Blob Tracking

- Our goal is to build an app that detects when something enters the visual range of a webcam
- Example use: Detect when someone enters your room.
The Basic Process

- Get an image from the webcam
- Subtract some notion of ‘background’
- ‘Grow’ regions of pixels to reduce noise

Getting the Image
Java Media Framework

- Java itself doesn’t ship with much support for video and audio media.
- There are software libraries to get access however
- We will be using the Java Media Framework (JMF)
  - It is an ‘official’ Sun library
  - Gives us access to our webcam reasonably simply

Abstraction, abstraction, abstraction...

- We don’t want to be stuck with the JMF though
  - Might need a different library for a different webcam
  - Might even want to input from a video file
- Try to identify the fundamental components of something that supplies us with images (=“frames”)
- Abstract into an interface

```java
public interface FrameInterface {
    public void open(String descriptor);
    public BufferedImage getNextFrame();
    public void close();
}
```
Concrete Instantiation

- The code in JMFFrameSource is not pretty because frankly the JMF is pretty awful to work with.
- You can ignore JMFFrameSource.java for this course – I refuse to teach JMF!!
- This design *encapsulates* everything about getting images from a webcam in a single class.
- The interface *decouples* the program from the JMF so you can easily substitute a better library

Aside: BufferedImage

- The FrameInterface returns objects of type BufferedImage
- This is a standard class in the class library to handle images
- The “Buffered” bit means that the image is represented by an accessible buffer – a grid of pixels
Aside: RGB Pixels

How do you represent the colour of a single pixel?

Actually many ways, but commonly we use RGB

The colour is made up from only Red, Green and Blue.

We use 24 bits to represent the colour

8 bits red (0-255)
8 bits green (0-255)
8 bits blue (0-255)

E.g. Purple = 160-32-240 (R-G-B)

We can think of this as a colour space

Like 3D space but xyz becomes rgb

Aside: RGB Pixels

We will need to get at individual pixels
Buffered image provides getRGB(...) methods
Each pixel is given as an int:

```
int r = ((colour) >> 16) & 0x000000FF;
int g = ((colour) >> 8) & 0x000000FF;
int b = (colour) & 0x000000FF;
```
Software So Far

A DisplayPanel is derived from JPanel and allows us to display the output image on the screen.

BlobTracker extends JFrame to make a window on the screen holding a DisplayPanel. It contains the main() method.

Subtracting the Background
Strategy Pattern

There are lots of algorithms for background subtraction
- We should structure our software to make it easy to select between multiple algorithms
- This of course means using the Strategy pattern

```
import java.util.List;

public interface BackgroundSubtractor {
    public List<Pixel> Subtract(int[] pixels);
}
```

- We provide an array of pixels (in ARGB as discussed)
- We get back a List of Pixels representing the foreground pixels

- Class Pixel just stores an (x,y) pair – we’ll come back to it
### SimpleBackgroundSubtractor

- Remember the notion of RGB as a space?
- We have two readings for each pixel – the saved background reading and the latest webcam reading
- We treat them as two vectors \((rb, gb, bb)\) and \((rw, gw, gb)\)
  - Then we compute the Euclidian distance apart in rgb space
  - If it’s greater than some threshold, we take it as different to the background
  - The threshold allows us to account for noise which is there even for a static background

### Things to Note

- The background image is ‘saved’ using clone() the first time we get a picture
  ```java
  if (mBackground==null)
  {
    mBackground = pixels.clone();
    return new LinkedList<Pixel>();
  }
  ```

- Arrays have clone() implemented by default (shallow)
  - This is an array of primitive ints so that’s all we need
Things to Note

§ Always think about efficiency – avoid expensive calls if you can (e.g. sqrt):

```java
LinkedList<Pixel> foregroundlist = new LinkedList<Pixel>();
for (int i=0; i<pixels.length; i++) {
    int r = ((pixels[i]>>16) & 0x000000FF);
    ...
    int distsq = (r-br)*(r-br) + (b-bb)*(b-bb) + (g-bg)*(g-bg);
    if (distsq > mThreshold*mThreshold) {
        foregroundlist.add(new Pixel(i, mlmageWidth));
    }
}
```

Region Growing
Why Bother?

- There is always so much noise that it isn’t enough to just count the foreground pixels and take that as an indicator of the size of object in view
- We need to find regions in the image where all the adjacent pixels are marked as foreground
- So how to we go from a list of foreground pixels to lists of neighbouring, connected pixels?
- Unsurprisingly, there are lots of algorithms...

Strategy Pattern Again

```
<<Interface>> BlobTracker
  
<<Interface>> RegionExtractor
  extractRegions()

<<Interface>> FrameInterface

RecursiveRegionExtractor
  extractRegions()

ScanningRegionExtractor
  extractRegions()
```

```
<<Interface>> JMFrameSource

1
```

```
<<Interface>> BlobTracker
```

```
<<Interface>> FrameInterface
```

```
<<Interface>> JMFrameSource
```
RegionExtractor

```java
public interface RegionExtractor {
    public List<Region> extractRegions(List<Pixel> pixels);
}

// We get back a List of Regions
// Choose a List because we will want ordering
// Sort by size
// Remove small regions (noise)
// How will the program know to sort the Regions by size?
// We have to tell it
```

Region

```java
// To sort objects, there must be a way to compare them
// Java offers us the Comparable interface

public class Region extends LinkedList<Pixel> implements Comparable<Region>
{
    public int compareTo(Region r) {
        if (this.size() > r.size()) return -1;
        if (this.size() < r.size()) return 1;
        return 0;
    }
}
```
Region

§ Now we can use Region in structures that sort
§ Either we use a structure that is always sorted
  (TreeSet, keys in a TreeMap, etc.)
§ Or we use the static sort() method in Collections

List<Region> regionlist = mRegionExtractor.extractRegions(fgpixels);

Iterator<Region> it = regionlist.iterator();
while (it.hasNext()) {
    Region r = it.next();
    if (r.size()<10) it.remove();
}
Collections.sort(regionlist);

RecursiveRegionExtractor

§ So how do we get Regions anyhow?
§ Start by translating the list of foreground pixels to an
  array of ints because it’s easier to search through
§ -1 means the cell is background
§ 0 means the cell is believed to be foreground
First we write a function that, given a pixel that is foreground:
- Labels it with a region number
- Runs itself on any neighbouring pixels that are foreground
- See `extractRegion(...)`

![RecursiveRegionExtractor](image.png)

-1 -1 -1  -1 -1 -1  -1 -1 -1  
-1 0 -1  -1 1 -1  -1 1 -1  
0 0 -1  0 0 -1  0 1 -1  

Start on (1,1)  Mark (1,1) with a unique region ID  Run function on (1,2)
- Look at the neighbours to see whether (1,1) has any foreground neighbours

Just these three pixels need a series of recursive calls to `extractRegion` (recursive = calls itself)

- `extractRegion (Pixel: 1 1 Label 1)`
- `extractRegion (Pixel: 1 2 Label 1)`
- `extractRegion (Pixel: 0 2 Label 1)`

So, the bigger the region, the more function calls Java has to keep track of simultaneously

Things can go wrong...
- We see a StackOverflowException
- Always a potential problem with recursive functions
## ScanningRegionExtractor

- In reality we want a non-recursive algorithm
  - No StackOverflow
  - Better performance anyway
  - Much easier to debug!!

- I have implemented such an algorithm for you in `ScanningRegionExtractor.java`
  - It’s a neat algorithm but I don’t intend to go through it here
  - Can you work it out and describe it in < 150 words?

## Making it all Fly...

![Blob Tracker](image.png)