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...