));
    int stimX_Back = bgX;
    int stimY_Back = bgY;
    int stimX_Front = bgX;
    int stimY_Front = bgY + (stimBG.getHeight() - stim[0].getHeight());
    match = (BufferedImage) createImage(MAX_DISK_SIZE, MAX_DISK_SIZE);
    matchX = getWidth() / 2 + (getWidth() / 2 - match.getWidth()) / 2;
    matchY = (getHeight() - match.getHeight()) / 2;

    List<Integer> trials = new ArrayList<Integer>(N_BACK + N_FRONT);
    for (int i = 0; i < N_BACK + N_FRONT; i++)
      trials.add(i);

    displayText(100, 200, "Task:");
    displayText(100, 350,
        "  Use the arrow keys to adjust the size of the disk on the right");
    displayText(100, 400,
        "  to match the image size of the ball in the scene");
    displayText(100, 450,
        "  (Up/Down for fine adjustment; Left/Right for fast adjustment)");
    displayText(100, 500, "  Press space bar to finalize trial");
    displayText(100, 600, "  Press ESC to quit");
    displayText(100, 800, "Now hit the \"s\" key to start");
    updateScreen();
```

```
while(!getKeyTyped(-1).equals("s")){}

for (int t = 0; t < N_TRIALS; t++) {

  Collections.shuffle(trials);
  Iterator<Integer> it = trials.listIterator();

  while (it.hasNext()) {

    boolean trialDone = false;
    matchSize = (int) (random() * 150);
    generateMatch();
    int nStim = it.next();
    String message = new String("Test Size = "
        + Integer.toString((nStim % 5 + 1) * 20 * 179 / 100) + ";");
    s = stim[nStim];
    if (nStim < N_FRONT) {
      x = stimX_Front;
      y = stimY_Front;
    }
    else {
      x = stimX_Back;
      y = stimY_Back;
    }
    updateScene();
    flushKeyPressed();

    while (!trialDone) {

      Integer keyPressedCode = getKeyPressed(-1);
      switch (keyPressedCode) {
        case LARGER:
          matchSize = min(MAX_DISK_SIZE, matchSize + 2);
          break;
        case SMALLER:
          matchSize = max(0, matchSize - 2);
          break;
        case fLARGER:
          matchSize = min(MAX_DISK_SIZE, matchSize + 20);
          break;
        case fSMALLER:
          matchSize = max(0, matchSize - 20);
          break;
        case COMPLETE:
          trialDone = true;
          break;
      }
      if (!trialDone) {
```

```
                generateMatch();
                updateScene();
              }
            }
            message = message
                + new String(" your response = " + matchSize + " (pixels)");
            blankScreen();
            displayText(message);
            updateScreen();
            Thread.sleep(3000);
          }
        }
      } catch (IOException e) {
        System.err.println("File not found");
        e.printStackTrace();
      } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
        e.printStackTrace();
      }
      finally {
        closeScreen();
      }
    }

    public void updateScene() {

      blankScreen();
      displayImage(bgX, bgY, stimBG);
      displayImage(x, y, s);
      displayImage(matchX, matchY, match);
      updateScreen();
    }

    public void generateMatch() {

      gMatch = match.createGraphics();
      gMatch.setRenderingHints(HINTS);
      gMatch.setColor(Color.BLACK);
      gMatch.fillRect(0, 0, MAX_DISK_SIZE, MAX_DISK_SIZE);
      int x = (int) floor((MAX_DISK_SIZE - matchSize) / 2);
      int y = x;
      gMatch.setColor(Color.GRAY);
      gMatch.fillOval(x, y, matchSize, matchSize);
      gMatch.dispose();
    }
  }
```

Here is another example which allows you to test the timing accuracy of key events

```
/*
 * chapter 6: KeyPressTest.java
 *
 * Test the observer response through keyboard
 *
 */
import java.awt.Font;
import static java.lang.Math.*;

public class KeyPressTest extends FullScreen2 implements Runnable {

  public static void main(String[] args) {

    KeyPressTest kpt = new KeyPressTest();
    kpt.setNBuffers(2);
    new Thread(kpt).start();
  }

  public void run(){

    try {
      displayText(100, 100, "Press keys on your keyboard during this test");
      displayText(100, 150, "Press q to finish the test");
      displayText(100, 200, "Now, press any key to start");
      updateScreen();
      getKeyPressed(-1);
      blankScreen();

      String res;
      int count = 0;
      while (true) {
        res = getKeyTyped(-1);
        blankScreen();
        displayText(count++ + " You pressed: " + res);
        updateScreen();
        if (res.equals("q"))
          break;
        flushKeyTyped();
      }
      flushKeyPressed();
      blankScreen();
      displayText(100, 100, "Next we test the reaction time");
      displayText(100, 150, "Press a key during the counting");
      displayText(100, 200, "Press q to finish the test");
      displayText(100, 250, "Now, press any key to start");
      updateScreen();
      getKeyPressed(-1);
```

```
      setFont(new Font("SansSerif", Font.BOLD, 46));
      while (true) {
        res = null;
        flushKeyTyped();
        count = 20;
        long start = System.currentTimeMillis();
        while (count>=0) {
          start = System.currentTimeMillis();
          blankScreen();
          displayText(300, 400, "counter: " + String.valueOf(count--));
          updateScreen();
          Thread.sleep(max(0,160-System.currentTimeMillis()+start));
        }
        res = getKeyTyped(1000);
        blankScreen();
        if(res != null){
          displayText(300, 600, "  You pressed: "
            + res + "   Reaction time: "
            + (getWhenKeyTyped() - start));
          if (res.equals("q"))
            break;
        }
        else
          displayText(300,600, "Time out! You are too slow!");
        updateScreen();
        Thread.sleep(3000);
      }
      Thread.sleep(1000);
    } catch (InterruptedException e) {}
    finally {
      closeScreen();
    }
  }
}
```

6.4 Summary

6.4.1 On using the FullScreen methods

To capture observer responses using the built-in methods of FullScreen class:

- Use getKeyTyped() method(s) to capture the characters the observer presses. For instance you can capture a capital a, “A”, with these methods (even though what the observer presses is possible “shift” and “a” keys together.)
- Use getKeyPressed() method(s) to get the code of the key hit by the observer. This method can report keys that don’t correspondond to ASCII characters such as arrow keys. Use getKeyPressed() methods if you need to capture such key presses. Similarly use getKeyReleased() method(s) to get the code of the released keys.

- Use getWhenKeyTyped(), getWhenKeyPressed() and getWhenKeyReleased() methods to get the time of the keyTyped, keyPressed and keyReleased events.
- getKeyTyped(), getKeyPressed() and getKeyReleased() methods are all overloaded. Invoke them with no argument if you need to immediately get the first key the observer has pressed. Those methods return null if the observer hasn't pressed anykey. Invoke those methods with a negative argument (for example with -1) if you want your program wait until the observer presses any key. Invoke them with a positive value (for example 1000) if you want to wait for the observer press a key for the time you specified in the argument (in milliseconds). If no key is pressed during the specified time the methods return null.
- use flushKeyTyped(), flushKeyPressed() and flushKeyReleased() methods to clear the event buffers.

6.4.2 On writing your own specialized event handler

...

8 Color look up tables

In this chapter

- What is look-up table? What is (inverse) look-up operation?
- Preparing color look-up tables
- Building your own custumized object for (inverse) look-up operaration
- Built-in objects for look-up operations in core Java

8.1 What is a look-up table? Why do you need an inverse look-up operation?

When you want to draw anything on your screen, say a uniform gray square at the center of the monitor, you do this by creating an object with spatial attributes (its location and dimensions, etc.) and with a certain pixel value. The position and dimensions of the square are in units of number of pixels, which means that their actual size in centimeters will depend on your monitor. Similarly, the *luminance* of, i.e. the amount of light emitted by, the square will depend not only on the pixel value you assigned, but also on your monitor, its brightness and color settings, and on your video card. In a visual psychophysics experiment you naturally want to make sure that the luminance is under your control, just as you want to make sure the visual angle the stimulus subtends. If you can *assume* that your monitor's output is stable, i.e. the amount of light it emits does not change in time, you can measure its output once and then use that measurement as your reference. The first step is preparing a *look-up table*, a table which contains the luminance of your display corresponding to each pixel value, from 0 to 255. Second, upon deciding what luminance to display on your screen, you check your look-up table to find the pixel value whose luminance is closest to the luminance you desired to display. I refer to this second step as *inverse look-up operation*.

8.1.1 Preparing the look-up table

Computer monitors have three color channels (also called as *gun*, referring to the electron guns in conventional CRT monitors), R, G, and B. Therefore you have to prepare a look-up table for each channel. In principle one would have to measure and record the luminance for each pixel value at every location on the monitor. Not only that, one would also have to measure luminance for each channel with varying the other two channels. Moreover, one would also have to measure luminance at each location in relation to neighboring locations. That is a task almost impossible to perform. Fortunately there are some sound assumptions you can make to reduce the number of measurements. Those assumptions include

- Spatial independence, i.e. output of a pixel at one particular location on the screen doesn't depend on the values of other pixels,
- Channel constancy, i.e. the relative spectrum of a channel is independent of the pixel value,

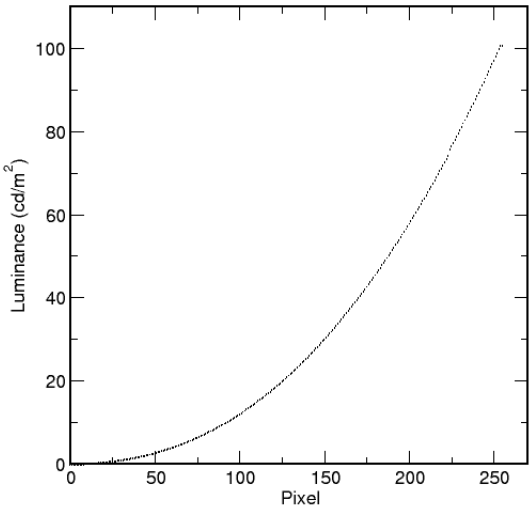

Figure 8.1: A plot showing how monitor output typically depends on pixel values

- Channel independence, i.e. one channel's output doesn't affect the outputs of other two channels,
- Spatial homogeneity, i.e. the output of pixels do not depend on where they are on the screen.

These assumptions reasonably hold for most modern CRTs (See David H. Brainard, 1989, for more details).

Given that the above conditions are satisfied, you can prepare a look up table by simply presenting a uniform patch on your screen and measuring its luminance using a spectro-photometer for all possible pixel values and for all three color channels. Best is to place the patch at the center of the screen. Also remember that a CRT monitor has to *warm up* for at least half hour before it stabilizes. I had to wait up to 2 hours for some monitors until they finally warmed up completely and the luminance measurements stabilized. You can easily observe it: turn the monitor on and present a very bright patch at the center, (255,255,255). Start measuring the luminance at 5 minute intervals. You will notice that the measurement will fluctuate in the beginning and stabilize later. Only after that you should start the actual measurement. Then you simply measure the luminance for different pixel values for each color channel and store them somewhere (on a piece of paper or better in a file on your hard disk), ideally you make 256x3=768 measurements. It is advisable to repeat each measurement 3 times and to take the average and inspect the standard deviations. This is your look-up table. (Note: CRTs are most unstable at highest and lowest ends of their output range, they are more stable around middle values.) See Figure 8.1.

Although you normally have to prepare a separate look-up table for each channel (R,G,B) you can get away with only one table if the output of each channel is exact same. This is also not an uncommon property for modern computer monitors. Manufacturers pay attention to keeping the chromaticity of "gray" same throughout the entire pixel range (say D65). A fortunate consequence of this is that the contribution from all three channels vary in exact same way to maintain a constant gray, apart from its intensity.

A further reduction in number of measurements is possible. You can limit the number of measurements to every other 5th, 10th or even 20th pixel value, and then either fit a function to the data or use an interpolation algorithm to assign values to those pixel values you didn't measure. The functional relation

between the pixel value and the luminance can be represented as

$$L(p) = f(p) \;\; p = 0, ..., 255$$

where L is the luminance value, p is the pixel value, for example a common form is

$$f(p) = b + g * p^{\gamma},$$

where b, g, and γ are free parameters determined by fitting a curve to the data at hand (b is usually referred to as *bias,* g as *gain*.) For interpolation, linear or cubic spline are usually sufficient. (Matlab users: use fminsearch() method for function fit, interp1() method for interpolation, or see Chapter XXX on numerical methods in this guide - *Coming soon!*). Note that in the functional form where there will always be some disagreement between $f(p)$ and the actual luminance presented on the screen. The function will even approximate the values you measured, and alter them. In case of interpolation those values you measured remain untouched, only the values you didn't measure are approximated. On the other hand the functional form may sometimes be advantageous because finding the inverse of a function may be easier than "inverting" a table. In the rest of this chapter, however, I will focus only on the tabular form as it is the more accurate. Whether functional fit or interpolation, you store the luminance values corresponding to entire range of pixel values at some safe place for later use. Later when you need, you export the data and store the table in an array, say pix2Lum - pixel to luminance table

$$L(p) = pix2Lum[p], \;\; p = 0, ..., 255.$$

(Maybe more on functional form later?)

8.2 Inverse operation: finding out which pixel gives the desired luminance

Suppose that the luminance value you want to show on the screen is Lum. You must first understand that you may not be able to present this exact value on the screen because of the limited number of pixel values. What you do is try and find the pixel whose luminance is closest to the desired value. Let's denote it by p^*

$$p^* = \arg\min_p(pix2Lum[p] - Lum), \;\; p = 0 \ldots 255.$$

This is the essence of what I call the inverse look-up operation. This operation would be costly if you repeat it every time for every pixel on the screen, searching the entire 0 to 255 range. However there are some ways to simplify it. I will show them step by step below.

First, let's normalize the luminance values to fit the range from 0 to 255 for each pixel

$$Lum(p) \rightarrow Lum(p)/MaxLum * 255,$$

where MaxLum is the maximum possible luminance. The resulting $pix2Lum[p]$ array is plotted in Figure 8.2.

Second, given the normalized table it is possible to determine p^* for integer (normalized) luminance values

$$p^* = \arg\min_p(pix2Lum[p] - Lum), \;\; \text{where } Lum \text{ is integer}$$

and store this in a new inverse look-up table, say lum2Pix[lum] - luminance-to-pixel table. You can just use this table for your inverse look up operation and this wouldn't be such a bad approximation. The result is plotted in Figure 8.3. But how accurate is this inverse look-up table? If you examine the tableGray255.txt file in this directory (it holds the normalized luminance values), you will notice that the spacing between

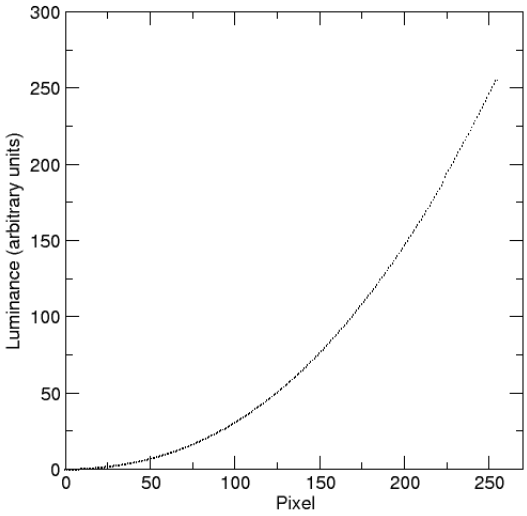

Figure 8.2: Normalized look-up table. pix2Lum[p] array, p=0..255.

the luminance values are usually larger than 1, but what happens when this isn't the case? Consider the following situation: suppose that you want to display a value of 49.40 in relative luminance. Look at the tableGray255.dat file, the closest luminance value in the table is 49.52. But if you are using the inverse look up table (lum2Pix[l] array), you will have to show a relative luminance value closest to 49 (because 49.40 rounds to 49), which is 48.60, not 49.52! Of course it is more accurate to use 49.52, not 48.60. In case you want more accuracy, you still have to go back to the original equation and find the p^* whose luminance is closest to your desired, floating value, luminance.

Fortunately, even if you prefer to be more accurate, the pix2Lum[] array you created will considerably reduce your computational expenses. Again, suppose that you want to display a luminance of 49.40. Assuming monotonicity, i.e. luminance is a monotonically increasing function of pixel values (or decreasing for that matter), you can restrict the search to the neighborhood of 49.40, for example from $p_{start} = lum2Pix[48]$ to $p_{end} = lum2Pix[50]$. This will reduce the search range from 256 to only 4 in this example.

Let's go back and look at the implementation of each of the above steps. After normalizing the luminance values, we construct the crude inverse look-up table as follows, assuming monotonically increasing form here,

```
int DIM = 256;
//...
int lastPixel = 0;
for(int lum=0; lum<DIM; lum++){
  double diff = Double.MAX_VALUE;
  for(int j=lastPixel; j<DIM; j++){
    double diffTmp = abs(pix2Lum[j]-lum);
    if(diffTmp == 0){
      lum2Pix[lum] = j;
      lastPixel = j;
      break;
```

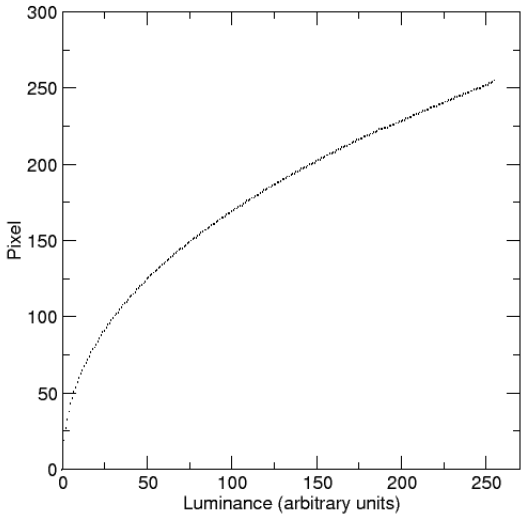

Figure 8.3: Inverse look-up table. lum2Pix[l] array, l=0..255 (integer).

```
    }
    else if(diffTmp <= diff){
      lum2Pix[lum] = j;
      lastPixel = j;
      diff = diffTmp;
    }
    else
      break;
  }
}
```

Similar algorithms can be implemented for monotonically decreasing tables (see next section). For tables neither monotonically increasing nor decreasing, you have to live without the simplifications. The more accurate inverse look-up operation can be done as follows, assuming monotonicity

```
// lum is the desired luminance
int intLum = (int)round(lum);
int pixel = lum2Pix[intLum];

if(lum == (double)intLum)
  return pixel;
else {
  int pixelStart = lum2Pix[max(intLum-1,0)];
  int pixelEnd = lum2Pix[min(intLum+1,DIM-1)];
  double diff = Double.MAX_VALUE;
  for(int j=pixelStart; j<=pixelEnd; j++){
    double diffTmp = abs(pix2Lum[j] - lum);
```

```
      if(diffTmp == 0){
        pixel = j;
        break;
      }
      else if(diffTmp <= diff){
        pixel = j;
        diff = diffTmp;
      }
      else
        break;
    }
    return pixel;
  }
```

I will show the entire code below in the next section.

8.2.1 A class to perform inverse look-up operation

In this section I will build a class, CLUT8, for inverse look-up operations. CLUT8 has two constructors. You can construct a CLUT8 object either by providing a file name which holds the output luminance values for pixels from 255 to 0 in descending order, or you provide a double array which holds the output values for each pixel value. Both constructors invoke the setClut() method to prepare the look-up table, i.e. pix2Lum array, and the crude inverse look-up table, i.e. lum2Pix array. First, the constructor with the file name:

```
public CLUT8(String filename) throws IllegalArgumentException{
  InputStream stream = CLUT8.class.getResourceAsStream(filename);
  Scanner in = new Scanner(stream);
  maxLum = 0;
  double[] table = new double[DIM];
  for (int pix = DIM - 1; pix > -1; pix--) {
    try {
      table[pix] = in.nextDouble();
    } catch (NoSuchElementException e) {
      throw new IllegalArgumentException(
          "Error in CLUT8(String): wrong input file - file too short");
    }
  }
  if (in.hasNext())
    throw new IllegalArgumentException(
        "Error in CLUT8(String): suspicious input file - file too long");
  in.close();
  setClut(table);
}
```

The constructor makes sure that the file contains exactly 256 double valued numbers (see the tableGray.txt file provided in the same directory.) In case the number of elements is not equal to 256 (=DIM), the program throws an exception. The other constructor is similar, except the user provides the data in a double array

```
public CLUT8(double[] table) throws IllegalArgumentException{
```

```
    if (table.length != DIM)
      throw new IllegalArgumentException(
          "Error in CLUT8(double[]): incompatible data");
    else
      setClut(table);
  }
```

Both constructors invoke the setClut() method. setClut() method normalizes the look up table and prepares the inverse look up table. It utilizes different approaches depending on whether the table is monotonically increasing or decreasing, or not monotonic at all. In case it is not monotonic, it warns the user but not throws an exception.

```
  public void setClut(double[] table){

    if (table.length != DIM)
      throw new IllegalArgumentException(
          "Error in CLUT8.setClut(double[]): incompatible data");

    maxLum = 0;
    pix2Lum = table;
    double lastLum = pix2Lum[0];
    for (double lum : pix2Lum) {
      maxLum = max(maxLum, lum);
      if (monotonicIncrease && lastLum > lum)
        monotonicIncrease = false;
      else if(monotonicDecrease && lastLum < lum)
        monotonicDecrease = false;
      lastLum = lum;
    }

    if(!monotonicIncrease && !monotonicDecrease){
      System.err
      .println("Warning in CLUT8.setClut(double[]): "
          + "LUT is not monotonically increasing or decreasing, "
          + "speed may degrade");
    }

    for (int pix = 0; pix < DIM; pix++)
      pix2Lum[pix] *= 255 / maxLum;

    if(monotonicIncrease){
      int lastPixel = 0;
      for (int lum = 0; lum < DIM; lum++) {
        double diff = Double.MAX_VALUE;
        for (int j = lastPixel; j < DIM; j++) {
          double diffTmp = abs(pix2Lum[j] - lum);
          if (diffTmp == 0) {
```

```
        lum2Pix[lum] = j;
        lastPixel = j;
        break;
      }
      else if (diffTmp <= diff) {
        lum2Pix[lum] = j;
        lastPixel = j;
        diff = diffTmp;
      }
      else
        break;
    }
  }
}
else if(monotonicDecrease){
  int lastPixel = 0;
  for (int lum = DIM-1; lum >= 0; lum--) {
    double diff = Double.MAX_VALUE;
    for (int j = lastPixel; j < DIM; j++) {
      double diffTmp = abs(pix2Lum[j] - lum);
      if (diffTmp == 0) {
        lum2Pix[lum] = j;
        lastPixel = j;
        break;
      }
      else if (diffTmp <= diff) {
        lum2Pix[lum] = j;
        lastPixel = j;
        diff = diffTmp;
      }
      else
        break;
    }
  }
}
else{
  for (int lum = 0; lum < DIM; lum++) {
    double diff = Double.MAX_VALUE;
    for (int j = 0; j < DIM; j++) {
      double diffTmp = abs(pix2Lum[j] - lum);
      if (diffTmp == 0) {
        lum2Pix[lum] = j;
        break;
      }
      else if (diffTmp <= diff) {
        lum2Pix[lum] = j;
        diff = diffTmp;
      }
```

```
      }
    }
  }
}
```

Note that setClut() is a public method and the user can invoke it as well. This would be useful if you do a look up table animation (see Chapter XX on Bits++).

There in one method to obtain the outcome of inverse look up operation, it is lum2Pix() method. However this method is overloaded. If you invoke it with an integer value, it simply returns that element of lum2Pix[] array

```
public int lum2Pix(int lum) {

  return lum2Pix[lum];
}
```

if you invoke it with a double argument, then it performs the more accurate inverse look up operation as described above

```
public int lum2Pix(double lum) {
  int intLum = (int) round(lum);
  int pixel = lum2Pix[intLum];
  if (lum == (double) intLum)
    return pixel;
  else {
    if(monotonicIncrease){
      int pixelStart = lum2Pix[max(intLum - 1, 0)];
      int pixelEnd = lum2Pix[min(intLum + 1, DIM - 1)];
      double diff = Double.MAX_VALUE;
      for (int j = pixelStart; j <= pixelEnd; j++) {
        double diffTmp = abs(pix2Lum[j] - lum);
        if (diffTmp == 0) {
          pixel = j;
          break;
        }
        else if (diffTmp <= diff) {
          pixel = j;
          diff = diffTmp;
        }
        else
          break;
      }
    }
    else if(monotonicDecrease){
      int pixelStart = lum2Pix[max(intLum + 1, 0)];
      int pixelEnd = lum2Pix[min(intLum - 1, DIM - 1)];
      double diff = Double.MAX_VALUE;
      for (int j = pixelStart; j <= pixelEnd; j++) {
        double diffTmp = abs(pix2Lum[j] - lum);
```

```
            if (diffTmp == 0) {
              pixel = j;
              break;
            }
            else if (diffTmp <= diff) {
              pixel = j;
              diff = diffTmp;
            }
            else
              break;
          }
        }
        else {
          double diff = Double.MAX_VALUE;
          for (int j = 0; j < DIM; j++) {
            double diffTmp = abs(pix2Lum[j] - lum);
            if (diffTmp == 0) {
              pixel = j;
              break;
            }
            else if (diffTmp <= diff) {
              pixel = j;
              diff = diffTmp;
            }
          }
        }
        return pixel;
      }
    }
```

Here are the other methods of the class. To inquire the maximum luminance in your look up table use getMaxLum() method

```
  public double getMaxLum() {

    return maxLum;
  }
```

To obtain the relative luminance of a pixel

```
    public double pix2Lum(int pixel){

      return pix2Lum[pixel];
    }
```

Here is the entire code of CLUT8 class

```
  /*
   * Chapter 8: CLUT8.java
```

```
 *
 * Provides methods to perform (inverse) look up operations.
 *
 */
import static java.lang.Math.*;
import java.awt.Color;
import java.awt.Graphics2D;
import java.awt.event.KeyEvent;
import java.awt.image.BufferedImage;
import java.io.InputStream;
import java.util.NoSuchElementException;
import java.util.Scanner;
public class CLUT8 {
  private final static int DIM = 256;
  private double maxLum;
  private double[] pix2Lum = new double[DIM];
  private int[] lum2Pix = new int[DIM];
  private boolean monotonicIncrease = true;
  private boolean monotonicDecrease = true;
  public CLUT8(String filename) throws IllegalArgumentException{
    InputStream stream = CLUT8.class.getResourceAsStream(filename);
    Scanner in = new Scanner(stream);
    maxLum = 0;
    double[] table = new double[DIM];
    for (int pix = DIM - 1; pix > -1; pix--) {
      try {
        table[pix] = in.nextDouble();
      } catch (NoSuchElementException e) {
        throw new IllegalArgumentException(
            "Error in CLUT8(String): wrong input file - file too short");
      }
    }
    if (in.hasNext())
      throw new IllegalArgumentException(
          "Error in CLUT8(String): suspicious input file - file too long");
    in.close();
    setClut(table);
  }
  public CLUT8(double[] table) throws IllegalArgumentException{
    if (table.length != DIM)
      throw new IllegalArgumentException(
          "Error in CLUT8(double[]): incompatible data");
    else
      setClut(table);
  }

  public void setClut(double[] table){
```

```
if (table.length != DIM)
  throw new IllegalArgumentException(
      "Error in CLUT8.setClut(double[]): incompatible data");

maxLum = 0;
pix2Lum = table;
double lastLum = pix2Lum[0];
for (double lum : pix2Lum) {
  maxLum = max(maxLum, lum);
  if (monotonicIncrease && lastLum > lum)
    monotonicIncrease = false;
  else if(monotonicDecrease && lastLum < lum)
    monotonicDecrease = false;
  lastLum = lum;
}

if(!monotonicIncrease && !monotonicDecrease){
  System.err
  .println("Warning in CLUT8.setClut(double[]): "
      + "LUT is not monotonically increasing or decreasing, "
      + "speed may degrade");
}

for (int pix = 0; pix < DIM; pix++)
  pix2Lum[pix] *= 255 / maxLum;

if(monotonicIncrease){
  int lastPixel = 0;
  for (int lum = 0; lum < DIM; lum++) {
    double diff = Double.MAX_VALUE;
    for (int j = lastPixel; j < DIM; j++) {
      double diffTmp = abs(pix2Lum[j] - lum);
      if (diffTmp == 0) {
        lum2Pix[lum] = j;
        lastPixel = j;
        break;
      }
      else if (diffTmp <= diff) {
        lum2Pix[lum] = j;
        lastPixel = j;
        diff = diffTmp;
      }
      else
        break;
    }
  }
}
else if(monotonicDecrease){
```

```
    int lastPixel = 0;
    for (int lum = DIM-1; lum >= 0; lum--) {
      double diff = Double.MAX_VALUE;
      for (int j = lastPixel; j < DIM; j++) {
        double diffTmp = abs(pix2Lum[j] - lum);
        if (diffTmp == 0) {
          lum2Pix[lum] = j;
          lastPixel = j;
          break;
        }
        else if (diffTmp <= diff) {
          lum2Pix[lum] = j;
          lastPixel = j;
          diff = diffTmp;
        }
        else
          break;
      }
    }
  }
  else{
    for (int lum = 0; lum < DIM; lum++) {
      double diff = Double.MAX_VALUE;
      for (int j = 0; j < DIM; j++) {
        double diffTmp = abs(pix2Lum[j] - lum);
        if (diffTmp == 0) {
          lum2Pix[lum] = j;
          break;
        }
        else if (diffTmp <= diff) {
          lum2Pix[lum] = j;
          diff = diffTmp;
        }
      }
    }
  }
}

public double getMaxLum() {
  return maxLum;
}

public int lum2Pix(double lum) {
  int intLum = (int) round(lum);
  int pixel = lum2Pix[intLum];
  if (lum == (double) intLum)
    return pixel;
  else {
```

```
if(monotonicIncrease){
  int pixelStart = lum2Pix[max(intLum - 1, 0)];
  int pixelEnd = lum2Pix[min(intLum + 1, DIM - 1)];
  double diff = Double.MAX_VALUE;
  for (int j = pixelStart; j <= pixelEnd; j++) {
    double diffTmp = abs(pix2Lum[j] - lum);
    if (diffTmp == 0) {
      pixel = j;
      break;
    }
    else if (diffTmp <= diff) {
      pixel = j;
      diff = diffTmp;
    }
    else
      break;
  }
}
else if(monotonicDecrease){
  int pixelStart = lum2Pix[max(intLum + 1, 0)];
  int pixelEnd = lum2Pix[min(intLum - 1, DIM - 1)];
  double diff = Double.MAX_VALUE;
  for (int j = pixelStart; j <= pixelEnd; j++) {
    double diffTmp = abs(pix2Lum[j] - lum);
    if (diffTmp == 0) {
      pixel = j;
      break;
    }
    else if (diffTmp <= diff) {
      pixel = j;
      diff = diffTmp;
    }
    else
      break;
  }
}
else {
  double diff = Double.MAX_VALUE;
  for (int j = 0; j < DIM; j++) {
    double diffTmp = abs(pix2Lum[j] - lum);
    if (diffTmp == 0) {
      pixel = j;
      break;
    }
    else if (diffTmp <= diff) {
      pixel = j;
      diff = diffTmp;
    }
```

```
      }
    }
    return pixel;
  }
}

public int lum2Pix(int lum) {
  return lum2Pix[lum];
}

public double pix2Lum(int pixel) {
  return pix2Lum[pixel];
}

public static void main(String[] args) {

  FullScreen fs = new FullScreen();
  try {
    // Choose one of the two constructors
    //CLUT8 lut8 = new CLUT8("tableGray.txt");
    double[] table = new double[256];
    for (int i = 0; i < 256; i++)
      table[i] = 1.0 - 1.0 * pow((i / 255.0), 1.0);

    CLUT8 lut8 = new CLUT8(table);

    BufferedImage rectangle = new BufferedImage(fs.getWidth() / 2, fs
        .getHeight() / 2, BufferedImage.TYPE_BYTE_GRAY);
    double maxLum = lut8.getMaxLum();
    double lum = 127.5;
    int pix = lut8.lum2Pix(lum);
    fs.blankScreen();
    fs.displayText(20, 50, "Use arrow keys to change the luminance");
    fs.displayText(20, 100, "(Press any key to continue, press ESC to quit)");
    fs.updateScreen();
    while (true) {
      int res = fs.getKeyPressed(-1);
      if (res == KeyEvent.VK_UP)
        lum += 0.5;
      else if (res == KeyEvent.VK_DOWN)
        lum -= 0.5;
      else if (res == KeyEvent.VK_LEFT)
        lum -= 10;
      else if (res == KeyEvent.VK_RIGHT)
        lum += 10;
      lum = Math.max(0, lum);
      lum = Math.min(255, lum);
      pix = lut8.lum2Pix(lum);
```

```
            Graphics2D g = rectangle.createGraphics();
            g.setColor(new Color(pix, pix, pix));
            g.fillRect(0, 0, rectangle.getWidth(), rectangle.getHeight());
            g.dispose();
            fs.blankScreen();
            fs.displayImage(rectangle);
            fs.displayText(10, 50, "Desired relative Luminance was: " + lum);
            fs.displayText(10, 120, "Displayed relative Luminance is: "
                + lut8.pix2Lum(pix));
            fs.displayText(10, fs.getHeight() - 40, "(Actual luminance : "
                + lut8.pix2Lum(pix) * maxLum / 255.0 + " cd/m2)");
            fs.updateScreen();
          }
        }
        catch (IllegalArgumentException e){
          e.printStackTrace();
        }
        finally {
          fs.closeScreen();
        }
      }
    }
```

Note that in this class I included a main() method to test it. Including main() method like this one is a useful strategy to test your classes quickly. You can execute the LUT8 class as usual and inspect how the luminance values shown on your screen and the desired values differ.

8.3 Example: (inverse) Look-up operation on an image

One often stores stimuli in image files of standard formats, for example jpeg or png. In case you have such an image, whose pixel values are to be treated as luminance values, and want to apply an inverse look up operation, you can use the CLUT8 class. Note that in case of standard images you will always be using the integer valued relative luminances. Here is sample program to show how you can filter your images by an inverse look up operation

```
/*
 * chapter 8: CLUT8Test.java
 *
 * demonstrates how to apply look up operation on an image using CLUT8
 *
 */
import java.awt.Graphics2D;
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;
import javax.imageio.ImageIO;
public class CLUT8Test {
```

```
public static void main(String[] args){

  FullScreen fs = new FullScreen();

  try {
    BufferedImage bi = ImageIO.read(new File("fechner.png"));

    fs.displayImage((fs.getWidth()/2-bi.getWidth())/2,
        (fs.getHeight()-bi.getHeight())/2, bi);

    double[] table = new double[256];
    for(int i=0; i<256; i++)
      table[i]=255-i;

    CLUT8 lut8 = new CLUT8(table);

    bi = filter(lut8,bi);

    fs.displayImage((fs.getWidth()+bi.getWidth())/2,
        (fs.getHeight()-bi.getHeight())/2, bi);

    fs.displayText(15,fs.getHeight()-15,"(press any key to finish)");

    fs.updateScreen();
    fs.getKeyPressed(-1);

  } catch (IOException e) {
    e.printStackTrace();
  } finally {
    fs.closeScreen();
  }

}

public static BufferedImage filter(CLUT8 lut8, BufferedImage bi){

  BufferedImage result = new BufferedImage(bi.getWidth(), bi.getHeight(),
      BufferedImage.TYPE_BYTE_GRAY);
  Graphics2D g = result.createGraphics();
  g.drawImage(bi,0,0,null);
  g.dispose();
  if(bi.getType() != BufferedImage.TYPE_BYTE_GRAY){
    System.err.println("Image must be gray scale, cannot filter");
    return result;
  }
  else {
    int[] gResult = new int[result.getWidth() * result.getHeight()];
    int[] gBi = new int[bi.getWidth() * bi.getHeight()];
```

```
        bi.getRaster().getPixels(0,0,bi.getWidth() , bi.getHeight(), gBi);
        for (int j = 0; j<gBi.length; j++)
          gResult[j] = lut8.lum2Pix(gBi[j]);

        result.getRaster().setPixels(0, 0, result.getWidth() ,
            result.getHeight(), gResult);
        return result;
      }
    }
  }
```

This example demonstrates how you can use CLUT8 class to perform image filtering. However, see below the section on Standard Java utilities for look up operations. The utilities provided by Standard Java should be preferred for image look up operations.

8.4 Experiment: Cornsweet illusion

In this section I will show how to use CLUT8 in an actual experiment. In this experiment we want to measure the magnitude of the brightness illusion in Craik-O'Brien-Cornsweet stimulus. In the Crack-O'Brien-Cornsweet stimulus two flanking territories of equal luminance are perceived to have different luminances because of a contrast border between them. See Figure 8.4. In this experiment we want to psychophysically determine the magnitude of this illusion in a systematic way and find out how the illusion depends on the contrast at the border.

To determine the magnitude of illusion we use two interval forced choice method (2IFC). The experiment is designed as follows: There are two intervals in which we present stimuli, temporally separated from each other. One of the stimuli is the Craik-O'Brein-Cornsweet stimulus, the other one is a "real" stimulus. The "real" stimulus is composed of two flanking territories with un-equal luminances and a contrast border at the center. Observer's task is to indicate the interval in which the perceived luminance difference was larger between the two f1anks. To make the experiment more controlled, we superimpose two square frames on the flanks, and ask the observer to make the judgment by comparing the luminance within the square frames. We use three fixed contrast levels: 0.3, 0.6, and 0.9. For each contrast level we seek to find the corresponding perceptually equivalent real stimulus. The difference between the flanks of that perceptually equivalent real stimulus will be assigned as the magnitude of the illusion. Every time the observer responds that the illusory stimulus has larger difference between its flanks, the program increases the contrast of the real stimulus for the next trial, otherwise the next trial has a lower real contrast. This is a 1 up 1 down staircase procedure. The number of trials for each staircase is fixed, then the data can easily be analyzed to determine the subjectively equivalent real stimulus for each contrast level. See Figure 8.5.

Let's start with the methods which create the illusory and real stimuli. The real stimulus is created as follows

```
public BufferedImage prepReal(int polarity, double contrast) {
  // Luminances of Uniform flanks
  double meanLeft = meanLum * (1 + contrast / 2 * polarity);
  double meanRight = meanLum * (1 - contrast / 2 * polarity);
  int[] gStim = new int[stimWidth * stimHeight];
  // Convert the Luminance to Pixel using the LUT
  for (int j = 0; j < stimWidth / 2; j++)
    gStim[j] = lut8.lum2Pix(meanLeft);
```

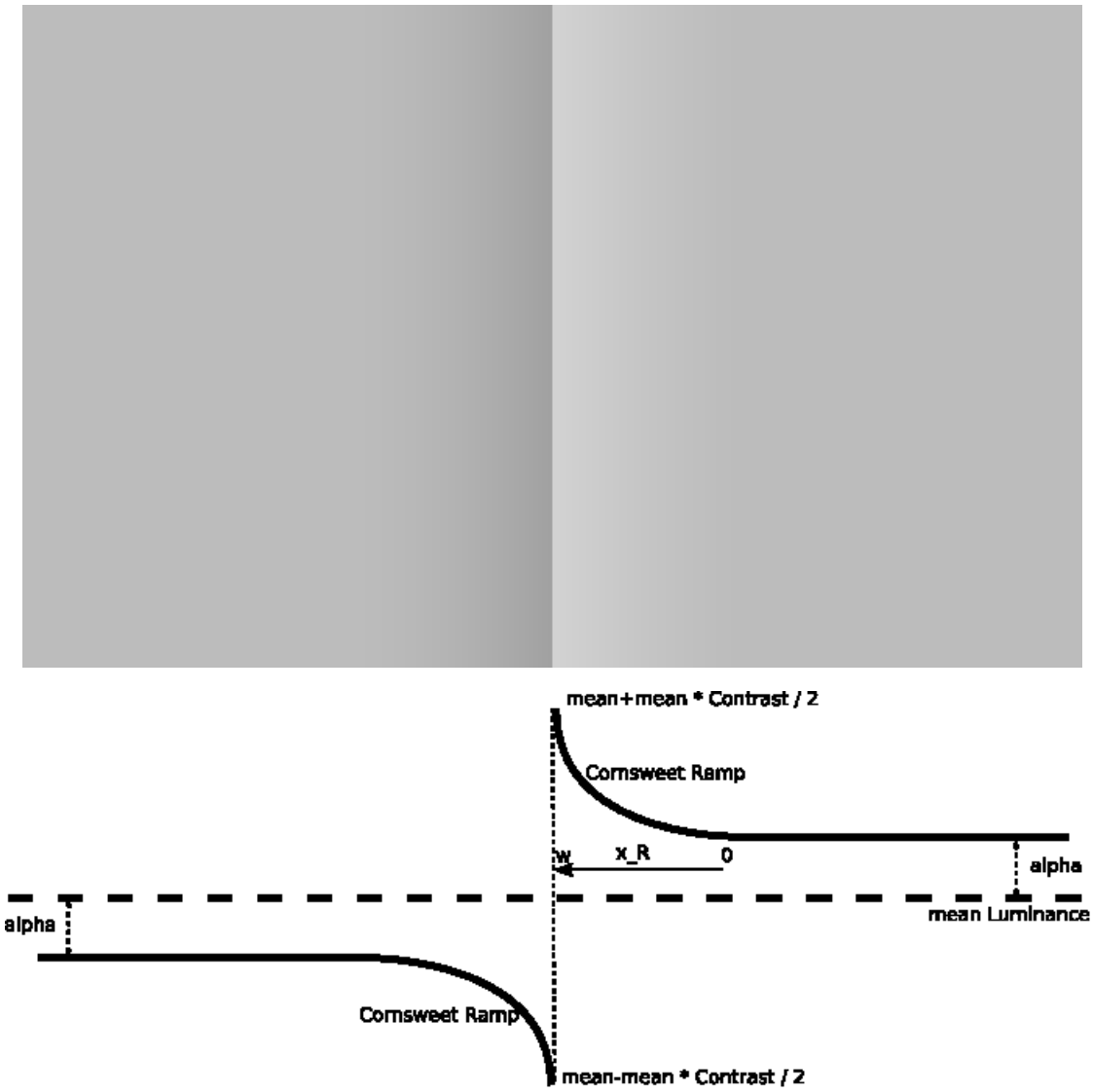

Figure 8.4: Craik-O'Brein-Cornsweet illusion. Cornsweet luminance profile. The equation of the Right ramp is: $L_R = (mean + \alpha) + (mean * Contrast/2 - \alpha) * (\frac{x_R}{w})^{\xi}$. This gives us the flexibility to choose $Contrast$,α and ξ. $\alpha = 0$ in the original Cornsweet stimulus at the top.

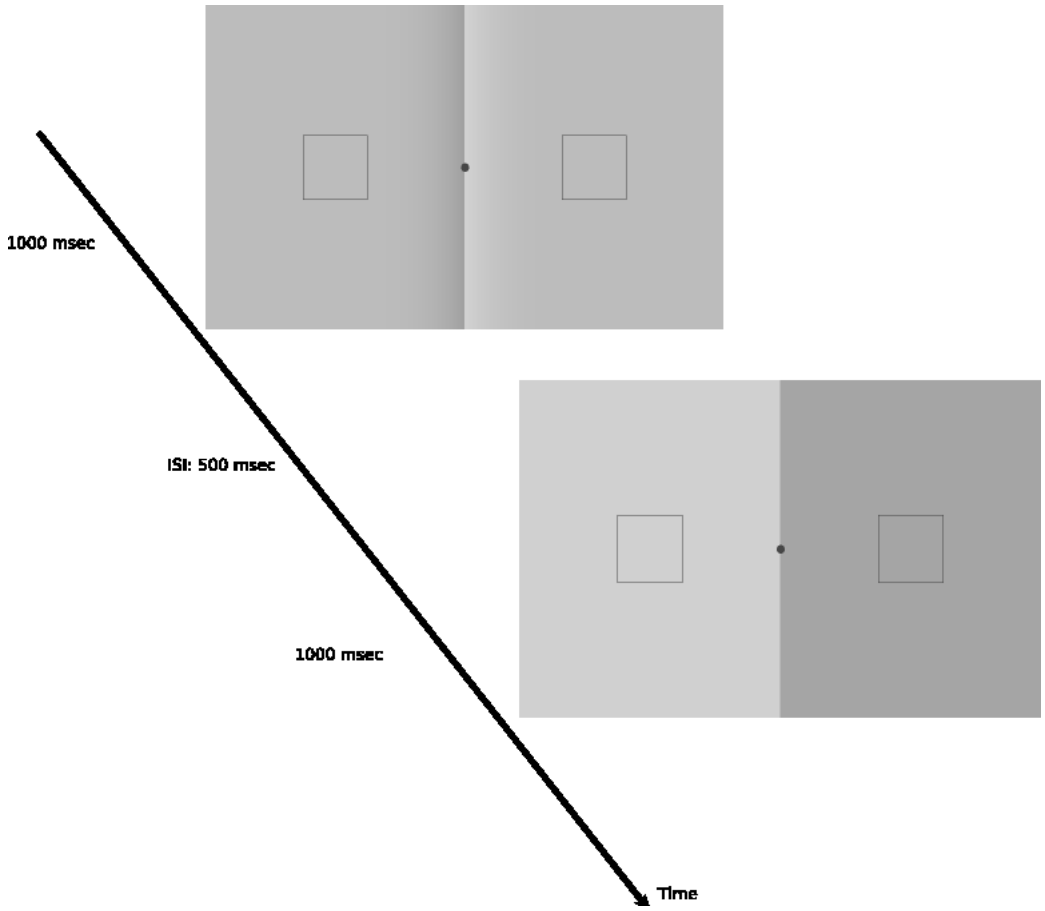

Figure 8.5: Experimental design. Observer's task is to indicate the interval with a larger difference between the flanks. The thin frames are provided to help the observers with the task. Observers were required to fixate at the center. A staircase procedure is used with one up one down rule, this results in a subjective equality between illusory and actual luminance variation. During ISI the screen is blank except the fixation mark. In order to eliminate cognitive effects for the non-naive observers we include trials with positive $\alpha$where there are actual differences between the flanks (always in the direction of illusion, not in the nulling direction) of Cornsweet stimulus.

```
  for (int j = stimWidth / 2; j < stimWidth; j++)
    gStim[j] = lut8.lum2Pix(meanRight);
  for (int i = 1; i < stimHeight; i++)
    for (int j = 0; j < stimWidth; j++)
      gStim[i * stimWidth + j] = gStim[j];
  // write the result in an image
  BufferedImage stim = new BufferedImage(stimWidth, stimHeight,
      BufferedImage.TYPE_BYTE_GRAY);
  stim.getRaster().setPixels(0, 0, stimWidth, stimHeight, gStim);
  return stim;
}
```

If polarity=1, the left flank is brighter, if polarity=-1 then the right flank is brighter. This method returns a BufferedImage that you can easily display on the screen using the displayImage() method of FullScreen class. And for the illusory stimulus

```
public BufferedImage prepAsymCorn(int polarity, double alpha,
    double contrast) {
  // Luminances of Uniform part of flanks
  double meanLeft = meanLum + alpha * polarity;
  double meanRight = meanLum - alpha * polarity;
  // Convert the Luminance to Pixel using the LUT
  int[] gStim = new int[stimWidth * stimHeight];
  for (int j = 0; j < stimWidth / 2 - rampWidth; j++)
    gStim[j] = lut8.lum2Pix(meanLeft);
  for (int j = stimWidth / 2 - rampWidth; j < stimWidth; j++)
    gStim[j] = lut8.lum2Pix(meanRight);
  // Luminances of Cornsweet2AFC ramps
  double delta = contrast / 2 * meanLum;
  for (int j = 0; j < rampWidth; j++) {
    // Left ramp
    int k = j + stimWidth / 2 - rampWidth;
    double tmp = pow((double) j / (double) (rampWidth - 1), exponent)
        * (-alpha + delta) * polarity;
    gStim[k] = lut8.lum2Pix(meanLeft + tmp);
    // Right ramp
    k = stimWidth / 2 + rampWidth - j - 1;
    gStim[k] = lut8.lum2Pix(meanRight - tmp);
  }
  for (int i = 1; i < stimHeight; i++)
    for (int j = 0; j < stimWidth; j++)
      gStim[i * stimWidth + j] = gStim[j];
  // write the result in an image
  BufferedImage stim = new BufferedImage(stimWidth, stimHeight,
      BufferedImage.TYPE_BYTE_GRAY);
  stim.getRaster().setPixels(0, 0, stimWidth, stimHeight, gStim);
  return stim;
}
```

Even though we assign some fixed contrast levels, the actual displayed contrast may not always be equal to those desired values, therefor we first check what actually the contrast and α values of the Craik-O'Brein-Cornsweet stimuli are

```
// compute the actual alpha, and contrastCorn values:
BufferedImage tmp;
for (int i = 0; i < contrastCorn.length; i++) {
  tmp = prepAsymCorn(1, 0, contrastCorn[i]);
  System.err.println("Requested contrastCorn = " + contrastCorn[i]);
  int[] px = new int[tmp.getWidth()];
  tmp.getRaster().getPixels(0, 0, tmp.getWidth() - 1, 1, px);
  contrastCorn[i] = (lut8.pix2Lum(px[tmp.getWidth() / 2 - 1]) - lut8
      .pix2Lum(px[tmp.getWidth() / 2]))
      / (lut8.pix2Lum(px[tmp.getWidth() / 2 - 1]) + lut8.pix2Lum(px[tmp
          .getWidth() / 2])) * 2;
  System.err.println("Received contrastCorn = " + contrastCorn[i]);
}
for (int i = 0; i < alphaCorn.length; i++) {
  tmp = prepAsymCorn(1, alphaCorn[i], 0);
  System.err.println("Requested alphaCorn = " + alphaCorn[i]);
  int[] px = new int[tmp.getWidth()];
  tmp.getRaster().getPixels(0, 0, tmp.getWidth() - 1, 1, px);
  alphaCorn[i] = (lut8.pix2Lum(px[1]) - lut8
      .pix2Lum(px[tmp.getWidth() - 2])) / 2;
  System.err.println("Received alphaCorn = " + alphaCorn[i]);
}
```

The trial order is taken care as follows: We create two HashMaps each holding the parameters of the illusory stimuli and the real stimuli respectively. In both HashMaps the keys are trial numbers, values are the parameters:

```
// trial initialization: corn holds the
// (KEY=)trial number - (VALUE=)cornsweet parameters (contrast,alpha)
HashMap<Integer, ArrayList<Double>> corn =
  new HashMap<Integer, ArrayList<Double>>();
// real holds the (KEY=)trial number - (VALUE=)real contrast
HashMap<Integer, Double> real = new HashMap<Integer, Double>();
// nTrial holds the entire trial number in a List
List<Integer> nTrial = new LinkedList<Integer>();
int t = 0;
for (int j = 0; j < contrastCorn.length; j++) {
  for (int k = 0; k < alphaCorn.length; k++) {
    ArrayList<Double> tmp = new ArrayList<Double>();
    tmp.add(contrastCorn[j]);
    tmp.add(alphaCorn[k]);
    corn.put(t, tmp);
    real.put(t, initContrast[k][j]);
    nTrial.add(t);
```

```
      t++;
    }
  }
```

I initialize the contrast of real stimuli deterministically, rather than randomly. It is done as follows, I create two staircases for each contrast value, one starting from very high initial real contrast values, one from very low. Of course first few trials of them will be trivial for the observer, but this strategy usually results in a data set which is easier to fit a psychometric function. Note that we also create a List for trial numbers. The next piece of code below shows how you randomize the order of trials and then extract the corresponding parameters

```
Collections.shuffle(nTrial);
Iterator<Integer> itTrial = nTrial.listIterator();
while (itTrial.hasNext()) {
  int it = itTrial.next();
  // extract the corresponding cornsweet parameters for this trial
  ArrayList<Double> tmp = corn.get(it);
  double contrast = tmp.get(0);
  double alpha = tmp.get(1);
  // extract the corresponding real contrast for this trial
  double contrastReal = real.get(it);
  // Code to present the stimuli with above parameters ....
}
```

Then we get the observer's response and decide what real contrast we should show in the next iteration for that trial number: if observer responded "illusory stimulus had larger difference between flanks" we set cornIsBigger=true and set the real contrast for the next trial higher than the current one. We do the opposite if the observer responds the other way

```
  resp = getKeyTyped(-1);
  if ((resp.equals(resp1) && order == -1)
      || (resp.equals(resp2) && order == 1)) {
    cornIsBigger = true;
    real.put(it, min(MAXCONTRAST, contrastReal + deltaContrastReal));
    break;
  }
  else if ((resp.equals(resp1) && order == 1)
      || (resp.equals(resp2) && order == -1)) {
    cornIsBigger = false;
    real.put(it, max(0.0, contrastReal - deltaContrastReal));
    break;
  }
  else
    continue;
}
```

In the end of the trial, before reporting the result we compute the actual contrast level of the real stimulus, because it may be different than the desired value

```
            // check the actual displayed contrastReal and report the result
            int tmp2 = (-order + 1) / 2;
            int[] px = new int[stimulus[tmp2].getWidth()];
            stimulus[tmp2].getRaster().getPixels(0, 0,
                stimulus[tmp2].getWidth() - 1, 1, px);
            double contrastRealActual = abs(lut8.pix2Lum(px[0])
                - lut8.pix2Lum(px[px.length - 2]))
                / (lut8.pix2Lum(px[0]) + lut8.pix2Lum(px[px.length - 2])) * 2;
            output.printf("%f %f %f %f %s \n", exponent, contrast, alpha,
                contrastRealActual, cornIsBigger);
            output.flush();
```

Here is the entire code

```
/*
 * chapter 8: Cornsweet2IFC.java
 *
 * An actual experiment demonstrating how to use CLUT8 class
 *
 */
import static java.lang.Math.*;
import java.awt.*;
import java.awt.geom.Ellipse2D;
import java.awt.image.BufferedImage;
import java.awt.image.ColorModel;
import java.io.*;
import java.util.*;
import java.util.List;
public class Cornsweet2IFC extends FullScreen implements Runnable {
  // How many pixels is oneDegree? This will depend on screen resolution,
  // monitor dimensions and viewing distance
  static final int oneDegree  = 51;
  static final int stimWidth  = 16 * oneDegree;
  static final int stimHeight = 10 * oneDegree;
  static final int rampWidth  = 3 * oneDegree;
  static final int frameWidth = 2 * oneDegree;
  // "desired" Luminance (0-255) and Contrast values
  static final double meanLum = 127.0;
  static final double[] contrastCorn = { .6, .6, .9, .9 };
  static final double exponent = 2.75;
  static final double[] alphaCorn = { 0.0, 6.35 };
  static final double[][] initContrastReal = { { .1, .5, .1, .5 },
    { .2, .7, .2, .7 } };
  static final double deltaContrastReal = .04;
  static final double MAXCONTRAST = 2.0;
  // time parameters for the 2IFC
  static final long INTERVAL1 = 1000;
  static final long INTERVAL2 = 1000;
```

```
static final long ISI = 500;
static final int NTRIALS = 15;
static final int fixSize = oneDegree / 4;
static final Color fixationMarkColor = new Color(255,70,70);
static final Shape fixShape = new Ellipse2D.Double(0,0,fixSize,fixSize);
BufferedImage fixation;
int fixX;
int fixY;

BufferedImage frame;
int[] frameX = new int[2];
int frameY;
// Response Buttons:
static final String resp1 = "1";
static final String resp2 = "2";

CLUT8 lut8;
public Cornsweet2IFC(int i) {
  super(i);

  lut8 = new CLUT8("tableGray.txt");
  // Fixation mark
  fixX = (getWidth() - fixSize) / 2;
  fixY = (getHeight() - fixSize) / 2;
  fixation = new BufferedImage(fixSize,fixSize,
      BufferedImage.TRANSLUCENT);
  Graphics2D fsG = (Graphics2D) fixation.getGraphics();
  fsG.setPaint(fixationMarkColor);
  fsG.fill(fixShape);
  fsG.dispose();
  // rectangular wire frames
  frame = new BufferedImage(frameWidth, frameWidth,
      BufferedImage.TRANSLUCENT);
  fsG = (Graphics2D) frame.getGraphics();
  fsG.setPaint(new Color(70, 70, 70));
  fsG.setStroke(new BasicStroke(.5f));
  fsG.drawRect(0, 0, frameWidth - 1, frameWidth - 1);
  fsG.dispose();
  frameX[0] = (getWidth() - stimWidth) / 2 + (int) (3 * oneDegree);
  frameX[1] = getWidth() - frameX[0] - frame.getWidth();
  frameY = (getHeight() - frame.getHeight()) / 2;
}
public static void main(String[] args) {
  Cornsweet2IFC cafc = new Cornsweet2IFC(0);
  cafc.setNBuffers(2);
  Thread exp = new Thread(cafc);
  exp.start();
}
```

```
public void run() {
  try {
    { // compute the actual alpha, and contrastCorn values
      //(as opposed to "desired"):
      BufferedImage tmp;
      for (int i = 0; i < contrastCorn.length; i++) {
        tmp = prepAsymCorn(1, 0, contrastCorn[i]);
        System.err.println("Requested contrastCorn = " + contrastCorn[i]);
        int[] px = new int[tmp.getWidth()];
        tmp.getRaster().getPixels(0, 0, tmp.getWidth() - 1, 1, px);
        contrastCorn[i] = (lut8.pix2Lum(px[tmp.getWidth() / 2 - 1]) - lut8
            .pix2Lum(px[tmp.getWidth() / 2]))
            / (lut8.pix2Lum(px[tmp.getWidth() / 2 - 1]) + lut8.pix2Lum(px[tmp
                .getWidth() / 2])) * 2;
        System.err.println("Received contrastCorn = " + contrastCorn[i]);
      }
      for (int i = 0; i < alphaCorn.length; i++) {
        tmp = prepAsymCorn(1, alphaCorn[i], 0);
        System.err.println("Requested alphaCorn = " + alphaCorn[i]);
        int[] px = new int[tmp.getWidth()];
        tmp.getRaster().getPixels(0, 0, tmp.getWidth() - 1, 1, px);
        alphaCorn[i] = (lut8.pix2Lum(px[1]) - lut8
            .pix2Lum(px[tmp.getWidth() - 2])) / 2;
        System.err.println("Received alphaCorn = " + alphaCorn[i]);
      }
    }
    // trial initialization: corn holds the
    // (KEY=)trial number - (VALUE=)cornsweet parameters (contrast,alpha)
    HashMap<Integer, ArrayList<Double>> corn =
      new HashMap<Integer, ArrayList<Double>>();
    // real holds the (KEY=)trial number - (VALUE=)real contrast
    HashMap<Integer, Double> real = new HashMap<Integer, Double>();
    // nTrial holds the entire trial number in a List
    List<Integer> nTrial = new LinkedList<Integer>();
    int t = 0;
    for (int j = 0; j < contrastCorn.length; j++) {
      for (int k = 0; k < alphaCorn.length; k++) {
        ArrayList<Double> tmp = new ArrayList<Double>();
        tmp.add(contrastCorn[j]);
        tmp.add(alphaCorn[k]);
        corn.put(t, tmp);
        real.put(t, initContrastReal[k][j]);
        nTrial.add(t);
        t++;
      }
    }
    hideCursor();
    blankScreen();
```

```
displayText(10, 50, "Indicate the interval with");
displayText(10, 100, "a larger difference between flanks");
displayText(10, 150, "Press 1 for first, 3 for second");
displayText(10, 250, "Now press anykey to start");
updateScreen();
getKeyPressed(-1);
blankScreen();

PrintWriter output = new PrintWriter(new File("data.txt"));

BufferedImage[] stimulus = new BufferedImage[2];
for (int i = 0; i < NTRIALS; i++) {
  // initialize the trial:
  Collections.shuffle(nTrial);
  Iterator<Integer> itTrial = nTrial.listIterator();
  while (itTrial.hasNext()) {
    int it = itTrial.next();
    // extract the corresponding cornsweet parameters for this trial
    ArrayList<Double> tmp = corn.get(it);
    double contrast = tmp.get(0);
    double alpha = tmp.get(1);
    // extract the corresponding real contrast for this trial
    double contrastReal = real.get(it);
    // order = -1 : first cornsweet; second real
    int order = 1 - 2 * (int) round(random());
    // polarity = 1: Left is +ive ramp,
    // polarity = -1 : Right is +ive
    int polarity = 1 - 2 * (int) round(random());
    stimulus[(-order + 1) / 2] = prepReal(polarity, contrastReal);
    // different polarity for the cornsweet
    polarity = 1 - 2 * (int) round(random());
    stimulus[(order + 1) / 2] = prepAsymCorn(polarity, alpha, contrast);
    // First Interval Stimulus
    displayImage(stimulus[0]);
    displayImage(frameX[0], frameY, frame);
    displayImage(frameX[1], frameY, frame);
    displayImage(fixX,fixY,fixation);
    updateScreen();
    Thread.sleep(INTERVAL1);
    // ISI
    blankScreen();
    updateScreen();
    Thread.sleep(ISI);
    // Second Interval Stimulus
    displayImage(stimulus[1]);
    displayImage(frameX[0], frameY, frame);
    displayImage(frameX[1], frameY, frame);
    displayImage(fixX,fixY,fixation);
```

```
      updateScreen();
      flushKeyTyped();
      Thread.sleep(INTERVAL2);
      blankScreen();
      displayImage(fixX,fixY,fixation);
      updateScreen();
      boolean cornIsBigger = false;
      String resp = getKeyTyped();
      while (true) {
        resp = getKeyTyped(-1);
        if ((resp.equals(resp1) && order == -1)
            || (resp.equals(resp2) && order == 1)) {
          cornIsBigger = true;
          real.put(it, min(MAXCONTRAST, contrastReal + deltaContrastReal));
          break;
        }
        else if ((resp.equals(resp1) && order == 1)
            || (resp.equals(resp2) && order == -1)) {
          cornIsBigger = false;
          real.put(it, max(0.0, contrastReal - deltaContrastReal));
          break;
        }
        else
          continue;
      }
      // check the actual displayed contrastReal and report the result
      int tmp2 = (-order + 1) / 2;
      int[] px = new int[stimulus[tmp2].getWidth()];
      stimulus[tmp2].getRaster().getPixels(0, 0,
          stimulus[tmp2].getWidth() - 1, 1, px);
      double contrastRealActual = abs(lut8.pix2Lum(px[0])
          - lut8.pix2Lum(px[px.length - 2]))
          / (lut8.pix2Lum(px[0]) + lut8.pix2Lum(px[px.length - 2])) * 2;
      output.printf("%f %f %f %f %s \n", exponent, contrast, alpha,
          contrastRealActual, cornIsBigger);
      output.flush();
      // extra sleep between trials - helps observer to refocus
      Thread.sleep(2*ISI);
    }
  }
  output.close();
} catch (InterruptedException e) {
  e.printStackTrace();
} catch (FileNotFoundException e) {
  e.printStackTrace();
}
finally {
  closeScreen();
```

```
  }
}
public BufferedImage prepReal(int polarity, double contrast) {
  // Luminances of Uniform flanks
  double meanLeft = meanLum * (1 + contrast / 2 * polarity);
  double meanRight = meanLum * (1 - contrast / 2 * polarity);
  int[] gStim = new int[stimWidth * stimHeight];
  // Convert the Luminance to Pixel using the LUT
  for (int j = 0; j < stimWidth / 2; j++)
    gStim[j] = lut8.lum2Pix(meanLeft);
  for (int j = stimWidth / 2; j < stimWidth; j++)
    gStim[j] = lut8.lum2Pix(meanRight);
  for (int i = 1; i < stimHeight; i++)
    for (int j = 0; j < stimWidth; j++)
      gStim[i * stimWidth + j] = gStim[j];
  // write the result in an image
  BufferedImage stim = new BufferedImage(stimWidth, stimHeight,
      BufferedImage.TYPE_BYTE_GRAY);
  stim.getRaster().setPixels(0, 0, stimWidth, stimHeight, gStim);
  return stim;
}
public BufferedImage prepAsymCorn(int polarity, double alpha,
    double contrast) {
  // Luminances of Uniform part of flanks
  double meanLeft = meanLum + alpha * polarity;
  double meanRight = meanLum - alpha * polarity;
  // Convert the Luminance to Pixel using the LUT
  int[] gStim = new int[stimWidth * stimHeight];
  for (int j = 0; j < stimWidth / 2 - rampWidth; j++)
    gStim[j] = lut8.lum2Pix(meanLeft);
  for (int j = stimWidth / 2 - rampWidth; j < stimWidth; j++)
    gStim[j] = lut8.lum2Pix(meanRight);
  // Luminances of Cornsweet2IFC ramps
  double delta = contrast / 2 * meanLum;
  for (int j = 0; j < rampWidth; j++) {
    // Left ramp
    int k = j + stimWidth / 2 - rampWidth;
    double tmp = pow((double) j / (double) (rampWidth - 1), exponent)
        * (-alpha + delta) * polarity;
    gStim[k] = lut8.lum2Pix(meanLeft + tmp);
    // Right ramp
    k = stimWidth / 2 + rampWidth - j - 1;
    gStim[k] = lut8.lum2Pix(meanRight - tmp);
  }
  for (int i = 1; i < stimHeight; i++)
    for (int j = 0; j < stimWidth; j++)
      gStim[i * stimWidth + j] = gStim[j];
  // write the result in an image
```

```
    BufferedImage stim = new BufferedImage(stimWidth, stimHeight,
        BufferedImage.TYPE_BYTE_GRAY);
    stim.getRaster().setPixels(0, 0, stimWidth, stimHeight, gStim);
    return stim;
  }
}
```

8.5 Look up operation in Standard Java Libraries

Java Standard library contains classes to perform look up operations. Here I will show how to perform a look up operation using those classes. The example is the same as the one above in which I showed how to filter an image using CLUT8. You first create a ByteLookupTable or a ShortLookupTable by providing an array which tells the relation between displayed pixel values and the image pixel values, i.e. an inverse look up table. Then you construct a LookupOp with this LookupTable. Finally you invoke the filter() method of the LookupOp class to apply the look up operation on your image. Here is the code

```
/*
 * chapter 8: JavaCoreLUTTest.java
 *
 * Demonstrates how to use tools of Standard Java library for look up operation
 */
import java.awt.image.*;
import java.io.*;
import javax.imageio.ImageIO;
public class JavaCoreLUTTest {

  public static void main(String[] args){

    FullScreen fs = new FullScreen();

    try {
      BufferedImage bi = ImageIO.read(new File("fechner.png"));

      fs.displayImage((fs.getWidth()/2-bi.getWidth())/2,
          (fs.getHeight()-bi.getHeight())/2, bi);

      byte[] table = new byte[256];
      for(int i=0; i<256; i++)
        table[i]=(byte)(255-i);

      LookupTable lTable = new ByteLookupTable(0, table);
      BufferedImageOp op = new LookupOp(lTable,null);
      op.filter(bi,bi);

      fs.displayImage((fs.getWidth()+bi.getWidth())/2,
          (fs.getHeight()-bi.getHeight())/2, bi);
```

```
        fs.displayText(15,fs.getHeight()-15,"(press any key to finish)");

        fs.updateScreen();
        fs.getKeyPressed(-1);

      } catch (IOException e) {
        e.printStackTrace();
      } finally {
        fs.closeScreen();
      }
    }
  }
```

8.6 Summary

If you need a more accurate inverse look up operation, you need to implement it yourself as I showed above. But if you are only concerned with filtering images of standard format you should better use the tools provided by the standard Java library. Whatever route you take, though, you must first prepare a look up table by carefully measuring the luminance output of your monitor.

8.6.1 Using CLUT8

- Construct a CLUT8 object using one of the two constructors
- etc.... coming more soon

8.6.2 Using Standard Java

- Construct a LookupTable object
- etc.... coming more soon

10 Managing the Display

In this chapter

- Managing multiple display systems, including stereo
- Getting and modifying screen characteristics in FSEM

10.1 Multiple Displays

Each display connected your computer is represented by a GraphicsDevice object. getScreenDevices() method of GraphicsEnvironment class returns a list of all available displays

```
private static GraphicsEnvironment gEnvironment =
    GraphicsEnvironment.getLocalGraphicsEnvironment();
private static GraphicsDevice[] gDevices =
    gEnvironment.getScreenDevices();
```

having a list of available graphics devices allows us to implement a new constructor, which takes an integer argument representing the display id

```
public FullScreen2(int displayID) {
  super(gDevices[displayID].getDefaultConfiguration());
  //...
}
```

now an application can construct a FullScreen2 object on a particular display by invoking this constructor

```
int displayID = 1;
FullScreen2 fs = new FullScreen2(displayID);
```

If displayID+1 is larger than the number of available displays the constructor throws an ArrayIndexOutOf-Bound exception and terminates.

With multiple displays, the JFrame is not automatically palced on the display which switches to FSEM. It does have the correct dimensions but by default it is placed at (0,0), which is the upper left cornert of the default display device. To correctly place it on the display which swithed to FSEM we first get the coordinates of that display

```
gDevice = gDevices[displayID];
gConfiguration = gDevice.getDefaultConfiguration();
Rectangle gcBounds = gConfiguration.getBounds();
```

```
int xoffs = gcBounds.x;
int yoffs = gcBounds.y;
int width = gcBounds.width;
int height = gcBounds.height;
```

then use those coordinates to place the JFrame in correct position

```
setBounds(xoffs, yoffs, width, height);
```

This line must come after gDevice.setFullScreenWindow(this), that is after swithing to FSEM.

There should still be a default constructor (Java insists on the default constructor - the constructor with no arguments - if your class has a non-default constructor). It is best that the default constructor uses the first available screen, displayID = 0, this way all the former examples of the book will still work with this new version of FullScreen class

```
public FullScreen2() {
  this(0);
}
```

Note for MacOS X: By default FSEM on a multi-display OS X system captures all displays, i.e. all displays turn blank once the FSEM is initiated. You can adjust this behavior and tell Java Virtual Machine not to capture all displays. You can add the following line in your code

```
System.setProperty("apple.awt.fullscreencapturealldisplays”," false”);
```

for instance in your main() method, or you can run your program with the following switch

```
java -Dapple.awt.fullscreencapturealldisplays=false MyProgram
```

10.2 Screen characteristics

The simple example HelloPsychophysicist of Chapter 2 used default screen characteristics. But in FSEM you could do more, you could, for example change the screen resolution. In this section I will show how to safely change the characteristics of the secreen.

Why would you change the screen parameters? One reason is performance: rendering and presenting a 512 by 512 image on a 1280x960 screen is probably faster than rendering and presenting a 1024x1024 image on a 1920x1200 screen. Moreover, monitors usually have higher temporal resolution (refresh rate) at lower spatial resolutions. Another reason might be getting the desired visual angle in an experiment in situations where you cannot adjust the distance between the observer and the computer screen. Having said that, it is doubtful whether or not letting your experimental program adjust the screen characteristics is a very bright idea. A word of caution deserves space here: better use your OS's dedicated applications to adjust the screen characteristics on the computers that you run your experiments. Better yet, do it permanantly so that there is little room for making errors between different sessions of the same experiment.

You can set or get the width, height, bitDepth and refreshRate of our display through a DisplayMode object. However, before attempting to change the DisplayMode, you should first check whether or not FSEM and DisplayMode change is available by using the GraphicsDevice's isFullScreenSupported() and isDisplayChangeSupported() methods. They return true if the qurried functionality is supported, false if not. If FSEM or DisplayMode change is not available you can't do much other than using the current native mode. If FSEM and DisplayMode change is avialable there are probably more than one available DisplayModes,

you can get a list of available DisplayModes by invoking the getDisplayModes() method of the GraphicsDevice class. But this wouldn't be a very useful piece of information to inspect, because you can't print it out. Here is a more eye-friendly way to see the available display modes

```
public String[] reportDisplayModes() {
  DisplayMode[] dms = getDisplayModes();
  String[] message = new String[dms.length];
  for (int i = 0; i < dms.length; i++) {
    StringBuilder m = new StringBuilder("("
        + dms[i].getWidth() + ","
        + dms[i].getHeight() + ","
        + dms[i].getBitDepth() + ","
        + dms[i].getRefreshRate() + ") ");
    message[i] = m.toString();
  }
  return message;
}
```

this method returns all the available display modes in a String array, wich can easily be printed. The getBitRate() returns BIT_DEPTH_MULTI if this mode supports multiple bit depths, and getRefreshRate() returns REFRESH_RATE_UNKNOWN if refreshRate is unknown or unconfigurable.

Tip: Why using StringBuilder, why not String? A String object is *immutable*, which means that we cannot modify a String object once it is created, for example we cannot append any more characters to an existing String. It is designed in this way for better performance. But a StringBuilder can be modified after it is constructed.

To change the DisplayMode, java API provides GraphicsDevice.setDisplayMode() method. However this method is known to lack robustness. I will instead use the following more robust method to change the DisplayMode

```
  public void setDisplayMode(DisplayMode dm) {
    if(displayMode.equals(dm))
      return;
    else if (!gDevice.isDisplayChangeSupported() || dm == null) {
      System.err.println("Exception in FullScreen.setDisplayMode(): "
          + "Display Change not Supported or DisplayMode is null");
      closeScreen();
      System.exit(0);
    }
    else if (!isDisplayModeAvailable(dm)) {
      System.err.println("Exception in FullScreen.setDisplayMode(): "
          + "DisplayMode not available");
      System.err.println(" ");
      System.err.println("Supported DisplayModes are:");
      String[] dms = reportDisplayModes();
      for (int i = 0; i < dms.length; i++) {
        System.err.print(dms[i]);
        if ((i + 1) % 4 == 0)
          System.err.println();
```

```
      }
      closeScreen();
      System.exit(0);
    }
    else {
      try {
        gDevice.setDisplayMode(dm);
        displayMode = dm;
        setBounds(0, 0, dm.getWidth(), dm.getHeight());
        Thread.sleep(200);
      } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
      } catch (RuntimeException e) {
        System.err.println("Exception in FullScreen.setDisplayMode(): "
            + "DisplayMode not available or " + "Display Change not Supported");
        System.err.println("");
        System.err.println("Supported DisplayModes are:");
        String[] dms = reportDisplayModes();
        for (int i = 0; i < dms.length; i++) {
          System.err.println(dms[i]);
          if ((i + 1) % 4 == 0)
            System.err.println();
        }
        closeScreen();
        System.exit(0);
      }
    }
  }
```

The method first compares the current DisplayMode to the requested one, if they are equal it returns. Next the method checks whether or not display mode change is available at all. If it isn't available it terminates the program. I chose to terminate the program in case the method fails to change to the requested DisplayMode, because you wouldn't want to present the stimulus in a wrong resolution by mistake in an actual experiment. After that we check whether the requested DisplayMode is among the available ones with a call to the following isDisplayModeAvailable() method

```
  private boolean isDisplayModeAvailable(DisplayMode dm) {

    DisplayMode[] mds = gDevice.getDisplayModes();
    for (int i = 0; i < mds.length; i++) {
      if (mds[i].getWidth() == dm.getWidth()
          && mds[i].getHeight() == dm.getHeight()
          && mds[i].getBitDepth() == dm.getBitDepth()
          && mds[i].getRefreshRate() == dm.getRefreshRate())
        return true;
    }
    return false;
  }
```

If the requested DisplayMode is not among the available ones, we warn the user, produce a list of available DisplayModes, then terminate. Finally we invoke the GraphicsDevice.setDisplayMode() method, and wait for one second, to let that method finish its job - much like in the createBufferStrategy() method we came across in Chapter 2. Nothe how I set the dimensions of FullScreen (JFrame) to fit exactly those of the new DisplayMode by invoking the setBounds() method of JFrame. The GraphicsDevice.setDisplayMode() method may throw a RuntimeException. We also catch this exception and terminate the program after printing out a list of available DisplayModes.

Finally I will include another getter to get the current DisplayMode

```
public DisplayMode getDisplayMode() {
  return displayMode;
}
```

and its acompanying reportDisplayMode() method

10.3 Stereo display systems

Coming soon...

Here is the entire FullScreen3 class

```
/*
 * chapter 9: FullScreen3.java
 *
 * Provides methods to manage (multiple) displays
 *
 */
import java.awt.*;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;
import java.awt.font.FontRenderContext;
import java.awt.geom.Rectangle2D;
import java.awt.image.BufferedImage;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;
import java.util.concurrent.TimeUnit;
import javax.swing.JFrame;
public class FullScreen3 extends JFrame implements KeyListener{
  private static final GraphicsEnvironment gEnvironment = GraphicsEnvironment
      .getLocalGraphicsEnvironment();
  private static final GraphicsDevice[] gDevices = gEnvironment
      .getScreenDevices();
  private GraphicsDevice gDevice;
  private GraphicsConfiguration gConfiguration;

  private int nBuffers = 1;
  private Color bgColor = Color.BLACK;
  private Color fgColor = Color.LIGHT_GRAY;
  private final Font defaultFont = new Font("SansSerif", Font.BOLD, 36);
```

```
private DisplayMode oldDisplayMode;
private DisplayMode displayMode;

private BlockingQueue<String> keyTyped;
private BlockingQueue<Long> whenKeyTyped;
private BlockingQueue<Integer> keyPressed;
private BlockingQueue<Long> whenKeyPressed;
private BlockingQueue<Integer> keyReleased;
private BlockingQueue<Long> whenKeyReleased;

public void keyTyped(KeyEvent ke) {
  keyTyped.offer(String.valueOf(ke.getKeyChar()));
  whenKeyTyped.offer(ke.getWhen());
}
public void keyReleased(KeyEvent ke) {

  keyReleased.offer(ke.getKeyCode());
  whenKeyReleased.offer(ke.getWhen());
}

public void keyPressed(KeyEvent ke) {

  if (ke.getKeyCode() == KeyEvent.VK_ESCAPE) {
    closeScreen();
    System.exit(0);
  }
  else {
    keyPressed.offer(ke.getKeyCode());
    whenKeyPressed.offer(ke.getWhen());
  }
}

public FullScreen3() {
  this(0);
}

public FullScreen3(int displayID) {
  super(gDevices[displayID].getDefaultConfiguration());
  gDevice = gDevices[displayID];
  gConfiguration = gDevice.getDefaultConfiguration();
  oldDisplayMode = gDevice.getDisplayMode();
  displayMode = gDevice.getDisplayMode();
  try {
    setUndecorated(true);
    setIgnoreRepaint(true);
    setResizable(false);
    setFont(defaultFont);
```

```
    setBackground(bgColor);
    setForeground(fgColor);
    gDevice.setFullScreenWindow(this);

    Rectangle gcBounds = gConfiguration.getBounds();
    int xoffs = gcBounds.x;
    int yoffs = gcBounds.y;
    int width = gcBounds.width;
    int height = gcBounds.height;
    setBounds(xoffs, yoffs, width, height);

    keyTyped = new LinkedBlockingQueue<String>();
    whenKeyTyped = new LinkedBlockingQueue<Long>();
    keyPressed = new LinkedBlockingQueue<Integer>();
    whenKeyPressed = new LinkedBlockingQueue<Long>();
    keyReleased = new LinkedBlockingQueue<Integer>();
    whenKeyReleased = new LinkedBlockingQueue<Long>();
    addKeyListener(this);

    setNBuffers(nBuffers);
  }
  finally {}
}
public void setNBuffers(int n) {
  try {
    createBufferStrategy(n);
    nBuffers = n;
  } catch (IllegalArgumentException e) {
    System.err.println(
        "Exception in FullScreen.setNBuffers(): "
        +"requested number of Buffers is illegal - falling back to default");
    createBufferStrategy(nBuffers);
  }
  try{
    Thread.sleep(200);
  } catch (InterruptedException e) {
    Thread.currentThread().interrupt();
  }
}
public int getNBuffers() {
  return nBuffers;
}
public void updateScreen() {

  if (getBufferStrategy().contentsLost())
    setNBuffers(nBuffers);
  getBufferStrategy().show();
}
```

```
public void displayImage(BufferedImage bi) {
  if( bi!=null){
    double x = (getWidth() - bi.getWidth()) / 2;
    double y = (getHeight() - bi.getHeight()) / 2;
    displayImage((int) x, (int) y, bi);
  }
}
public void displayImage(int x, int y, BufferedImage bi) {
  Graphics2D g =  (Graphics2D)getBufferStrategy().getDrawGraphics();
  try {
    if(g!=null && bi!=null)
      g.drawImage(bi, x, y, null);
  }
  finally {
    g.dispose();
  }
}
public void displayText(String text) {
  Graphics2D g = (Graphics2D)getBufferStrategy().getDrawGraphics();
  if( g!=null && text!= null){
    Font font = getFont();
    g.setFont(font);
    FontRenderContext context = g.getFontRenderContext();
    Rectangle2D bounds = font.getStringBounds(text, context);
    double x = (getWidth() - bounds.getWidth()) / 2;
    double y = (getHeight() - bounds.getHeight()) / 2;
    double ascent = -bounds.getY();
    double baseY = y + ascent;
    displayText((int) x, (int) baseY, text);
  }
  g.dispose();
}
public void displayText(int x, int y, String text) {
  Graphics2D g =  (Graphics2D)getBufferStrategy().getDrawGraphics();
  try {
    if(g!=null && text!=null){
      g.setFont(getFont());
      g.setColor(getForeground());
      g.drawString(text, x, y);
    }
  }
  finally {
    g.dispose();
  }
}
public void blankScreen() {
  Graphics2D g = (Graphics2D)getBufferStrategy().getDrawGraphics();
```

```
  try {
    if(g!=null){
      g.setColor(getBackground());
      g.fillRect(0, 0, getWidth(), getHeight());
    }
  }
  finally {
    g.dispose();
  }
}

public Color getBackground(){

  return bgColor;
}

public void setBackground(Color bg){

  bgColor = bg;
}

public void hideCursor() {
  Cursor noCursor = null;
  Toolkit tk = Toolkit.getDefaultToolkit();
  Dimension d = tk.getBestCursorSize(1,1);
  if((d.width|d.height)!=0)
    noCursor = tk.createCustomCursor(
        gConfiguration.createCompatibleImage(d.width, d.height),
        new Point(0, 0), "noCursor");
  setCursor(noCursor);
}

public void showCursor() {
  setCursor(new Cursor(Cursor.DEFAULT_CURSOR));
}

public void closeScreen() {
  if (gDevice.isDisplayChangeSupported())
    setDisplayMode(oldDisplayMode);
  gDevice.setFullScreenWindow(null);
  dispose();
}

public String getKeyTyped(long ms){

  String c = null;
  try {
    if(ms < 0)
```

```
      c = keyTyped.take();
    else
      c = keyTyped.poll(ms, TimeUnit.MILLISECONDS);
  } catch (InterruptedException e) {
    Thread.currentThread().interrupt();
    e.printStackTrace();
  }
  return c;
}

public String getKeyTyped(){

  return keyTyped.poll();
}

public Long getWhenKeyTyped(){

  return whenKeyTyped.poll();
}

public void flushKeyTyped(){

  keyTyped.clear();
  whenKeyTyped.clear();
}

public Integer getKeyPressed(long ms){

  Integer c = null;
  try {
    if(ms < 0)
      c = keyPressed.take();
    else
      c = keyPressed.poll(ms, TimeUnit.MILLISECONDS);
  } catch (InterruptedException e) {
    Thread.currentThread().interrupt();
    e.printStackTrace();
  }
  return c;
}

public Integer getKeyPressed(){

  return keyPressed.poll();
}

public Long getWhenKeyPressed(){
```

```
  return whenKeyPressed.poll();
}

public void flushKeyPressed(){

  keyPressed.clear();
  whenKeyPressed.clear();
}

public Integer getKeyReleased(long ms){

  Integer c = null;
  try {
    if(ms < 0 )
      c = keyReleased.take();
    else
      c = keyReleased.poll(ms, TimeUnit.MILLISECONDS);
  } catch (InterruptedException e) {
    Thread.currentThread().interrupt();
    e.printStackTrace();
  }
  return c;
}

public Integer getKeyReleased(){

  return keyReleased.poll();
}

public Long getWhenKeyReleased(){

  return whenKeyReleased.poll();
}

public void flushKeyReleased(){

  keyReleased.clear();
  whenKeyReleased.clear();
}
public boolean isFullScreenSupported() {
  return gDevice.isFullScreenSupported();
}
public void setDisplayMode(DisplayMode dm) {
  if(displayMode.equals(dm))
    return;
  else if (!gDevice.isDisplayChangeSupported() || dm == null) {
    System.err.println("Exception in FullScreen.setDisplayMode(): "
        + "Display Change not Supported or DisplayMode is null");
```

```
      closeScreen();
      System.exit(0);
    }
    else if (!isDisplayModeAvailable(dm)) {
      System.err.println("Exception in FullScreen.setDisplayMode(): "
          + "DisplayMode not available");
      System.err.println(" ");
      System.err.println("Supported DisplayModes are:");
      String[] dms = reportDisplayModes();
      for (int i = 0; i < dms.length; i++) {
        System.err.print(dms[i]);
        if ((i + 1) % 4 == 0)
          System.err.println();
      }
      closeScreen();
      System.exit(0);
    }
    else {
      try {
        gDevice.setDisplayMode(dm);
        displayMode = dm;
        setBounds(0, 0, dm.getWidth(), dm.getHeight());
        Thread.sleep(200);
      } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
      } catch (RuntimeException e) {
        System.err.println("Exception in FullScreen.setDisplayMode(): "
            + "DisplayMode not available or " + "Display Change not Supported");
        System.err.println("");
        System.err.println("Supported DisplayModes are:");
        String[] dms = reportDisplayModes();
        for (int i = 0; i < dms.length; i++) {
          System.err.println(dms[i]);
          if ((i + 1) % 4 == 0)
            System.err.println();
        }
        closeScreen();
        System.exit(0);
      }
    }
  }
  private boolean isDisplayModeAvailable(DisplayMode dm) {
    DisplayMode[] mds = gDevice.getDisplayModes();
    for (int i = 0; i < mds.length; i++) {
      if (mds[i].getWidth() == dm.getWidth()
          && mds[i].getHeight() == dm.getHeight()
          && mds[i].getBitDepth() == dm.getBitDepth()
          && mds[i].getRefreshRate() == dm.getRefreshRate())
```

```
        return true;
      }
      return false;
    }
    public String reportDisplayMode() {
      StringBuilder message = new StringBuilder();
      if (displayMode.getBitDepth() == DisplayMode.BIT_DEPTH_MULTI)
        message.append("  Bit Depth = -1 (MULTIPLE) \n");
      else
        message.append("  Bit Depth = " + displayMode.getBitDepth() + "\n");
      message.append("  Width = " + displayMode.getWidth() + "\n");
      message.append("  Height = " + displayMode.getHeight() + "\n");
      if (displayMode.getRefreshRate() == DisplayMode.REFRESH_RATE_UNKNOWN)
        message.append("  Refresh Rate = 0 (unknown/unmodifiable) \n");
      else
        message.append("  Refresh Rate = " + displayMode.getRefreshRate() + "\n");
      return message.toString();
    }
    public DisplayMode getDisplayMode() {
      return displayMode;
    }
    public boolean isDisplayChangeSupported() {
      return gDevice.isDisplayChangeSupported();
    }
    public String[] reportDisplayModes() {
      DisplayMode[] dms = getDisplayModes();
      String[] message = new String[dms.length];
      for (int i = 0; i < dms.length; i++) {
        StringBuilder m = new StringBuilder("(" + dms[i].getWidth() + ","
            + dms[i].getHeight() + "," + dms[i].getBitDepth() + ","
            + dms[i].getRefreshRate() + ") ");
        message[i] = m.toString();
      }
      return message;
    }
    public DisplayMode[] getDisplayModes() {
      return gDevice.getDisplayModes();
    }
  }
```

10.4 Examples

Here is a sample test program, which allows user set the DisplayMode safely

```
  /*
   * chapter 3: DisplayTest.java
   *
```

```
 * Tests FSEM and DisplayMode change support, if supported lets user set the
 * DisplayMode
 *
 */
import java.awt.DisplayMode;
import java.awt.GraphicsEnvironment;
import java.util.StringTokenizer;
import javax.swing.JOptionPane;

public class DisplayTest {

  public static int dialog(String message, String[] options) {
    int selectedValue = JOptionPane.showOptionDialog(null, message,
        "Information", JOptionPane.DEFAULT_OPTION,
        JOptionPane.INFORMATION_MESSAGE, null, options, options[0]);
    return selectedValue;
  }

  public static void dialog(String message) {

    JOptionPane.showMessageDialog(null, message);
  }

  public static String inputDialog(String message, String[] options) {

    String s = (String) JOptionPane.showInputDialog(null, message,
        "Query Dialog", JOptionPane.PLAIN_MESSAGE, null, options, options[0]);
    return s;
  }

  public static void main(String[] args) {

    if (dialog("Now we will start diagnosing the Display(s) \n"
        + "Are you ready?", new String[] { "YES", "NO/EXIT" }) == 1)
      System.exit(0);
    int numOfDisplays = GraphicsEnvironment.getLocalGraphicsEnvironment()
        .getScreenDevices().length;
    dialog("Found " + numOfDisplays + " display(s)");
    FullScreen2[] fs = new FullScreen2[numOfDisplays];
    for (int j=0; j< numOfDisplays; j++) {
      fs[j] = new FullScreen2(j);
      try {
        fs[j].displayText("Display: " + j);
        Thread.sleep(2000);
        fs[j].blankScreen();
        fs[j].displayText("Bounds: " + fs[j].getBounds().x + " "
            + fs[j].getBounds().y + " " + fs[j].getBounds().width + " "
            + fs[j].getBounds().height );
```

```
  Thread.sleep(2000);
  boolean supported = fs[j].isDisplayChangeSupported();
  fs[j].closeScreen();
  if (!fs[j].isFullScreenSupported())
    dialog("Full Screen Exclusive Mode is not supported for Display "
        + j + "\n"
        + "Java uses alternative methods to imitate Full Screen Mode");
  else
    dialog("Full Screen Exclusive Mode is supported for Display " + j);
  dialog("Current DisplayMode is: \n" + fs[j].reportDisplayMode());
  if (supported) {
    String[] dms = fs[j].reportDisplayModes();
    StringBuffer message = new StringBuffer();
    for (int i = 0; i < dms.length; i++) {
      message.append(dms[i]);
      if ((i + 1) % 4 == 0)
        message.append("\n");
    }
    dialog("DisplayMode change is supported for Display " + j +"\n"
        + "Available Modes are:\n" + message);
    String srep = inputDialog("Choose DisplayMode or click on Cancel",
        dms);
    if (srep != null) {
      StringTokenizer rep = new StringTokenizer(srep, "(,)");
      int[] dp = new int[4];
      for (int k = 0; k < 4; k++)
        dp[k] = Integer.parseInt(rep.nextToken());
      DisplayMode dm = new DisplayMode(dp[0], dp[1], dp[2], dp[3]);
      fs[j] = new FullScreen2();
      fs[j].setDisplayMode(dm);
      fs[j].blankScreen();
      fs[j].displayText("Display: " + j);
      fs[j].updateScreen();
      Thread.sleep(2000);
      fs[j].blankScreen();
      fs[j].displayText("Bounds: " + fs[j].getBounds().x + " "
          + fs[j].getBounds().y + " " + fs[j].getBounds().width + " "
          + fs[j].getBounds().height );
      fs[j].updateScreen();
      Thread.sleep(2000);
    }
  }
  else
    dialog("DisplayMode change is not supported for Display " + j);
} catch (InterruptedException e) {}
finally {
  fs[j].closeScreen();
}
```

```
        }
      }
    }
```

In this program I use some interesting features, which allow interaction with the user. They are implemented in dialog() and inputDialog() methods. Java's Swing API provides simple tools to interact with users through the JOPtionPane class. (See "How to make dialogs" chapter in "Trail: Creating a GUI with JFC/Swing" at `http://java.sun.com/docs/books/tutorial/uiswing/components/dialog.html`.)

We first determine whether FSEM and DisplayMode change are supported by invoking isFullScreenSupported() and isDisplayChangeSupported() methods. If DisplayMode change is supported we show a list of available modes in a dialog window. Then user can choose one of the available modes and the screen switches to that mode with a call to the setDisplayMode() method.

Another useful object in this example is StringTokenizer(String text, String delim) method. It breaks the given String text into smaller tokens stopping at any one of the given delimeters. Here our delimeters are "(", ")" and ",". This way we first get user's preferred DisplayMode as a String and by stripping it out of the delimeters we construct a new DisplayMode object.

10.5 Summary

- If you have a multiple display system you can choose the one to use to display your stimulus in full screen exclusive mode. Use the constructor FullScreen(int displayID) to choose the screen to display your stimulus. (Use DisplayTest.java to figure out display id's in a multi-display system.)
- For display characteristics use
  - isFullScreenSupported() method to learn whether Full Screen Exclusive Mode (FSEM) is supported,
  - reportDisplayMode() (or getDisplayMode()) method to learn your current display mode (width, height, bit depth, refresh rate),
  - isDisplayChangeSupported() method to learn whether you can change your display's characteristics,
  - reportDisplayModes() (or getDisplayModes()) method to see all available DisplayModes.
- Use setDisplayMode() method to change the DisplayMode in FSEM.

11 Applets, normal window applications, packaging and sharing your work

In this chapter

- Converting Full Screen experiments into normal window applications,
- Packaging and sharing applications
  - packaging in a self running jar archive
  - deploying as Java applets
  - deploying as Java Web Start application

Upto now all our examples were in Full Screen Exclusive mode. This, of course, was the appropriate choice for a psychophysics experiment, nevertheless one may need to test or implement a version of the actual psychophysics experiment in a normal window. This is particularly useful to convert the experiment into an applet and place in a web page. The first objective of this chapter is to show how to create normal window applications using core Java tools. I will do this by creating a new class called NormalWindow. With this new class, converting the FullScreen applications into normal window applications will be just a matter of changing a few lines in the experimental code.

The other objective of the current chapter is to introduce ways to share your work with colleagues and others. (Java platform is particularly attractive because it makes sharing possible. Ability to share the work enhances communication, boosts the productivity.) One mean of sharing or demonstrating your work is turning the application into a Java Applet and place it on a web page. This way visitors of your web page could run an Applet version of your experiments without installing any programs other then the Java plug-ins. Another method to share your work is to pack it in a self running, so called *jar* file. Depending on the OS and your configuration, those files can run by just a mouse click. Last, I will show how to use Java Web Start technology to share your work.

I will start describing the implementation of the NormalWindow class, after that I will show how to convert the HelloPsychophysicist example from Chapter 2 into a normal window application and into Java applets.

11.1 NormalWindow class

As we will see next, there is not so much difference between the FSEM application and a normal window application. Therefor I will present the NormalWindow class by highlighting its differences from the FullScreen class.

11.1.1 Constructor

First of all, NormalWindow extends JPanel instead of JFrame. *Because* .... The constructor of the NormalWindow class differs from the FullScreen class. We eliminate the method invocations related to FSEM: setUndecorated(true), setIgnoreRepaint(true), setResizable(false), gDevice.setFullScreenWindow(this). We also eliminate the method related to setting the BufferStrategy, setNBuffers(nBuffers) totally. Here is the constructor of NormalWindow

```
public class NormalWindow extends JPanel{
  public NormalWindow(){

    super();

    setFont(defaultFont);
    setForeground(fgColor);
    setBackground(bgColor);
    setFocusable(true);

    // ...
  }
```

11.1.2 storing the entire screen in a BufferedImage

We store the visible screen in a BufferedImage called screenImage. In displayImage(), displayText() and blankScreen() methods we manipulate the screenImage, in updateScreen we actually show it on the screen. Notice the similarity and distinction to the FSEM version. In FSEM version we were drawing the images or texts on the video buffer by using its Graphics2D contents. Here we use the Graphics2D contents of the screenImage. In FSEM, the show() method of BufferStrategy class was responsible of actually displaying whatever changes we had done so far, here we will manually update the screen using the screenImage.

11.1.3 Double Buffering and active/passive rendering in NormalWindow

As we eliminated the setNBuffers() method, we are now on our own to determine various double buffering strategies, and since we eliminated the setIgnoreRepaint(true) in the constructor, we should manually decide our active/passive rendering choices.

As I mentioned before, in active rendering you are responsible with all renderings, JVM does not intervene. This allows you to precisely adjust your stimulation. This suits well in a FSEM application. But in a Normal Window application there are good reasons why JVM should sometimes intervene: for instance, when you close your window, or your window is occluded by other applications, and later brought to front again you have to update the content. In passive rendering this is automatically done by the JVM. Active rendering was suitable for the FullScreen applications, because no other window could occlude our application window, or any other OS related menus or bars could actually appear. So we did not need JVM's intervention to repaint the screen. It was all up to us. But in a normal window mode such occlusions may happen. Therefore it is more suitable to use passive rendering in normal window mode. By default we will adopt the passive rendering strategy in NormalWindow class. We define a boolean parameter passiveRendering and set it to true

```
private boolean passiveRendering = true;
```

But if you decide that you don't want the JVM update the screen passively, you can set passiveRendering to false with setPassiveRendering() method

```
public void setPassiveRendering(boolean pr){

  passiveRendering = pr;
  setIgnoreRepaint(!passiveRendering);
}
```

This method invokes the setIgnoreRepaint method, thereby instructing the JVM to passively update or not the application's visible surface. isPassiveRendering() method returns whether the strategy is set to passive rendering or not

```
public boolean isPassiveRendering(){

  return passiveRendering;
}
```

How does passive rendering work? A JPanel has a method called paintComponent(), a class inheriting from JPanel should override that method and provide the rules to paint the stimulus in it. JVM uses this method to repaint the application window in case of passive rendering. However you never invoke paintComponent() directly. You invoke another method called repaint() and it indirectly takes care of invoking the paintComponent() method.

Here is the new updateScreen() method. If the passiveRendering variable is set to true, it invokes the repaint() method, if not it actively paints the screen

```
public void updateScreen(){

  if(screenImage == null)
    createScreenImage();

  if(passiveRendering)
    repaint();
  else {
    Graphics g = getGraphics();
    try {
      if(g!=null)
        g.drawImage(screenImage, 0,0, this);
      Toolkit.getDefaultToolkit().sync();
    }
    finally {
      g.dispose();
    }
  }
}
```

First we check wheter or not screenImage is null. Next, if we are in passive rendering mode we just invoke the repaint method. But if we are in active rendering we do the following: we first get the Graphics contents of the component, here NormalWindow, then we draw the screenImage on this component, and finally dispose the Graphics2D.

Next let's look at the paintComponent() method of NormalWindow. JVM invokes paintComponent() method under two conditions: One, if the application invokes repaint() method; Two, if the window needs to be updated for some reason, such as minimization/maximization. Normally, if you are using passive rendering strategy, what you have to place in your paintComponent() is invoking the JPanel's paintComponent() method with super.paintComponent(), then place your application specific lines. What if you are doing active rendering? Although we eliminated the repaint() invocation in updateScreen() above, JVM is still going to automatically invoke paintComponent() method if there is a need to re-draw the window. You have to make sure that JVM knows that you are in active rendering mode and it should not update the screen automatically. Here is our paintComponent() implementation:

```
protected void paintComponent(Graphics g){

  if(passiveRendering){
    super.paintComponent(g);
    if(screenImage != null)
      g.drawImage(screenImage,0,0,this);
  }
}
```

On the other hand, even though it is not FSEM, we still want to use double buffering. This is not too difficult, because we had already set up the FullScreen class in a way that makes it easy to accomplish this. Recall that we first draw the stimulus on an off-screen buffer and then display it on the screen after the rendering is completed, we will keep that same strategy here in NormalWindow:

```
public void displayImage(int x, int y, BufferedImage bi) {
   if(screenImage == null)
      createScreenImage();

    Graphics2D g = screenImage.createGraphics();
    try {
      if(g!=null && bi!=null)
        g.drawImage(bi, x, y, null);
      Toolkit.getDefaultToolkit().sync();
    }
    finally {
      g.dispose();
    }
  }
```

then, just as in the FSEM to actually display the stimulus the application should invoke the updateScreen() method.

Here is the complete NormalWindow.java code

```
/*
 * chapter 10: HPWindow.java
 *
 * displays the text "Hello Psychophysicist (Normal Window)" and two images
 * in a normal window
 *
```

```
 */
import javax.imageio.ImageIO;
import javax.swing.JFrame;
import java.awt.image.BufferedImage;
import java.io.IOException;
// HPWindow extends NormalWindow, instead of FullScreen
public class HPWindow extends NormalWindow implements Runnable {
  static JFrame mainFrame;

  public static void main(String[] args) {
    HPWindow nw = new HPWindow();

    // The only addition/change is here:

    //nw.setPassiveRendering(false);
    mainFrame = new JFrame();
    mainFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    mainFrame.add(nw);
    mainFrame.setBounds(100, 100, 612, 612);
    mainFrame.setResizable(false);
    mainFrame.setVisible(true);
    // up to here

    Thread hpnw = new Thread(nw);
    hpnw.start();
  }

  public void run(){
    try {
      displayText("Hello Psychophysicist (Normal Window)");
      updateScreen();
      Thread.sleep(2000);
      blankScreen();
      hideCursor();
      BufferedImage bi1 = ImageIO.read(
          HPWindow.class.getResource("psychophysik.png"));
      displayImage(bi1);
      updateScreen();
      Thread.sleep(2000);
      blankScreen();
      BufferedImage bi2 = ImageIO.read(
          HPWindow.class.getResource("fechner.png"));
      displayImage(bi2);
      updateScreen();
      Thread.sleep(2000);
    } catch (IOException e) {
      System.err.println("File not found");
      e.printStackTrace();
```

```
    } catch (InterruptedException e) {}
    finally {
      // and replace this
      //fs.closeScreen();
      mainFrame.dispose();
    }
  }
}
```

11.2 HelloPsychophysicist, normal window

Here is the HelloPsychophysicist.java example from Chapter 2, modified to act as a normal window application, the changes are marked with **bold font**

```
/*
 * chapter 10: HPWindow.java
 *
 * displays the text "Hello Psychophysicist (Normal Window)" and two images
 * in a normal window
 *
 */
import javax.imageio.ImageIO;
import javax.swing.JFrame;
import java.awt.image.BufferedImage;
import java.io.IOException;
// HPWindow extends NormalWindow, instead of FullScreen
public class HPWindow extends NormalWindow implements Runnable {
  static JFrame mainFrame;

  public static void main(String[] args) {
    HPWindow nw = new HPWindow();
    //nw.setPassiveRendering(false);

    // The only addition/change is from here:
    mainFrame = new JFrame();
    mainFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    mainFrame.add(nw);
    mainFrame.setBounds(100, 100, 612, 612);
    mainFrame.setResizable(false);
    mainFrame.setVisible(true);
    // up to here

    Thread hpnw = new Thread(nw);
    hpnw.start();
  }

  public void run(){
```

```
        try {
          displayText("Hello Psychophysicist (Normal Window)");
          updateScreen();
          Thread.sleep(2000);
          blankScreen();
          hideCursor();
          BufferedImage bi1 = ImageIO.read(
              HPWindow.class.getResource("psychophysik.png"));
          displayImage(bi1);
          updateScreen();
          Thread.sleep(2000);
          blankScreen();
          BufferedImage bi2 = ImageIO.read(
              HPWindow.class.getResource("fechner.png"));
          displayImage(bi2);
          updateScreen();
          Thread.sleep(2000);
        } catch (IOException e) {
          System.err.println("File not found");
          e.printStackTrace();
        } catch (InterruptedException e) {}
        finally {
          // and replace this
          //fs.closeScreen();
          mainFrame.dispose();
        }
      }
    }
```

When compiled and executed, this program opens a normal window and displays the text "Hello Psychophysicist (Normal Window)" and two other images. NormalWindow extends JPanel. But JPanels can't be displayed on the screen directly, it has to be put inside a, so called *heavier* component. Here I put it inside a JFrame:

```
        mainFrame = new JFrame();
        mainFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        mainFrame.add(nw);
        mainFrame.setBounds(100, 100, 612, 612);
        mainFrame.setResizable(false);
        mainFrame.setVisible(true);
```

setDefaultCloseOperation() method tells the JFrame to exit the program in case the user closes the window. In the next line we add the NormalWindow object nw to the JFrame. Then we set the position and size of the JFrame with setBounds(100, 100, 612, 612) method. It will be placed at (100,100) relative to upper left corner of the screen, with width and height equal to 612. We also invoke setResizable(false) so that the user can not resize the window. We make the JFrame visible by invoking setVisible(true) method. In the end, inside finally{}, instead of closeScreen() method of the FullScreen, we use the dispose() method of JFrame

```
        finally {
```

```
        // and replace this
        //fs.closeScreen();
        mainFrame.dispose();
      }
```

Another subtle difference is the way we load the images

```
BufferedImage bi1 = ImageIO.read(
          HPWindow.class.getResource("psychophysik.png"));
```

We have to this in order to get the self running jar files use the resources properly. In this case the resource file is an image, but this consideration extends to other types of files, as well. I will show how to use other types of resource files below in Section 11.7.

11.3 Java Applets

Applets don't have main() methods. Instead you should provide init() method for initialization of the applet. This method is invoked by the browser or appletviewer to inform the applet that it is loaded. The other relevant methods are start() and stop(). start() is called after the init() method and everytime the web page is revisited. Initializations, such as creation of new Threads can be done inside the init() method. start() and stop() can be used, for example, pause and resume a Threaded animation. I will provide two different versions of HelloPsychophysicist as Java applets below.

11.4 HelloPsychophysicist, as an applet

```
/*
 * chapter 10: HPApplet.java
 *
 * displays the text "Hello Psychophysicist (Applet)" and two images on an
 * otherwise entirely blank window
 *
 */
// <applet code="HPApplet.class" width=612 height=612>
// </applet>
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.image.BufferedImage;
import java.io.IOException;
import javax.imageio.ImageIO;
import javax.swing.JApplet;
import javax.swing.JButton;
import javax.swing.JPanel;
public class HPApplet extends JApplet {
  private AnimationPanel animationPanel;
  public void init() {
    animationPanel = new AnimationPanel();
    animationPanel.setPassiveRendering(true);
```

```
    animationPanel.initAnimation();
    add(animationPanel, "Center");

    final JButton playButton = new JButton("Play Again");
    JPanel buttonPanel = new JPanel();
    buttonPanel.add(playButton, "Center");
    add(buttonPanel, "South");
    setSize(612, 612);

    playButton.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        animationPanel.startAnimation();
      }
    });

  }
  public void start() {
    animationPanel.startAnimation();
  }

  public void stop() {

    animationPanel.stopAnimation();
  }

}
class AnimationPanel extends NormalWindow implements Runnable {
  private Thread animator;
  private boolean running = false;
  public void initAnimation() {
    if (animator == null || !animator.isAlive())
      animator = new Thread(this);
  }
  public void startAnimation() {
    if (!animator.isAlive())
      animator = new Thread(this);

    if (!running) {
      running = true;
      animator.start();
    }

  }

  public void stopAnimation(){

    running = false;
  }
```

```
  public void run() {
    try {
      blankScreen();
      displayText("Hello Psychophysicist (Applet)");
      updateScreen();
      Thread.sleep(2000);
      blankScreen();
      hideCursor();
      BufferedImage bi1 = ImageIO.read(HPApplet.class
          .getResource("psychophysik.png"));
      displayImage(bi1);
      updateScreen();
      Thread.sleep(2000);
      blankScreen();
      BufferedImage bi2 = ImageIO.read(HPApplet.class
          .getResource("fechner.png"));
      displayImage(bi2);
      updateScreen();
      Thread.sleep(2000);
    } catch (IOException e) {
      System.err.println("File not found");
      e.printStackTrace();
    } catch (InterruptedException e) {
      Thread.currentThread().interrupt();
    }
    finally {
      running = false;
    }
  }
}
```

Different from the FSEM and Normal Window versions, I included a button to replay the stimulus sequence

```
final JButton playButton = new JButton("Play Again");
```

For the JButton to be useful for anything, I add an action listener using the anonymous inner class technique (see Chapter XXX)

```
playButton.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    animationPanel.startAnimation();
  }
});
```

When the observer presses the “Play Again” button, the applet invokes the startAnimation() method of AnimationPanel class. We add the JButton first in a JPanel of itself and then add that JPanel to the Applet

```
JPanel buttonPanel = new JPanel();
```

```
buttonPanel.add(playButton, "Center");
add(buttonPanel, "South");
```

run() method of AnimationPanel class is the same as the former versions of HelloPsycophysicist. But this time I included 3 new methods initAnimation(), startAnimation(), stopAnimation(), to initialize, start and stop the Animation thread respectively. When HPApplet is first loaded by the browser (or by applet viewer) its init() method is invoked. Inside init() we initialize the AnimationPanel by invoking its initAnimation() method. Next, the start() method of the HPApplet is invoked. Note that you don't need to invoke this method, nor the init() and stop() methods, yourself. Their invocation is taken care of automatically. The start() method is invoked by the browser everytime you visit the web page holding the applet. In the start() method we invoke the startAnimation() method of the AnimationPanel to start the animation. So the animation starts as soon as the browser loads the page. stop() method of the HPApplet is invoked when you leave that page. In that method we invoke the stopAnimation() method of the AnimationPanel.

11.5 HelloPsychophysicist, a normal window as a pop-up in applet

Here I will show another way to deploy normal window applications. This is a combination of an Applet and a Normal Window. The Applet part displays a "Press to set in/visible" button and allows the user to set the normal window visible or invisible. The Normal Window part is the same as the AnimationPanel we used in Section 11.4, in fact we use it without any change. We don't need to re-write it in our code because it is in the same directory and accessible by the other java programs.

```
/*
 * chapter 10: HPApplet2.java
 *
 * By pressing a JButton
 * opens a JFrame and displays the text "Hello Psychophysicist (Applet)"
 * and two images on an otherwise entirely blank screen
 *
 */
// <applet code="HPApplet2.class" width=250 height=50>
// </applet>
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JApplet;
import javax.swing.JButton;
import javax.swing.JFrame;
public class HPApplet2 extends JApplet{
  public void init() {
    final AnimationPanel animationPanel = new AnimationPanel();
    animationPanel.initAnimation();

    final JFrame fs = new JFrame();
    fs.setTitle("Hello Psychophysicist");
    fs.setSize(612,612);
    fs.add(animationPanel);

    JButton playButton = new JButton("Press to set in/visible");
```

```
    add(playButton);
    setSize(250,50);

    playButton.addActionListener(new ActionListener(){
      public void actionPerformed(ActionEvent e){
        fs.setVisible(!fs.isVisible());
        if(fs.isVisible())
          animationPanel.startAnimation();
      }
    });
  }
}
```

11.6 Deploying applets

Applets can be deployed in various ways. If you are using Eclipse, you can run the HApplet.java by right-clicking on the HApplet.java file and choosing Run As →Java Applet. Or you can deploy it from a terminal by typing

```
appletviewer HApplet.java
```

and pressing enter. The commented lines

```
// <applet code="HPApplet.class" width=612 height=612>
// </applet>
```

instructs the appletviewer to initiate the HPApplet.class, within a frame with a size of 612 by 612. Below I will show initiating applets from within web pages.

To run applets from a web page, you create an html file like this one:

```
<html>
<body>
  <span style="font-family: sans-serif; font-weight: bold;">
    Sharing your work is easy, here is one possibility using applets:</span>
  <br>
  <br>

  <applet code="HPApplet.class" height="612" width="612">
    </applet>
  </body>
</html>
```

Again, you can run this html file directly from within Eclipse by double-clicking on it. Or you can deploy it with your java-enabled browser. Or you can use appletviewer

```
appletviewer HApplet.html
```

11.6.1 Packaging resources for Applets

If there are a number resource files that your applet uses, then the client's browser will have to download each one separately. For each such request a new connection has to be established. This would cause an overhead. Instead, you can put all the necessary files in a single bundled .jar file. You can do that directly from within Eclipse. Right click on the file(s) and choose Export → jar file (multiple .java files can be chosen from the Package explorer by pressing and holding the ctrl key and then left clicking on each java file). Click next and select the export destination name (for example HPApplet2.jar) under the same directory. Also check the image files psychophysik.png and fechner.png. Eclipse will include the corresponding files in the exported jar file. You can also do the export manually. Open a terminal and type

```
jar -cvf HPApplet2.jar *.class *.png
```

and press enter. In any case, the line in your web page changes slightly:

```
<applet code="HPApplet2.class" archive="HPApplet2.jar" width="250" height="50">
</applet>
```

This html file can similarly be deployed either directly from within Eclipse or by using a web browser, or by using appletviewer.

When packaging resources you must pay attention to how your application loads those resources. For instance, as we saw before, to load images you must use something like this:

```
BufferedImage bi1 = ImageIO.read(
          HPWindow.class.getResource("psychophysik.png"));
```

For opening files to read, we used to use a Scanner object like this

```
Scanner in = new Scanner(new BufferedReader(new FileReader(filename)));
```

you can still use a Scanner object but with a slightly different construction

```
Scanner in = new Scanner(HPWindow.class.getResourceAsStream(filename));
```

11.7 Preparing self running .jar files

An effective way of sharing your applications with your collaborators is preparing self-running jar files. To prepare a self-running jar file, first crate a manifest file, here manifest.mf. This file in its simplest form contains only two lines

```
Manifest-Version: 1.0
Main-Class: HPWindow
```

then pack your application in a .jar file using this manifest file

```
jar -cvfm HPWindow.jar manifest.mf HPWindow.class NormalWindow.class *.png
```

and press enter. Eclipse can also create self running jar files. You should still create a manifest file. Then choose the .java file(s) from the Package explorer, and right click and choose export. As in previous section, check the image resources and this time click Next. Click Next again, until you reach the window where you are asked for manifest file. Here you can either use an existing manifest file, or let Eclipse create one for you by providing the name of the class which holds the main class.

Once the self-running jar file is created, you can deploy it either by double-clicking on the file on your hard rive (this may work even if the file is on a web page), or manually from a terminal by typing

```
java -jar HPWindow.jar
```

11.8 Applications deployed with Java Web Start (JWS)

Java Web Start is a newer technology. To deploy applications with JWS, first prepare a self running .jar file as described above (for example HPWindow.jar). Then prepare the HPWindow.jnlp launch file

```
<?xml version="1.0" encoding="utf-8"?>
<jnlp
  spec="1.0+"
  codebase="file:./"
  href="HPWindow.jnlp">
  <information>
    <title>HelloPsychophysicist Window </title>
    <vendor>Huseyin Boyaci</vendor>
    <description>A simple test of JWS
    </description>
    <offline-allowed/>
  </information>
  <resources>
    <j2se version="1.5+"/>
    <jar href="HPWindow.jar"/>
  </resources>
  <application-desc/>
</jnlp>
```

Now you can launch the JWS application either pointing your web browser to the location of this .jnlp file, by double-clicking on the file on your hard drive, or using the javaws program included in JRE. For the later option, open a terminal and type

```
javaws HPWindow.jnlp
```

then press enter. There a few advantages of JWS over an Applet. A Java applet runs in a “sandbox” for security reasons. Therefore, for example, an applet can not touch your hard drive, can not read or write. Also an applet can not open a FSEM application. But JWS can (in principle). Try it using the HelloPsychophysicist program from Chapter 2. *(Failed under FC4?!)*

Here is a .html file which demonstrates all possible ways of sharing the work as we saw above

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
```

```
<head>
  <meta content="text/html; charset=ISO-8859-1"
 http-equiv="content-type">
  <title>Sharing your work</title>
</head>
<body>
  <span style="font-family: sans-serif; font-weight: bold;">
    Sharing your work is easy, here are some possibilities</span>
  <br>
  <br>
  <span style="font-family: sans-serif;">
    1) package your application in a
    .jar file, place a link to the jar file in your web page and double
    click on the link
    <a href="HPWindow.jar">HPWindow.jar</a>
    (if the click doesn't work, your visitors can copy it to local
    disk and double click on it. Or do
    <span style="font-family: courier;">
      java -jar HPWindow.jar
    </span>
    from a terminal)
    <br>
    2) prepare it in an applet like
    this embedded in your web page
    <br>
    <applet code="HPApplet.class" height="612" width="612">
    </applet>
    <br>
    <br>
    3) Or click on this button to start the applet in a new window:
    <br>
    <applet code="HPApplet2.class" archive="HPApplet2.jar"
    height="40" width="250">
    </applet>
    <br>
     4) Prepare it as a Java Web Start Application and tell them to click
     <a href="HPWindow.jnlp">here</a>, or copy the content of that file
     and run from a terminal
     <span style="font-family: courier;">
        javaws HPWindow.jnlp
     </span>
  </body>
</html>
```

11.9 Summary

To convert from FullScreen (FSEM) to normal window:

- Make your class extend NormalWindow instead of FullScreen
- create a JFrame and place your NormalWindow JPanel in it
- Eliminate calls to those methods:
  - setNBuffers(); getNBuffers(); setDisplayMode(); getDisplayMode(); reportDisplayMode();
- Decide whether you want passive rendering or not, set your preference using setPassiveRendering(pr) where pr is a boolean either true or false
- instead of closeScreen(), invoke the dispose() method of JFrame to close your normal window application.

To prepare an applet of your FullScreen application:

- create a JApplet, provide init(), start() and stop() methods
- Create a class extending NormalWindow and place it in the applet
- if it is an animation, supply the necessary method invokations to the NormalWindow from the init() or start() methods of the JPanel

16 Bits++

In this chapter

- What is Bits++?
- Preparing and loading a 14 bits look up table to Bits++
- A fake Bits++ class

Bits++ is a hardware device manufactured by Cambridge Research Systems, which allows to display 2^{14} different luminance values (14 bits) for each color channel as opposed to the 2^8 (8 bits) capacity of common graphics cards. The device is connected to the video output of the computer on one end and to a computer monitor on the other end. To display images of higher resolution you don't need to create new versions of those images. Instead you provide a table which establishes a relation from 8 bits conventional pixels to 2^{14} different "pixel" levels, which I will call *bits++ pixels*, or *b-pixels* for short. I will name this table as *bits++ look up table* or *b-look up table* for short.

16.1 Why do you need 14 bits?

Let's suppose that you want to present a sinusoidal grating whose luminance profile is shown in Figure 16.1a. Normally you create an image with the desired luminance variation

```
for (int j = 0; j < width; j++) {
  //....
  lum[j] = mean + amp * mean * cos(x * cpp * 2 * Math.PI);
  // ...
}
```

and then apply an inverse look up operation as described in Chapter 8 to determine the correct pixel values in order to display the desired luminance pattern on the screen. In the above pseudo code amp is amplitude, mean is mean luminance level, cpp is frequency in units of cycles per pixel size. The resulting pattern could look like the one shown in Figure 16.1b.

Now suppose that you want to be able to vary the amplitude of your grating in very fine steps, because you are interested in contrast discrimination thresholds. For this example let's look at Figure 16.2a. The two curves in the plot are very close to each other, one has amplitude of 0.04, the other 0.045. And in terms of luminance values even their peaks differ less than a unit (in units of arbitray luminance levels ranging from 0 to 255.) As we have seen in Chapter 8, it is not always possible to display the exact luminance value you desire. And definetely the small difference shown in 16.2a will get lost in translation to pixel values and you will end up displaying the exact same stimulus for the two theoretically different luminance patterns. But check the second plot where the luminance values are seperated by finer intervals, it is possible to capture the difference between the peaks of two curves now. Obviously finer steps are more useful!

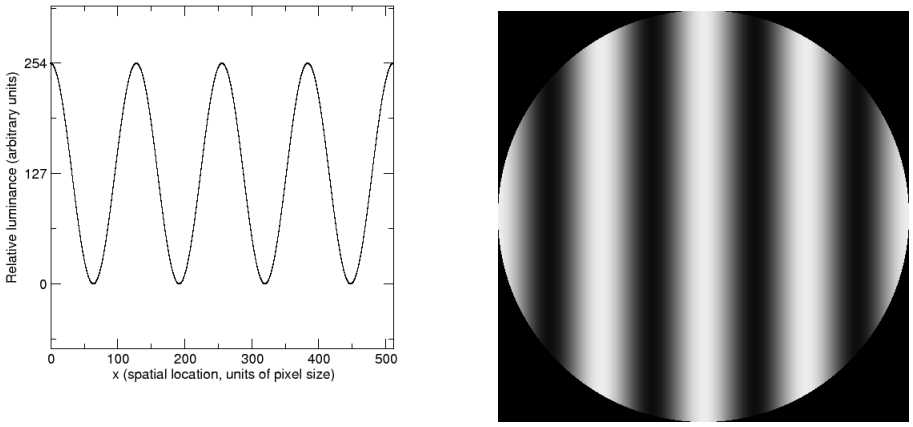

Figure 16.1: The luminance profile (left) along a horizontal line at the middle of a sample grating (right).

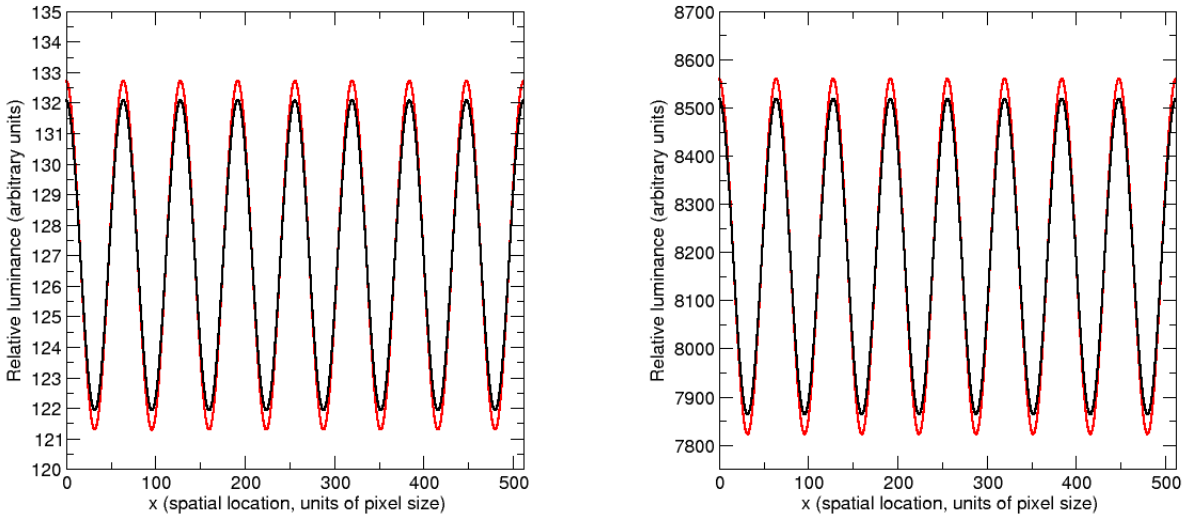

Figure 16.2: A finer amplitude difference. Black curve amplitude=0.04, red curve amplitude=0.045.

But you got a problem: you have a conventional image with pixel values ranging from 0 to 255. How can you get the 14 bit range from that 8 bit image? The answer lies in the bits++ look up table. You don't change the image but manipulate the b-look up table to get the finer scaled luminance pattern on the screen. Let's go back to the grating example. How can you present two slightly different gratings back to back (in time) on the screen? I will explain a method to do this: You first create a pixel value map of your grating, like in the pseudo code above using the entire 8 bit range (0-255). You store this map in an image. Suppose the equation governing the pixel values is

$$pix(x) = 128 + 127 * \cos(2\pi x/128),$$

now you have something like the one shown in Figure 16.3a below. In general one can write the pixel equation as follows

$$pix(x) = meanPix + pixAmplitude/2 * \cos(2\pi x * cpp).$$

This pixel map has a mean value of 128, and amplitude of 254, and ranges from 1 to 255 (0 is spared, which turns out to be useful as we'll see later). Now suppose that the desired relative luminance pattern, in arbitrary units ranging from 0 to $2^{14} - 1$ is

$$lum(x) = 8192 + 0.04 * 8192 * \cos(2\pi x/128).$$

Note that $8192 = 2^{14}/2 = 2^{13}$. The corresponding pattern is shown in Figure 16.3b below in black. The red curve in the same plot corresponds to the same equation above except with a slightly higher contrast (0.045). In general you can write the luminance pattern of a grating as follows

$$lum(x) = meanLum + contrast * meanLum * \cos(2\pi x * cpp)$$

where contrast is defined as

$$contrast = \frac{maxLum - minLum}{2 * meanLum} = \frac{lumAmplitude}{2 * meanLum}.$$

As you see, the problem is reduced to finding a suitable mapping from the 8 bit pixel map to 14 bit relative luminance values. By inspection one finds that the suitable mapping is

$$b{-}table(pix) = meanLum + slope * (pix - meanPix),$$

where slope is given by

$$slope = \frac{lumAmplitude}{pixAmplitude} = \frac{2 * contrast * meanLum}{pixAmplitude}.$$

For our example we get

$$b{-}table(pix) = 2^{13} + \frac{2^{15} * 10^{-2}}{2^7 - 1} * (pix - 2^7).$$

The validity of the above relation can be easily checked: at $x = 0$ we have $pix(0) = 255$ and we should have $lum(0) = 2^{13} + 2^{15} * 10^{-2}$. We find that $b{-}table(255) = 2^{13} + \frac{2^{15} * 10^{-2}}{2^7 - 1} * (255 - 2^7) = 2^{13} + 2^{15} * 10^{-2}$, which checks. Similarly at $x = 32$ we have $pix(32) = 128$ while the desire luminance is $lum(32) = 2^{13}$, and $b{-}table(128) = 2^{13}$, which again checks. You can use the above b-table form for any such grating problem. The mappings corresponding to the two desired luminance profiles are plotted in Figure 16.3c below.

One question which may come to the reader's mind is the following: Are we not loosing precision by storing the base grating profile in integer valued pixels as an image? The answer is yes, but unfortunately there is no other way around that. You will do your best to eliminate artifactual mappings by designing your b-table and base image carefully. The whole process is not 100% automatic.

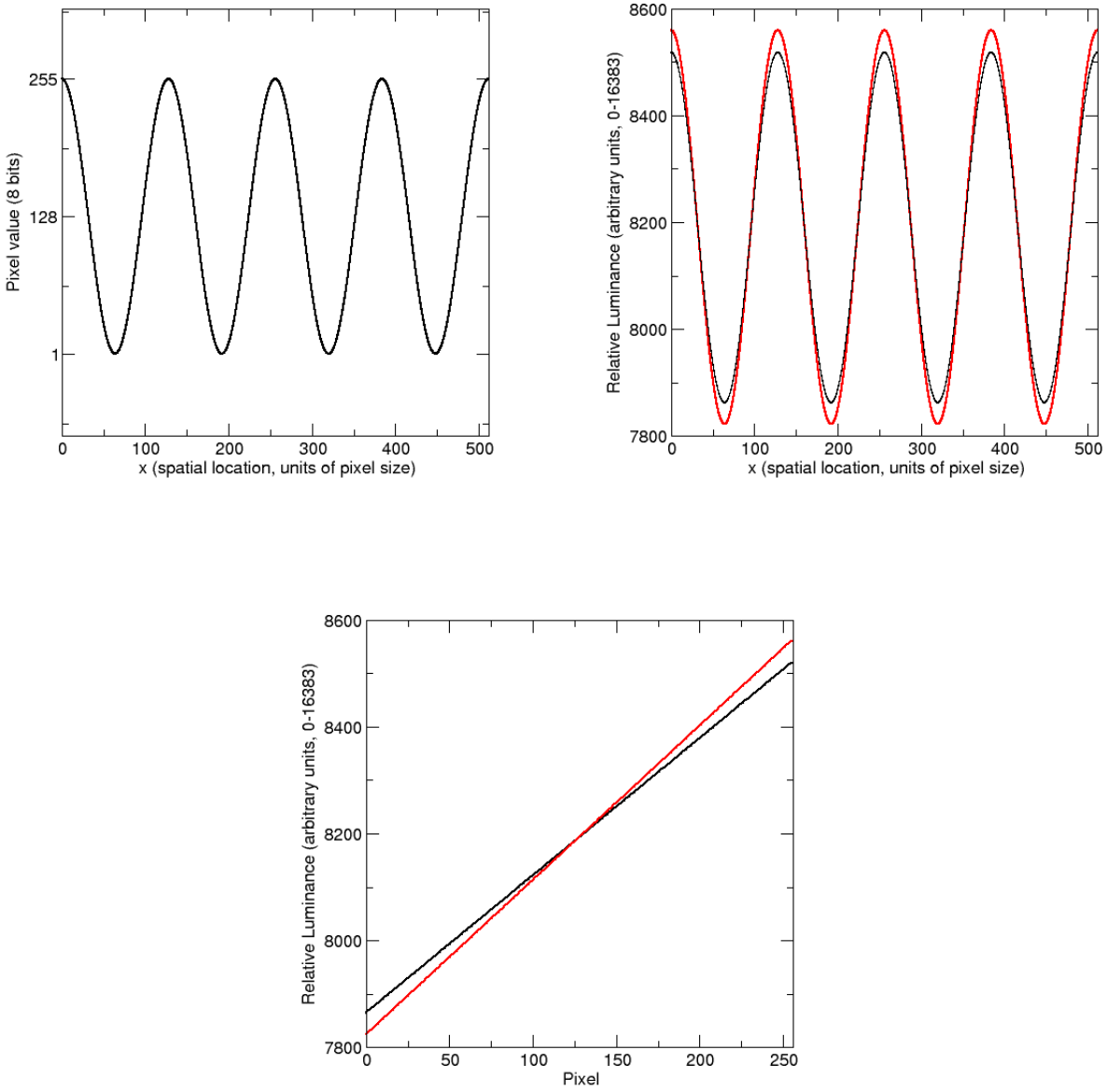

Figure 16.3: A) Grating base pixel profile, B) desired (slightly different) luminance patterns, C) the two slightly different b-tables which provide the mapping from (A) to corresponding (color coded) luminance pattern in (B).

From b-pixel to luminance and vice versa: 14 Bits look up table

We now have a finer scale, a 14 bits luminance map at hand, but what are the right b++ pixel values which give those luminance values? Just like we did in Chapter 8, we need an inverse look up operation, but this time in 14 bits. For this purpose I will use the same class from Chapter 8, only overload the constructor to allow the user specify number of bits. Not to break your former code, I keep the constructors in the original class and default to 8 bits, in other words you can use this class with your implementations which were using the older version. Here are the additions

```
//...
public class CLUT {

  private int DIM;
  //

  public CLUT(String filename){

    this(filename,8);
  }

  public CLUT(double[] table){

    this(table,8);
  }

  public CLUT(String filename, int bits) throws IllegalArgumentException{

    InputStream stream = CLUT.class.getResourceAsStream(filename);
    Scanner in = new Scanner(stream);

    DIM = (int)pow(2,bits);
    pix2Lum = new double[DIM];
    lum2Pix = new int[DIM];

    //...
  }
  public CLUT(double[] table, int bits) throws IllegalArgumentException{
    DIM = (int)pow(2,bits);
    pix2Lum = new double[DIM];
    lum2Pix = new int[DIM];
    //...
  }
  //...
}
```

Just as in Chapter 8 you can employ an inverse look up operation to find the b-pixel that provides the luminance closest to the luminance of your desire

```
CLUT lut = new CLUT("table14.txt");
```

```
for(int i=0; i<btable.length; i++)
  btable[i]=lut.lum2Pix(meanLum+slope*(i-meanPix));
```

A sample b-pixel versus luminance is plotted in Figure 16.4 below.

16.2 Communicating the bits++ look up table

You can dynamically load a b-table to Bits++ with every refresh of the screen. The way to load the table is to "draw" it on a single line and "display" it. It is sensible to put this line at the upper left corner of the screen. This way the lookup operation applies to the rest of the vertical scan. For Bits++ to understand that you are trying to upload a table, it must first be triggered by a specific combination of pixel values. This specific trigger consists of 36 values, or 12 pixels to draw. On the same line, right after those 12 pixels in the next 512 pixels, you provide the values of your new lookup table. Those 512 pixels contain 16 bits of information for each channel and for each of 256 conventional pixel values (note that Bits++ manufacturers require 16 bit mapping rather than 14 bits, this is done for future compatibility). First pixel holds the most significant 8 bit information for each channel for your conventional pixel of 0, second pixel holds the least significant 8 bits for each channel. Hence you have 2 pixels devoted to all three channels for each one of 256 levels (2*256=512). Figure 16.5 show the trigger and images which correspond to several b-look tables. Once it is triggered, Bits++ doesn't send that line to screen for display.

Note: Make sure that your video card has a linear ramp gamma. Because if your video card has anything but linear gamma it will destroy the values in trigger pixels and you will never be able to load your look up table.

16.3 BitsPP class

In this section I will develop a sample class, BitsPP, with methods to control Bits++. It is a good idea to inherit all the useful methods of FullScreen class which do not need any modification, therefore BPP extends FullScreen

```
public class BitsPP extends FullScreen {
  //...
  public BitsPP(int screen_id) {
    super(screen_id);
    initClut();
  }
  //...
}
```

The constructors of BitsPP, apart from invoking FullScreen's constructor, invokes initClut() method which initialiazes the look up table line as a BufferedImage and places the trigger part in first 12 pixels

```
  private BufferedImage lutBI;
  private WritableRaster lutWR;
  public void initClut() {
    int[] unlock = { 36, 106, 133, 63, 136, 163, 8, 19,
        138, 211, 25, 46, 3, 115, 164, 112, 68, 9, 56, 41,
        49, 34, 159, 208, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
```

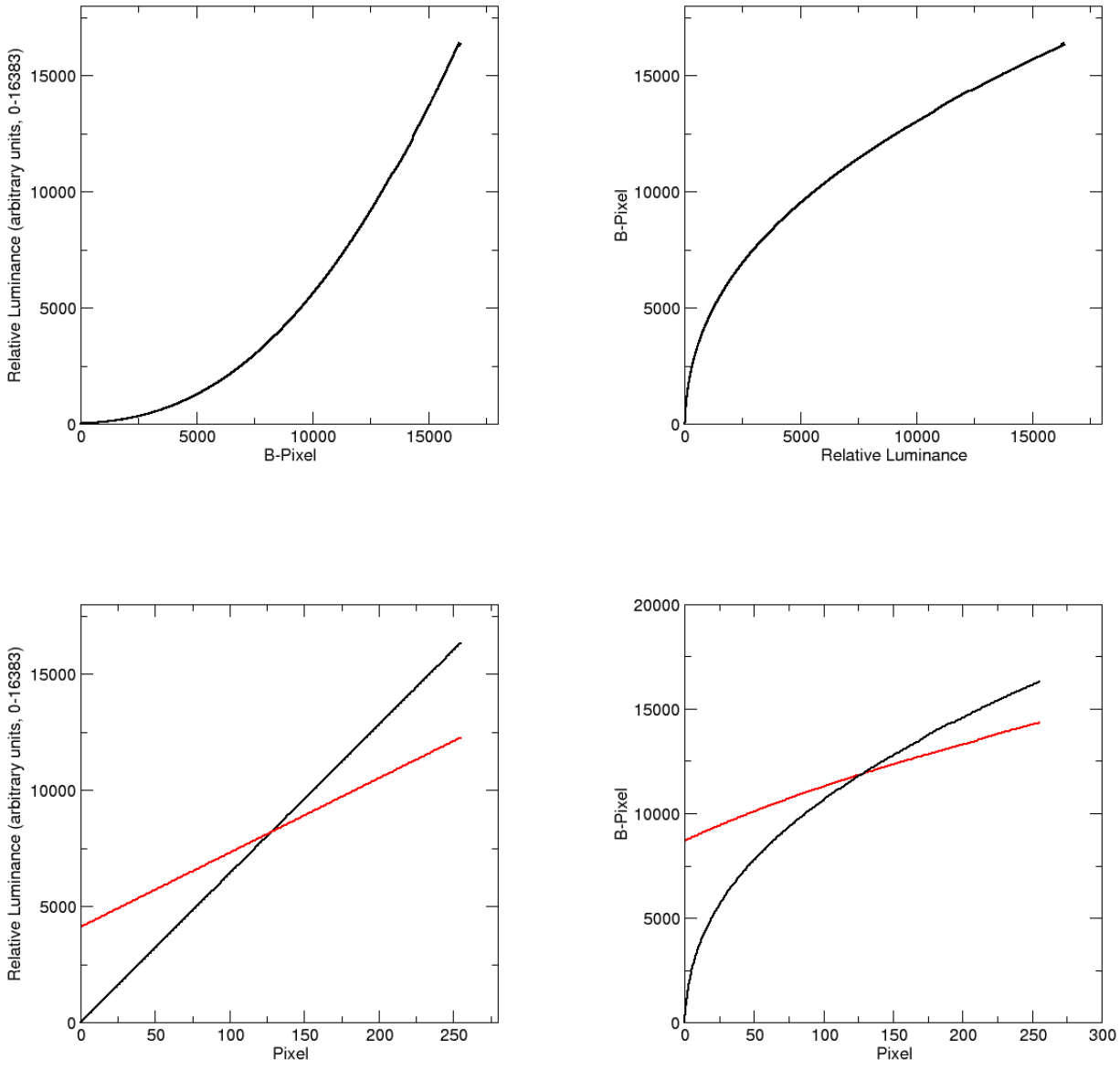

Figure 16.4: A) B-pixel vs. relative luminance, B) Inverse of (A), C) Desired relations between conventional pixels and the finer scale relative luminance, D) Bits++ look up table, b-table: relation between the conventional pixel and b-pixel to achive the desired pixel vs. luminance relation in (C) given the relation between b-pixels and luminance in (A) and (B).

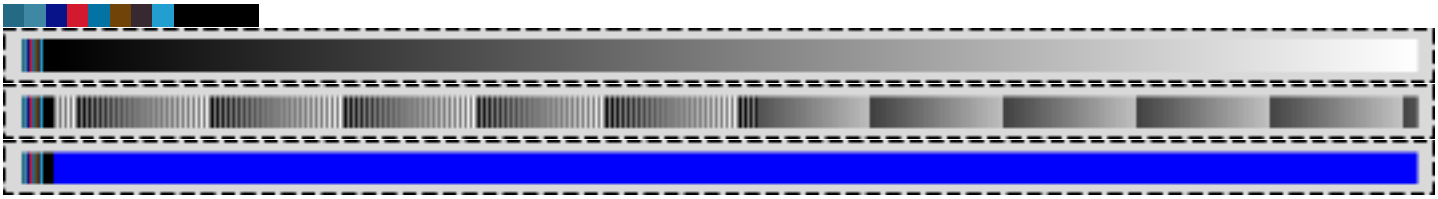

Figure 16.5: The look-up line uploaded to Bits++ device. On top, the trigger header is shown. Below, the slope=1 table for all three channels. The next image corresponds to the b-table given in the example equation in text (Eqn XXX). Bottom image is a chromatic version where red and green channels are all set to zero, blue channel set to maximum.

```
    int width = getWidth();
    lutBI = new BufferedImage(width, 1, BufferedImage.TYPE_INT_RGB);
    lutWR = (WritableRaster) lutBI.getRaster();
    try {
      lutWR.setPixels(0, 0, unlock.length / 3, 1, unlock);
    } catch (ArrayIndexOutOfBoundsException e) {
      System.err.println("Error in BitsPP.initClut(): cannot initialize " +
          "clut row (is your screen wide enough (>524 pixels)?)");
      e.printStackTrace();
    }
    int[] table = new int[256];
    for (int i = 0; i < table.length; i++) {
      table[i] = i << 6;
    }
    setClut(table);
  }
```

initClut() initializes the lutBI, a BufferedImage with 1 pixel height, and generates its WritableRaster lutWR. lutBI is going to be the bits++ look up table line. initCLut() first places the trigger sequence at the first 12 pixels of lutBI, then invokes setCLut() method with a dummy look up table, which is just a straight line. setCLut in turn prepares the rest of the lutBI

```
  public void setClut(int clut[]) throws ArrayIndexOutOfBoundsException{
    int len = 256;
    if (clut.length != len)
      throw new ArrayIndexOutOfBoundsException(
        "Clut should have " + len + " elements");

    int[] row = new int[len * 2 * 3];
    int r;
    for (int i = 0; i < len; i++) {
      r = (clut[i]) << 2;
      row[i * 6] = r >> 8;
      row[i * 6 + 1] = row[i * 6];
      row[i * 6 + 2] = row[i * 6];
      row[i * 6 + 3] = r & 255;
      row[i * 6 + 4] = row[i * 6 + 3];
```

```
      row[i * 6 + 5] = row[i * 6 + 3];
    }
    lutWR.setPixels(12, 0, row.length / 3, 1, row);
  }
```

In the final version of the class I will include an overloaded version of this method to accept 3 lookup tables for each of R,G and B channels.

Once you have the b-table as an image all you have to do is to "display" it (remember Bits++ doesn't really display that line on the screen, as soon as it sees the trigger, the rest of the line is interpreted as b-table and is not sent to screen). It is sensible to put this line at the upper left corner of the screen. This way the lookup operation applies to the rest of the vertical scan. I will override the displayImage() method of FullScreen to automatically draw the b-table on the top line everytime it is invoked

```
public void displayImage(int x, int y, BufferedImage bi) {
    Graphics2D g = (Graphics2D)getBufferStrategy().getDrawGraphics();
    try {
      if(g!=null && bi!=null){
        g.drawImage(bi, x, y, null);
        g.drawImage(lutBI, 0, 0, null);
      }
    }
    finally {
      g.dispose();
    }
  }
```

The changes to the other two methods, displayText() and blankScreen(), follow the same logic. In order to leave the system in a nice condition (loaded with a linear b-table) after the program is terminated I will override the closeScreen() method

```
public void closeScreen() {
  initClut();
  blankScreen();
  updateScreen();
  try{
    Thread.sleep(100);
  }catch(InterruptedException e){}
  super.closeScreen();
}
```

That's all, and here is the complete listing of BitsPP.java followed by the new CLUT class

```
/*
 * Chapter A: BitsPP
 *
 * Provides Bits++ triggering mechanism and methods to load
 * Bits++ look up tables.
 *
 */
```

```
import java.awt.*;
import java.awt.image.*;
import java.io.File;
import java.io.IOException;
import javax.imageio.ImageIO;
public class BitsPP extends FullScreen {
  private BufferedImage lutBI;
  private WritableRaster lutWR;
  public BitsPP() {
    this(0);
  }
  public BitsPP(int screen_id) {
    super(screen_id);
    initClut();
  }
  public void initClut() {
    int[] unlock = { 36, 106, 133, 63, 136, 163, 8, 19,
        138, 211, 25, 46, 3, 115, 164, 112, 68, 9, 56, 41,
        49, 34, 159, 208, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
    int width = getWidth();
    lutBI = new BufferedImage(width, 1, BufferedImage.TYPE_INT_RGB);
    lutWR = (WritableRaster) lutBI.getRaster();
    try {
      lutWR.setPixels(0, 0, unlock.length / 3, 1, unlock);
    } catch (ArrayIndexOutOfBoundsException e) {
      System.err.println("Error in BitsPP.initClut(): cannot initialize " +
          "clut row (is your screen wide enough (>524 pixels)?)");
      e.printStackTrace();
    }
    int[] table = new int[256];
    for (int i = 0; i < table.length; i++) {
      table[i] = i << 6;
    }
    setClut(table);
  }
  public void setClut(int[] redClut, int[] greenClut, int[] blueClut)
      throws ArrayIndexOutOfBoundsException{
    int len = 256;
    if (redClut.length != len || greenClut.length != len
        || blueClut.length != len)
      throw new ArrayIndexOutOfBoundsException(
        "Clut should have " + len + " elements");

    int[] row = new int[len * 2 * 3];
    int r;
    int g;
    int b;
    for (int i = 0; i < len; i++) {
```

```
    r = redClut[i]   << 2;
    g = greenClut[i] << 2;
    b = blueClut[i]  << 2;
    row[i * 6] = r >> 8;
    row[i * 6 + 1] = g >> 8;
    row[i * 6 + 2] = b >> 8;
    row[i * 6 + 3] = r & 255;
    row[i * 6 + 4] = g & 255;
    row[i * 6 + 5] = b & 255;
  }
  lutWR.setPixels(12, 0, row.length / 3, 1, row);
}
public void setClut(int clut[]) throws ArrayIndexOutOfBoundsException{
  int len = 256;
  if (clut.length != len)
    throw new ArrayIndexOutOfBoundsException(
      "Clut should have " + len + " elements");

  int[] row = new int[len * 2 * 3];
  int r;
  for (int i = 0; i < len; i++) {
    r = (clut[i]) << 2;
    row[i * 6] = r >> 8;
    row[i * 6 + 1] = row[i * 6];
    row[i * 6 + 2] = row[i * 6];
    row[i * 6 + 3] = r & 255;
    row[i * 6 + 4] = row[i * 6 + 3];
    row[i * 6 + 5] = row[i * 6 + 3];
  }
  lutWR.setPixels(12, 0, row.length / 3, 1, row);
}
public void displayImage(int x, int y, BufferedImage bi) {
  Graphics2D g = (Graphics2D)getBufferStrategy().getDrawGraphics();
  try {
    if(g!=null && bi!=null){
      g.drawImage(bi, x, y, null);
      g.drawImage(lutBI, 0, 0, null);
    }
  }
  finally {
    g.dispose();
  }
}
public void displayText(int x, int y, String text) {
  Graphics2D g = (Graphics2D)getBufferStrategy().getDrawGraphics();
  try {
    if(g!=null && text!=null){
      g.setFont(getFont());
```

```
        g.setColor(getForeground());
        g.drawString(text, x, y);
        g.drawImage(lutBI, 0, 0, this);
      }
    }
    finally {
      g.dispose();
    }
  }

  public void blankScreen() {
    Graphics2D g = (Graphics2D)getBufferStrategy().getDrawGraphics();
    try {
      if(g!=null){
        g.setColor(getBackground());
        g.fillRect(0, 0, getWidth(), getHeight());
        g.drawImage(lutBI, 0, 0, this);
      }
    }
    finally {
      g.dispose();
    }
  }
  public void closeScreen() {
    initClut();
    blankScreen();
    updateScreen();
    try{
      Thread.sleep(100);
    }catch(InterruptedException e){}
    super.closeScreen();
  }
}
```

and here is the new CLUT

```
/*
 * Chapter A: CLUT8.java
 *
 * Provides methods to perform (inverse) lookup operations.
 *
 */
import static java.lang.Math.*;
import java.awt.Color;
import java.awt.Graphics2D;
import java.awt.event.KeyEvent;
import java.awt.image.BufferedImage;
import java.io.InputStream;
```

```
import java.util.NoSuchElementException;
import java.util.Scanner;
public class CLUT {
  private int DIM;
  private double maxLum;
  private double[] pix2Lum;
  private int[] lum2Pix;
  private boolean monotonicIncrease = true;
  private boolean monotonicDecrease = true;
  public CLUT(String filename){

    this(filename,8);
  }

  public CLUT(double[] table){

    this(table,8);
  }

  public CLUT(String filename, int bits) throws IllegalArgumentException{
    InputStream stream = CLUT.class.getResourceAsStream(filename);
    Scanner in = new Scanner(stream);

    DIM = (int)pow(2,bits);
    pix2Lum = new double[DIM];
    lum2Pix = new int[DIM];

    maxLum = 0;
    double[] table = new double[DIM];
    for (int pix = DIM - 1; pix > -1; pix--) {
      try {
        table[pix] = in.nextDouble();
      } catch (NoSuchElementException e) {
        throw new IllegalArgumentException(
            "Error in CLUT8(String): wrong input file - file too short");
      }
    }
    if (in.hasNext())
      throw new IllegalArgumentException(
          "Error in CLUT8(String): suspicious input file - file too long");
    in.close();
    setClut(table);
  }
  public CLUT(double[] table, int bits) throws IllegalArgumentException{
    DIM = (int)pow(2,bits);
    pix2Lum = new double[DIM];
    lum2Pix = new int[DIM];
```

```
  if (table.length != DIM)
    throw new IllegalArgumentException(
        "Error in CLUT(double[], int): incompatible data");
  else
    setClut(table);
}

public void setClut(double[] table){

  if (table.length != DIM)
    throw new IllegalArgumentException(
        "Error in CLUT8.setClut(double[]): incompatible data");

  maxLum = 0;
  pix2Lum = table;
  double lastLum = pix2Lum[0];
  for (double lum : pix2Lum) {
    maxLum = max(maxLum, lum);
    if (monotonicIncrease && lastLum > lum)
      monotonicIncrease = false;
    else if(monotonicDecrease && lastLum < lum)
      monotonicDecrease = false;
    lastLum = lum;
  }

  if(!monotonicIncrease && !monotonicDecrease){
    System.err
    .println("Warning in CLUT8.setClut(double[]): "
        + "LUT is not monotonically increasing or decreasing, "
        + "speed may degrade");
  }

  for (int pix = 0; pix < DIM; pix++)
    pix2Lum[pix] *= (double)(DIM-1) / maxLum;

  if(monotonicIncrease){
    int lastPixel = 0;
    for (int lum = 0; lum < DIM; lum++) {
      double diff = Double.MAX_VALUE;
      for (int j = lastPixel; j < DIM; j++) {
        double diffTmp = abs(pix2Lum[j] - lum);
        if (diffTmp == 0) {
          lum2Pix[lum] = j;
          lastPixel = j;
          break;
        }
        else if (diffTmp <= diff) {
          lum2Pix[lum] = j;
```

```
          lastPixel = j;
          diff = diffTmp;
        }
        else
          break;
      }
    }
  }
  else if(monotonicDecrease){
    int lastPixel = 0;
    for (int lum = DIM-1; lum >= 0; lum--) {
      double diff = Double.MAX_VALUE;
      for (int j = lastPixel; j < DIM; j++) {
        double diffTmp = abs(pix2Lum[j] - lum);
        if (diffTmp == 0) {
          System.err.println(lum);
          lum2Pix[lum] = j;
          lastPixel = j;
          break;
        }
        else if (diffTmp <= diff) {
          lum2Pix[lum] = j;
          lastPixel = j;
          diff = diffTmp;
        }
        else
          break;
      }
    }
  }
  else{
    for (int lum = 0; lum < DIM; lum++) {
      double diff = Double.MAX_VALUE;
      for (int j = 0; j < DIM; j++) {
        double diffTmp = abs(pix2Lum[j] - lum);
        if (diffTmp == 0) {
          lum2Pix[lum] = j;
          break;
        }
        else if (diffTmp <= diff) {
          lum2Pix[lum] = j;
          diff = diffTmp;
        }
      }
    }
  }
}
public double getMaxLum() {
```

```
  return maxLum;
}
public int lum2Pix(double lum) {
  int intLum = (int) round(lum);
  int pixel = lum2Pix[intLum];
  if (lum == (double) intLum)
    return pixel;
  else {
    if(monotonicIncrease){
      int pixelStart = lum2Pix[max(intLum - 1, 0)];
      int pixelEnd = lum2Pix[min(intLum + 1, DIM - 1)];
      double diff = Double.MAX_VALUE;
      for (int j = pixelStart; j <= pixelEnd; j++) {
        double diffTmp = abs(pix2Lum[j] - lum);
        if (diffTmp == 0) {
          pixel = j;
          break;
        }
        else if (diffTmp <= diff) {
          pixel = j;
          diff = diffTmp;
        }
        else
          break;
      }
    }
    else if(monotonicDecrease){
      int pixelStart = lum2Pix[min(intLum + 1, DIM-1)];
      int pixelEnd = lum2Pix[max(intLum - 1, 0)];
      double diff = Double.MAX_VALUE;
      for (int j = pixelStart; j <= pixelEnd; j++) {
        double diffTmp = abs(pix2Lum[j] - lum);
        if (diffTmp == 0) {
          pixel = j;
          break;
        }
        else if (diffTmp <= diff) {
          pixel = j;
          diff = diffTmp;
        }
        else
          break;
      }
    }
    else {
      double diff = Double.MAX_VALUE;
      for (int j = 0; j < DIM; j++) {
        double diffTmp = abs(pix2Lum[j] - lum);
```

```
          if (diffTmp == 0) {
            pixel = j;
            break;
          }
          else if (diffTmp <= diff) {
            pixel = j;
            diff = diffTmp;
          }
        }
      }
      return pixel;
    }
  }
  public int lum2Pix(int lum) {
    return lum2Pix[lum];
  }
  public double pix2Lum(int pixel) {
    return pix2Lum[pixel];
  }
}
```

16.4 A Fake BitsPP class

One often needs to work on the experimental code on a different computer rather than the one connected to Bits++. Of course it wouldn't be very productive to have two separate code for development and actual experiment. However, we can easily create a fake Bits++ class which has the exact same structure except it performs software lookup operation within the limits of a standard 8 bit video card. Here are the initClut() and setClut() methods of the fake Bits++ class

```
private BufferedImageOp op;
private LookupTable table;

public void initClut() {
  int[] convert = new int[256];
  for (int i = 0; i < 256; i++)
    convert[i] = i << 6;
  setClut(convert);
}

public void setClut(int clut[]) throws ArrayIndexOutOfBoundsException{
    int len = 256;
    if (clut.length != len)
      throw new ArrayIndexOutOfBoundsException(
          "Clut should have " + len + " elements");
    byte[] row = new byte[len];
    for (int i = 0; i < len; i++)
      row[i] = (byte) (clut[i] >> 6);
```

```
        table = new ByteLookupTable(0, row);
        op = new LookupOp(table, new RenderingHints(
            RenderingHints.KEY_ANTIALIASING,
            RenderingHints.VALUE_ANTIALIAS_ON));
    }
```

Since standard video cards have 8 bit resolution, in the fake mode we use only the most significant part of the table provided by the application. Note that you can omit the RenderingHints and pass null instead. This would improve the speed of the filtering but result in poorer quality image. As in the actual BitsPP class, I overload the setClut() method to accept 3 tables, one for each of R, G and B channels. In the displayImage() method we use the BufferedImageOP.filter() method to perform the software lookup operation

```
    public void displayImage(int x, int y, BufferedImage bi) {

        Graphics2D g = (Graphics2D) getBufferStrategy().getDrawGraphics();
        try {
          // check image type ...
          g.drawImage(op.filter(bi, null), (int) x, (int) y, null);
          getBufferStrategy().show();
        }
        finally {
          g.dispose();
        }
      }
```

However a software lookup operation is much slower than a hardware one, especially for larger images. Another problem is that the BufferedImageOP and the image we provide may not be compatible. If they are incompatible, the result of the operation would be unpredictable. To address these issues, I implement the displayImage() method to accept only two certain types of BufferedImages and to check the compatibility of the BufferedImage and the BufferedImageOP. The images accepted are of type TYPE_BYTE_GRAY or TYPE_3BYTE_BGR. In the examples below I will show how to create images of these types. As for the lookup tables, if the image to display is of TYPE_BYTE_GRAY, then the lookup table should be prepared with setClut(clut) method, if it is of TYPE_3BYTE_BGR, the lookup table should be prepared with setClut(redClut, greenClut, blueClut). In this sample fake version I will not override displayText(), blankScreen() and closeScreen() methods, you can easily improve those methods to fit your needs. The final listing of BitsPPFake.java follows

```
/*
 * Chapter A: BitsPPFake.java
 *
 * Provides Fake Bits++ methods for development
 *
 */
import java.awt.*;
import java.awt.image.*;
public class BitsPPFake extends FullScreen {
  private BufferedImageOp op;
  private LookupTable table;
  public BitsPPFake() {
```

```
  this(0);
}
public BitsPPFake(int screen_id) {
  super(screen_id);
  initClut();
}
public void initClut() {
  int[] convert = new int[256];
  for (int i = 0; i < 256; i++)
    convert[i] = i << 6;
  setClut(convert);
}
public void setClut(int[] redClut, int[] greenClut, int[] blueClut)
throws ArrayIndexOutOfBoundsException{
  int len = 256;
  if (redClut.length != len || greenClut.length != len
      || blueClut.length != len)
    throw new ArrayIndexOutOfBoundsException(
        "Clut should have " + len + " elements");
  byte[][] row = new byte[3][len];
  for (int i = 0; i < len; i++) {
    row[0][i] = (byte) (blueClut[i] >> 6);
    row[1][i] = (byte) (greenClut[i] >> 6);
    row[2][i] = (byte) (redClut[i] >> 6);
  }
  table = new ByteLookupTable(0, row);
  op = new LookupOp(table, new RenderingHints(
      RenderingHints.KEY_ANTIALIASING,
      RenderingHints.VALUE_ANTIALIAS_ON));
}
public void setClut(int clut[]) throws ArrayIndexOutOfBoundsException{
  int len = 256;
  if (clut.length != len)
    throw new ArrayIndexOutOfBoundsException(
        "Clut should have " + len + " elements");
  byte[] row = new byte[len];
  for (int i = 0; i < len; i++)
    row[i] = (byte) (clut[i] >> 6);
  table = new ByteLookupTable(0, row);
  op = new LookupOp(table, new RenderingHints(
      RenderingHints.KEY_ANTIALIASING,
      RenderingHints.VALUE_ANTIALIAS_ON));
}
public void displayImage(int x, int y, BufferedImage bi) {
  Graphics g =  getBufferStrategy().getDrawGraphics();
  try {
    if(g!=null && bi!=null){
      if (bi.getRaster().getNumBands() != table.getNumComponents()
```

```
            || (bi.getType() != BufferedImage.TYPE_3BYTE_BGR
                && bi.getType() != BufferedImage.TYPE_BYTE_GRAY)) {
          System.err.println(
          "Exception in BitsPPFake.displayImage(): Wrong image or table type");
          System.err.println(
          "use either a BufferedImage of TYPE_BYTE_GRAY with setClut(table)");
          System.err.println(
          "        or a TYPE_3BYTE_BGR with setClut(tableR, tableG, tableB)");
          closeScreen();
          System.exit(0);
        }
        else
          g.drawImage(op.filter(bi, null), x, y, null);
      }
    }
    finally {
      g.dispose();
    }
  }
}
```

16.5 Examples

I will now show how to dynamically load lookup tables to Bits++ and animate otherwise stationary images. In the first example we generate a sinusoidal grating and make it counter-phase flicker by inverting the lookup table. In the second example, we generate a sinusoidal grating and allow the observer change its contrast by pressing up and down arrow keys. Here is the listing of the first example

```
/*
 * chapter A: BitsPlusFlicker.java
 *
 * displays a flickering grating using bits++
 *
 */
import java.awt.image.BufferedImage;
import static java.lang.Math.*;
//Choose either fake or actual mode
//public class BitsPlusFlicker extends BitsPP implements Runnable {
public class BitsPlusFlicker extends BitsPPFake implements Runnable {

  private static final long flickDuration = 64;
  private static final int repeat = 100;

  BitsPlusFlicker(){

    super(0);
  }
```

```
BitsPlusFlicker(int i){

  super(i);
}
public static void main(String[] args) {
  int screen_id = 0;
  BitsPlusFlicker fs = new BitsPlusFlicker(screen_id);
  fs.setNBuffers(2);
  new Thread(fs).start();
}

public void run(){

  try {

    int dd = (int) pow(2, 14) - 1;
    float onePix = dd/255f;
    int[][] table = new int[2][256];
    for (int i = 1; i < table[0].length; i++){
      table[0][i] = (int)(i * onePix);
      table[1][i] = dd - table[0][i];
    }

    BufferedImage bi =
      Tools.aGrating(128, (double)1/128, (double)127/128, 513);
    blankScreen();
    displayImage(bi);
    updateScreen();

    Thread.sleep(200);

    long total = System.currentTimeMillis();

    for (int k = 0; k < repeat / table.length; k++) {

      for (int j=0; j < table.length; j++){
        long start = System.currentTimeMillis();
        setClut(table[j]);
        displayImage(bi);
        updateScreen();
        Thread.sleep(
            max(flickDuration - System.currentTimeMillis() + start, 0));
      }
    }

    System.err.println("rendered and showed " + repeat +
        " images in: " + (System.currentTimeMillis()-total) + " msec");
```

```
    } catch (PixelOutOfRangeException e) {
      e.printStackTrace();
    } catch (InterruptedException e) {
      e.printStackTrace();
      Thread.currentThread().interrupt();
    }
    finally {
      closeScreen();
    }
  }
}
```

to choose either the fake or actual mode, just uncomment the mode you like to use and comment out the other one. This example works in both modes without any further change. You should also choose a screen device. If you have only one screen, leave screen_id = 0. If you have more than one screen, find out the order by playing with screen_id variable. Next, the program prepares two linear lookup tables with inverse slopes, and prepares the grating using Tools.aGrating() method. This image stays constant for the entire duration of the test. But in the loop below, we use the two tables prepared before. After 100 iteration, the program reports the total time spent on setting the lookup table and displaying the image.

I've placed the aGrating() method in a different class because we will need it once agaoin for the next example below, here is the Tools class

```
/*
 * Chapter A: Tools.java
 *
 * Provides the method to prepare sinusoidal grating
 *
 */
import java.awt.Graphics2D;
import java.awt.image.BufferedImage;
public class Tools {
  public static BufferedImage aGrating(int mean, double cpp,
      double contrast, int size) throws PixelOutOfRangeException {
    int[] gPixel = new int[size * size];
    int size2 = size * size / 4;
    for (int j = 0; j < size; j++) {
      int x = (j - size / 2);
      int val = (int) (contrast * mean * Math.cos(x * cpp * 2 * Math.PI));
      x = x * x;
      int iLow = (int) (size / 2 - Math.sqrt(size2 - x));
      int iHigh = (int) (size / 2 + Math.sqrt(size2 - x));
      for (int i = iLow; i <= iHigh; i++) {
        int k = i * size + j;
        if ((gPixel[k] = mean + val) > 255)
          throw new PixelOutOfRangeException(
            "Exception in setGrating: calculated pixel value exceeds 255");
      }
```

```
    }
    BufferedImage bi = new BufferedImage(size, size,
        BufferedImage.TYPE_BYTE_GRAY);
    bi.getRaster().setPixels(0, 0, size, size, gPixel);
    return bi;
  }
  public static BufferedImage aGratingBGR(int mean, double cpp,
      double contrast, int size) throws PixelOutOfRangeException {
    BufferedImage bi = aGrating(mean, cpp, contrast, size);
    BufferedImage bic = new BufferedImage(size, size,
        BufferedImage.TYPE_3BYTE_BGR);
    Graphics2D g2 = (Graphics2D) bic.getGraphics();
    g2.drawImage(bi, 0, 0, null);
    return bic;
  }
}
class PixelOutOfRangeException extends Exception {
  PixelOutOfRangeException(String s) {
    super(s);
  }
}
```

This class has methods to create TYPE_BYTE_GRAY or TYPE_3BYTE_BGR BufferedImages. Note how one can convert an image to a different format. Using a TYPE_BYTE_GRAY is advantageous especially in the fake mode, because the otherwise slower software lookup operation is fast enough with a single band. The methods throw PixelOutOfRangeExceptions in case the pixel level exceeds 255.

The next example is interactive: user can change the contrast of the grating by pressing the up and down arrow keys

```
/*
 * chapter A: BitsPlusAdjustContrast.java
 *
 * displays a varying contrast grating using bits++
 *
 */
import java.awt.image.BufferedImage;
import java.awt.event.KeyEvent;
import static java.lang.Math.*;
//choose either fake or actual mode
public class BitsPlusAdjustContrast extends BitsPPFake implements Runnable{
//public class BitsPlusAdjustContrast extends BitsPP implements Runnable{
  public BitsPlusAdjustContrast(){

    super(0);
  }

  public BitsPlusAdjustContrast(int i){
```

```
  super(i);
}

public static void main(String[] args) {
  int screen_id = 0;
  BitsPlusAdjustContrast fs = new BitsPlusAdjustContrast(screen_id);
  fs.setNBuffers(2);
  new Thread(fs).start();
}

public void run(){

  try {

    int dd = (int) Math.pow(2, 14) - 1;
    float onePix = dd / 255f;
    BufferedImage bi =
      Tools.aGrating(128, (double)1/128, (double)127/128, 513);
    int[] table = new int[256];
    for (int i = 0; i < table.length; i++)
      table[i] = (int)(i * onePix);

    setClut(table);

    displayText(50, 100,
        "Press up/down arrow keys to increase/decrease contrast");
    displayText(50, 200, "Press q to quit");
    displayText(50, 300, "Now Press any key to start");
    updateScreen();

    getKeyPressed(-1);

    blankScreen();
    displayImage(bi);
    updateScreen();
    Integer res = null;

    int meanPix = 128;
    int meanLum = (int)pow(2,13);
    float maxSlope = (float)(meanLum/meanPix);
    float slope = maxSlope;
    while (true) {

      res = getKeyPressed(-1);

      if (res == KeyEvent.VK_UP)
        slope = Math.min(maxSlope, slope + .05f * maxSlope);
      else if (res == KeyEvent.VK_DOWN)
```

```
          slope = Math.max(-maxSlope, slope - .05f * maxSlope);
        else if (res == KeyEvent.VK_Q)
          break;

        // leave 0 always black
        for (int i = 1; i < table.length; i++)
          table[i] = (int)(meanLum + slope * (i - meanPix));

        setClut(table);
        displayImage(bi);
        updateScreen();
      }
      blankScreen();
    } catch (PixelOutOfRangeException e) {}
    finally {
      closeScreen();
    }
  }
}
```

The logic of this example is similar to the previous one. We create a stationary grating, then manipulate the contrast by altering the lookup table. Here we use a different scheme: we have a linear table, whose slope can vary between 0 and 1. When the user presses up arrow the slope increases, with down arrow press it decreases. The table is pivoted at the mean value of the grating, so that the mean is always fixed.