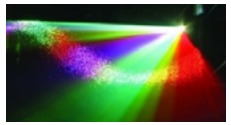
Chapter 4
Visual Content

The Design and Implementation of Multimedia Software

David Bernstein

Jones and Bartlett Publishers

www.jbpub.com

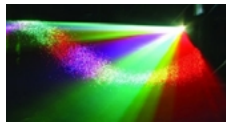


About this Chapter

1. The physics of light.

2. The biology of vision and the psychology of visual perception.

3. Visual output devices, and how they can be used to present visual content.

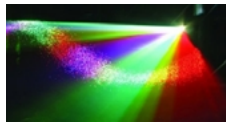


Electromagnetic Waves

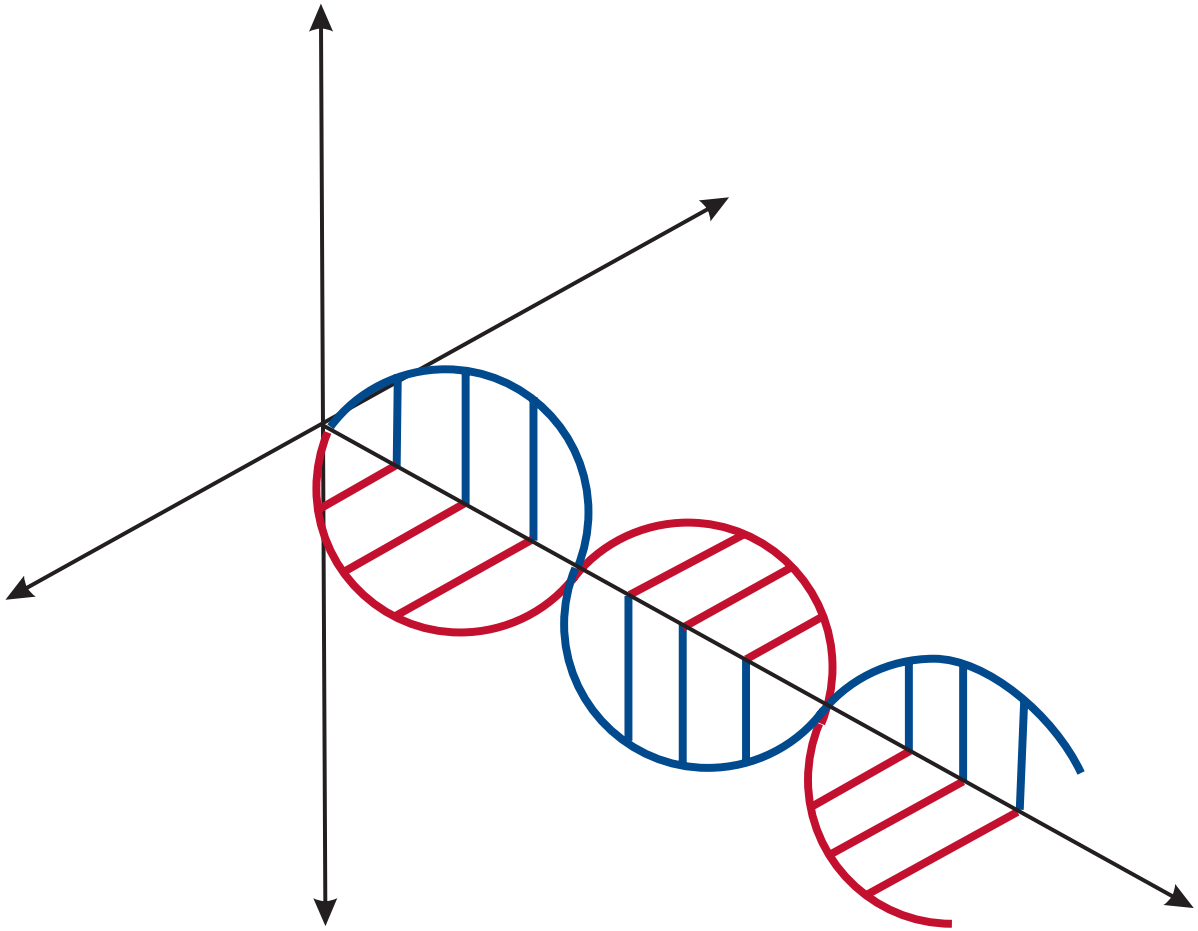
- Visible light is the portion of the *electromagnetic (EM) spectrum* with wavelengths ranging from about 380 nanometers to 740 nanometers (where $1\text{nm} = 10^{-9}\text{m}$).

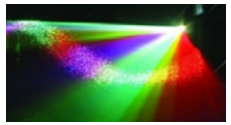
- Sources of EM waves create both electric and magnetic fields, and the fields it produces are perpendicular to each other.

- These fields travel away from the source, and both are perpendicular to the direction of travel (i.e., they are *transverse waves*).



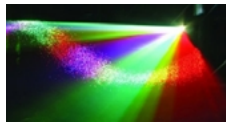
Electromagnetic Waves (cont.)





Electromagnetic Waves (cont.)

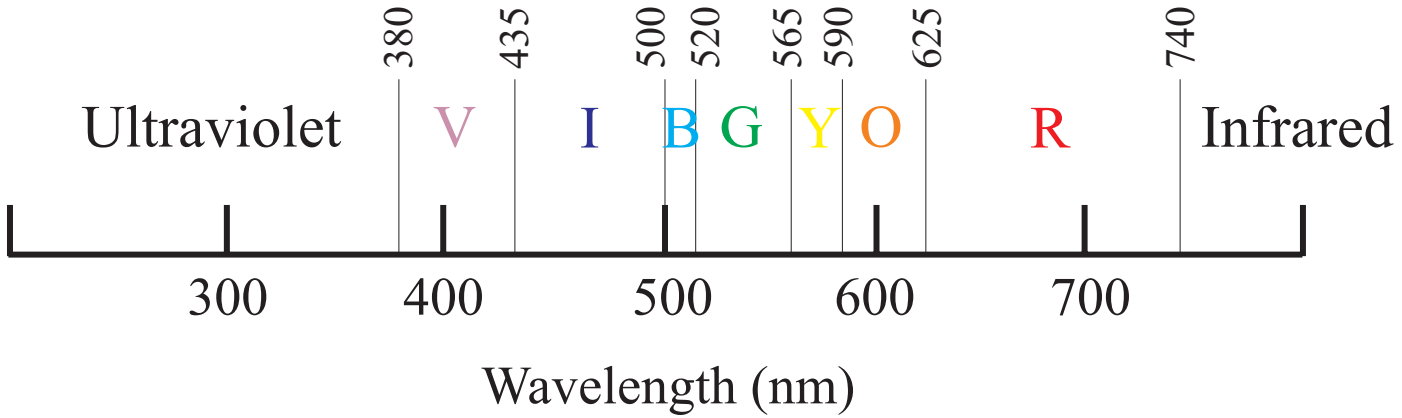
- An EM wave carries no mass but does carry energy.
- The energy of an EM wave is stored in the electric and magnetic fields, and is proportional to the frequency of the wave.
- While sound waves require a medium for transmission, EM waves can travel through a vacuum.

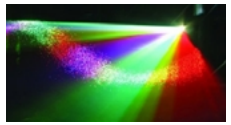


The Visible Spectrum

Definition

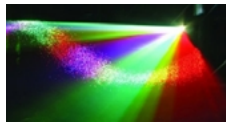
A *spectral color* is light of a single wavelength in the visible portion of the EM spectrum.



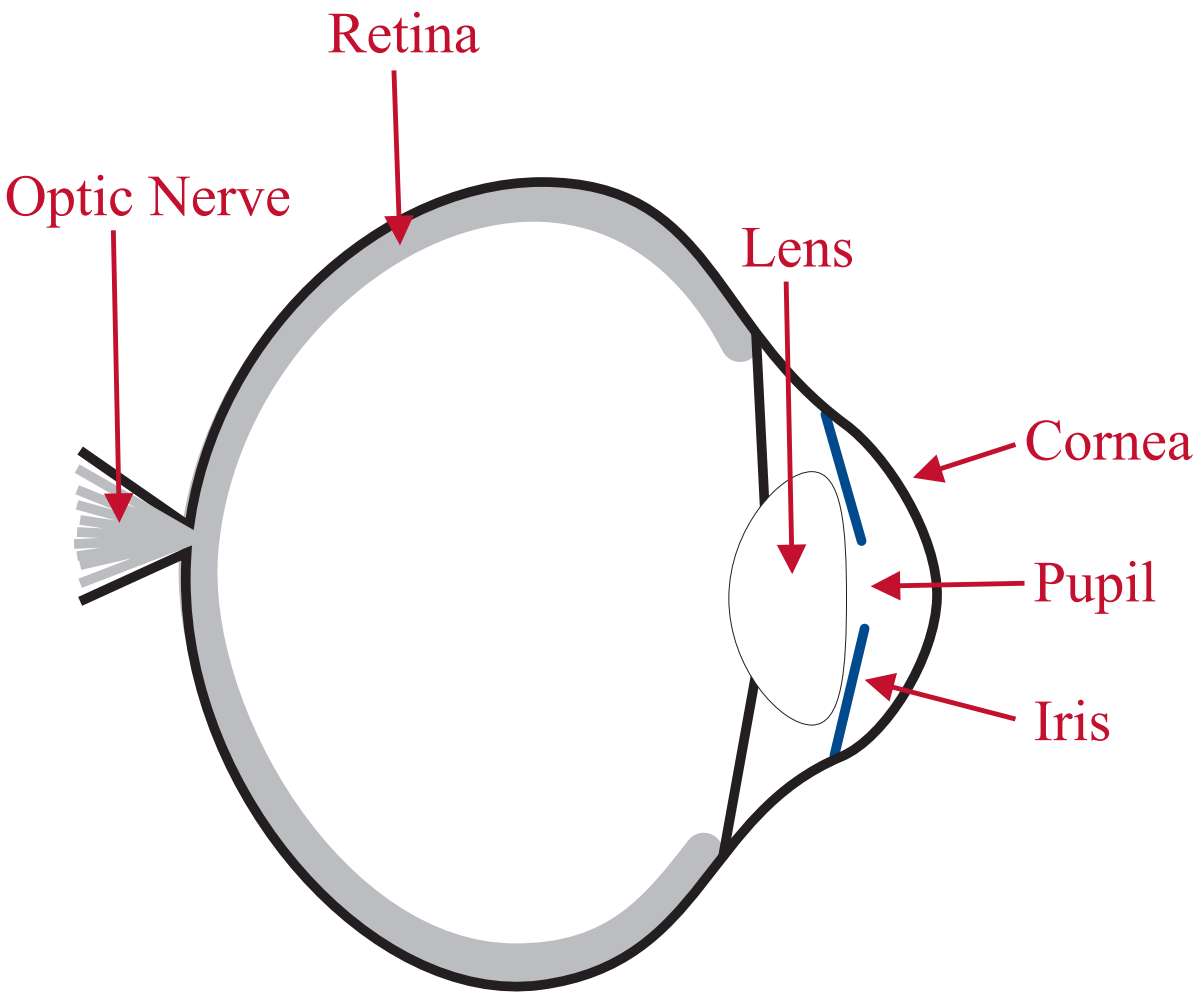


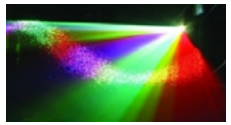
The Organs Involved

- Humans sense light using an organ called the eye.
- Humans interpret the sensation using the brain.



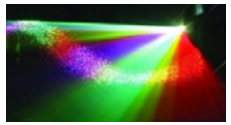
The Human Eye





Photoreceptors in the Retina

- *Rods* and *Cones*
- *On* (respond to the presence of light), *Off* (respond to the absence of light) and *On-Off* (respond to both)



Kinds of Cones

- ρ-Cones:

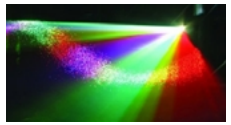
  Sensitive to wavelengths between 400nm and 700nm; most sensitive to a wavelength of about 580nm. (64% of the cones)

- γ-Cones:

  Sensitive to wavelengths between 420nm and 660nm; most sensitive to a wavelength of about 540 nm. (32% of the cones)

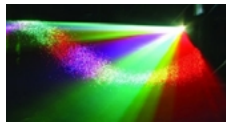
- β-Cones:

  Sensitive to wavelengths between 400nm and 550nm; most sensitive to a wavelength of about 450nm. (2% of the cones)



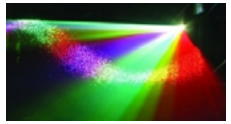
Transmission

- Data from the eye is transmitted to the brain along the *optic nerve*, which consists of about 1 million individual nerve fibers.
- One stream is used to transmit the signals that control eye movements.
- A second stream is used to transmit information that is highly variable over time but not space.
- A third is used to transmit information that is highly variable over space but not time.



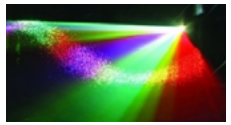
Processing

- Most information arrives at the portion of the *occipital lobe* of the brain called the *primary visual cortex*.
- About 20 other areas in the *cortex* seem to receive visual input.



Getting Started

The data that are received by the brain are perceived in a variety of different ways.



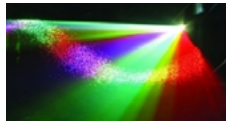
Important Points

- An increase in output from the rods is perceived as an increase in brightness.

  Light waves with a higher amplitude cause the rods to generate stronger signals and are perceived as being bright.

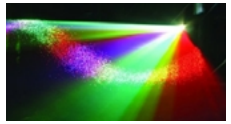
- Perceived brightness is approximately logarithmic.

  A doubling of output from the rods does not lead to a perceived doubling of the brightness.



Some Intuition

- Cones do not capture the spectral power distribution of the light.
- Hence, the data processed by the brain is just the total levels of activity sensed by the three types of cones.
- So, if several waves with different wavelengths interfere with each other and then enter the pupil, only one color is perceived.



A Formalization

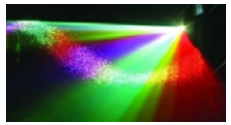
Letting P denote the spectral power distribution of the light passing through the pupil, and s_ρ, s_γ, and s_β denote the sensitivity of the three types of cones, then:

$$R = \int P(\lambda) s_\rho(\lambda) d\lambda \tag{1}$$

$$G = \int P(\lambda) s_\gamma(\lambda) d\lambda \tag{2}$$

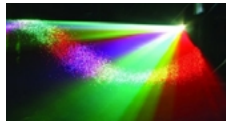
$$B = \int P(\lambda) s_\beta(\lambda) d\lambda \tag{3}$$

where R, G, and B denote the total activity captured in the red, green, and blue ranges.



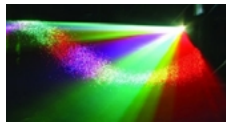
Metameric Substitution

- Light with different spectral power distributions can be perceived as the same color.
- The average person can only perceive between 7 million and 10 million colors.
- Increasing or decreasing the power of R, G, and B by the same factor changes the overall perceived brightness but not the perceived color.



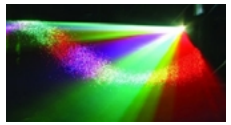
Some Observations

- The retina is a two-dimensional array of sensors; it does not collect any three-dimensional data.
- The eye does not have a sensor that measures depth/distance.
- All perception of depth and distance is inferred by the brain.

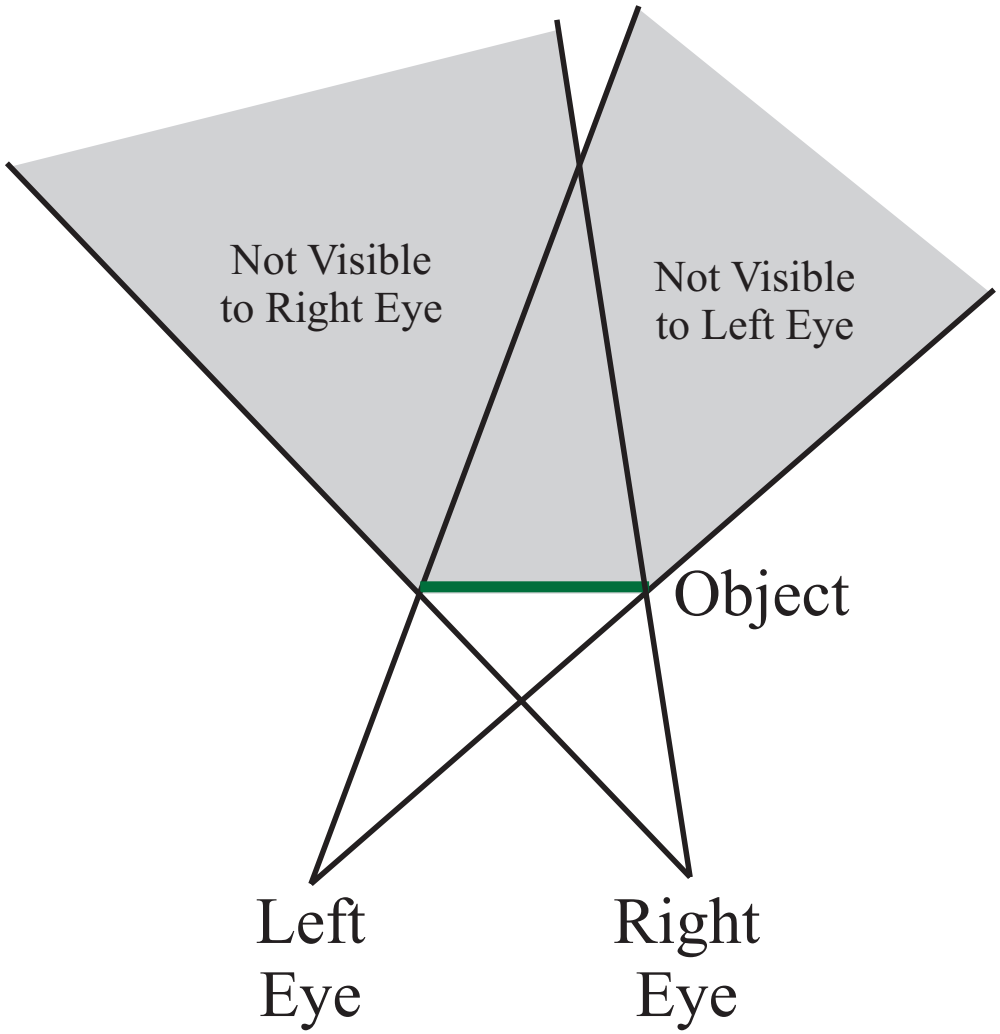


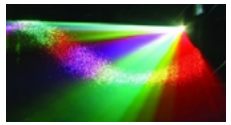
Retinal Disparity

- For objects that are close to the viewer, the two eyes collect data from a slightly different position and a slightly different angle.
- For objects that are far from the viewer, this is not the case.

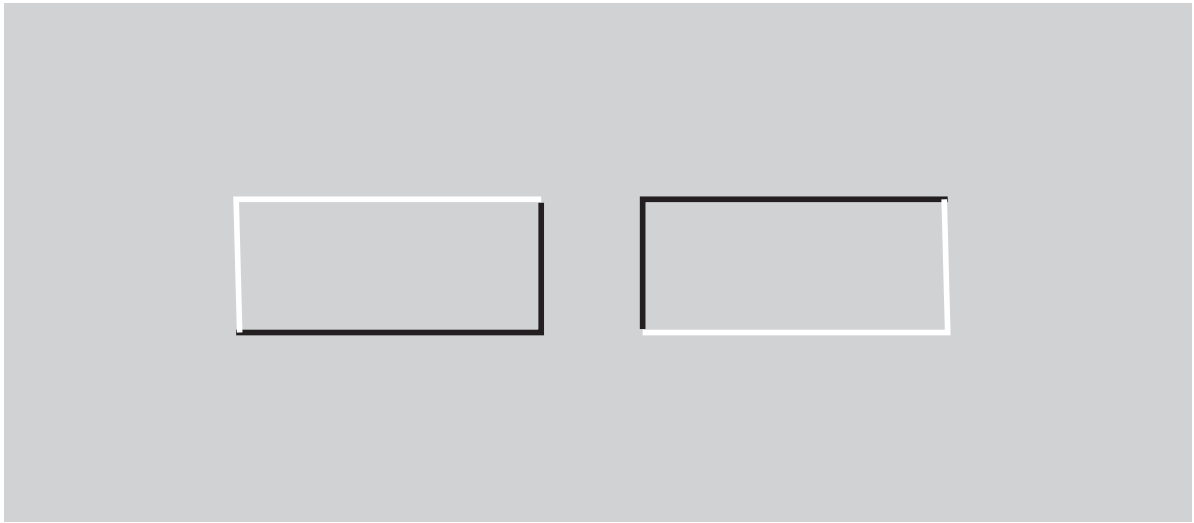


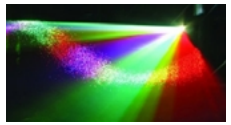
Occluded and Half-Occluded Regions



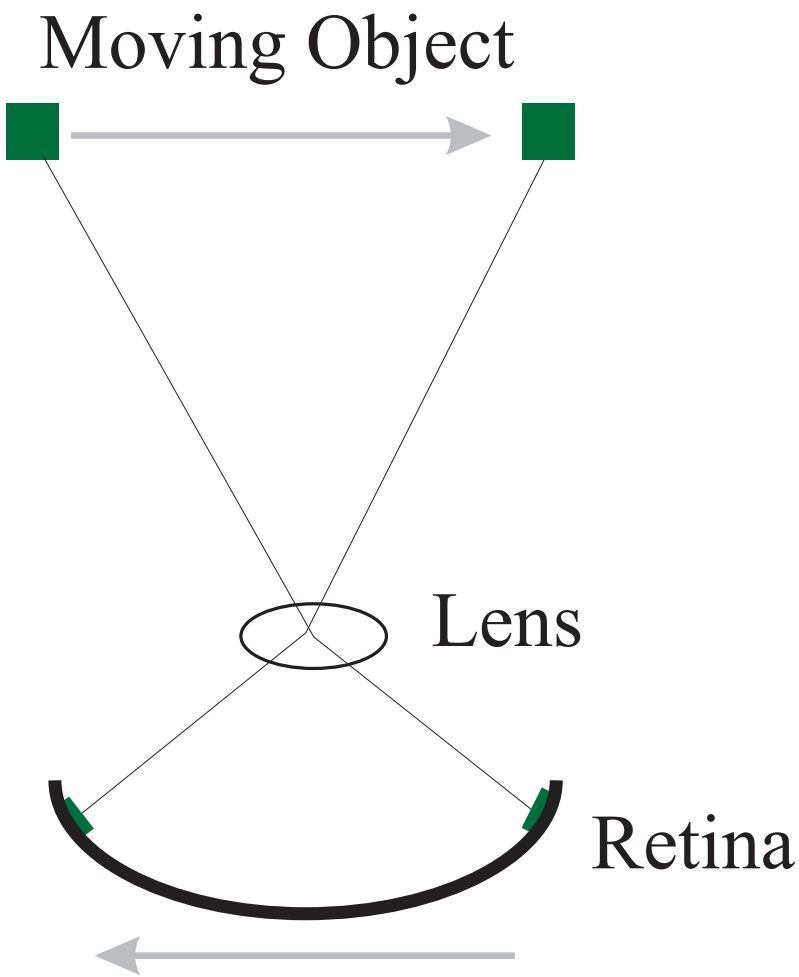


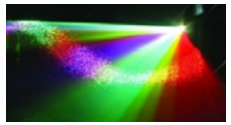
Shadows and Depth





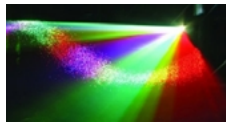
Sequential Firing of Receptors





Sequential Firing of Receptors (cont.)

- Since each photoreceptor takes some amount of time to react, the brain is receiving a **sample** of all of the data that are actually available.
- As a result, people can not perceive temporal signals higher than about 60Hz.



Head and Eye Movements

- Inflow Theory (Older):

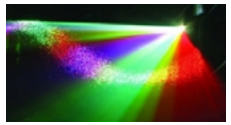
  Signals to move the eye go to the eye, and signals from the eye muscles and the retina go to the brain.

  The brain uses the retinal data and the actual eye movement data to infer motion.

- Outflow Theory (Modern):

  Signals to move the eye go both to the eye and the brain, and signals from the retina go to the brain.

  The brain uses the retinal data and the eye movement commands to infer motion.



Basics

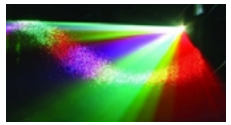
- Visual Output Device:

  A machine that is capable of presenting visual information transmitted from a computer (i.e., making information stored in a computer perceivable by the human eye).

- Types:

  'Hardcopy' Devices (i.e., devices that put the information on a material like paper that reflects light)

  'Temporary' Output Devices (i.e., devices like monitors and displays that emit light).



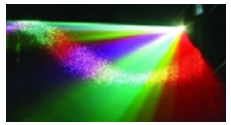
Categories

Definition

A visual output device with a discrete display space is called a *raster device*.

Definition

A visual output device with a continuous display space is called a *vector device*.



Examples

- Raster Devices:

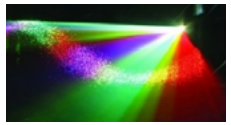
  Liquid Crystal Displays (LCDs) and Light Emitting Diode (LED) displays

  Ink jet printers and laser printers

- Vector Devices:

  Pen plotter

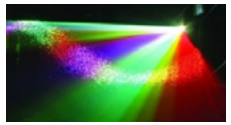
  Laser lightshow projectors



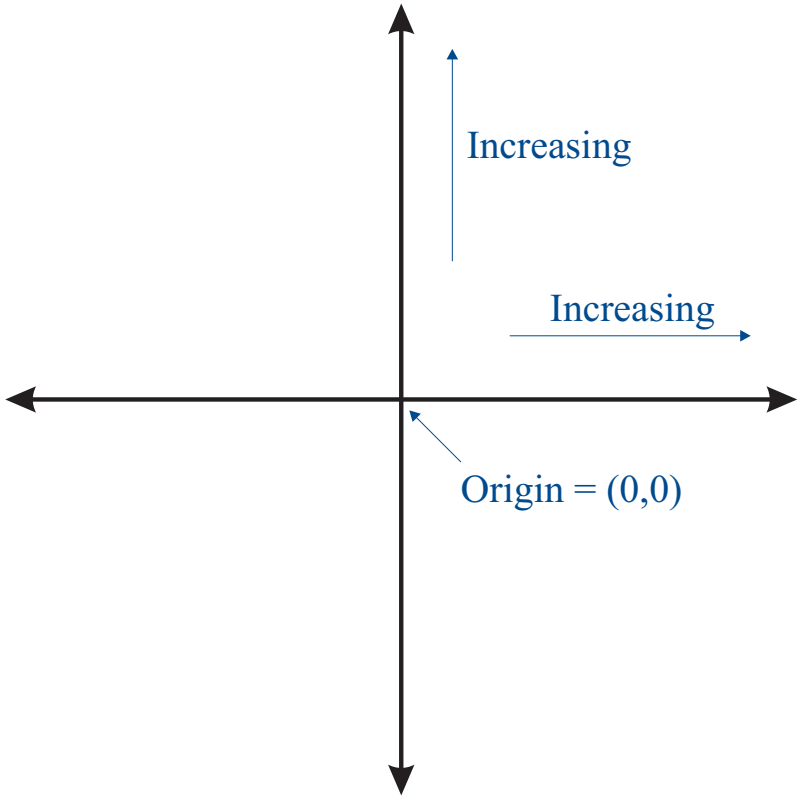
Coordinates

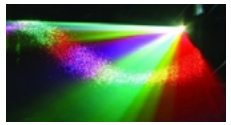
Definition

Coordinates are quantities (either linear or angular) that designate the position of a point in relation to a given reference frame.

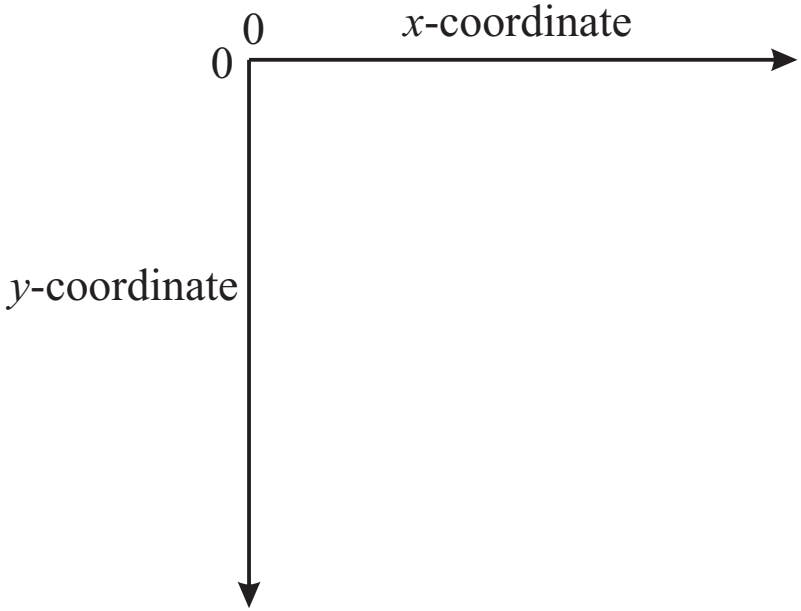


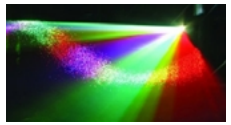
Cartesian Coordinates





Output Device Coordinates





Rectangular Output Devices

Definition

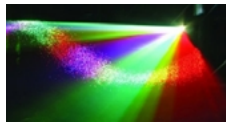
The *aspect ratio* of a rectangle is the ratio of the size of its larger side to the size of its smaller side.

- *Portrait* Orientation:

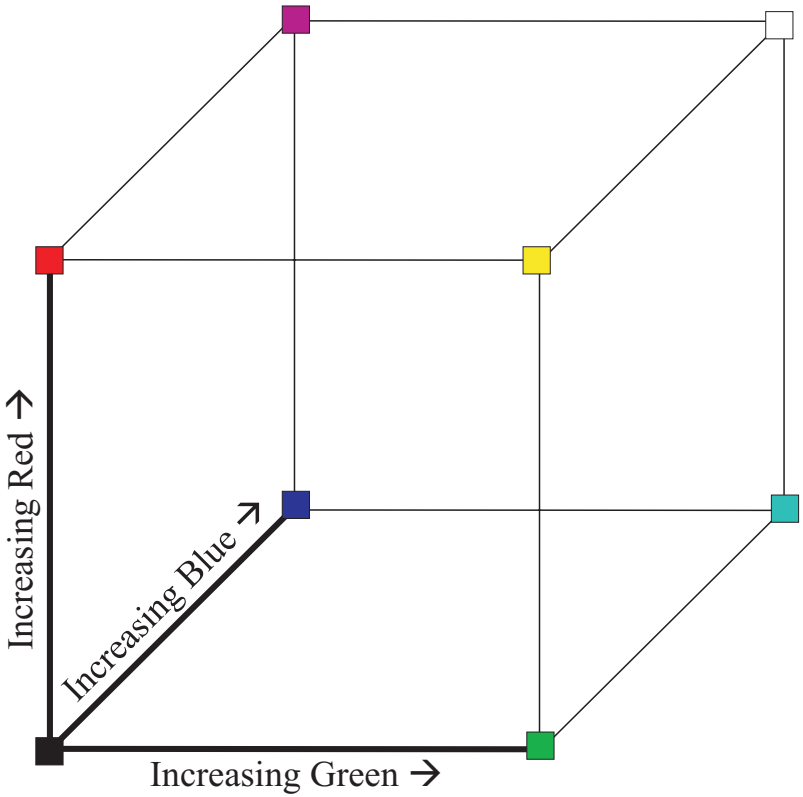
  Taller than it is wide.

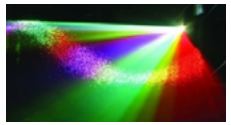
- *Landscape* Orientation:

  Wider than it is tall.

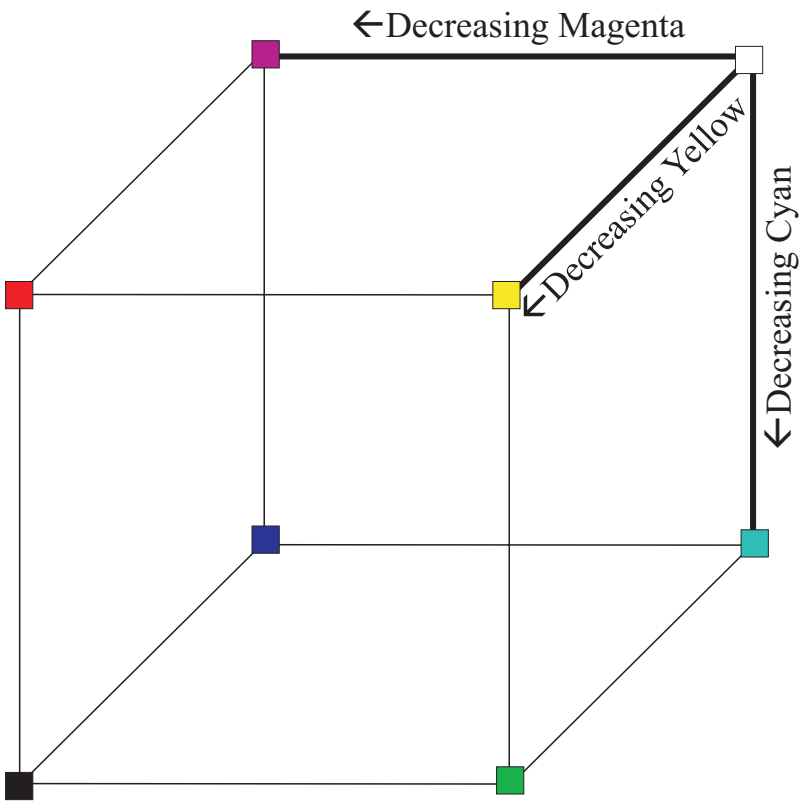


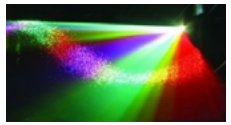
An Additive Model - The RGB Cube





A Subtractive Model - The CYM Cube

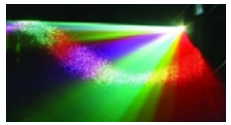




Limitations of RGB and CYM

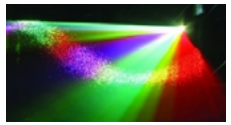
- Some colors that people can perceive can only be generated using negative amounts of one component (e.g., red in the RGB model), which is not physically possible.

- They are both linear in their components and the cones in the eye do not respond in a linear fashion.



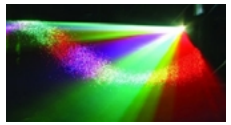
Colors in Java

- A `Color` consists of an array of (normalized) components.
- The meaning of each component is encapsulated in a `ColorSpace` object.



Rendering in Different Disciplines

- In cooking and industrial applications, rendering is the separation of fats from other organic materials.
- In art, rendering is the process by which a work of art is created.



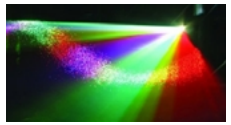
Visual Rendering

Definition

Visual rendering is the process of taking an internal representation of visual content and presenting it on a visual output device.

Rendering Engines in Java:

- `Graphics`
- `Graphics2D`



The Visual Rendering Process

- Not Our Concern:

  Conversion of the color space

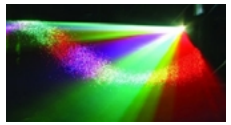
  Rasterization or vectorization of the internal representation

- Our Concern:

  Coordinate transformation

  Clipping

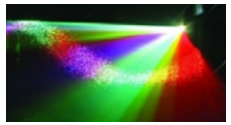
  Composition



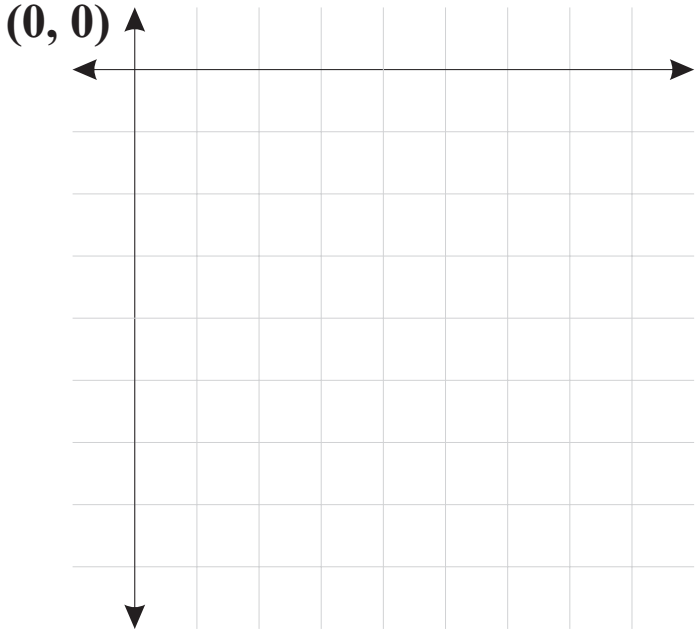
Translation

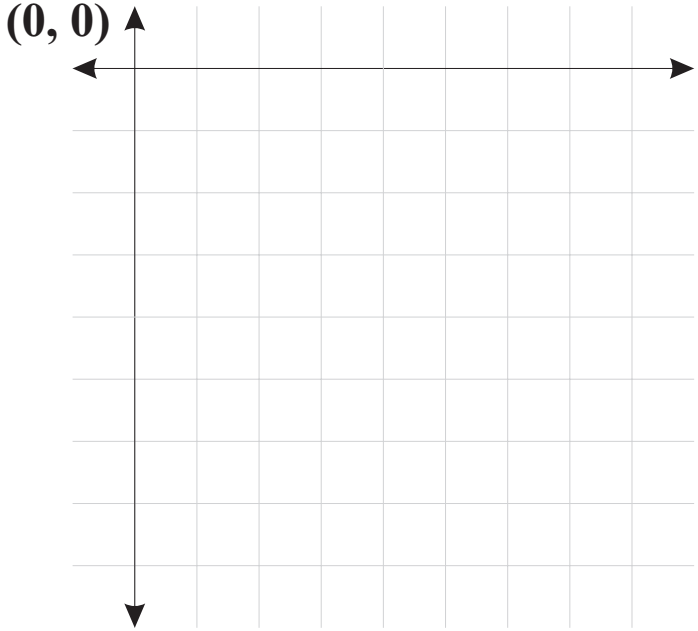
Definition

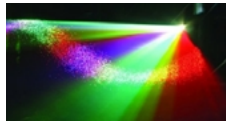
A *translation* involves a horizontal and/or vertical displacement.



Translation (cont.)



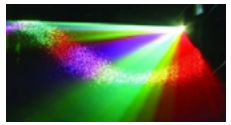




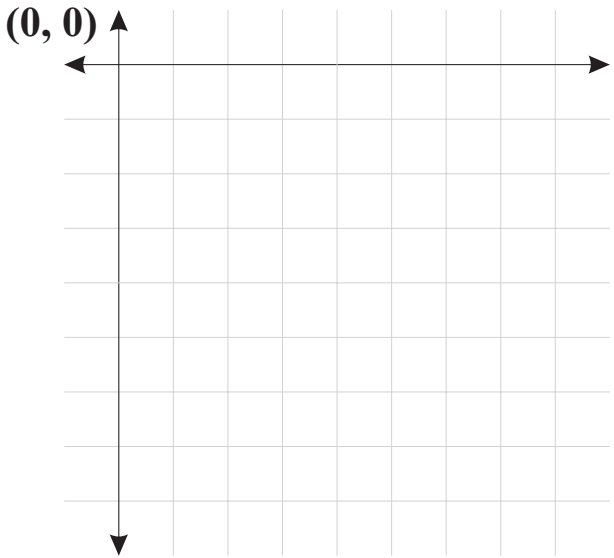
Scaling

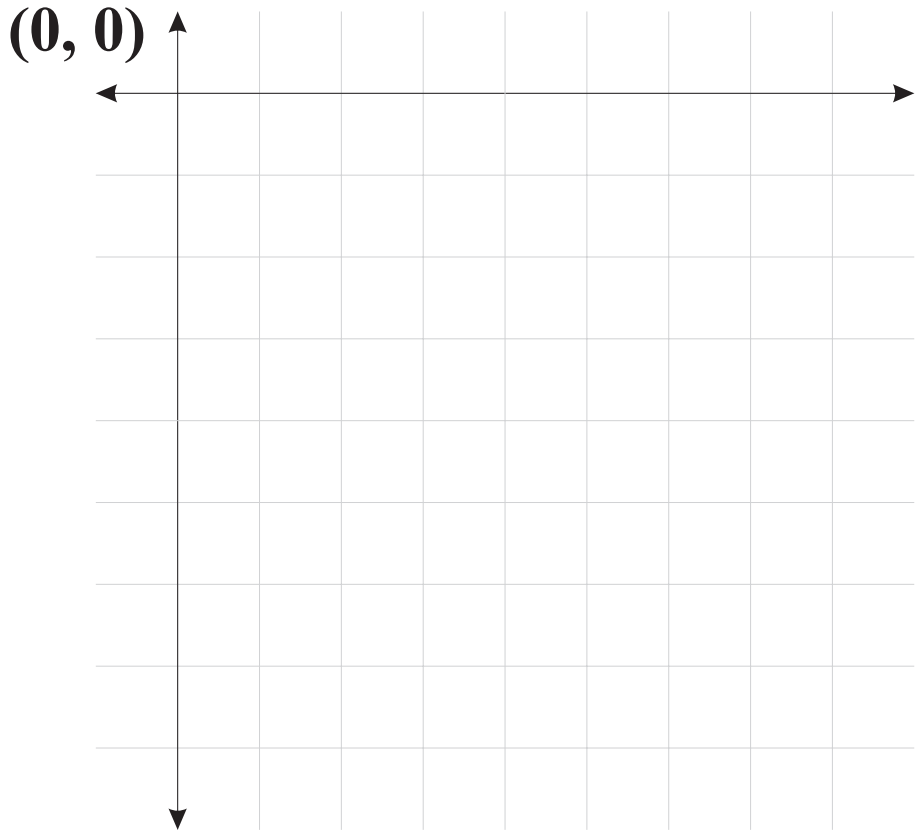
Definition

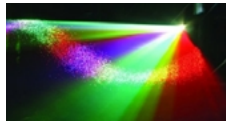
A *scaling* is a multiplication of each of the coordinates by a particular value (which need not be the same in both dimensions).



Scaling (cont.)



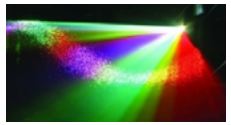




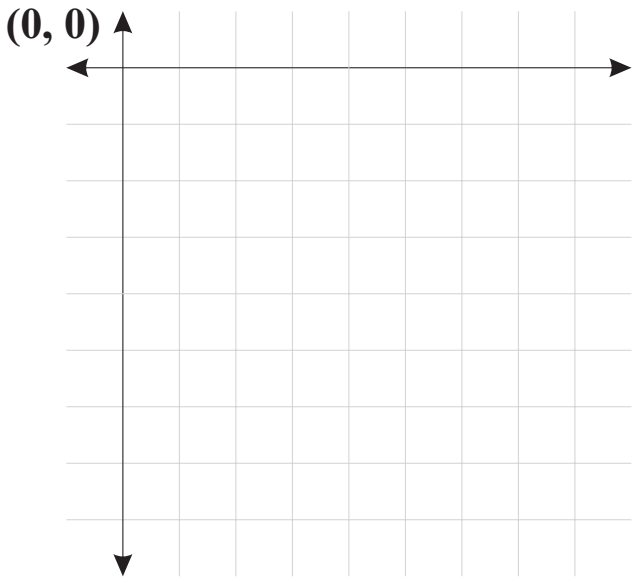
Rotation

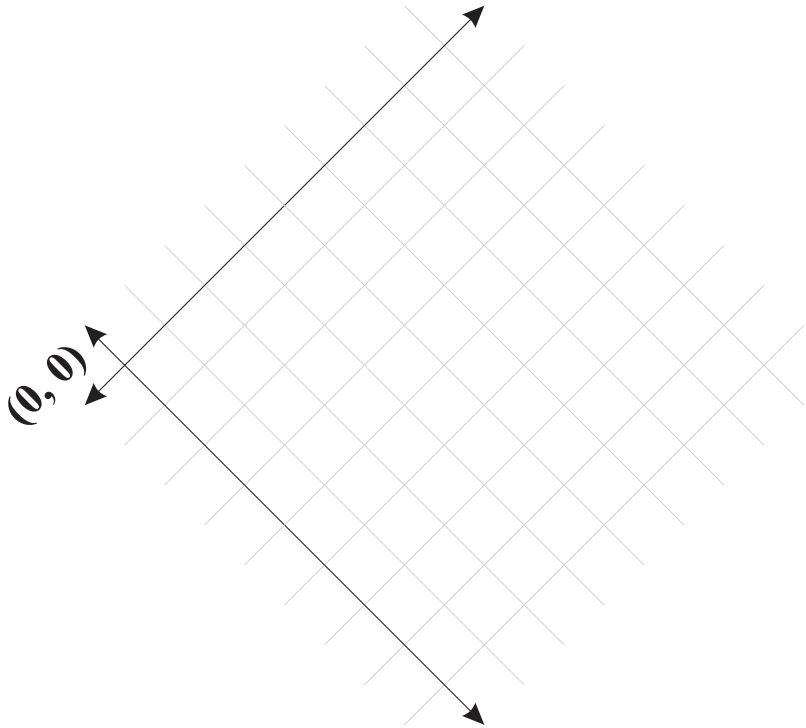
Definition

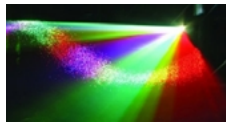
A *rotation* is an angular transformation of the coordinate system.



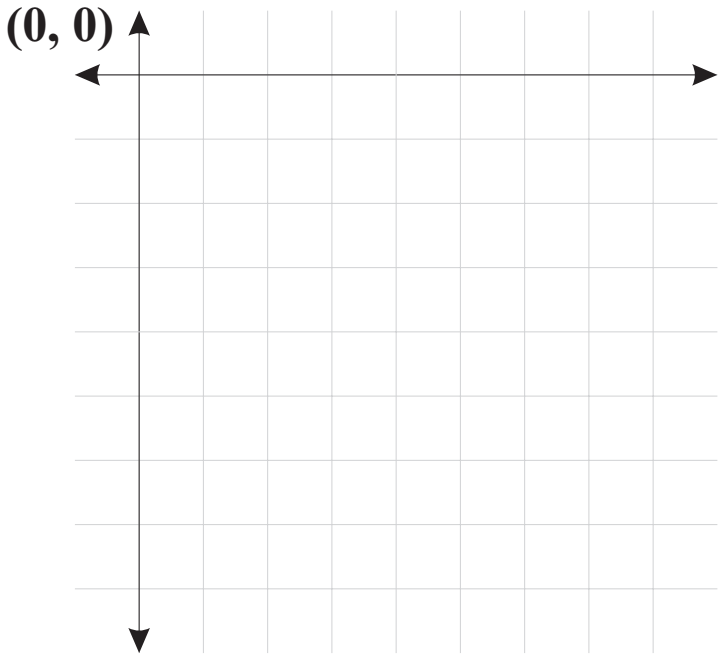
Rotation (cont.)

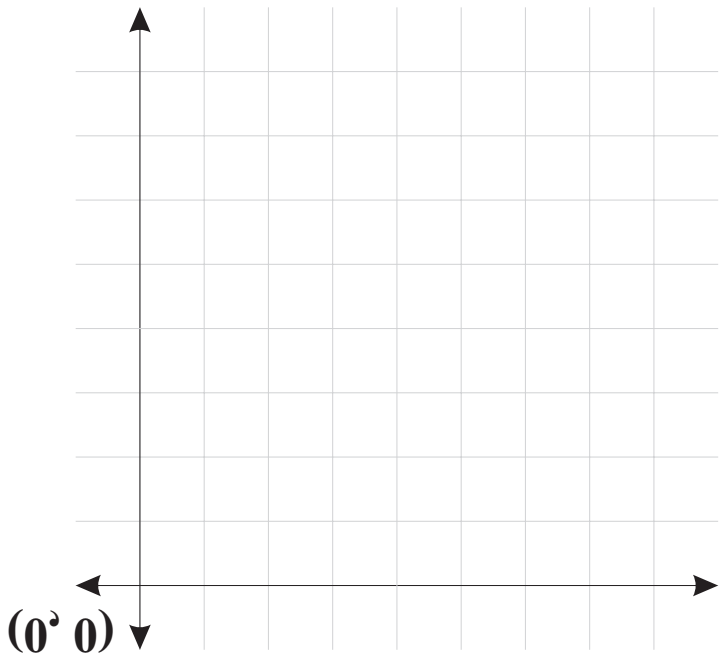


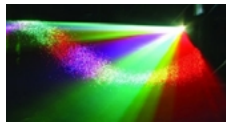




Reflection

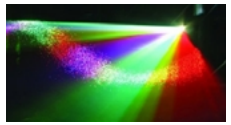






Transformations in Java

These transforms, and others, are encapsulated in the `AffineTransform` class.



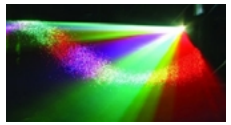
Limiting what is Rendered

Definition

Clipping is the process of limiting the portion of the internal representation that will be rendered.

Clipping in Java:

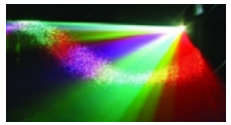
Use the `setClip(int, int, int, int)` method in the `java.awt.Graphics` or `java.awt.Graphics2D` object.



Combining Content

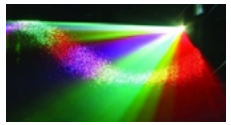
Definition

Composition is the process of combining the content being presented (the source) with the background (the destination).



Alpha Blending

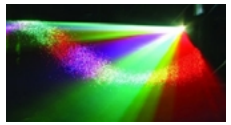
- Every 'point' has four channels, one each for red, green and blue, and one for α.
- The α channel is a weighting factor.



The Source Over Destination Rule

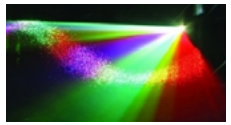
Letting d and s represent the destination and source, respectively:

$$R_d = R_s + R_d(1 - \alpha_s) \quad (4)$$
$$G_d = G_s + G_d(1 - \alpha_s) \quad (5)$$
$$B_d = B_s + B_d(1 - \alpha_s) \quad (6)$$
$$\alpha_d = \alpha_s + \alpha_d(1 - \alpha_s) \quad (7)$$



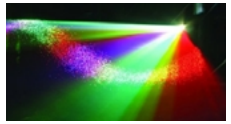
The Porter-Duff Rules

Rule	**Definition**
Destination	$C_d = C_d$
Destination Atop Source	$C_d = C_s(1 - \alpha_d) + C_d\alpha_s$
Destination Held Out By Source	$C_d = C_d(1 - \alpha_s)$
Destination In Source	$C_d = C_d\alpha_s$
Destination Over Source	$C_d = C_s(1 - \alpha_d) + C_d$
Source	$C_d = C_s$
Source Atop Destination	$C_d = C_s\alpha_d + C_d(1 - \alpha_s)$
Source Held Out By Destination	$C_d = C_s(1 - \alpha_d)$
Source In Destination	$C_d = C_s\alpha_d$
Source XOR Destination	$C_d = C_s(1 - \alpha_d) + C_d(1 - \alpha_s)$
Source Over Destination	$C_d = C_s + C_d(1 - \alpha_s)$



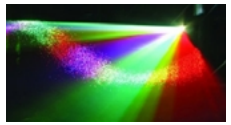
Composition in Java

- The `java.awt.Composite` Interface
- The `java.awt.AlphaComposite` Class
- The `Graphics2D.setComposite(Composite)` Method



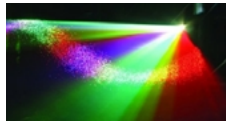
The Process

- When a `JComponent` 'needs to be' rendered, its `paint()` method is called and is passed a rendering engine.
- A `JComponent` (or a specialization of a `JComponent`) then uses this rendering engine to render itself.



An Example

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

public class BoringComponent extends JComponent
{
    public void paint(Graphics g)
    {
       Graphics2D          g2;

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


       // Put the rendering code here
    }
}
```



Requirements

F4.1 Support the addition of visual content.

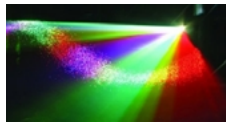
F4.2 Support the removal of visual content.

F4.3 Support 'front-to-back' ordering (i.e., z-ordering) of visual content.

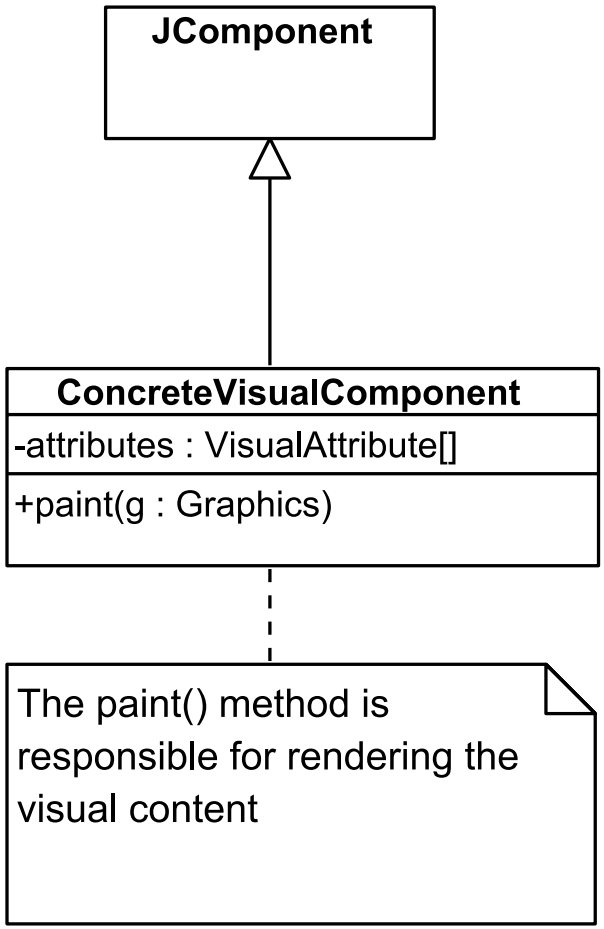
F4.4 Support multiple different presentations of visual content at the same time.

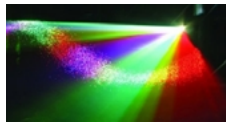
F4.5 Support the transformation of visual content.

N4.6 Be thread safe.



Alternative 1





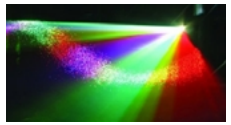
Alternative 1 - Shortcomings

- The `ConcreteVisualComponent` class will almost never be cohesive.

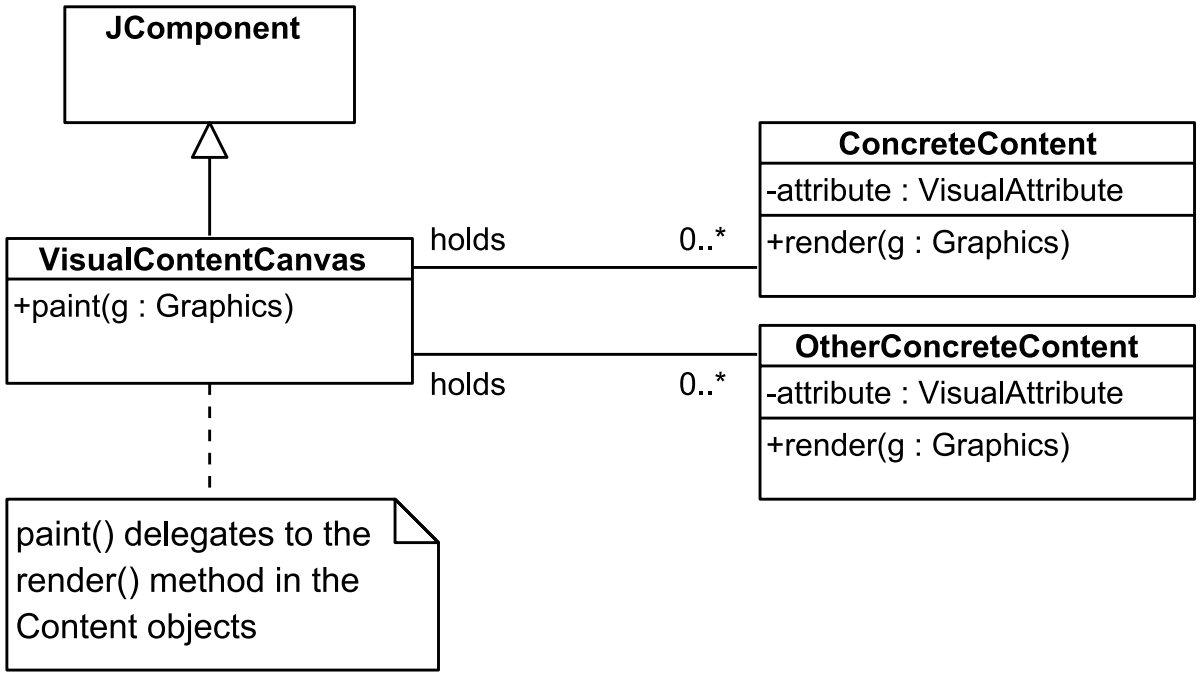
  It will be responsible for rendering both many different 'pieces' of visual content and many different types of visual content.

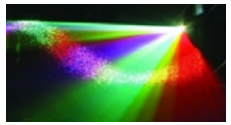
- Can be difficult to implement because of the need to manage different visual attributes for different pieces and types of visual content.

  For example, some pieces of content may have multiple visual attributes (e.g., position, color, width) that need to be accounted for and some may have only one (e.g., position).



Alternative 2





Alternative 2 - Advantages and Disadvantages

- Advantages:

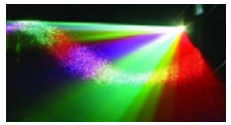
  The `VisualContentCanvas` can be used in any number of different applications.

  Each `ConcreteContent` and `OtherConcreteContent` object is only responsible for rendering itself and is, by definition, cohesive.

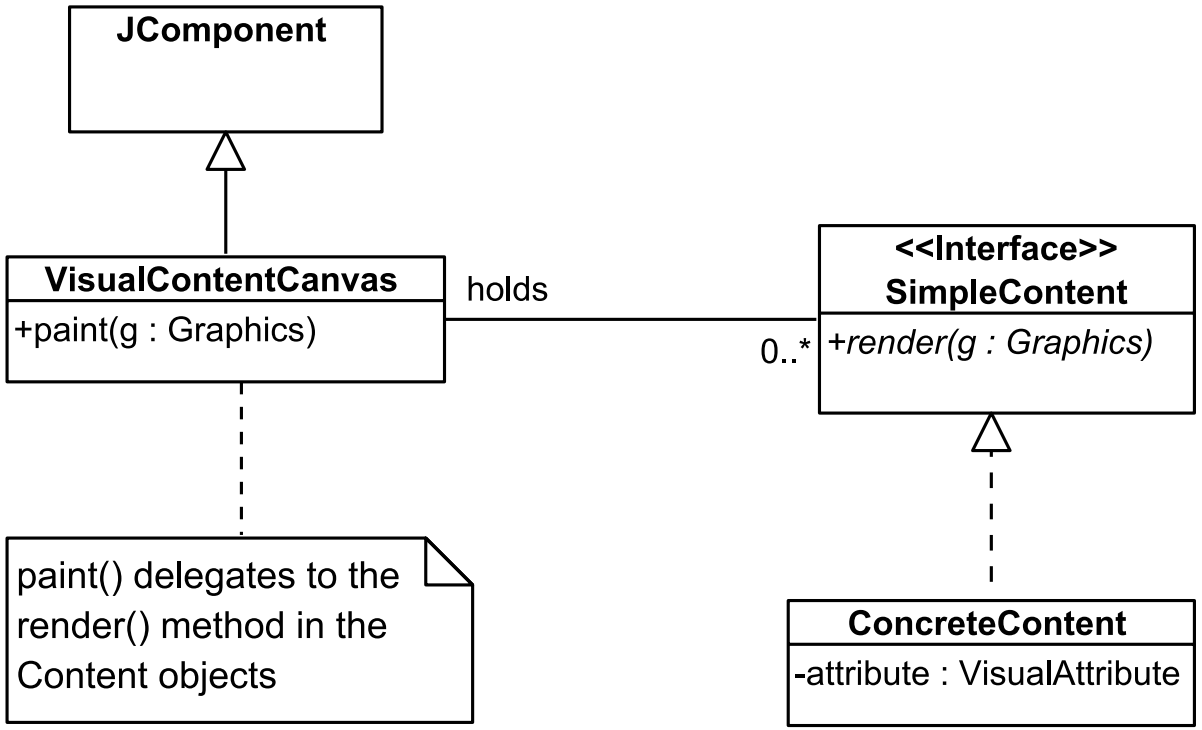
- Disadvantage:

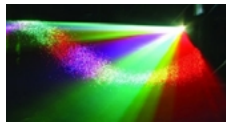
  The `VisualContentCanvas` class is too closely coupled to the `ConcreteContent` and `OtherConcreteContent` classes.

  As a result, the `VisualContentCanvas` class would have to manage several different collections, one for each type of visual content.



Alternative 3





Alternative 3 - Advantages and Disadvantages

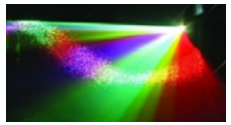
- Advantage:

  The `SimpleContent` interface decouples the `VisualContentCanvas` class from the concrete content classes.

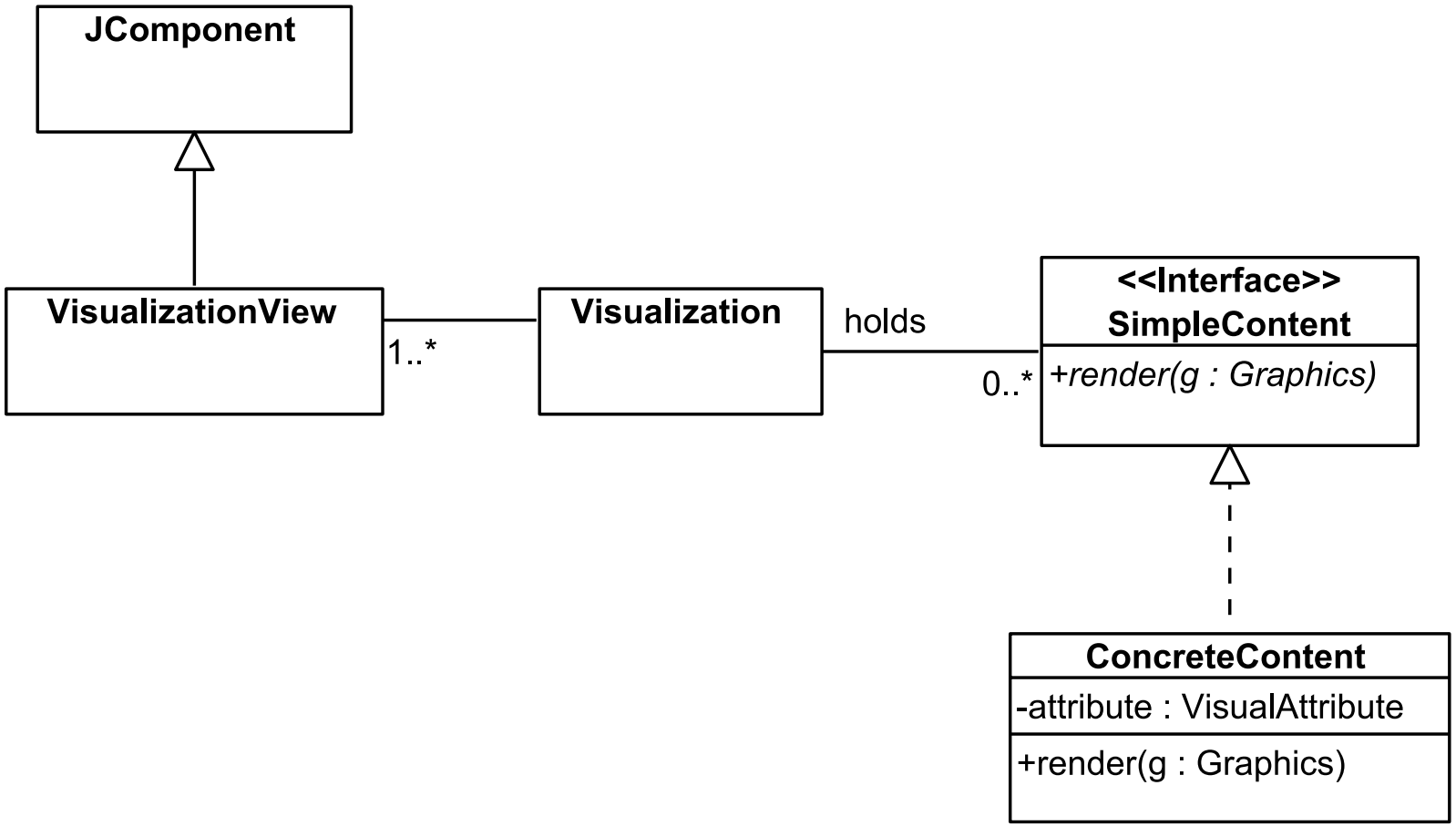
- Disadvantage:

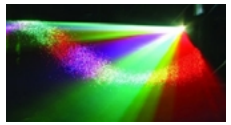
  The `VisualContentCanvas` class is not cohesive.

  It is responsible for managing the collection of `SimpleContent` objects (sometimes called 'model' responsibilities) and for the presentation functions (sometimes called 'view' responsibilities).



Alternative 4





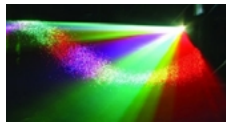
Alternative 4 - Advantages and Disadvantages

- Advantage:

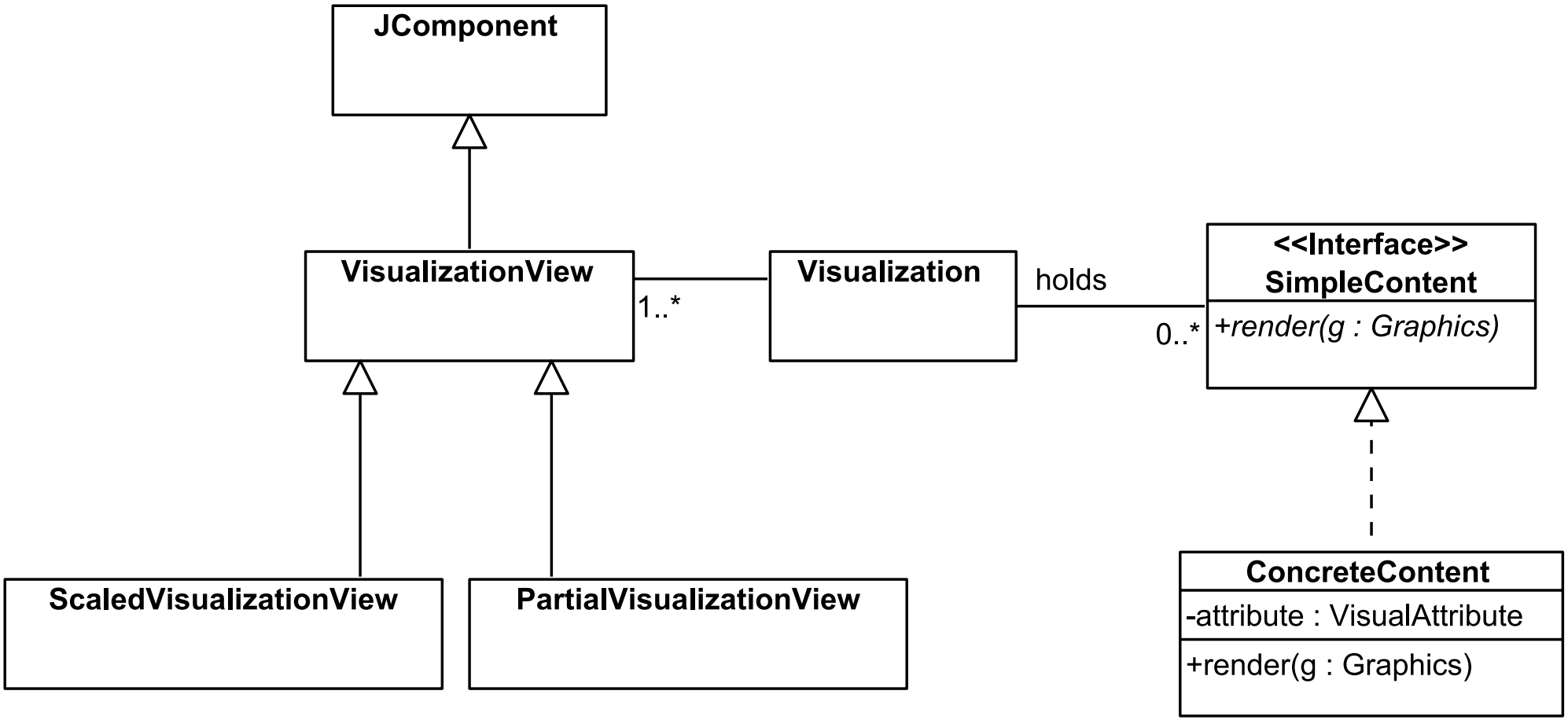
  Gives us the ability to have multiple 'views' associated with a single model.

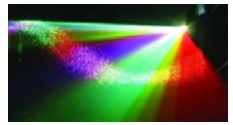
- Disadvantage:

  Does not completely satisfy Requirement 4.4 because the 'views' cannot have different capabilities/properties.



Alternative 5

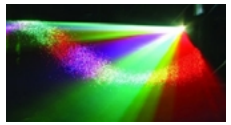




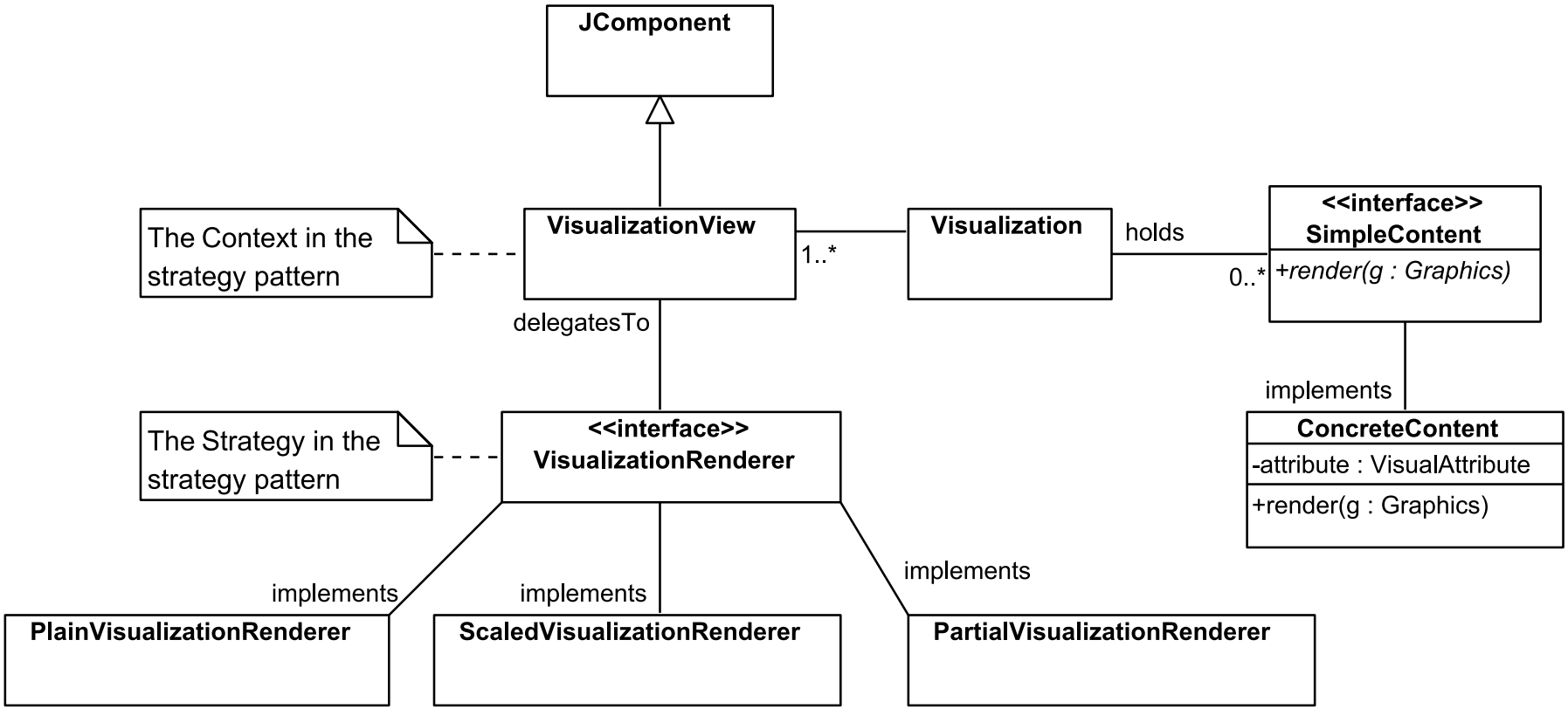
Alternative 5 - Shortcoming

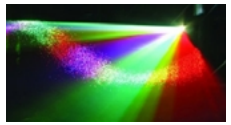
A Desirable Feature

Be able to 'enhance' individual `VisualizationView` objects at run time (e.g., picture-in-picture needs to be able to scale an existing `VisualizationView` object).



Alternative 6





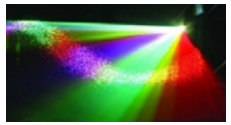
Alternative 6 - Shortcoming

- The Issue:

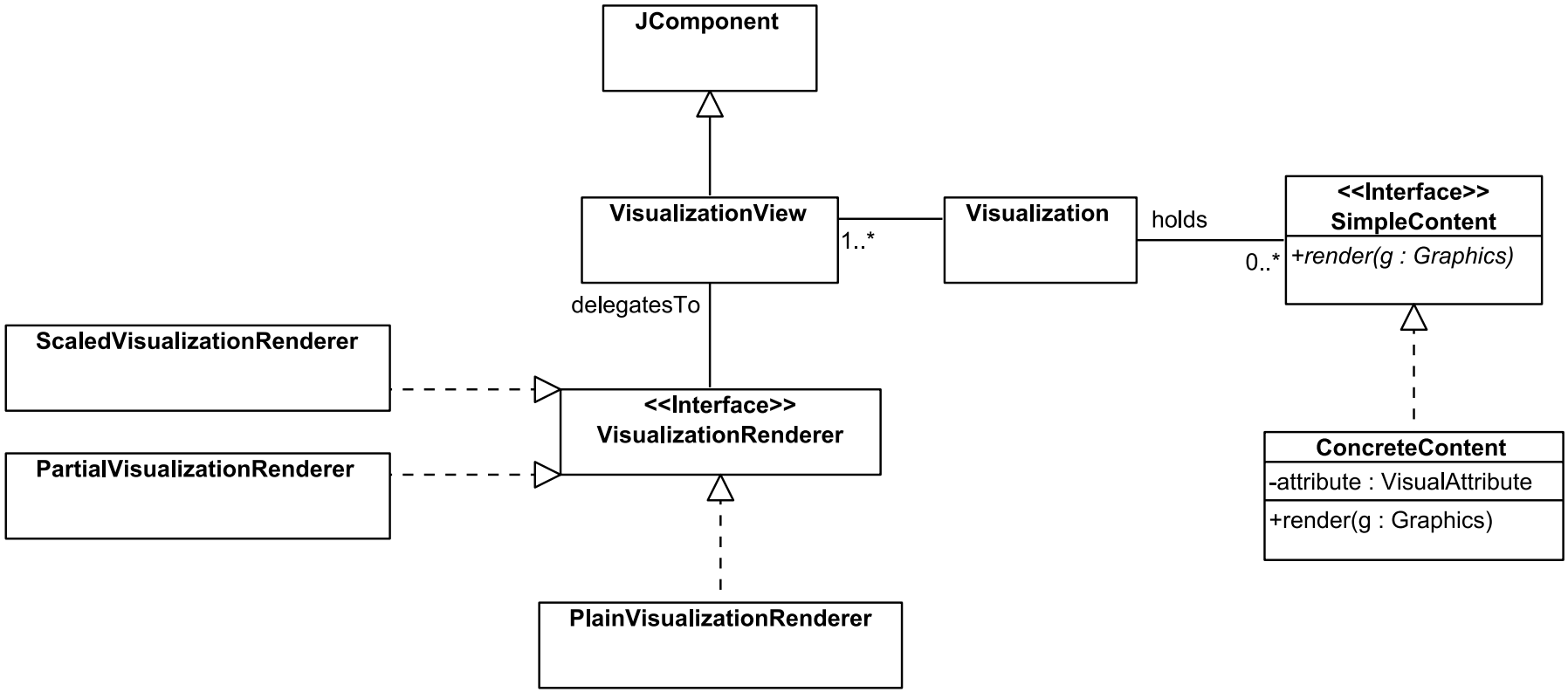
  A `VisualizationView` object can only use a single strategy at a time.

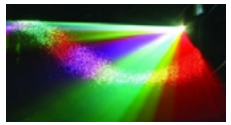
- An Example:

  If one wanted to create an application that supports zooming (which must be able to show part of a view and scale it at the same time) one would need to create a `ZoomingVisualizationRenderer` that includes the capabilities of both the `ScaledVisualizationRenderer` and the `PartialVisualizationRenderer`.



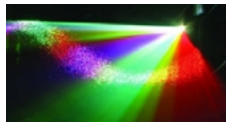
Alternative 7





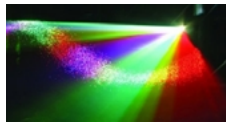
The SimpleContent Interface

```
package visual.statik;

import java.awt.*;

public interface SimpleContent
{
    public abstract void render(Graphics g);
}
```



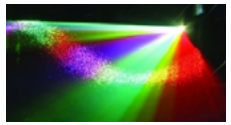
Structure of the `Visualization` Class

```
package visual;

import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.util.concurrent.CopyOnWriteArrayList;

import visual.statik.SimpleContent;

public class Visualization
{
    private   CopyOnWriteArrayList<SimpleContent> content;
    private   LinkedList<VisualizationView>       views;

    public Visualization()
    {
       content = new CopyOnWriteArrayList<SimpleContent>();
       views   = new LinkedList<VisualizationView>();

       views.addFirst(createDefaultView());
    }
}
```



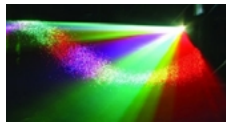
Managing Content in the `Visualization` Class

```
public void add(SimpleContent r)
{
   if (!content.contains(r))
   {
      content.add(r);
      repaint();
   }
}

public void clear()
{
   content.clear();
}

public Iterator<SimpleContent> iterator()
{
   return content.iterator();
}

public void remove(SimpleContent r)
{
   if (content.remove(r)) repaint();
}
```



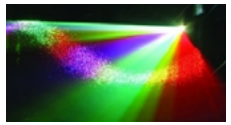
Ordering Content in the `Visualization` Class

```
public void toBack(SimpleContent r)
{
   boolean         removed;

   removed = content.remove(r);
   if (removed)
   {
      content.add(r);
   }
}

public void toFront(SimpleContent r)
{
   boolean         removed;

   removed = content.remove(r);
   if (removed)
   {
      content.add(0, r);
   }
}
```



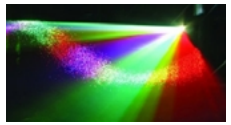
Managing Views in the `Visualization` Class

```
public void addView(VisualizationView view)
{
   views.addLast(view);
}

public VisualizationView getView()
{
   return views.getFirst();
}

public Iterator<VisualizationView> getViews()
{
   return views.iterator();
}

public void removeView(VisualizationView view)
{
   views.remove(view);
}

public void setView(VisualizationView view)
{
   views.removeFirst();
   views.addFirst(view);
}
```



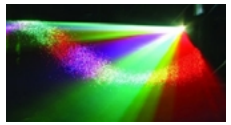
repaint() in the Visualization Class

```
protected  void repaint()
{
   Iterator<VisualizationView>   i;
   VisualizationView             view;

   i = views.iterator();
   while (i.hasNext())
   {
      view = i.next();
      view.repaint();
   }
}
```



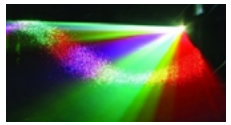
Structure of the `VisualizationView` Class

```
package visual;

import java.awt.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.*;

import visual.statik.*;

public class      VisualizationView
       extends    JComponent
       implements MouseListener
{
    protected Visualization         model;
    protected VisualizationRenderer renderer;

    public VisualizationView(Visualization         model,
                             VisualizationRenderer renderer)
    {
       super();
       this.model    = model;
       this.renderer = renderer;
    }
}
```



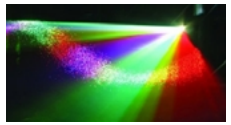
Rendering in the `VisualizationView` Class

```
public void setRenderer(VisualizationRenderer renderer)
{
   this.renderer = renderer;
}

protected void postRendering(Graphics g)
{
   renderer.postRendering(g, model, this);
}

protected void preRendering(Graphics g)
{
   renderer.preRendering(g, model, this);
}

protected void render(Graphics g)
{
   renderer.render(g, model, this);
}
```



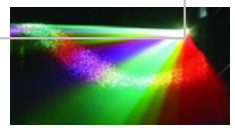
Rendering in the PlainVisualizationRenderer

```
public void render(Graphics           g,
                   Visualization      model,
                   VisualizationView  view)
{

   Iterator<SimpleContent>   iter;
   SimpleContent             c;

   iter = model.iterator();
   while (iter.hasNext())
   {
      c  = iter.next();
      if (c != null) c.render(g);
   }
}
```



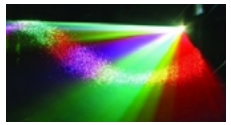
Structure of the ScaledVisualizationRenderer

```
package visual;

import java.awt.*;
import java.util.*;

public class      ScaledVisualizationRenderer
       implements VisualizationRenderer
{
    private double                   height, scaleX, scaleY, width;
    private VisualizationRenderer    decorated;

    public ScaledVisualizationRenderer(VisualizationRenderer decorated,
                                       double width, double height)
    {
       this.decorated = decorated;
       this.width     = width;
       this.height    = height;
    }

    public void render(Graphics           g,
                       Visualization      model,
                       VisualizationView  view)
    {
       decorated.render(g, model, view);
    }
}
```



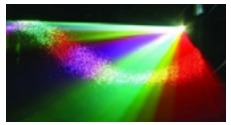
Pre-Rendering in the ScaledVisualizationRenderer

```
public void preRendering(Graphics          g,
                         Visualization     model,
                         VisualizationView view)
{
   Dimension         size;
   Graphics2D        g2;

   g2   = (Graphics2D)g;
   size = view.getSize();
   scaleX = size.getWidth()  / width;
   scaleY = size.getHeight() / height;

   g2.scale(scaleX, scaleY);

   decorated.preRendering(g, model, view);
}
```

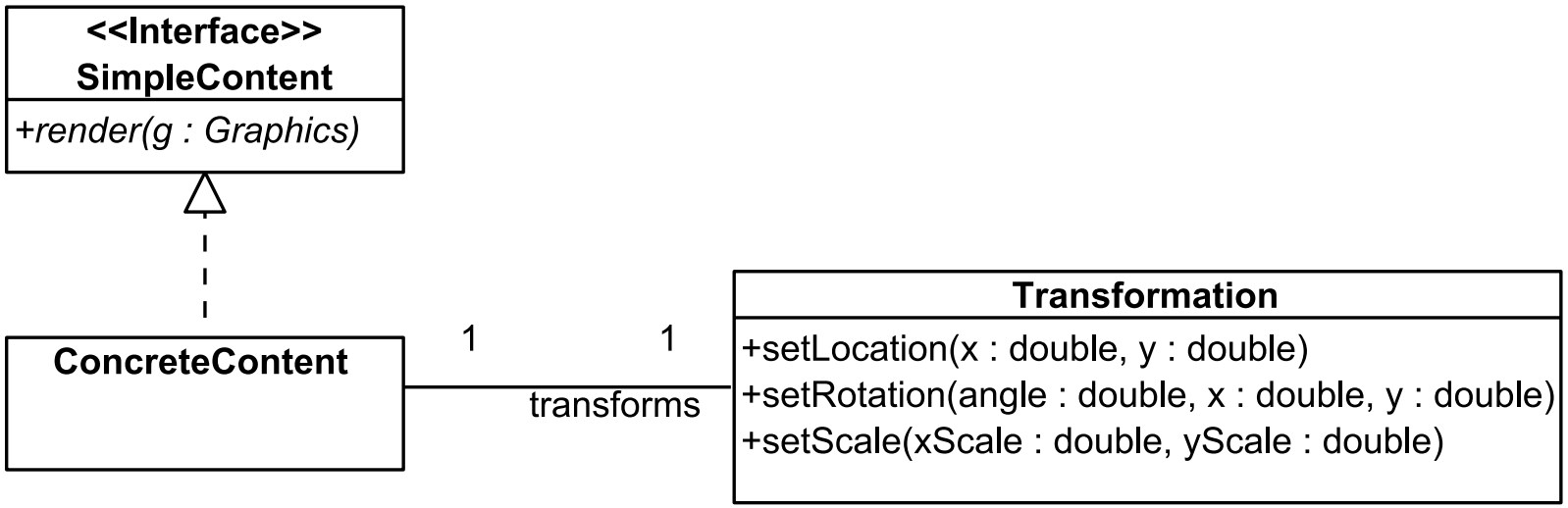


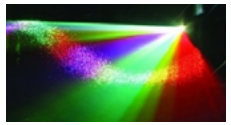
Post-Rendering in the ScaledVisualizationRenderer

```
public void postRendering(Graphics           g,
                          Visualization      model,
                          VisualizationView  view)
{
   Graphics2D   g2;

   g2 = (Graphics2D)g;
   g2.scale(1.0/scaleX, 1.0/scaleY);

   decorated.postRendering(g, model, view);
}
```



Alternative 1





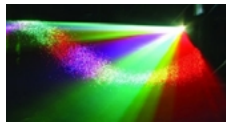
Alternative 1 - Advantages and Disadvantages

- Advantage:

  The classes are very cohesive

- Disadvantage:

  Since `Transformation` objects are completely coupled to the `SimpleContent` objects that they operate on, there is no real reason to think of them as distinct objects.



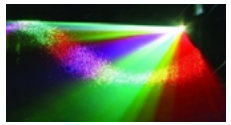
Alternative 2

- The Change:

  Add the required methods to the `SimpleContent` interface.

- Shortcomings:

  What are the shortcomings?



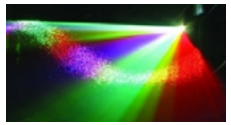
Alternative 2

- The Change:

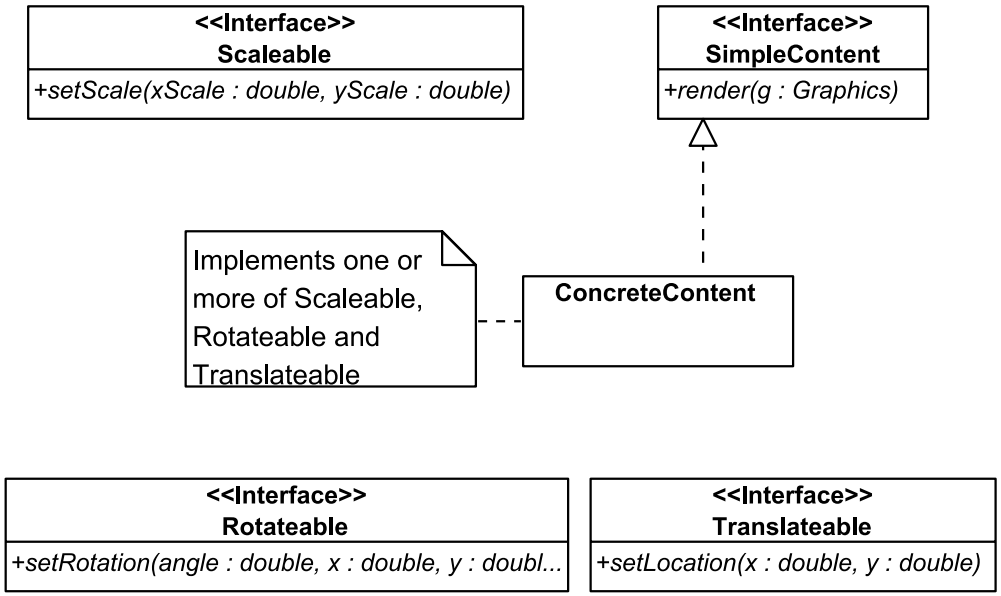
  Add the required methods to the `SimpleContent` interface.

- Shortcomings:

  Seems like 'overkill' and could be confusing in situations that do not require transformations.

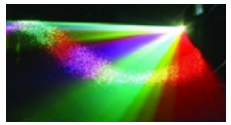


Alternative 3

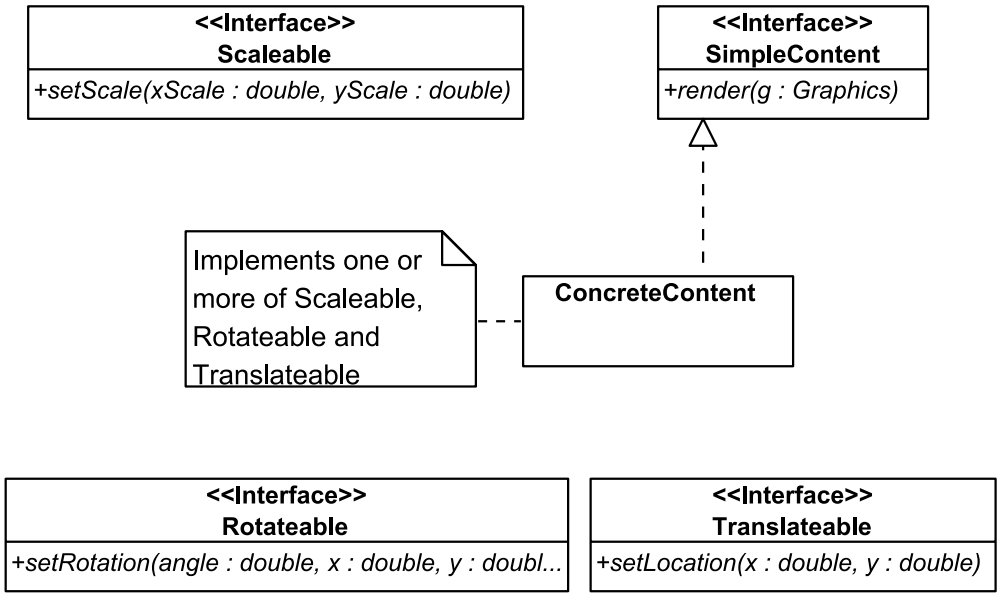


Shortcomings:

What are the shortcomings?

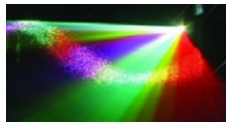


Alternative 3

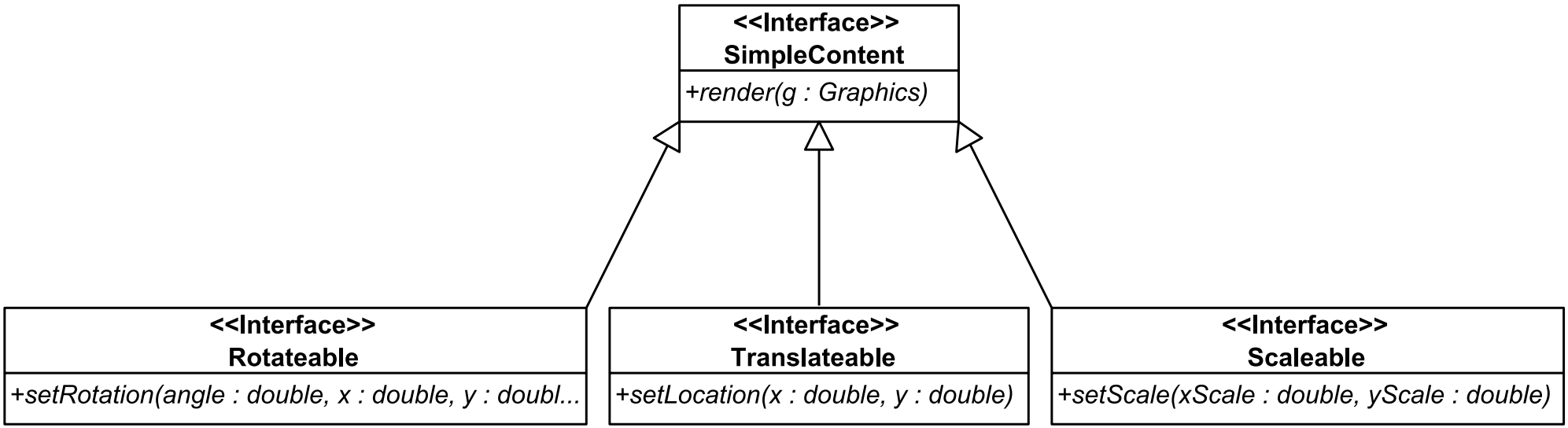


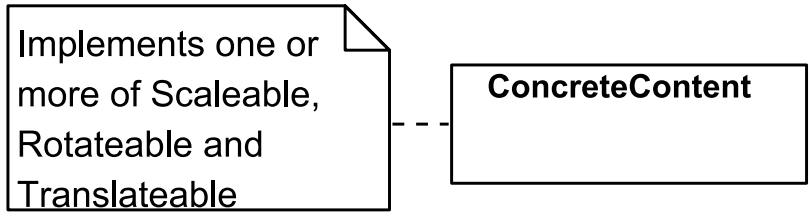
Shortcomings:

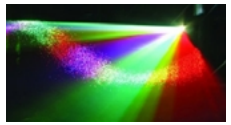
It is hard to imagine a multimedia program that would transform visual content without also rendering it.



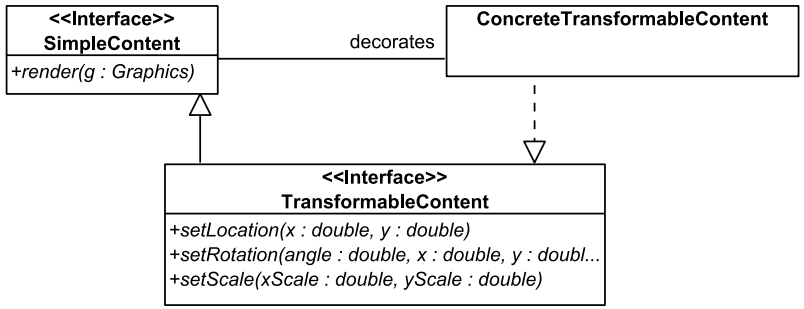
Alternative 4





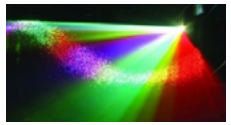


Alternative 4.1

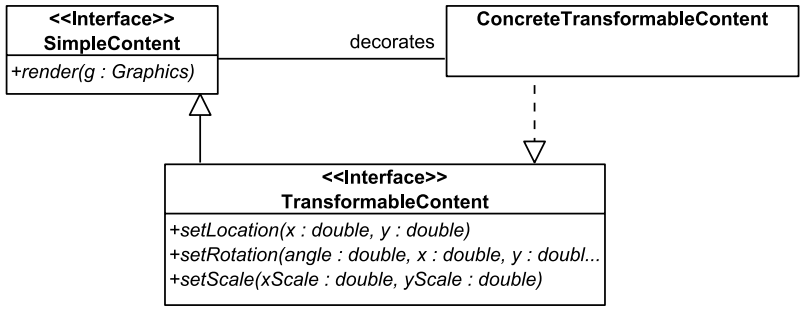


Shortcomings:

What are the shortcomings?

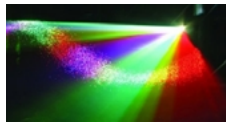


Alternative 4.1

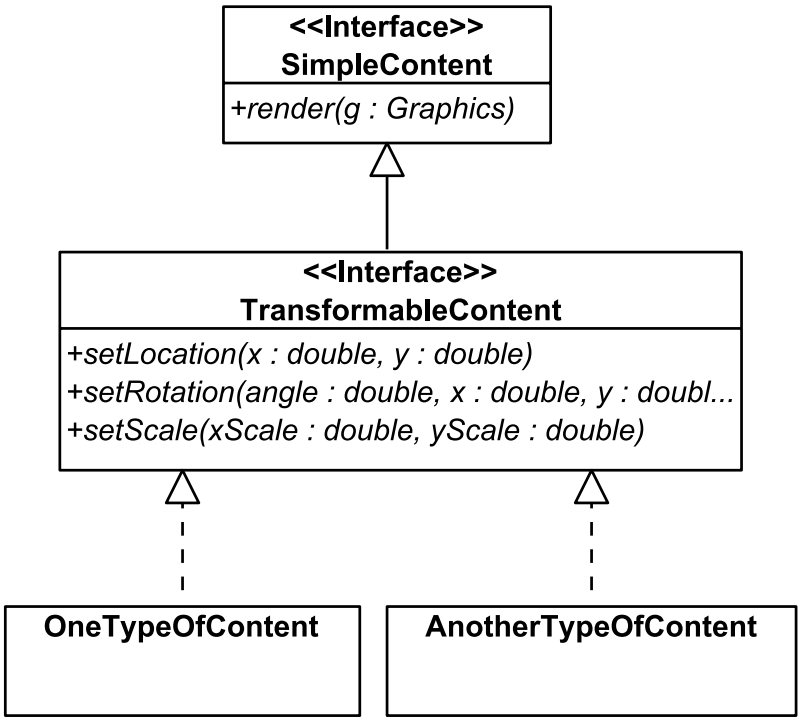


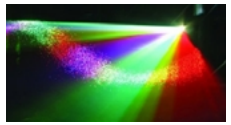
Shortcomings:

It does not seem like there is any need to add capabilities to a particular content object rather than to the class of content objects.

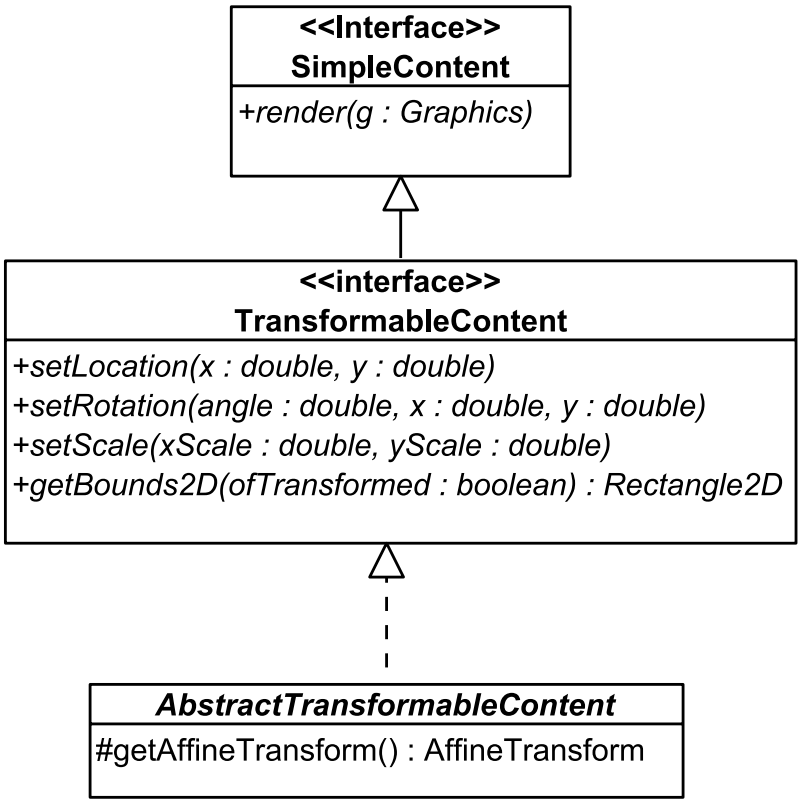


Alternative 4.2



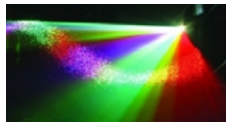


Alternative 4.3

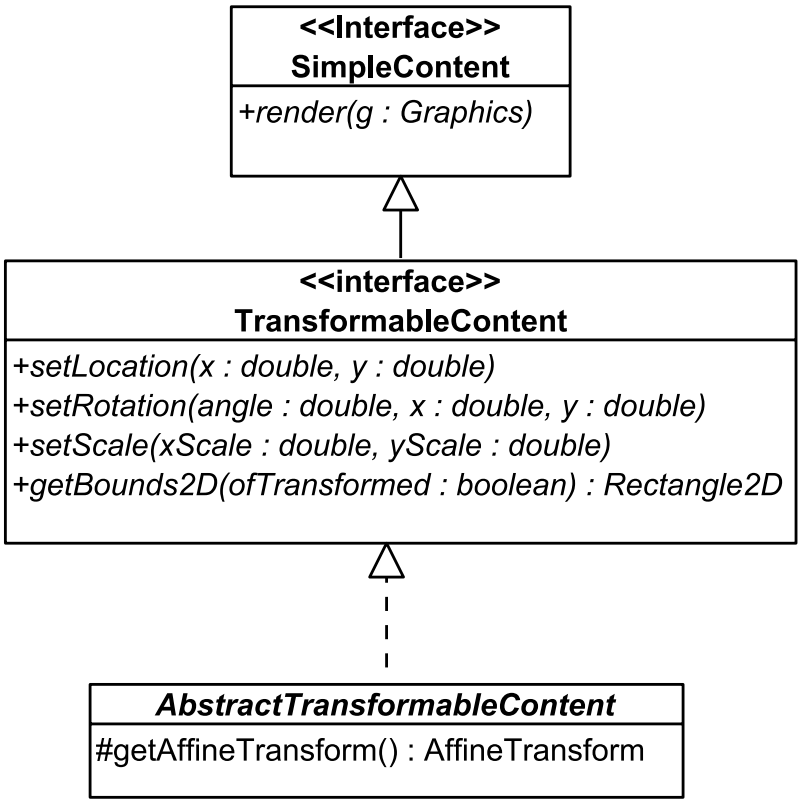


Advantages:

What are the advantages?

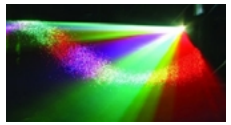


Alternative 4.3



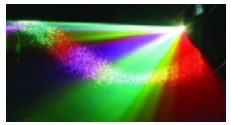
Advantages:

Reduces code duplication and coupling with no loss of cohesiveness



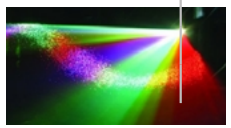
TransformableContent

```
package visual.statik;

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

public interface TransformableContent extends SimpleContent
{
    public abstract void setLocation(double x, double y);

    public abstract void setRotation(double angle,
                                     double x, double y);

    public abstract void setScale(double xScale, double yScale);

    public abstract Rectangle2D getBounds2D(boolean ofTransformed);
}
```



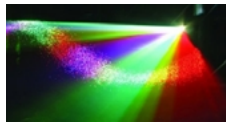
Structure of AbstractTransformableContent

```
package visual.statik;

import java.awt.geom.*;

public abstract class AbstractTransformableContent
                implements TransformableContent
{
    protected boolean          relocated, rerotated, rescaled;
    protected double           angle;
    protected double           xScale, yScale;
    protected double           x, y;
    protected double           xRotation, yRotation;

    public AbstractTransformableContent()
    {
       setTransformationRequired(false);

       angle     = 0.0;
       xScale    = 1.0;
       yScale    = 1.0;
       x         = 0.0;
       y         = 0.0;
       xRotation = 0.0;
       yRotation = 0.0;
    }
    protected void setTransformationRequired(boolean required)
    {
       relocated = required;
```



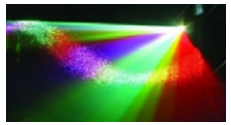
Structure of AbstractTransformableContent (cont.)

```
        rerotated = required;
        rescaled  = required;
    }

    protected boolean isTransformationRequired()
    {
        return (relocated || rerotated || rescaled);
    }
}
```



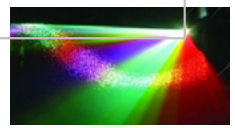
Getters in AbstractTransformableContent

```
public Rectangle2D getBounds2D()
{
   return getBounds2D(true);
}

public abstract Rectangle2D getBounds2D(boolean transformed);
```



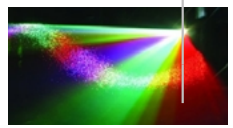
Setters in AbstractTransformableContent

```
public void setLocation(double x, double y)
{
   this.x = x;
   this.y = y;
   relocated = true;
}

public void setRotation(double angle, double x, double y)
{
   this.angle = angle;
   xRotation  = x;
   yRotation  = y;
   rerotated  = true;
}

public void setScale(double xScale, double yScale)
{
   this.xScale = xScale;
   this.yScale = yScale;
   rescaled    = true;
}

public void setScale(double scale)
{
   setScale(scale, scale);
}
```



getAffineTransform()

```
protected AffineTransform getAffineTransform()
{
   // We could use an object pool rather than local
   // variables.  Rough tests estimate that this method
   // would require 1/3 as much time with an object pool.
   AffineTransform     at, rotation, scaling, translation;
   Rectangle2D         bounds;

   // Start with the identity transform
   at = AffineTransform.getTranslateInstance(0.0, 0.0);

   if (rerotated)
   {
      bounds = getBounds2D(false);


      rotation = AffineTransform.getRotateInstance(angle,
                                          xRotation, yRotation);
      at.preConcatenate(rotation);
   }

   if (rescaled)
   {
      scaling = AffineTransform.getScaleInstance(xScale, yScale);
      at.preConcatenate(scaling);
   }

   if (relocated)
```



getAffineTransform() (cont.)

```
        {
            translation = AffineTransform.getTranslateInstance(x, y);
            at.preConcatenate(translation);
        }

        return at;
    }
```