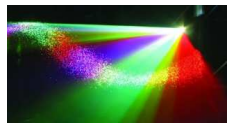
Chapter 5
Sampled Static Visual Content

# The Design and Implementation of Multimedia Software

David Bernstein

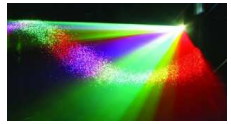
Jones and Bartlett Publishers

www.jbpub.com



# About this Chapter

- The sampling of static visual content involves the sampling of both the color spectrum and space (usually in that order)
- The two are similar in that they both involve the discretization of continuous information.
- They are different in their dimensionality.

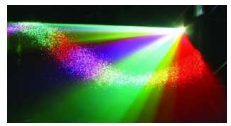


# Color Sampling

## Definition

*Color sampling* is a process for converting from a continuous (infinite) set of colors to a discrete (finite) set of colors.

- Such processes are sometimes called *quantization schemes*.
- The result of color sampling (i.e., the discrete set of colors) is often referred to as a *palette*

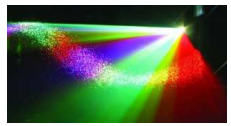


# Spatial Sampling

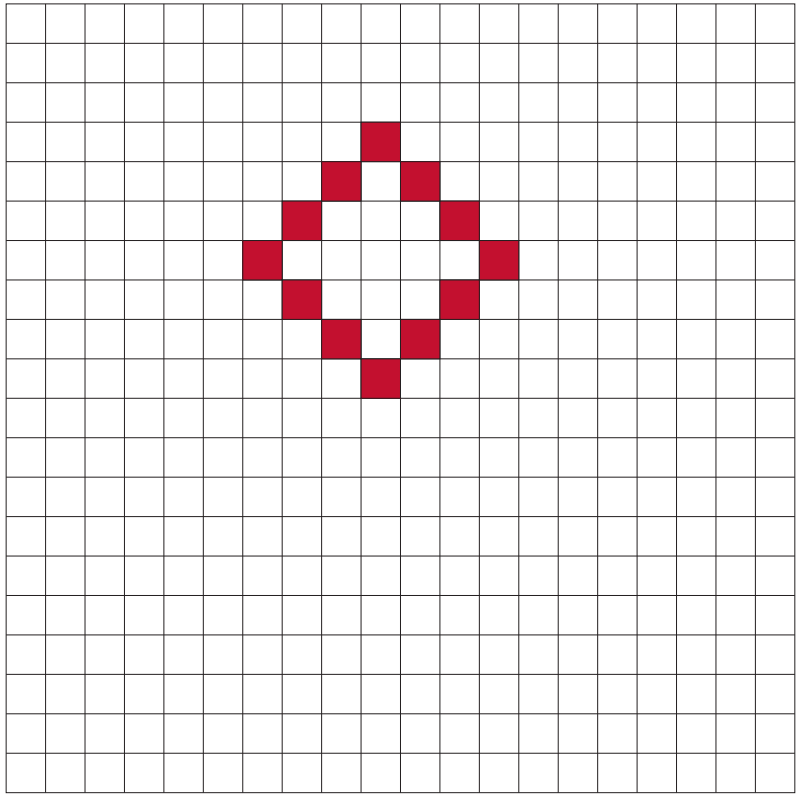
## Definition

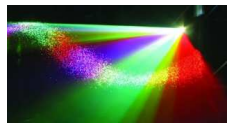
*Spatial sampling* is a process for discretizing space.

- In essence, one places a regular grid on a planar surface and assigns a single color to each cell in the grid.
- The result of color sampling and spatial sampling is a matrix/table of picture elements (or pixels).
- Such a matrix/table is often called a *raster representation*.



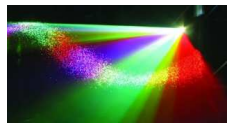
# A Raster Representation





# What's Next

Some immediate gratification.



# Sampled Static Visual Content in Java

- The Encapsulation:

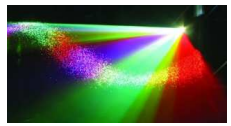
  `Image`

- Creating an `Image`:

  Use the `read()` method in the `ImageIO`

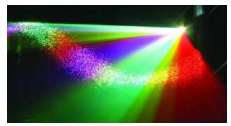
- Rendering an `Image`:

  Use the `drawImage()` method in the `Graphics` class.



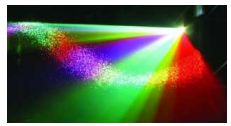
# Structure of the ImageCanvas Class

```
import java.awt.*;
import javax.swing.*;

public class ImageCanvas extends JComponent
{
  private static final long serialVersionUID = 1L;

  private Image        image;

  public ImageCanvas(Image image)
  {
    this.image = image;
  }
}
```



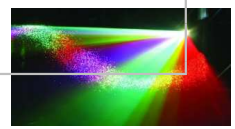
# paint() in ImageCanvas

```
public void paint(Graphics g)
{
  Graphics2D         g2;

  // Cast the rendering engine appropriately
  g2 = (Graphics2D)g;

  // Render the image
  g2.drawImage(image,  // The Image to render
      0,       // The horizontal coordinate
      0,       // The vertical coordinate
      null);  // An ImageObserver
}
```



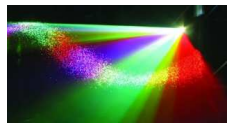
# Structure of the ImageCanvasDemo Class

```
import java.awt.Image;
import java.io.*;
import javax.imageio.*;
import javax.swing.*;

import app.*;
import io.*;

public class ImageCanvasDemo extends JApplication
{
  private String                    name;


  public static void main(String[] args)
  {
    JApplication demo = new ImageCanvasDemo(args, 640, 480);
    invokeInEventDispatchThread(demo);
  }

  public ImageCanvasDemo(String[] args, int width, int height)
  {
    super(args, width, height);

    if (args.length >= 1) name = args[0];
    else                  name = "scribble.png";
  }
}
```



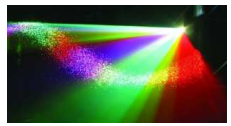
# The `init()` Method

```
public void init()
{
  Image                    image;
  InputStream              is;
  ResourceFinder           finder;
```



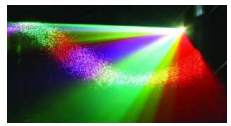
# The `init()` Method (cont.)

```
// Construct a ResourceFinder
finder = ResourceFinder.createInstance(new resources.Marker());

// Read the image
image  = null;
try
{
  is = finder.findInputStream(name);
  if (is != null)
  {
    image = ImageIO.read(is);
    is.close();
  }
}
catch (IOException io)
{
  // image will be null
}
```

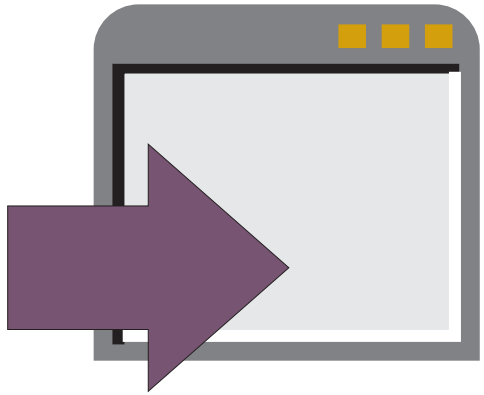


# The `init()` Method (cont.)

```
    // Create the component that will render the image
    ImageCanvas canvas      = new ImageCanvas(image);
    canvas.setBounds(0,
        0,
        image.getWidth(null),
        image.getHeight(null));

    // Add the ImageCanvas to the main window
    JPanel contentPane = (JPanel)getContentPane();
    contentPane.add(canvas);
  }
```

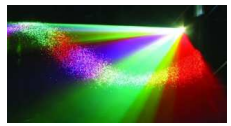


# ImageCanvasDemo - Demonstration



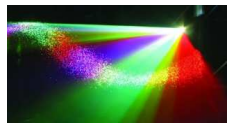
In `examples/chapter`:

```
java -cp multimedia2.jar:examples.jar ImageCanvasDemo woods.png
```



# What's Next

We need to consider the encapsulation of sampled static visual content.



# The `BufferedImage` Class

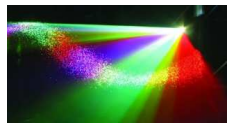
A `BufferedImage` object has two important attributes.

- `ColorModel`:

  Contains methods for translating a pixel value (i.e., the value contained in a particular grid cell) into color components

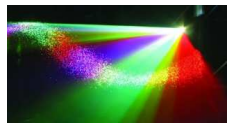
- `Raster`:

  Contains the image data (i.e., the pixel values).



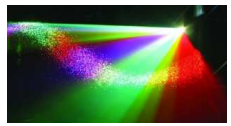
# An Example: Making a Copy

```
private void copy(BufferedImage src, BufferedImage dst)
{
  ColorModel       dstColorModel, srcColorModel;
  int              dstRGB, height, srcRGB, width;
  int[]            srcColor;

  width         = src.getWidth();
  height        = src.getHeight();
  srcColorModel = src.getColorModel();
  srcColor      = new int[4];
  dstColorModel = dst.getColorModel();

  for (int x=0; x<width; x++)
  {
    for (int y=0; y<height; y++)
    {
      srcRGB = src.getRGB(x, y);
      srcColorModel.getComponents(srcRGB,srcColor,0);
      dstRGB = dstColorModel.getDataElement(srcColor,0);
      dst.setRGB(x, y, dstRGB);
    }
  }
}
```



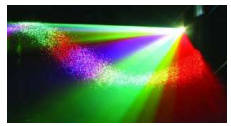
# Creating a BufferedImage from an Image

```
int                   type;
if (channels == 3) type = BufferedImage.TYPE_INT_RGB;
else               type = BufferedImage.TYPE_INT_ARGB;
```



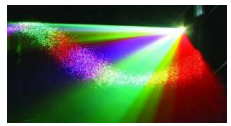
# Creating a BufferedImage from an Image (cont.)

```
BufferedImage       bi;
bi = null;
bi = new BufferedImage(image.getWidth(null),
    image.getHeight(null),
    type);
```



# Creating a BufferedImage from an Image (cont.)

```
Graphics2D          g2;
g2 = bi.createGraphics();
g2.drawImage(image, null, null);
```



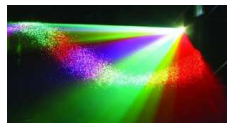
# Some Requirements

It would be both tedious and repetitive to include this code in every class that needs to work with an `Image`. Instead, it would be better to have a system that can:

F5.1 Create a `BufferedImage` from an `Image`.

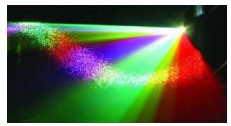
F5.2 Create a `BufferedImage` from a file containing an `Image` of any kind.

F5.3 Create either RGB or $\alpha$RGB images.



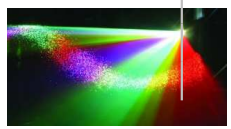
# The ImageFactory Class

```
public BufferedImage createBufferedImage(Image image,
    int   channels)
{
  int                 type;
  if (channels == 3) type = BufferedImage.TYPE_INT_RGB;
  else               type = BufferedImage.TYPE_INT_ARGB;
  BufferedImage       bi;
  bi = null;
  bi = new BufferedImage(image.getWidth(null),
      image.getHeight(null),
      type);
  Graphics2D          g2;
  g2 = bi.createGraphics();
  g2.drawImage(image, null, null);
  return bi;
}
```



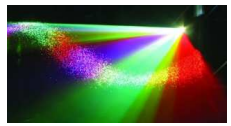
# More of the ImageFactory Class

```
public BufferedImage createBufferedImage(String name,
    int channels)
{
  BufferedImage   image, result;
  InputStream     is;
  int             imageType;


  image = null;
  is    = finder.findInputStream(name);

  if (is != null)
  {
    try
    {
      image       = ImageIO.read(is);
      is.close();
    }
    catch (IOException io)
    {
      image       = null;
    }
  }

  // Modify the type, if necessary
  result = image;
  if (image != null)
  {
```



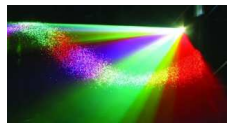
# More of the ImageFactory Class (cont.)

```
    imageType = image.getType();
    if (((channels == 3) &&
        (imageType != BufferedImage.TYPE_INT_RGB)) ||
        ((channels == 4) &&
            (imageType != BufferedImage.TYPE_INT_ARGB))    )
    {
      result = createBufferedImage(image, channels);
    }
  }

  return result;
}
```



# What's Next

We need to consider operating on sampled static visual content.



# Single-Input/Single-Output Operations

- The Interface:

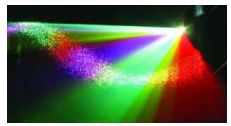
  `BufferedImageOp`

- The Method:

  ```
  public BufferedImage filter(BufferedImage src,
  BufferedImage dst)
  ```

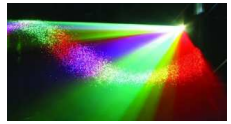
- The Convention:

  If this method is passed an existing `dst`, its pixels are 'filled in' by the operation. If `dst` is `null`, a new image is created and returned.



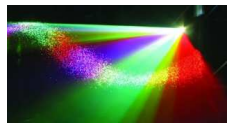
# IdentityOp

```
public BufferedImage createCompatibleDestImage(BufferedImage src,
    ColorModel    dstCM)
{
  BufferedImage      dst;
  int                height, width;
  WritableRaster     raster;

  if (dstCM == null) dstCM = src.getColorModel();

  height = src.getHeight();
  width  = src.getWidth();
  raster = dstCM.createCompatibleWritableRaster(width, height);

  dst = new BufferedImage(dstCM, raster,
      dstCM.isAlphaPremultiplied(), null);

  return dst;
}
```



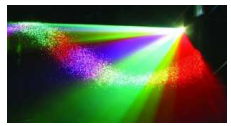
# IdentityOp (cont.)

```
public BufferedImage filter(BufferedImage src,
    BufferedImage dst)
{
  // Construct the destination image if one isn't provided
  if (dst == null)
    dst = createCompatibleDestImage(src, src.getColorModel());

  // Copy the source to the destination
  copy(src, dst);

  // Return the destination (in case it is new)
  return dst;
}
```



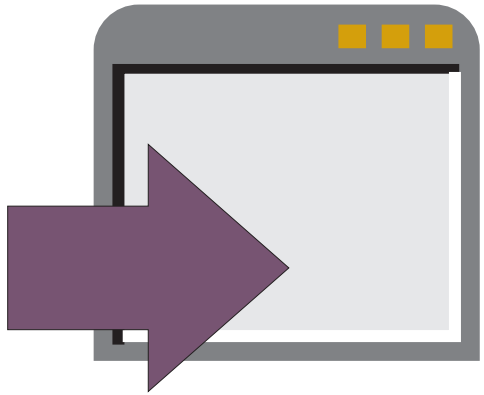
# IdentityOp (cont.)

```
public Rectangle2D getBounds2D(BufferedImage src)
{
  Raster raster = src.getRaster();

  return raster.getBounds();
}

public Point2D getPoint2D(Point2D srcPt, Point2D dstPt)
{
  if (dstPt == null) dstPt = (Point2D)srcPt.clone();
  dstPt.setLocation(srcPt);

  return dstPt;
}

public RenderingHints getRenderingHints()
{
  return null;
}
```



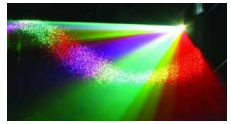
# GrayExceptOp

Convert a color image to a gray scale image, while preserving one particular color.



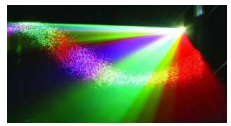
In `extras`:

```
java -cp multimedia2.jar:examples.jar ImageOperationsDemo firstsnow.png GrayExcept
```



# Structure of GrayExceptOp

```
import java.awt.image.*;

import math.*;

public class GrayExceptOp extends IdentityOp
{
  private int[]              highlightColor;

}
```



# "One Particular Color"

- Meaning:

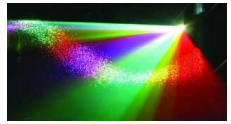
  All of the colors that are close to one RGB combination.

- Method:

  Use a *metric* (or distance) of some kind.

- Design:

  There are alternatives to consider.



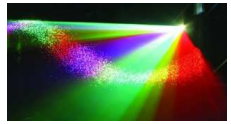
# Alternative 1

- Overview:

  Include the algorithm for calculating the distance in the `areSimilar()` method.

- Shortcomings:

  What are the shortcomings?



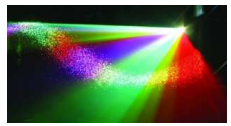
# Alternative 1 

- Overview:

  Include the algorithm for calculating the distance in the `areSimilar()` method.

- Shortcomings:

  It is inflexible (i.e., the only way to change the metric is to modify the `areSimilar()` method.



# Alternative 2 

- Overview:

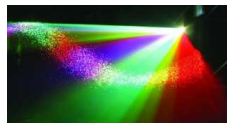
  Use the strategy pattern.

- Details:

  Create a `Metric` interface and a concrete realizations (e.g., a `RectilinearMetric` class).

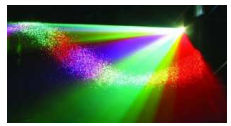
  Add a `Metric` attribute.

  Use the `Metric` attribute in the `areSimilar()` method.



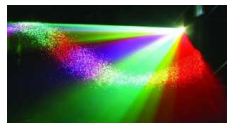
# The `Metric` Interface

```
package math;

public interface Metric
{
  public abstract double distance(double[] x, double[] y);
}
```



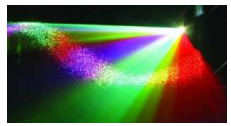
# The RectilinearMetric Class

```
package math;

public class      RectilinearMetric
       implements Metric
{
  public double distance(double[] x, double[] y)
  {
    double  result;
    int     n;

    result = 0.0;
    n      = Math.min(x.length, y.length);

    for (int i=0; i<n; i++)
    {
      result += Math.abs(x[i]-y[i]);
    }

    return result;
  }
}
```



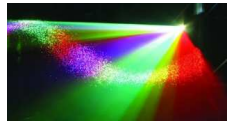
# Constructors in GrayExceptOp

```
public GrayExceptOp(int r, int g, int b)
{
  this(r, g, b, new RectilinearMetric());
}

public GrayExceptOp(int r, int g, int b, Metric metric)
{
  highlightColor    = new int[3];
  highlightColor[0] = r;
  highlightColor[1] = g;
  highlightColor[2] = b;

  this.metric = metric;
}
```



# areSimilar() in GrayExceptOp

```
private boolean areSimilar(int[] a, int[] b)
{
  boolean        result;
  double         distance;

  for (int i=0; i<3; i++)
  {
    x[i] = a[i];
    y[i] = b[i];
  }

  result   = false;
  distance = metric.distance(x, y);

  if (distance <= TOLERANCE) result = true;

  return result;
}
```



# `filter()` in `GrayExceptOp`

- Purpose:

  If the source pixel is close to the color being preserved copy the pixel, otherwise create a gray value with the appropriate luminance.

- Two Observations:

$$R_g + G_g + B_g = R_c + G_c + B_c \tag{1}$$

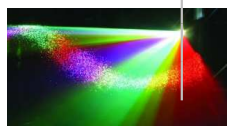
$$R_g = G_g = B_g \tag{2}$$

- The Implication:

$$R_g = G_g = B_g = (R_c + G_c + B_c)/3 \tag{3}$$

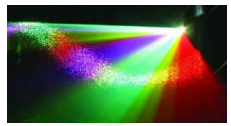
# Implementing `filter()` in `GrayExceptOp`

```
public BufferedImage filter(BufferedImage src,
    BufferedImage dest)
{
  ColorModel       destColorModel, srcColorModel;
  int              grayRGB, highlightRGB, srcRGB, srcHeight, srcWidth;
  int[]            gray, srcColor;

  srcWidth      = src.getWidth();
  srcHeight     = src.getHeight();
  srcColorModel = src.getColorModel();
  srcColor      = new int[4];
  gray          = new int[4];

  if (dest == null) dest = createCompatibleDestImage(src, srcColorModel);

  destColorModel = dest.getColorModel();
  highlightRGB   = destColorModel.getDataElement(highlightColor, 0);

  for (int x=0; x<srcWidth; x++)
  {
    for (int y=0; y<srcHeight; y++)
    {
      srcRGB = src.getRGB(x, y);
      srcColorModel.getComponents(srcRGB, srcColor, 0);

      if (areSimilar(srcColor, highlightColor))
      {
        dest.setRGB(x, y, highlightRGB);
```



# Implementing `filter()` in `GrayExceptOp` (cont.)

```
        }
        else
        {
          gray[0]=(srcColor[0]+srcColor[1]+srcColor[2])/3;
          gray[1]=gray[0];
          gray[2]=gray[0];
          grayRGB=destColorModel.getDataElement(gray,0);
          dest.setRGB(x, y, grayRGB);
        }
      }
    }
    return dest;
  }
```

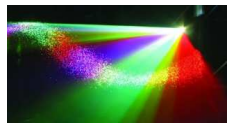


# Defined

In its most general usage, a convolution is a product of two functions. As it is being used here:

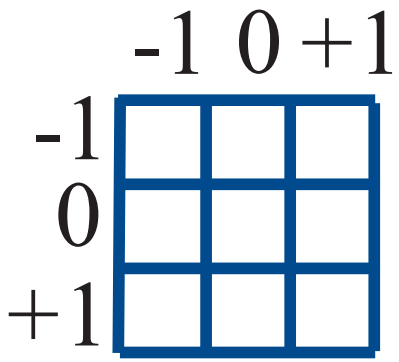
## Definition

A *convolution* is a way of calculating the color of a pixel in a destination image from that same pixel and its neighbors in a source image.

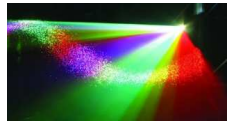


# Convolution Kernels

A *kernel* contains weights that are applied to the source pixels to obtain the destination pixels.



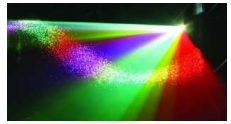

A $3 \times 3$ Convolution Kernel



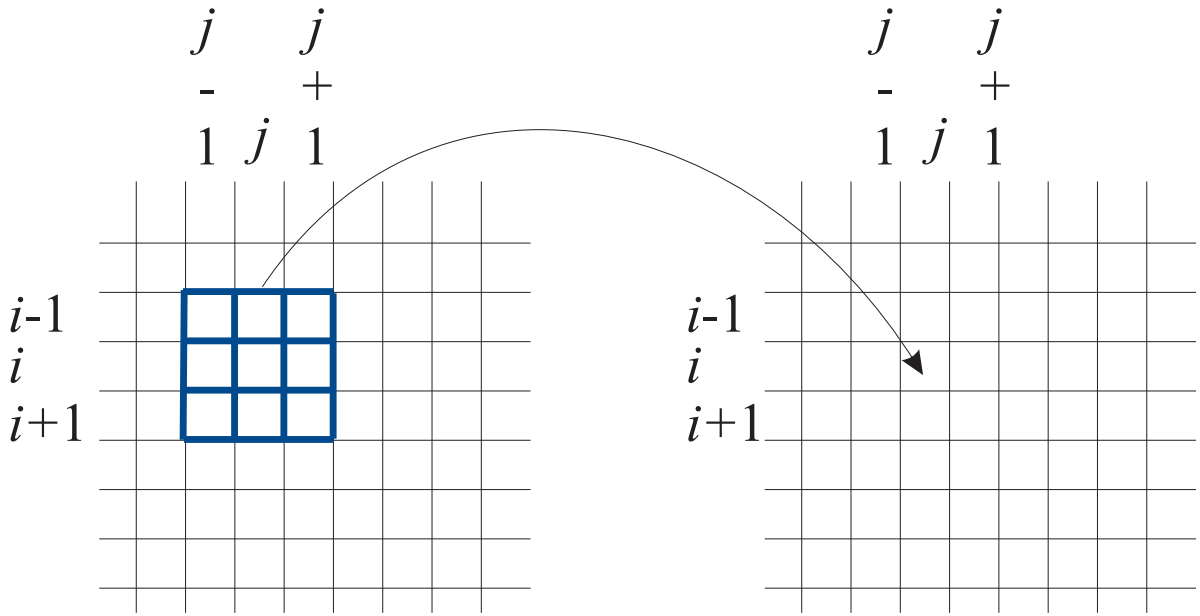
# Convolution Kernels (cont.)

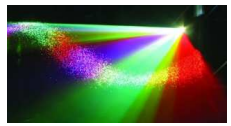
Letting $s_{i,j}$ denote the value in source pixel $(i, j)$, $k_{r,c}$ denote the value in element $(r, c)$ of the kernel, and $d_{i,j}$ denote the value in destination pixel $(i, j)$, a $3 \times 3$ spatial convolution can be defined as follows:

$$d_{i,j} = \sum_{r=-1}^{1} \sum_{c=-1}^{1} s_{i+r,j+c} k_{r,c} \tag{4}$$

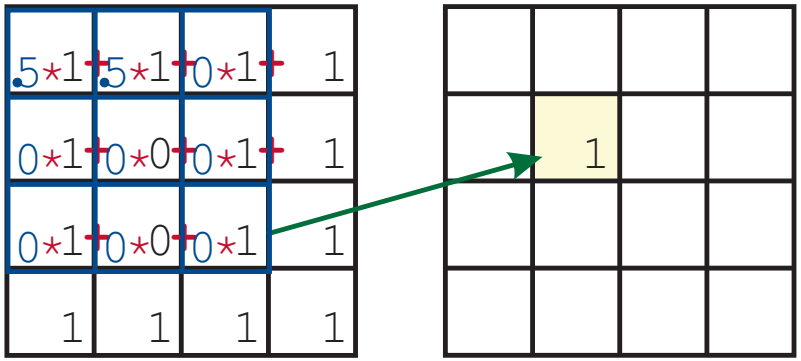


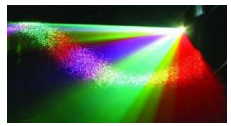
# Obtaining One Pixel in the Destination Image



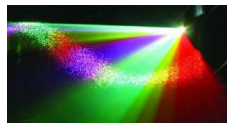
# Obtaining One Pixel in the Destination Image (cont.)



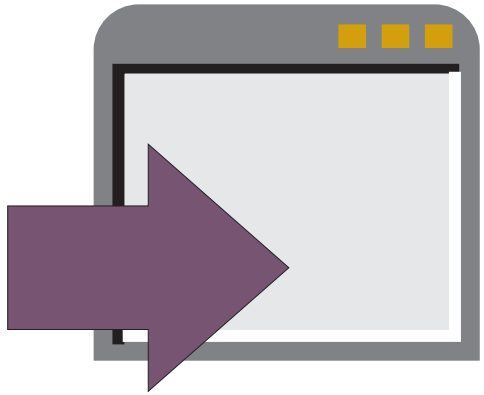



# The Identity Kernel

| | | |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | 0 |

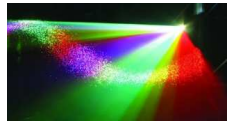


# Blurring - Demonstration



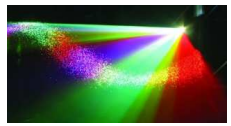
In `extras`:

```
java -cp multimedia2.jar:examples.jar ImageOperationsDemo wilson-hall.png Blur
```



# A Blurring Kernel

| | | |
|---|---|---|
| 1/9 | 1/9 | 1/9 |
| 1/9 | 1/9 | 1/9 |
| 1/9 | 1/9 | 1/9 |

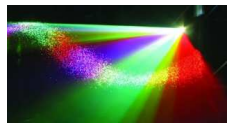


# A Blurring Kernel (cont.)

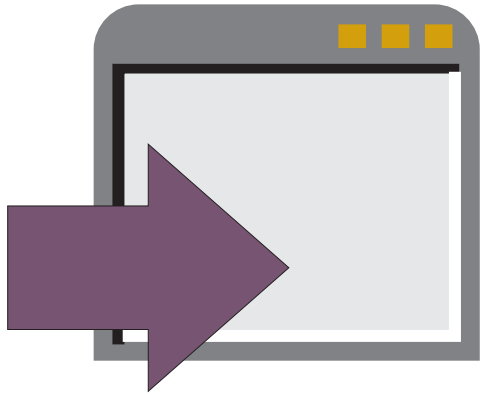
| | | | |
|---|---|---|---|
| 1 | 1 | 1 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

| | | |
|---|---|---|
| 1/9 | 1/9 | 1/9 |
| 1/9 | 1/9 | 1/9 |
| 1/9 | 1/9 | 1/9 |

| | | | |
|---|---|---|---|
| | | | |
| | 7/9 | 7/9 | |
| | 7/9 | 7/9 | |
| | | | |

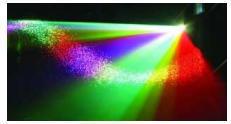


# Edge Detection - Demonstration



In `extras`:

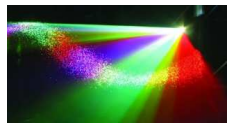
```
java -cp multimedia2.jar:examples.jar ImageOperationsDemo wilson-hall.png Edge
```



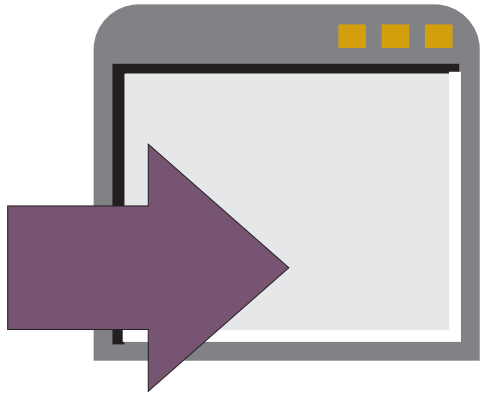
# An Edge-Detection Kernel

| | | |
|---|---|---|
| 0 | -1 | 0 |
| -1 | 4 | -1 |
| 0 | -1 | 0 |

- Suppose the value of the source pixel and its neighbors is given by $x$.
- Suppose the source pixel is brighter than its vertical and horizontal neighbors.

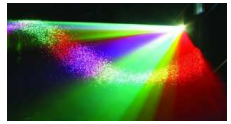


# Sharpening - Demonstration



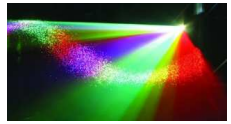
In `extras`:

```
java -cp multimedia2.jar:examples.jar ImageOperationsDemo firstsnow.png Sharpen
```



# A Sharpening Kernel

- Find the edges (i.e., start with an edge detection kernel).
- Add the edges to the original image (i.e., add an identity kernel).

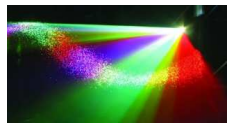


# Convolutions in Java

- `ConvolveOp`

  Implements the `BufferedImageOp` interface

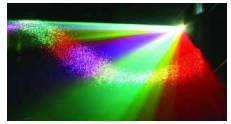
- `Kernel`



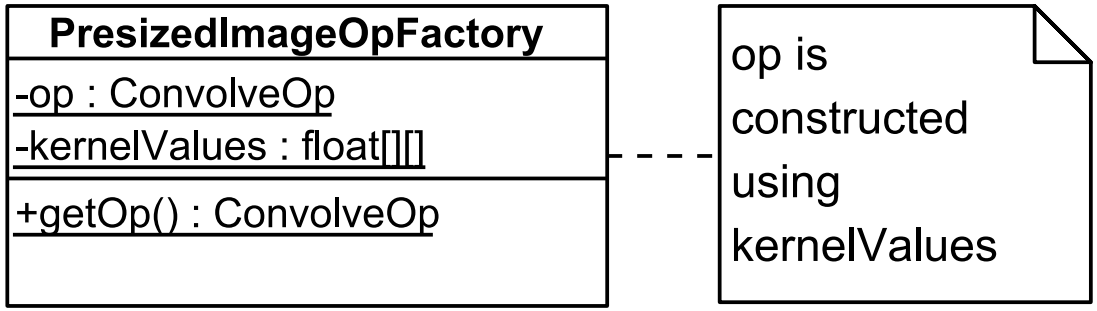
# Additional Requirements

F5.4 Create different `BufferedImageOp` objects.

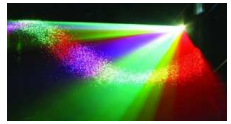
N5.5 Conserve the amount of memory used by kernels.



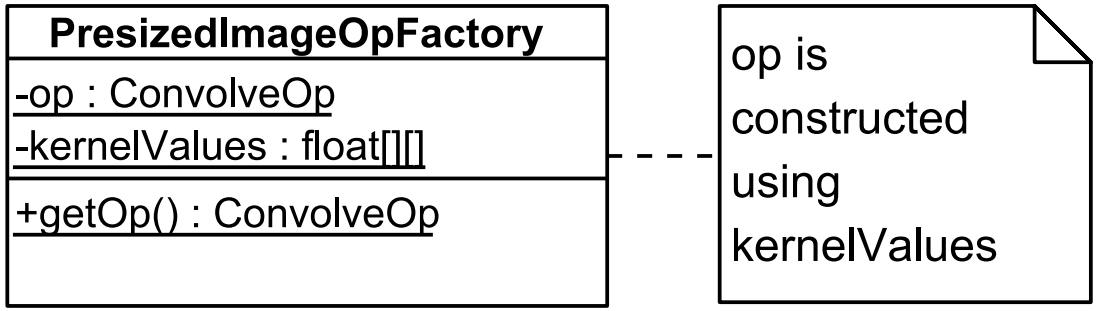
# Alternative 1



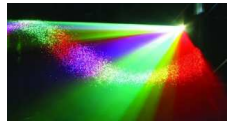

What are the shortcomings?



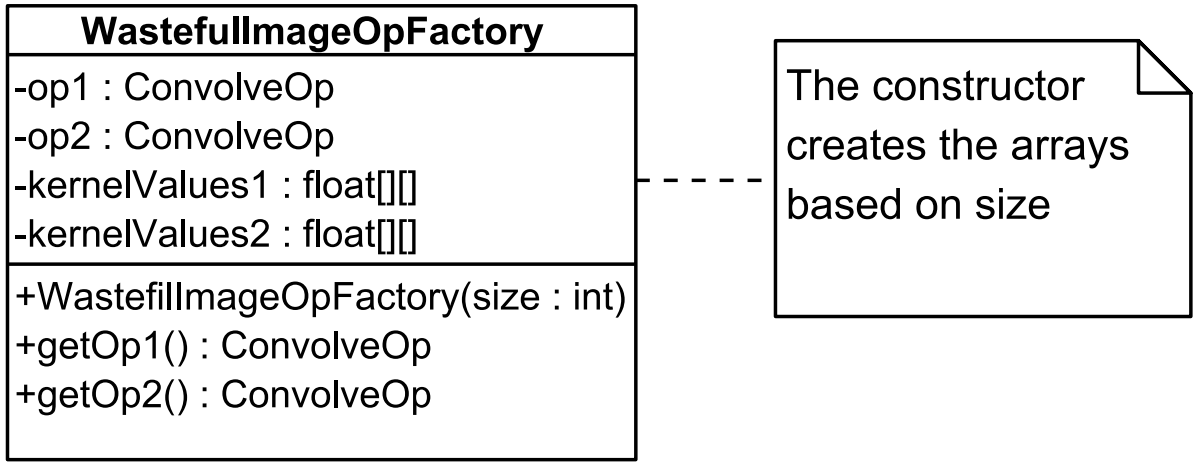
# Alternative 1



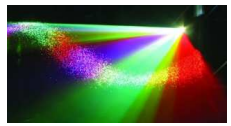

All of the kernels have to be pre-sized.



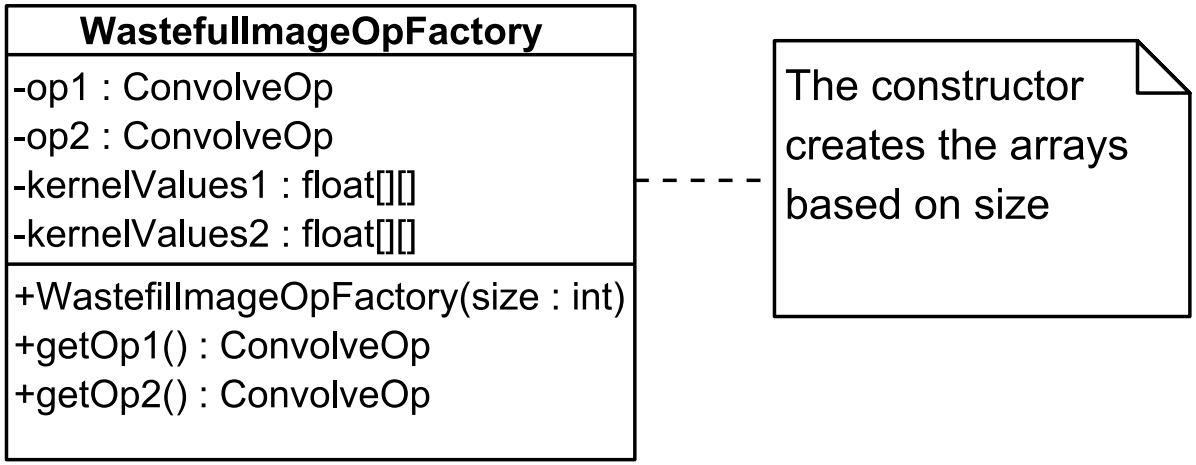
# Alternative 2



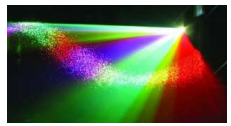

What are the shortcomings?



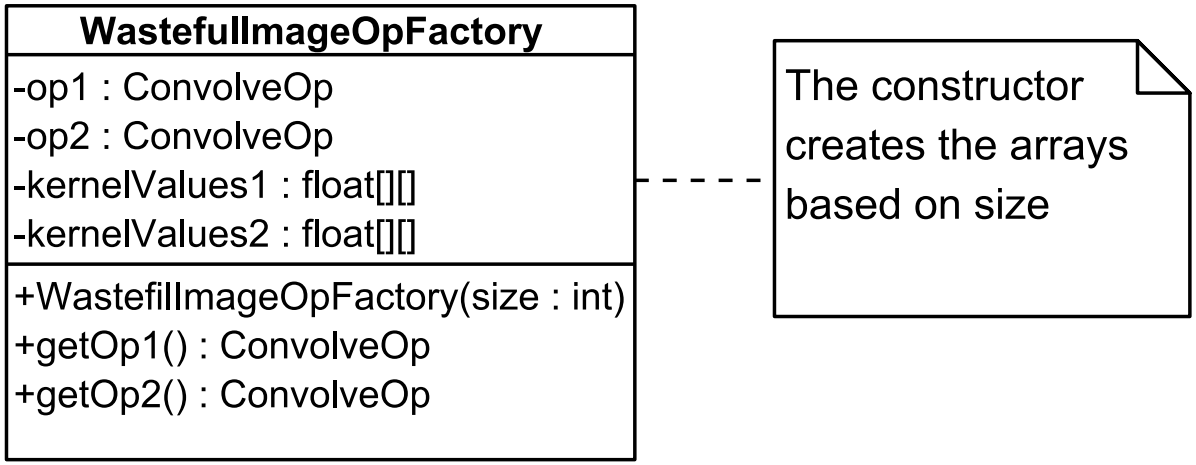
# Alternative 2




One factory must be constructed for each size and each must allocate memory for all of the kernel values.

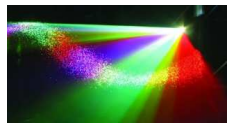


# Alternative 2

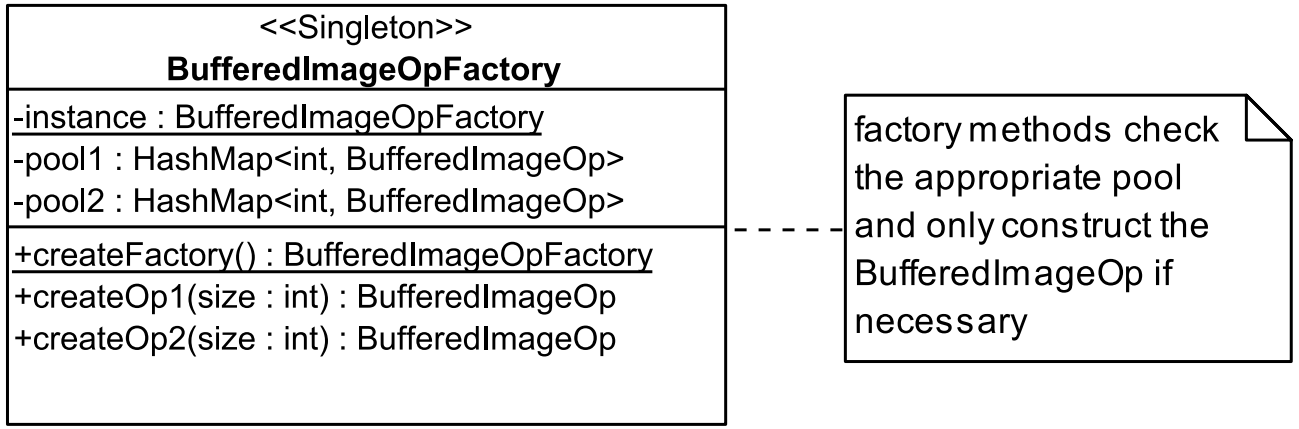


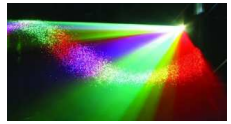

One factory must be constructed for each size and each must allocate memory for all of the kernel values.

`ConvolveOp` objects are size-dependent but other objects that implement `BufferedImageOp` are not.



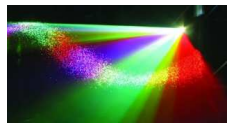
# Alternative 3



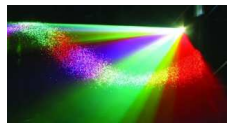
# Alternative 3.1

```
public ConvolveOp createBlurOp(int size)
{
   ConvolveOp    op;
   float         denom;
   float[]       kernelValues;
   Integer       key;

   key = new Integer(size);
   op  = blurOps.get(key);
   if (op == null)
   {
      kernelValues = getBlurValues(size);

      op = new ConvolveOp(new Kernel(size,size,kernelValues),
                          ConvolveOp.EDGE_NO_OP,
                          null);

      blurOps.put(key, op);
   }

   return op;
}
```



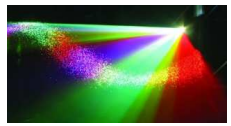
# Alternative 3.1

```
public ConvolveOp createEdgeDetectionOp(int size)
{
   ConvolveOp    op;
   float         denom;
   float[]       kernelValues;
   int           center;
   Integer       key;

   key = new Integer(size);
   op  = edgeOps.get(key);
   if (op == null)
   {
      kernelValues = getEdgeValues(size);

      op = new ConvolveOp(new Kernel(size,size,kernelValues),
                          ConvolveOp.EDGE_NO_OP,
                          null);

      edgeOps.put(key, op);
   }

   return op;
}
```



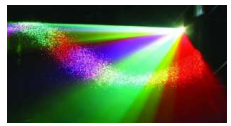
# Alternative 3.1

What are the shortcomings?



# Alternative 3.1

Very repetitive and, as a result, difficult to maintain (i.e., the `createBlurOp()` and `createEdgeDetectionOp()` methods are very similar).



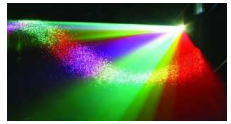
# Alternative 3.2

- Approach:

  Use the command pattern (i.e., pass the command to use to get the kernel values (along with the pool) to a 'generic' `createOp()` method.

- Shortcomings:

  What are the shortcomings?



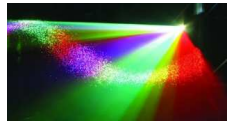
# Alternative 3.2

- Approach:

  Use the command pattern (i.e., pass the command to use to get the kernel values (along with the pool) to a 'generic' `createOp()` method.

- Shortcomings:

  Seems overly complicated.



# Alternative 3.2

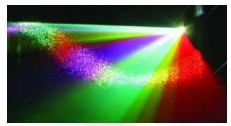
- Approach:

  Use the command pattern (i.e., pass the command to use to get the kernel values (along with the pool) to a 'generic' `createOp()` method.

- Shortcomings:

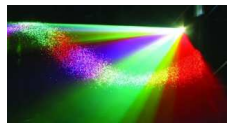
  Seems overly complicated.

  Results in a large number of classes.



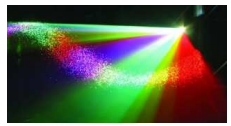
# Alternative 3.3 – Overview

```
private ConvolveOp createOp(Convolutions type, int size)
{
  ConvolveOp                          op;
  HashMap<Integer, ConvolveOp>        pool;
  Integer                             key;

  key  = new Integer(size);
  pool = convolutionPools.get(type);
  op   = pool.get(key);

  if (op == null)
  {
    op = new ConvolveOp(new Kernel(size,size,type.getKernelValues(size)),
        ConvolveOp.EDGE_NO_OP,null);
    pool.put(key, op);
  }
  return op;
}
```



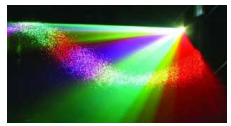
# Alternative 3.3.1

```
package visual.statik.sampled;

enum UnmaintainableConvolutions
{
   BLUR,
   EDGE;

   float[] getKernelValues(int size)
   {
      switch (this)
      {
         case BLUR:     return getBlurValues(size);
         case EDGE:     return getEdgeValues(size);

         default:       return getIdentityValues(size);
      }
   }
}
```



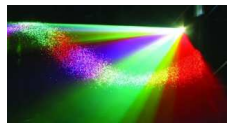
# Alternative 3.3.1  (cont.)

What are the shortcomings?



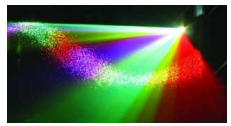
# Alternative 3.3.1 (cont.)

It is difficult to maintain (i.e., one can add an enumerated value and forget to add the corresponding `case`).



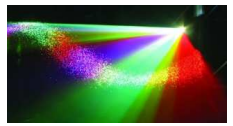
# Alternative 3.3.2 – Structure

```
package visual.statik.sampled;

enum Convolutions
{
  BLUR {float[] getKernelValues(int size)
  {return getBlurValues(size);}
  },
  EDGE {float[] getKernelValues(int size)
  {return getEdgeValues(size);}
  },
  EMBOSS {float[] getKernelValues(int size)
  {return getEmbossValues(size);}
  },
  IDENTITY {float[] getKernelValues(int size)
  {return getIdentityValues(size);}
  },
  SHARPEN {float[] getKernelValues(int size)
  {return getSharpenValues(size);}
  };

  abstract float[] getKernelValues(int size);
}
```



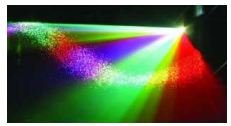
# Alternative 3.3.2 – getBlurValues()

```
private static float[] getBlurValues(int size)
{
  float       denom;
  float[]     result;

  denom  = (float)(size*size);
  result = new float[size*size];

  for (int row=0; row<size; row++)
    for (int col=0; col<size; col++)
      result[indexFor(row,col,size)] = 1.0f/denom;

  return result;
}
```



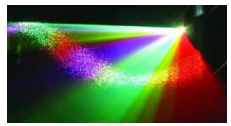
# Alternative 3.3.2 – getEdgeValues()

```
private static float[] getEdgeValues(int size)
{
  float[]     result;
  int         center;

  center = size/2;
  result = new float[size*size];

  result[indexFor(center-1, center  , size)] = -1.0f;
  result[indexFor(center  , center-1, size)] = -1.0f;
  result[indexFor(center  , center  , size)] =  4.0f;
  result[indexFor(center  , center+1, size)] = -1.0f;
  result[indexFor(center+1, center  , size)] = -1.0f;

  return result;
}
```



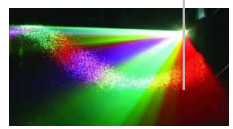
# Alternative 3.3.2 – indexFor()

```
private static int indexFor(int row, int col, int size)
{
  return row*size + col;
}
```



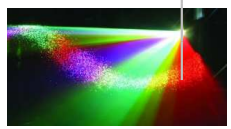
# Alternative 3.3 – Structure

```
package visual.statik.sampled;

import java.awt.*;
import java.awt.color.*;
import java.awt.geom.*;
import java.awt.image.*;
import java.util.*;

public class BufferedImageOpFactory
{
  private HashMap<Convolutions,
  HashMap<Integer, ConvolveOp>> convolutionPools;
  private static BufferedImageOpFactory     instance =
      new BufferedImageOpFactory();

  private BufferedImageOpFactory()
  {
    HashMap<Integer, ConvolveOp>    pool;

    // Initialize the pool of ConvolveOp objects
    convolutionPools = new HashMap<Convolutions,
        HashMap<Integer, ConvolveOp>>();

    for (Convolutions type : Convolutions.values())
    {
      pool = new HashMap<Integer, ConvolveOp>();
```



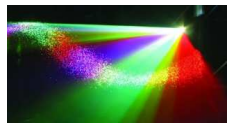
# Alternative 3.3 – Structure (cont.)

```
    convolutionPools.put(type, pool);
  }
  // Initialize the pool of grayExceptOp objects
  grayExceptPool = new HashMap<Integer, GrayExceptOp>();
}

public static BufferedImageOpFactory createFactory()
{
  return instance;
}
private ConvolveOp createOp(Convolutions type, int size)
{
  ConvolveOp                        op;
  HashMap<Integer, ConvolveOp>      pool;
  Integer                           key;

  key  = new Integer(size);
  pool = convolutionPools.get(type);
  op   = pool.get(key);

  if (op == null)
  {
    op = new ConvolveOp(new Kernel(size,size,type.getKernelValues(size)),
        ConvolveOp.EDGE_NO_OP,null);
    pool.put(key, op);
  }
  return op;
```



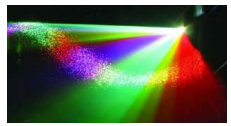
# Alternative 3.3 – Structure (cont.)

```
    }
}
```



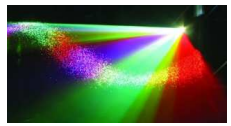
# Alternative 3.3 – createBlurOp()

```
public ConvolveOp createBlurOp(int size)
{
  return createOp(Convolutions.BLUR, size);
}
```



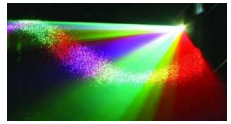
# Alternative 3.3 – createEdgeDetectionOp()

```
public ConvolveOp createEdgeDetectionOp(int size)
{
  return createOp(Convolutions.EDGE, size);
}
```



# Motivation

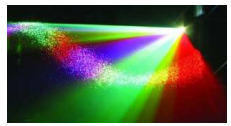
- In many situations, one doesn't want to transform the coordinate space.
- Instead one wants to transform the content itself.



# Affine Transformations in Java

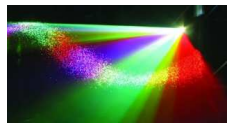
- `AffineTransform`
- `AffineTransformOp`

  Implements the `BufferedImageOp` interface



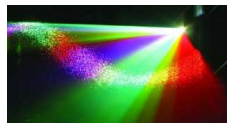
# An Example of Scaling

```
AffineTransform      at;
AffineTransformOp    op;

at = AffineTransform.getScaleInstance(xScale, yScale);
op = new AffineTransformOp(
    at,
    AffineTransformOp.TYPE_BILINEAR);
```



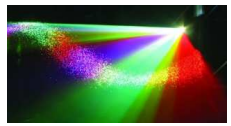
# An Example of Rotation – Getting the Midpoint

```
AffineTransform   rotate;

rotate = AffineTransform.getRotateInstance(
    theta,
    width/2.0,
    height/2.0);
```



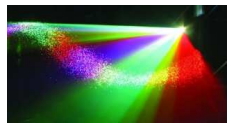
# An Example of Rotation – Rendering Hints

```
RenderingHints           hints;

// Value can be BILINEAR or NEAREST_NEIGHBOR
hints = new RenderingHints(
    RenderingHints.KEY_INTERPOLATION,
    RenderingHints.VALUE_INTERPOLATION_BILINEAR
    );
```

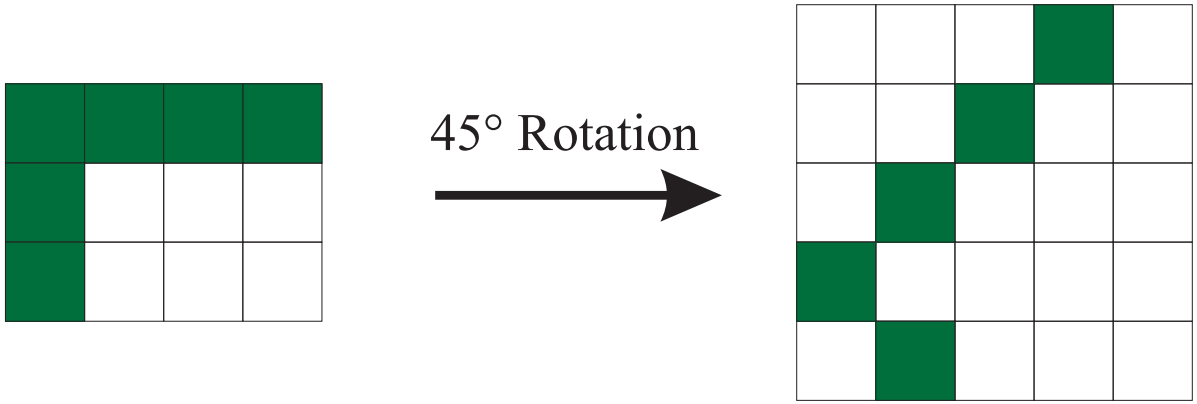


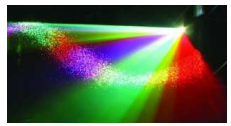
# An Example of Rotation – Create an `AffineTransform`

```
AffineTransformOp        op;

op = new AffineTransformOp(rotate, hints);
```

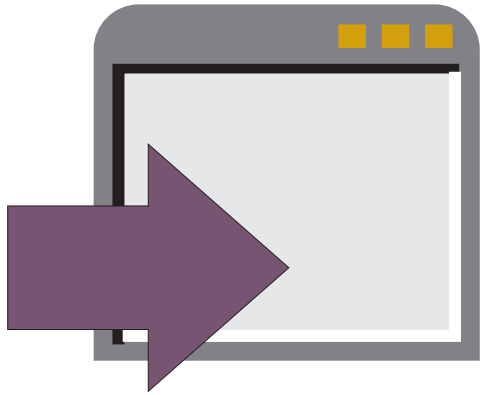


# Rotations Often Change the Size



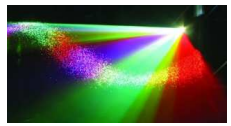



# Rotation - Demonstration



In `extras`:

```
java -cp multimedia2.jar:examples.jar RotationDemo car.png
```



# Component-Independent Lookup Tables

- Size:

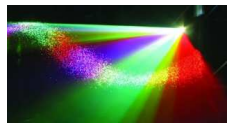
  A single array with as many elements as there are component values (e.g., 256 entries for the component values 0 through 255).

- Components:

  The element at index $i$ contains the destination value for source components with a value of $i$.

- An Example:

  An entry of 156 in element 2 indicates that values of 2 (in the red, green, and blue) components should be changed to 156.



# Component-by-Component Lookup Tables

- Size:

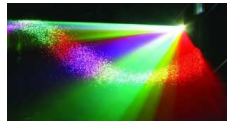
  One array for each color component (e.g., 3 arrays of length 256 for a 24-bit RGB system).

- Components:

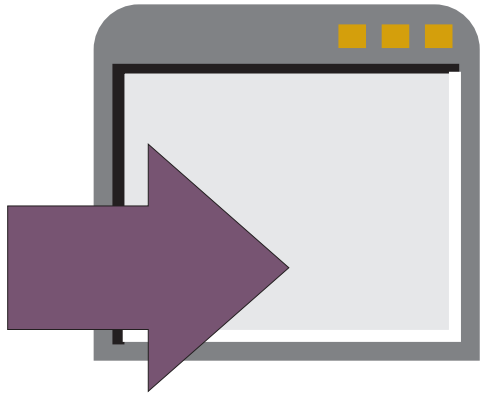
  The element at index $i$ in array $c$ contains the destination value for source components of color $c$ with a value of $i$.

- An Example:

  An entry of 0 in element 255 of the red array means that any red values of 255 should be replaced with 0.

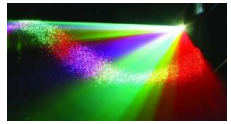


# Creating a Photo Negative - Demonstration



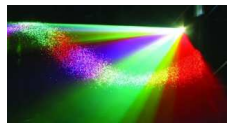
In `extras`:

```
java -cp multimedia2.jar:examples.jar ImageOperationsDemo wilson-hall.png Negative
```

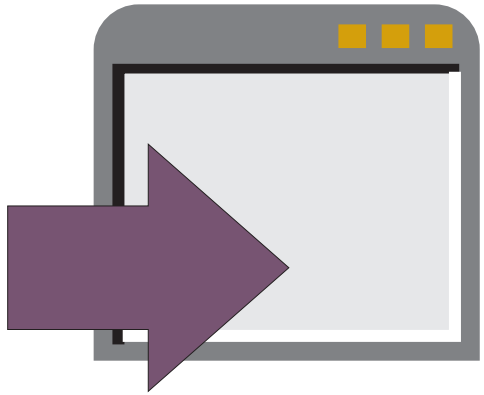


# Creating a Photo Negative

```
LookupTable                lookupTable;
short[]                    lookup;

lookup = new short[256];

for (int i=0; i<lookup.length; i++)
{
  lookup[i] = (short)(255 - i);
}

lookupTable = new ShortLookupTable(0, lookup);
negativeOp  = new LookupOp(lookupTable, null);
```

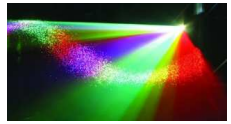


# Night Vision Effect - Demonstration



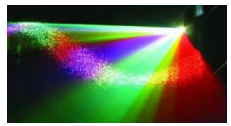
In `extras`:

```
java -cp multimedia2.jar:examples.jar ImageOperationsDemo wilson-hall.png Night
```



# A Night Vision Effect Filter

```
LookupTable               lookupTable;
short[]                   leave, remove;
short[][]                 lookupMatrix;


leave  = new short[256];
remove = new short[256];

for (int i=0; i < leave.length; i++) {

  leave[i]  = (short)(i);
  remove[i] = (short)(0);
}

lookupMatrix    = new short[3][];

lookupMatrix[0] = remove;
lookupMatrix[1] = leave;
lookupMatrix[2] = remove;


lookupTable   = new ShortLookupTable(0, lookupMatrix);
nightVisionOp = new LookupOp(lookupTable, null);
```



# The Basics

- Purpose:

  Apply a linear function to the source pixels when creating the destination pixels.

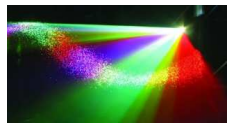
- In Java:

  `RescaleOp`

- Parameters:

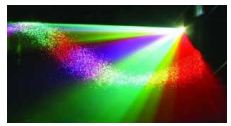
  Scale factor (i.e., the multiplicative parameter)

  Offset (i.e., the additive parameter)

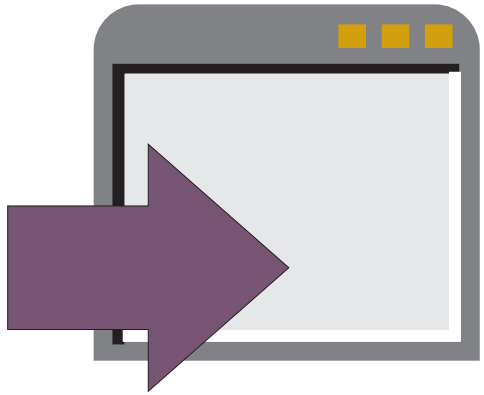


# Examples

```
brightenOp = new RescaleOp(1.5f, 0.0f, null);
darkenOp = new RescaleOp(0.5f, 0.0f, null);
metalOp = new RescaleOp(1.0f, 128.0f, null);
```

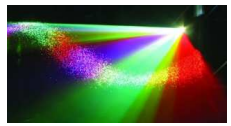


# Darkening - Demonstration

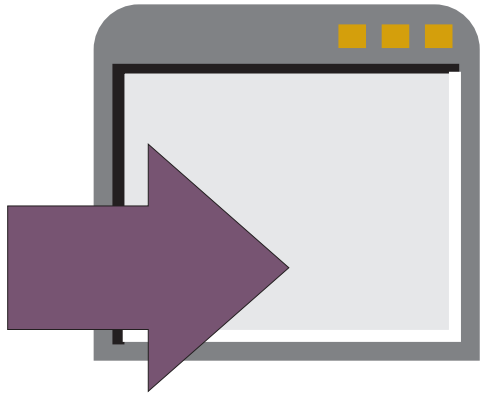


In `extras`:

```
java -cp multimedia2.jar:examples.jar ImageOperationsDemo fall.png Darken
```

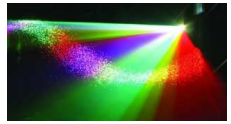


# Brightening - Demonstration

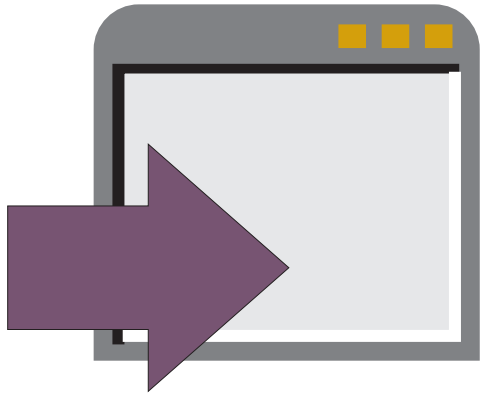


In `extras`:

```
java -cp multimedia2.jar:examples.jar ImageOperationsDemo fall.png Brighten
```

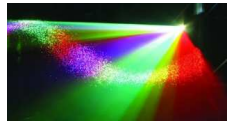


# Metal Effect - Demonstration



In `extras`:

```
java -cp multimedia2.jar:examples.jar ImageOperationsDemo fall.png Metal
```



# The Basics

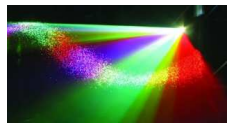
- A `ColorModel` Consists Of:

  A `ColorSpace` (e.g. GRAY, RGB, sRGB, CIEXYZ)

  A color depth (e.g., 24-bit)

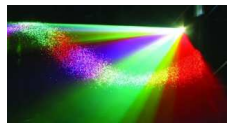
- Conversion:

  Use a `ColorConvertOp`



# An Example

```
public ColorConvertOp createGrayOp()
{
  if (grayOp == null)
  {
    ColorConvertOp          op;
    ColorSpace              cs;

    cs = ColorSpace.getInstance(ColorSpace.CS_GRAY);
    op = new ColorConvertOp(cs, null);
    grayOp = op;
  }
  return grayOp;
}
```



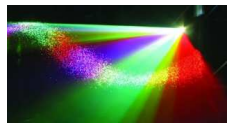
# The Basics

- Defined:

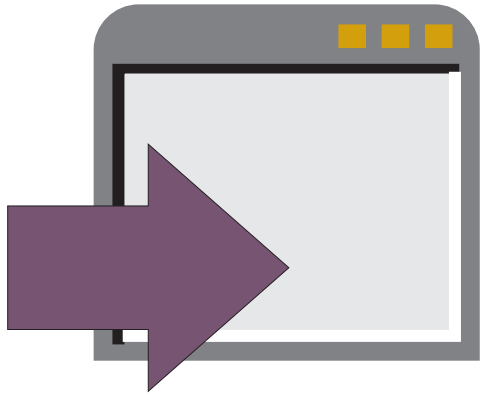
  The extraction of a rectangular sub-image

- In Java:

  Use the `getSubImage()` method in the `BufferedImage` class



# Cropping/Cutting and Zooming - Demonstration

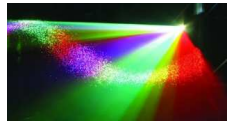


In `extras`:

```
java -cp multimedia2.jar:examples.jar SurveillanceDemo wilson-hall.png
```

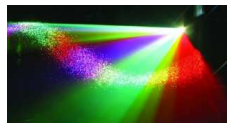
Imagine yourself in a command center and in an excited voice say:

1. Isolate those people [CropSW]
2. Can't you make it larger? [Zoom]
3. They need to be bigger! [CropSE][Zoom]
4. Sharpen it! [Sharpen][Sharpen]



# What's Next

Design of a sampled static visual content system.



# Requirements

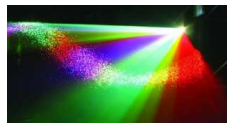
F5.6 Support the addition of sampled static visual content.

F5.7 Support the removal of sampled static visual content.

F5.8 Support $z$-ordering of sampled static visual content.

F5.9 Support the transformation of sampled static visual content.

F5.10 Support the rendering of sampled static visual content.



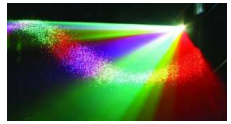
# Alternative 1 

- Approach:

  Specialize the `BufferedImage` class

- Shortcomings:

  What are the shortcomings?



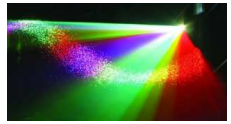
# Alternative 1 

- Approach:

  Specialize the `BufferedImage` class

- Shortcomings:

  Will likely lead to code duplication when, for example, described static visual content is added to the system.



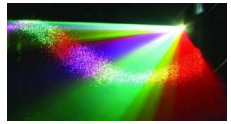
# Alternative 2

- Approach:

  Decorate `Image` objects with another class that provides the additional functionality.

- Shortcomings:

  What are the shortcomings?



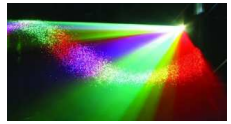
# Alternative 2

- Approach:

  Decorate `Image` objects with another class that provides the additional functionality.

- Shortcomings:

  Would probably lead to confusion because the `Image` interface include methods like `getGraphics()` and `getScaledInstance()` that are not relevant for most apps.



# Alternative 3 

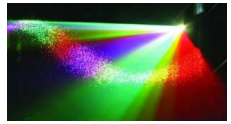
- Approach:

  Create a class that delegates to a `BufferedImage` object but doesn't implement the `Image` interface.

- The Participants:

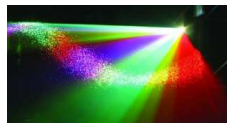
  A specialization of the `statik.TransformableContent` interface that contains the additional requirements of sampled content.

  A class that extends the `AbstractTransformableContent` that implements this interface, in part by delegating to a `BufferedImage` object.



# The statik.sampled.TransformableContent Interface

```
package visual.statik.sampled;

import java.awt.*;
import java.awt.image.*;

public interface TransformableContent
       extends   visual.statik.TransformableContent
{
  public abstract void setBufferedImageOp(BufferedImageOp op);

  public abstract void setComposite(Composite c);
}
```



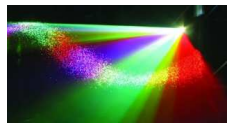
# Alternative 3.1

- One Implementation of `statik.sampled.Content`:

  Have a `BufferedImage` attribute that is transformed whenever the `render()` method or `getBounds()` method is called (using the `AffineTransform` returned by the `getAffineTransform()` class).

- Shortcomings:

  What are the shortcomings?



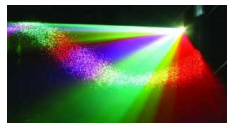
# Alternative 3.1

- One Implementation of `statik.sampled.Content`:

  Have a `BufferedImage` attribute that is transformed whenever the `render()` method or `getBounds()` method is called (using the `AffineTransform` returned by the `getAffineTransform()` class).

- Shortcomings:

  Transformations are relatively time-consuming.



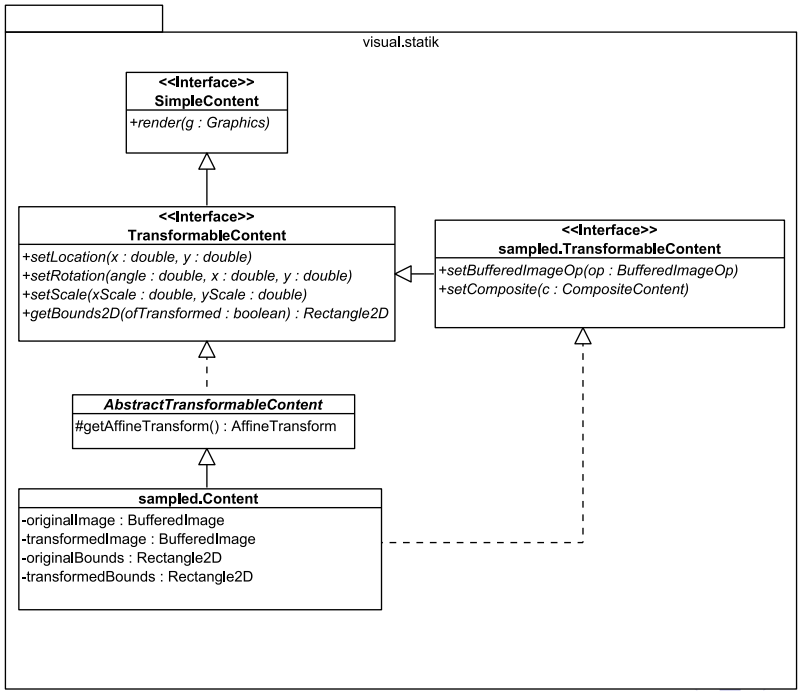
# Alternative 3.2

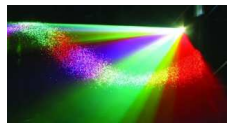
- Another Implementation of `statik.sampled.Content`:

  Keep both the original `BufferedImage` and the transformed `BufferedImage` as attributes.

  Keep the bounds of both.

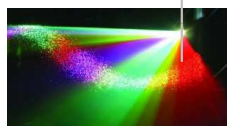
- An Illustration:



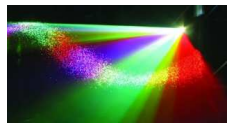
# Structure of the Content Class

```
package visual.statik.sampled;

import java.awt.*;
import java.awt.geom.*;
import java.awt.image.*;

public class      Content
       extends    visual.statik.AbstractTransformableContent
       implements TransformableContent
{
  private boolean            refiltered;
  private BufferedImageOp    imageOp;
  private Composite          composite;
  private BufferedImage      originalImage, transformedImage;
  private Rectangle2D.Double originalBounds,transformedBounds;

  private static final double       DEFAULT_X         = 0.0;
  private static final double       DEFAULT_Y         = 0.0;

  public Content()
  {
    this(null, DEFAULT_X, DEFAULT_Y);
  }

  public Content(BufferedImage image, double x, double y)
  {
    this(image, x, y, true);
```



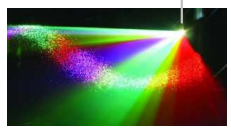
# Structure of the `Content` Class (cont.)

```
  }
}
```



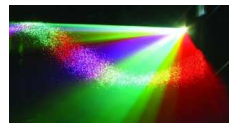
# Setters in the Content Class

```
public void setBufferedImageOp(BufferedImageOp op)
{
  imageOp    = op;
  refiltered = true;
}

public void setComposite(Composite c)
{
  composite = c;
}

public void setLocation(double x, double y)
{
  if (!rotating)
  {
    this.x = x;
    this.y = y;
    transformedBounds.x = this.x;
    transformedBounds.y = this.y;
  }
  else
    super.setLocation(x, y);
}

public void setRotation(double angle, double x, double y)
{
```



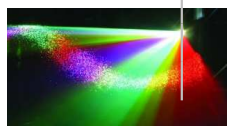
# Setters in the `Content` Class (cont.)

```
    if (rotating) super.setRotation(angle, x, y);
  }
```



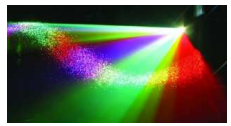
# Transformations in the Content Class

```
private void createTransformedContent(BufferedImageOp op)
{
  BufferedImage   tempImage;
  Rectangle2D     temp;

  try
  {
    // Apply the filter
    tempImage = originalImage;
    if (imageOp != null)
    {
      tempImage = imageOp.filter(originalImage, null);
    }

    // Create the transformed version
    transformedImage = op.filter(tempImage, null);

    temp = op.getBounds2D(originalImage);

    transformedBounds.x      = temp.getX();
    transformedBounds.y      = temp.getY();
    transformedBounds.width  = temp.getWidth();
    transformedBounds.height = temp.getHeight();

    if (!rotating)
    {
      transformedBounds.x += x;
      transformedBounds.y += y;
```



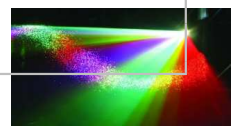
# Transformations in the `Content` Class (cont.)

```
      }

      setTransformationRequired(false);
    }
    catch (RasterFormatException rfe)
    {
      // Unable to transform
      transformedImage = null;
    }
  }
```



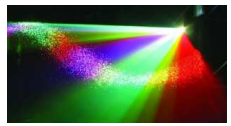
# render() in the Content Class

```
public void render(Graphics g)
{
  Composite      oldComposite;
  Graphics2D     g2;

  g2 = (Graphics2D)g;

  if (originalImage != null)
  {
    oldComposite = g2.getComposite();
    if (composite != null) g2.setComposite(composite);

    // Transform the Image (if necessary)
    if (isTransformationRequired())
    {
      createTransformedContent();
    }

    // Render the image
    if (!rotating)
      g2.drawImage(transformedImage,(int)x,(int)y,null);
    else
      g2.drawImage(transformedImage,0,0,null);

    g2.setComposite(oldComposite);
  }
}
```



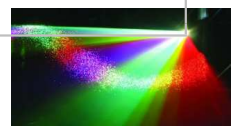
# getBounds2D() in the Content Class

```
public Rectangle2D getBounds2D(boolean transformed)
{
  if   (transformed) return transformedBounds;
  else               return originalBounds;
}
```



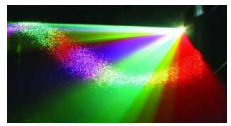
# Structure of the ContentFactory Class

```
package visual.statik.sampled;

import java.awt.*;
import java.awt.image.*;
import java.io.*;
import java.util.Arrays;

import io.ResourceFinder;

public class ContentFactory
{
  private ImageFactory          imageFactory;

  private static final int      DEFAULT_CHANNELS = 3;
  private static final boolean  ROTATABLE        = true;

  public ContentFactory()
  {
    imageFactory = new ImageFactory();
  }

  public ContentFactory(ResourceFinder finder)
  {
    imageFactory = new ImageFactory(finder);
  }
}
```



# Creating Content from an Image

```
public Content createContent(Image   image,
    int     channels,
    boolean rotatable)
{
  BufferedImage       bi;

  bi = imageFactory.createBufferedImage(image, channels);
  return createContent(bi, rotatable);
}
```



# Creating `Content` from a Named Resource

```
public Content createContent(String name,
    int channels,
    boolean rotatable)
{
  BufferedImage          bi;

  bi = imageFactory.createBufferedImage(name, channels);
  return createContent(bi, rotatable);
}
```