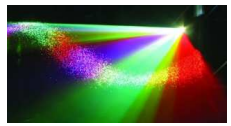
Chapter 8
Sampled Dynamic Visual Content

The Design and Implementation of Multimedia Software

David Bernstein

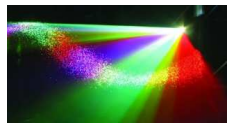
Jones and Bartlett Publishers

www.jbpub.com

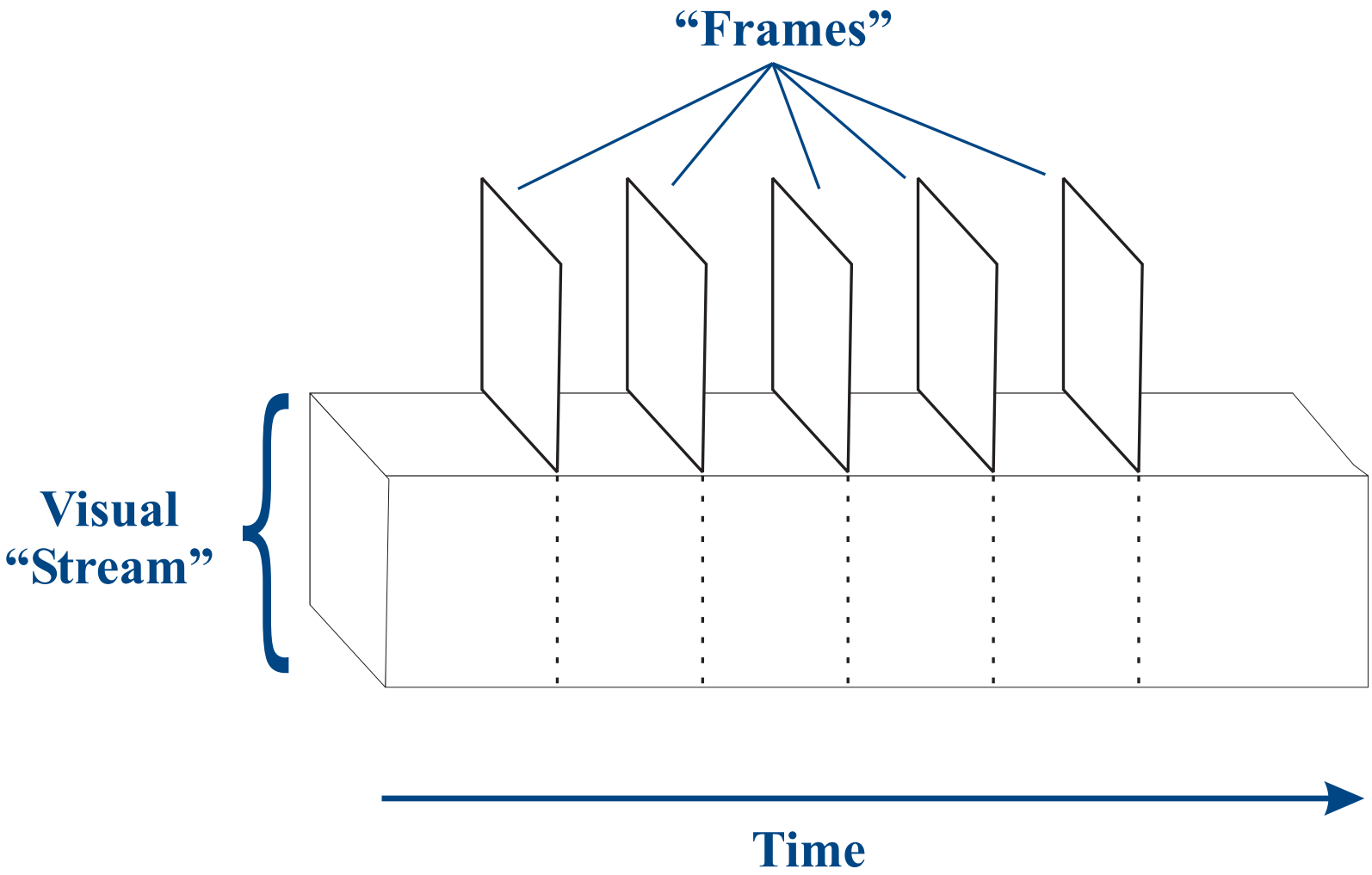


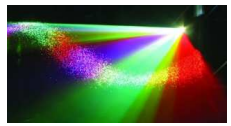
About this Chapter

- Dynamic visual content is any visual content that changes over time.
- Our interest is with visual content that changes over time in a way that causes the perception of *apparent motion*.
- The sampling of dynamic visual content involves sampling from all possible points in time.



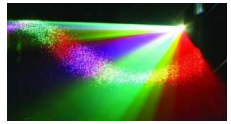
Sampling from a Visual Stream





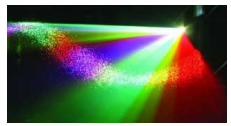
A Comparison

- In sampled **static** visual content each pixel is treated as atomic.
- In sampled **dynamic** visual content each frame is atomic.



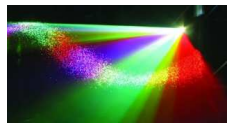
Levels of Abstraction/Resolution

- At a lower level of abstraction, each frame actually consists of static visual content, and that content can be either sampled or described.
- At the current (higher) level of abstraction, the nature of each frame is of no consequence.



What’s Next

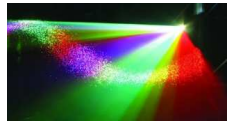
We need some instant gratification.



Requirements

Our goal is to create a system that behaves like a flip book for `SimpleContent` objects. That is, the system must:

F8.1 Render a sequence of `SimpleContent` objects over time.



Satisfying this Requirement

- What We Have:

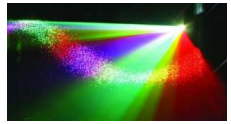
  An object that manages the collection of `SimpleContent` objects.

  A process that controls the rendering of the objects in the sequence.

  A GUI component that presents the `SimpleContent` objects.

- What We Need:

  An 'enhanced' `Visualization` object to manage the collection of frames (which are `SimpleContent` objects).



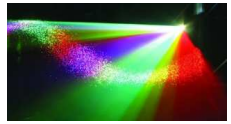
Alternative 1 

- Approach:

  Add code to the `Visualization` class.

- Shortcomings:

  What are the shortcomings?



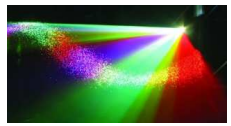
Alternative 1 

- Approach:

  Add code to the `Visualization` class.

- Shortcomings:

  It makes this class more complicated and more difficult to understand for those that have no interest in dynamic content.



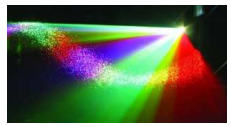
Alternative 2

- Approach:

  Use the decorator pattern.

- Shortcomings:

  What are the shortcomings?



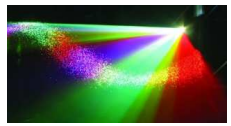
Alternative 2

- Approach:

  Use the decorator pattern.

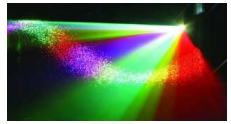
- Shortcomings:

  It is hard to imagine a situation in which, at run time, one would want to add these kinds of capabilities to a `Visualization` object.



Alternative 3 

Create a specialization of the `Visualization` class (called the `Screen` class) that includes the attributes and methods needed to manage and present dynamic content.



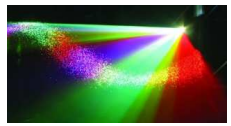
Alternative 3.1

- Approach:

  Have the `Screen` class iterate through the collection of `SimpleContent` objects sending each a `render()` message in turn.

- Shortcomings:

  What are the shortcomings?



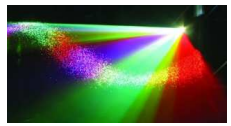
Alternative 3.1

- Approach:

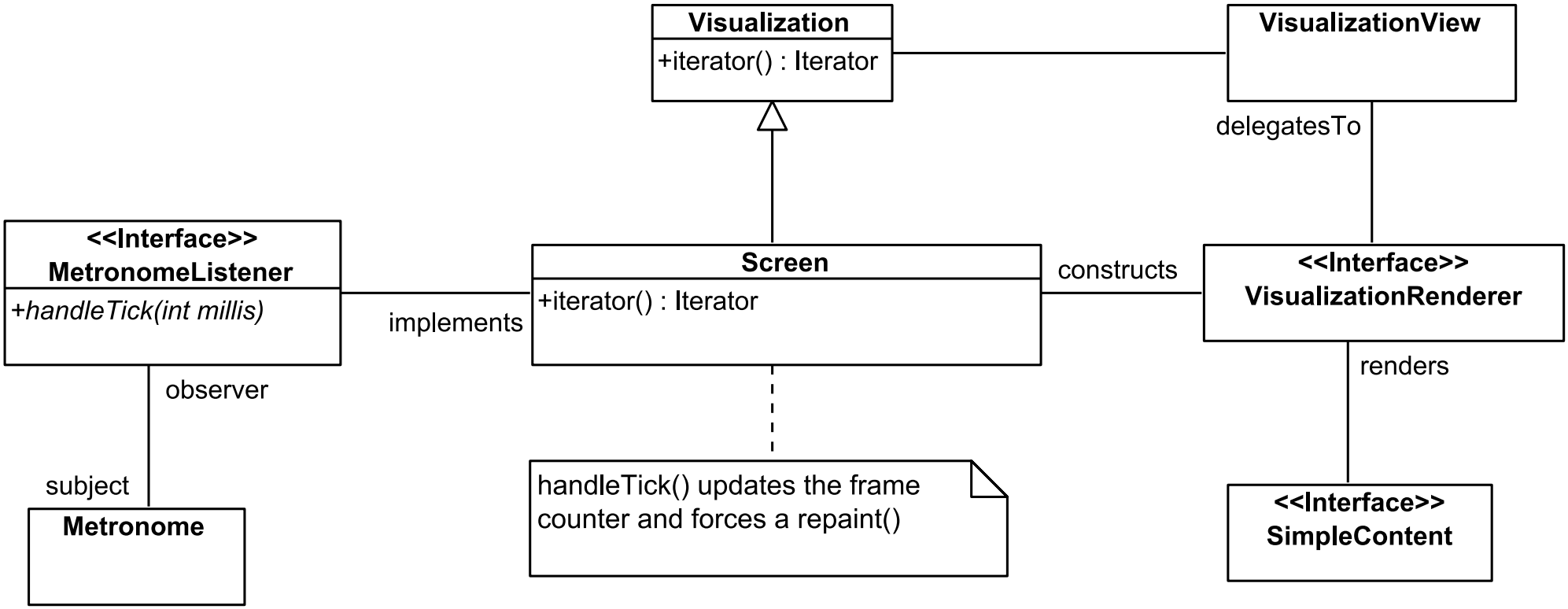
  Have the `Screen` class iterate through the collection of `SimpleContent` objects sending each a `render()` message in turn.

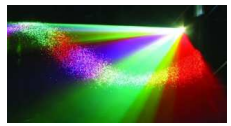
- Shortcomings:

  It does not provide control over the amount of time each `SimpleContent` object is visible.

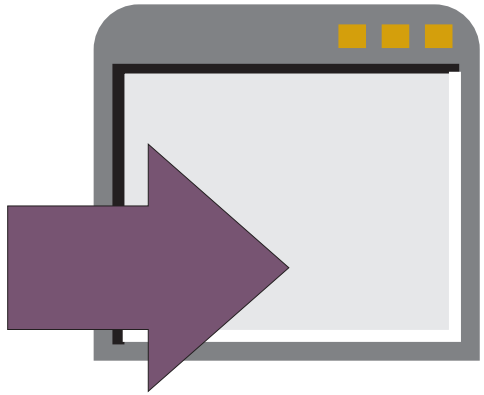


Alternative 3.2





Screen - Demonstrations

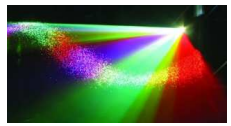


In `examples/chapter`:

```
java -cp multimedia2.jar:examples.jar ScreenDemo solidclock
```

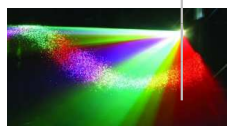
```
java -cp multimedia2.jar:examples.jar ScreenDemo scribble
```

```
java -cp multimedia2.jar:examples.jar ScreenDemo movingrectangle
```



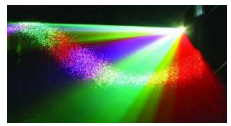
Screen – Structure

```
package visual.dynamic.sampled;

import java.util.*;

import collectionframework.*;
import event.*;
import visual.*;
import visual.statik.SimpleContent;

public class Screen extends    Visualization
                    implements MetronomeListener
{
  private   boolean                       repeating;
  private   int                           frameNumber;
  private   Iterator<SimpleContent>       frames;
  protected Metronome                     metronome;
  protected SimpleContent                 currentFrame;

  public static final int                 DEFAULT_FRAME_DELAY = 42;

  public Screen()
  {
    this(DEFAULT_FRAME_DELAY);
  }

  public Screen(int frameRate)
  {
    this(new Metronome((int)(1000.0 / frameRate)));
```



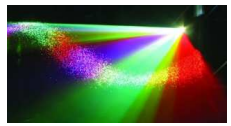
Screen – Structure (cont.)

```
  }

  public Screen(Metronome metronome)
  {
    super();
    this.metronome = metronome;
    metronome.addListener(this);
    setRepeating(false);
  }

  public void start()
  {
    reset();
    if (frames != null) metronome.start();
  }

  public void stop()
  {
    metronome.stop();
  }
}
```



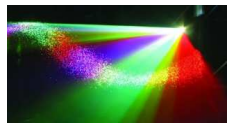
Screen – Setters

```
public void setRepeating(boolean repeating)
{
  this.repeating = repeating;
}
```



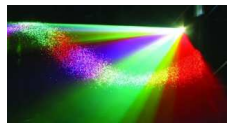
Screen – getFrameNumber()

```
public int getFrameNumber()
{
  return frameNumber;
}
```



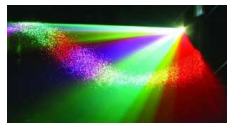
Screen – Initiating Rendering

```
public void handleTick(int time)
{
  if (frames != null)
  {
    // See if we're done
    if (frameNumber < 0)
    {
      if   (repeating) reset();
      else             stop();
    }

    // Start the rendering process (i.e., request that the
    // paint() method be called)
    repaint();

    // Advance the frame
    advanceFrame();
  }
}
```



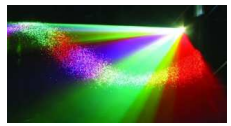
Screen – Advancing the Frame

```
private void advanceFrame()
{
  if ((frames != null) && (frames.hasNext()))
  {
    currentFrame = frames.next();
    frameNumber++;
  }
  else
  {
    currentFrame = null;
    frameNumber  = -1;
  }
}
```



An Important Modification

The `Screen` class modifies the behavior of its parent `Visualization` class in one important way – its `iterator()` method does not contain all of the frames, it only contains the current frame.

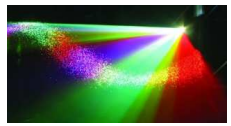


Screen – The Current Frame

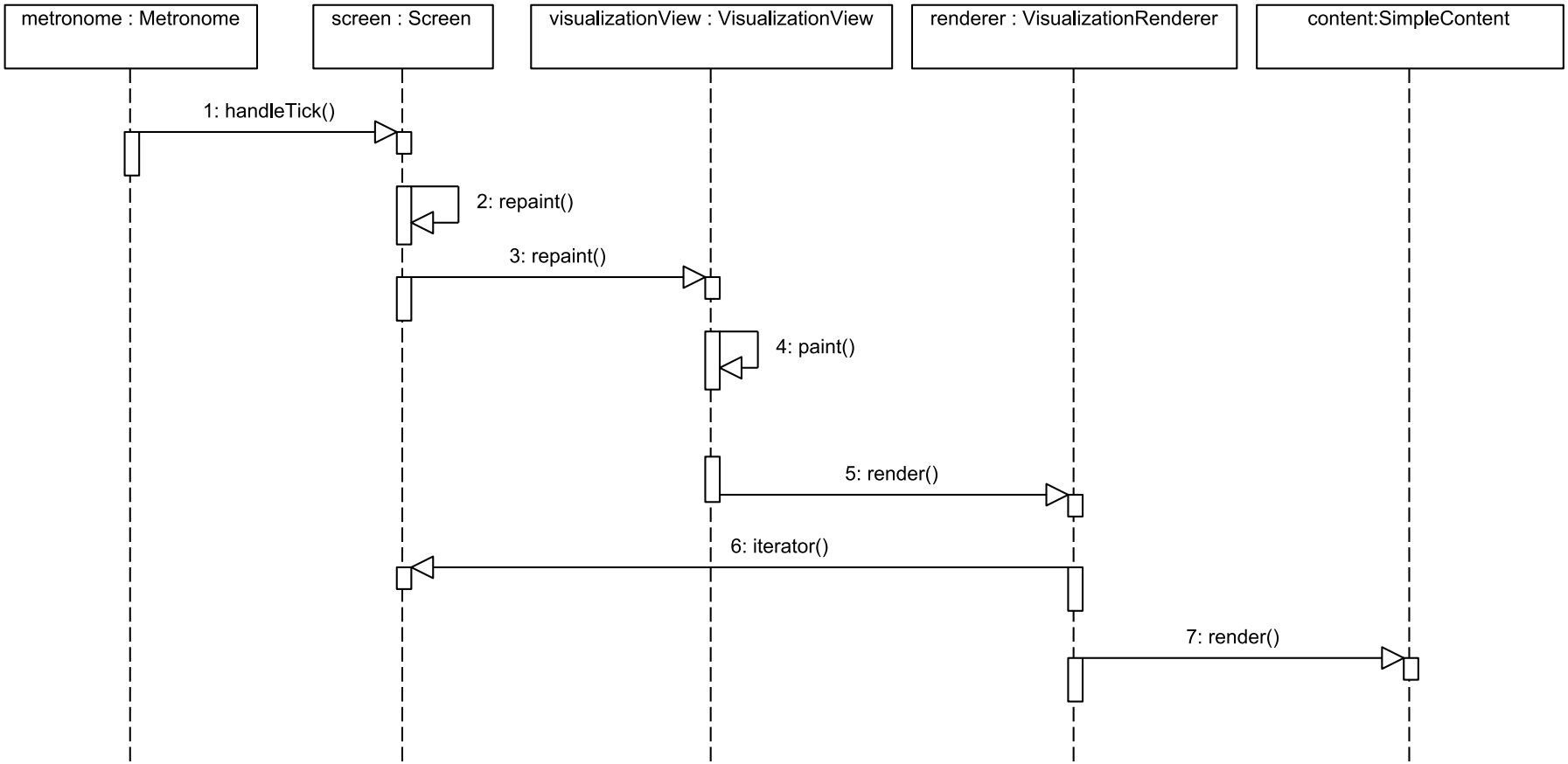
```
protected NullIterator<SimpleContent> currentFrameIterator;
```

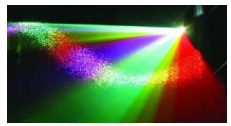
```
  currentFrameIterator = new NullIterator<SimpleContent>();
```

```
public Iterator<SimpleContent> iterator()
{
  currentFrameIterator.setElement(currentFrame);
  if (frameNumber < 0) currentFrameIterator.clear();

  return currentFrameIterator;
}

public Iterator<SimpleContent> iterator(boolean all)
{
  Iterator<SimpleContent>       result;

  if  (all) result = super.iterator();
  else      result = iterator();

  return result;
}
```



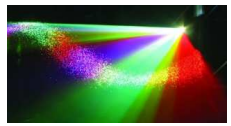
The Process of Rendering Sampled Dynamic Content





What's Next

We need to consider the encapsulation of sampled dynamic content.

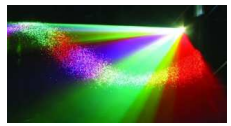


The Collection

- Desirable Properties:

  What are they?

- A Good Choice:



The Collection

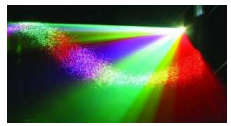
- Desirable Properties:

  Can return either an associated `Iterator` or an associated `ListIterator` (that supports bi-directional traversal).

  Thread safe.

- A Good Choice:

  Any thoughts?



The Collection

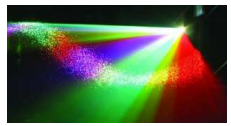
- Desirable Properties:

  Can return either an associated `Iterator` or an associated `ListIterator` (that supports bi-directional traversal).

  Thread safe.

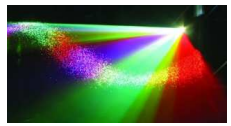
- A Good Choice:

  `CopyOnWriteArrayList`



What's Next

We need to consider the rendering of individual frames.



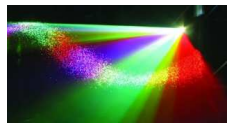
A Small Problem

- You Might Think:

  Since `sampled.Content` objects know how to render themselves there is no reason to discuss the rendering of individual frames.

- However:

  When rendered in quick succession, the frames have a tendency to "flicker".



Double Buffering

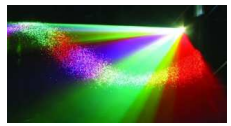
- The Idea:

  The rendering engine writes to a *background image* or *screen buffer*.

  Then this buffer is transfered to the screen in its entirety (very quickly).

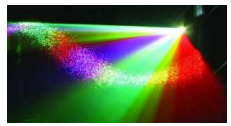
- Where to Put this Functionality:

  The `VisualizationView` class.



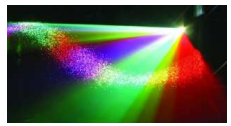
Attributes for Double Buffering

```
// Attributes used for double-buffering
protected boolean               useDoubleBuffering;
protected Graphics2D            bg;
protected Image                 offscreenImage;
protected int                   height, width;
```



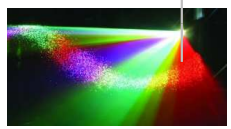
The Buffer

```
private Graphics2D createOffscreenBuffer()
{
  Dimension                    d;

  d = getSize();
  if ((d.height != height) || (d.width != width))
  {
    height = d.height;
    width  = d.width;

    offscreenImage = createImage(width, height);
    bg = (Graphics2D)offscreenImage.getGraphics();
    bg.setClip(0,0,width,height);
  }

  return bg;
}
```



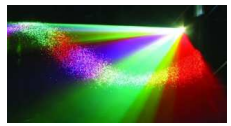
Rendering

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

  if (useDoubleBuffering) bg = createOffscreenBuffer();
  else                    bg = (Graphics2D)g;

  if (bg != null)
  {
    // Perform necessary operations before rendering
    preRendering(bg);

    // Render the visual content
    render(bg);

    // Perform necessary operations after rendering
    postRendering(bg);

    if (useDoubleBuffering)
    {
      // Put the offscreen image on the screen
      g.drawImage(offscreenImage, 0, 0, null);

      // Reset the clipping area
      bg.setClip(0,0,width,height);
    }
```



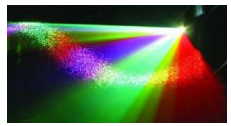
Rendering (cont.)

```
    }
}
```



What's Next

We need to consider operations on multiple frames.

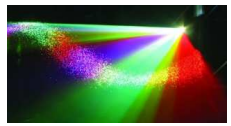


Motivation

Definition

A *straight cut* involves rendering frame $n + 1$, in its entirety, immediately after frame n.

- As it has been implemented thus far, the `Screen` class always uses straight cuts.
- An alternative way to render a sequence of frames is to use a *transition*.

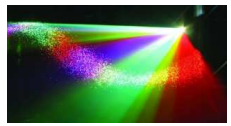


Common Fades

Definition

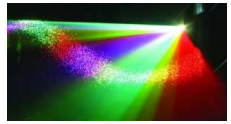
A *fade from black* is a progressive brightening and a *fade to black* is a progressive darkening.

The fade from black is a specific example of a *fade-in* (or *fade-up*) and the fade to black is a specific example of a *fade-out* (or *fade-to*).



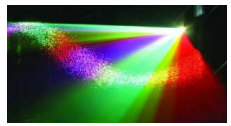
Setting the Destination Pixels to Black

```
protected void setDestinationPixels(Graphics g)
{
  Graphics2D       g2;
  Rectangle        r;

  g2 = (Graphics2D)g;

  r = g2.getClipBounds();

  g2.setComposite(AlphaComposite.Src);
  g2.setColor(g2.getBackground());
  g2.fill(r);
  g2.draw(r);
}
```



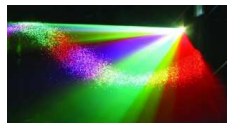
Pre-Rendering

```
g2 = (Graphics2D)g;
originalComposite = g2.getComposite();

alpha = ((float)(frame - first + 1))/(float)duration;
if (direction == FADE_OUT) alpha = 1.0f - alpha;

if      (alpha > 1.0f) alpha = 1.0f;
else if (alpha < 0.0f) alpha = 0.0f;

setDestinationPixels(g2);

ac  = AlphaComposite.getInstance(AlphaComposite.SRC_OVER,
    alpha);
g2.setComposite(ac);
```



Post-Rendering

```
        g2 = (Graphics2D)g;
        if (originalComposite != null) g2.setComposite(originalComposite);
```



Understanding Dissolves

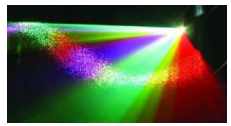
- Usual Description:

  Frame n fades out while frame $n + 1$ fades in.

  A fade-in without an intermediate solid background.

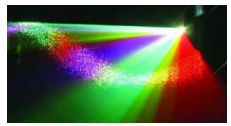
- Approach:

  Specialize the `Fade` class and override the `setDestinationPixels()` method so that it does nothing.



Dissolve

```
package visual.dynamic.sampled;

import java.awt.*;

public class Dissolve extends Fade
{
  public Dissolve(int first, int duration)
  {
    super(FADE_IN, first, duration);
  }

  protected void setDestinationPixels(Graphics g)
  {
    // Use the last frame
  }
}
```

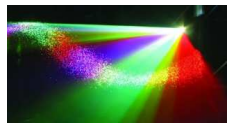


Basics

Definition

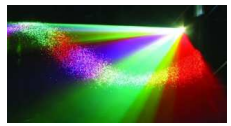
In a *wipe*, a frame (or frames) is clipped to a series of (one or more) geometric shapes.

- Circle Wipe
- Fly-On Wipe
- Line Wipe
- Quivering wipe (e.g., in "Wayne's World" in *Saturday Night Live*)
- Rectangle Wipe
- Star Wipe (e.g., Homer's video of Flanders in *The Simpsons*)



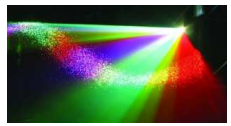
RectangleWipe – calculateClip()

```
protected Rectangle2D calculateClip(float width,
    float height, int   frame)
{
  float              h, w, x, y;
  Rectangle2D        clip;

  w = scale*frame*width;
  h = scale*frame*height;
  x = width/2.0f  - w/2.0f;
  y = height/2.0f - h/2.0f;

  clip = new Rectangle2D.Float(x, y, w, h);

  return clip;
}
```



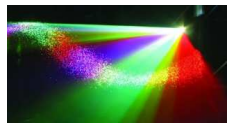
RectangleWipe – Pre-Rendering

```
g2 = (Graphics2D)g;
originalClip = g2.getClip();

bounds = g2.getClipBounds();
height = (float)(bounds.getHeight());
width  = (float)(bounds.getWidth());


clip = calculateClip(width, height, frame-first+1);
g2.setClip(clip);
```



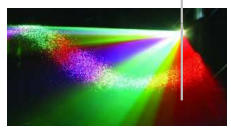
RectangleWipe – Post-Rendering

```
g2 = (Graphics2D)g;

if (originalClip != null) g2.setClip(originalClip);
```



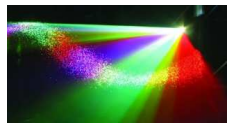
LineWipe

```
protected Rectangle2D calculateClip(float width,
    float height,
    int frame)
{
  float              h, w, x, y;
  Rectangle2D        clip;

  w = width;
  h = height;
  x = 0.0f;
  y = 0.0f;

  if        (direction == RIGHT)
  {
    w = scale*frame*width;
    h = height;
    x = 0.0f;
    y = 0.0f;
  }
  else if (direction == LEFT)
  {
    w = scale*frame*width;
    h = height;
    x = width - w;
    y = 0.0f;
  }
  else if (direction == UP)
  {
```

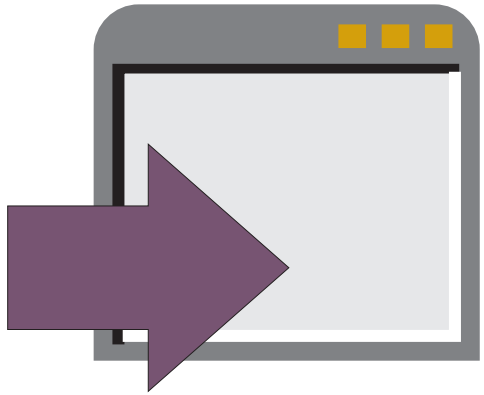


LineWipe (cont.)

```
      w = width;
      h = scale*frame*height;
      x = 0.0f;
      y = height - h;
    }
    else
    {
      w = width;
      h = scale*frame*height;
      x = 0.0f;
      y = 0.0f;
    }

    clip = new Rectangle2D.Float(x, y, w, h);

    return clip;
  }
```

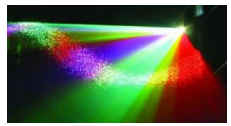


Transitions – Demonstration



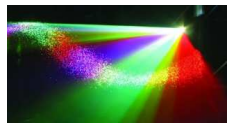
In `examples/chapter`:

```
java -cp multimedia2.jar:examples.jar TransitionDemo -Xmx256m
```



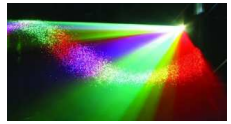
What's Next

We need to consider operations on individual frames.



Don't Get Confused

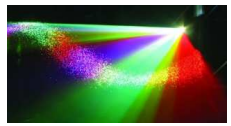
- A variety of operations can be performed on the individual frames when they are created/produced.
- One can also perform operations on individual frames while they are being presented in a dynamic setting.



Superimpositions

Definition

A *superimposition* is visual content that is to be added to an existing frame while it is being rendered.



Examples of Superimpositions

- 'Text':

  Films superimpose credits

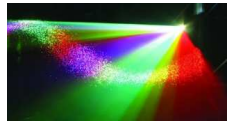
  Sporting events superimpose statistics

  Television shows superimpose closed-captioning

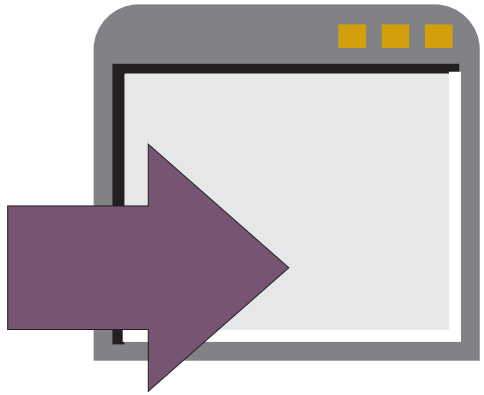
- 'Graphics':

  Television shows superimpose network logos

  Football games superimpose first-down lines



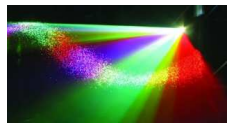
Superimpositions – Demonstrations



In `examples/chapter`:

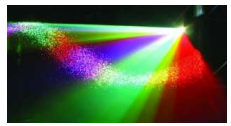
```
java -cp multimedia2.jar:examples.jar SuperimpositionDemo
```

```
java -cp multimedia2.jar:examples.jar SuperimpositionDemo buzzy
```



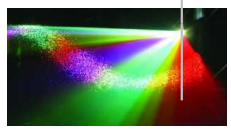
AbstractSuperimposition – Construction

```
public AbstractSuperimposition(int first,
    int duration, int position)
{
  super(first, duration);

  this.position = SwingConstants.SOUTH_EAST;
  if ((position == SwingConstants.NORTH)      ||
      (position == SwingConstants.NORTH_EAST) ||
      (position == SwingConstants.EAST)       ||
      (position == SwingConstants.SOUTH_EAST) ||
      (position == SwingConstants.SOUTH)      ||
      (position == SwingConstants.SOUTH_WEST) ||
      (position == SwingConstants.WEST)       ||
      (position == SwingConstants.NORTH_WEST) ||
      (position == SwingConstants.CENTER    )    )
  {
    this.position = position;
  }
}
```



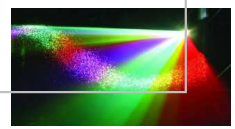
AbstractSuperimposition – Registration

```
protected Point2D calculateRegistrationPoint(double frameWidth,
    double frameHeight, double siWidth, double siHeight)
{
  double         left, top;

  top  = 0.0;
  left = 0.0;
  if       (position == SwingConstants.NORTH)
  {
    top  = siHeight;
    left = frameWidth/2.0 - siWidth/2.0;
  }
  else if  (position == SwingConstants.NORTH_EAST)
  {
    top  = siHeight;
    left = frameWidth - siWidth - 1;
  }
  else if  (position == SwingConstants.EAST)
  {
    top  = frameHeight/2.0 - siHeight/2.0;
    left = frameWidth - siWidth - 1;
  }
  else if  (position == SwingConstants.SOUTH_EAST)
  {
    top  = frameHeight - siHeight - 1;
    left = frameWidth - siWidth - 1;
  }
  else if  (position == SwingConstants.SOUTH)
```



AbstractSuperimposition – Registration (cont.)

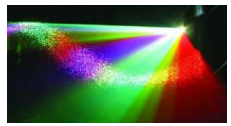
```
    {
      top  = frameHeight - siHeight - 1;
      left = frameWidth/2.0 - siWidth/2.0;
    }
    else if  (position == SwingConstants.SOUTH_WEST)
    {
      top  = frameHeight - siHeight - 1;
      left = 0.0;
    }
    else if  (position == SwingConstants.WEST)
    {
      top  = frameHeight/2.0 - siHeight/2.0;
      left = 0.0;
    }
    else if  (position == SwingConstants.NORTH_WEST)
    {
      top  = siHeight;
      left = 0.0;
    }
    else if  (position == SwingConstants.CENTER)
    {
      top  = frameHeight/2.0 - siHeight/2.0;
      left = frameWidth/2.0 - siWidth/2.0;
    }

    return new Point2D.Double(left, top);
  }
```



TransformableContentSuperimposition

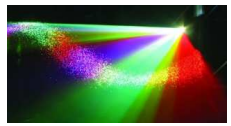
Post-Rendering

```
g2 = (Graphics2D)g;

// Transform the TransformableContent so that
// it is positioned properly
frameBounds   = g2.getClipBounds();
frameWidth    = (double)frameBounds.width;
frameHeight   = (double)frameBounds.height;

contentBounds = content.getBounds2D(false);
contentWidth  = contentBounds.getWidth();
contentHeight = contentBounds.getHeight();


rp = calculateRegistrationPoint(frameWidth,
    frameHeight,
    contentWidth,
    contentHeight);

content.setLocation(rp.getX(), rp.getY());
content.render(g);
```



What's Next

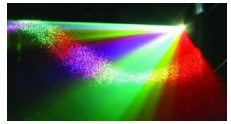
We need to design a sampled dynamic visual content system.



Requirements

F8.2 Support transitions (other than straight cuts) between frames of sampled dynamic content.

F8.3 Support the superimposition of static visual content on one or more frames of sampled dynamic content.

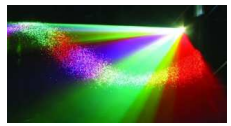


Alternative 1

AbstractTransition
+getLastFrame() : int
+getFirstFrame() : int
+postRendering(g : Graphics, frame : int)
+preRendering(g : Graphics, frame : int)
+hasFinishedAt(frame : int) : boolean
+shouldApplyAt(frame : int) : boolean

AbstractSuperimposition
+calculateRegistrationPoint() : Point2D
+getLastFrame() : int
+getFirstFrame() : int
+postRendering(g : Graphics, frame : int)
+preRendering(g : Graphics, frame : int)
+hasFinishedAt(frame : int) : boolean
+shouldApplyAt(frame : int) : boolean

What are the shortcomings?

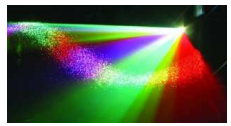


Alternative 1

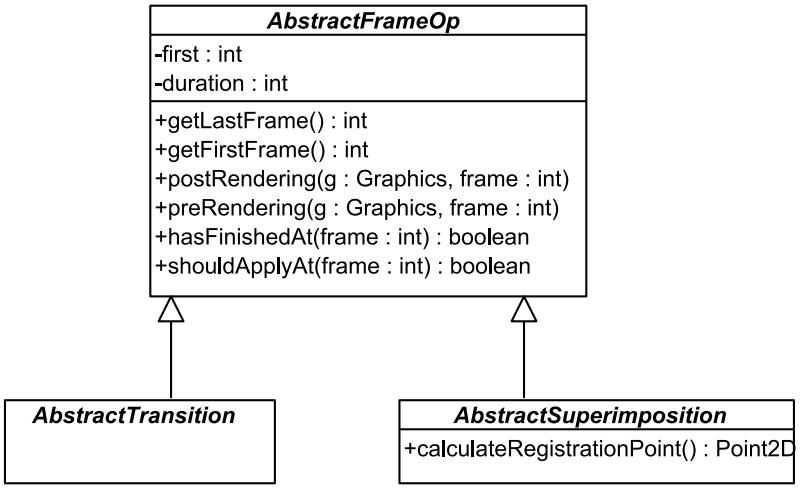
AbstractTransition
+getLastFrame() : int
+getFirstFrame() : int
+postRendering(g : Graphics, frame : int)
+preRendering(g : Graphics, frame : int)
+hasFinishedAt(frame : int) : boolean
+shouldApplyAt(frame : int) : boolean

AbstractSuperimposition
+calculateRegistrationPoint() : Point2D
+getLastFrame() : int
+getFirstFrame() : int
+postRendering(g : Graphics, frame : int)
+preRendering(g : Graphics, frame : int)
+hasFinishedAt(frame : int) : boolean
+shouldApplyAt(frame : int) : boolean

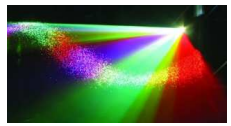
It leads to an enormous amount of code duplication.



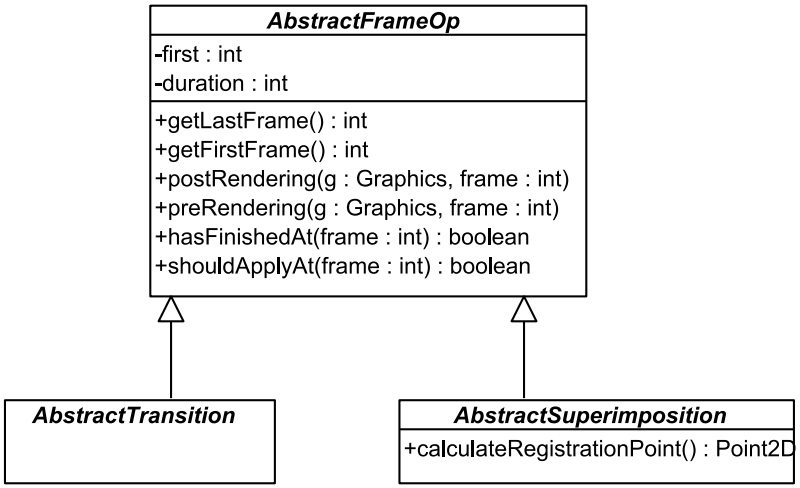
Alternative 2



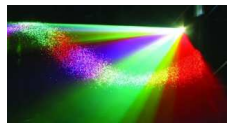
What are the shortcomings?



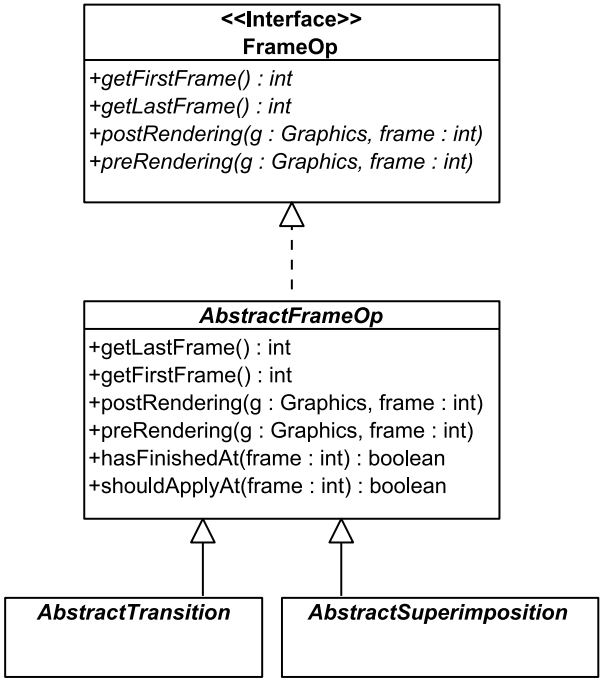
Alternative 2



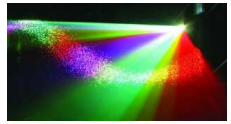
Inflexibility – it does not lend itself to the addition of transition/superimposition classes that do not extend the `AbstractTransition` or `AbstractSuperimposition` classes.



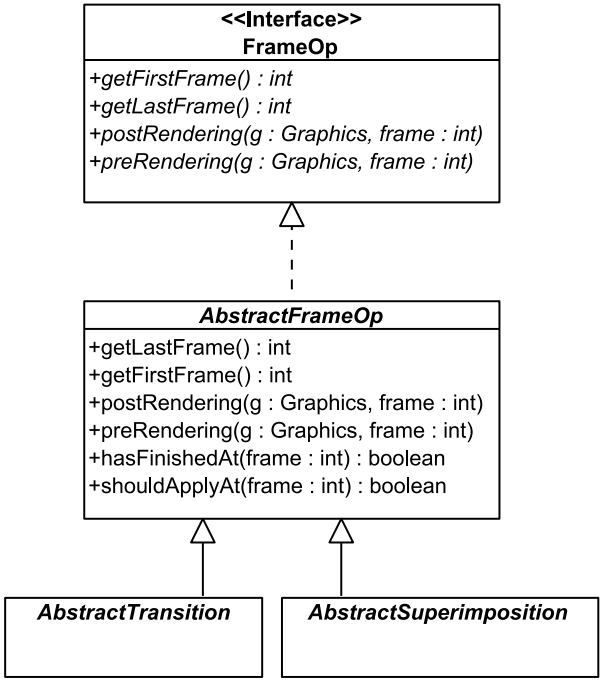
Alternative 3



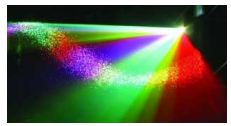
What are the shortcomings?



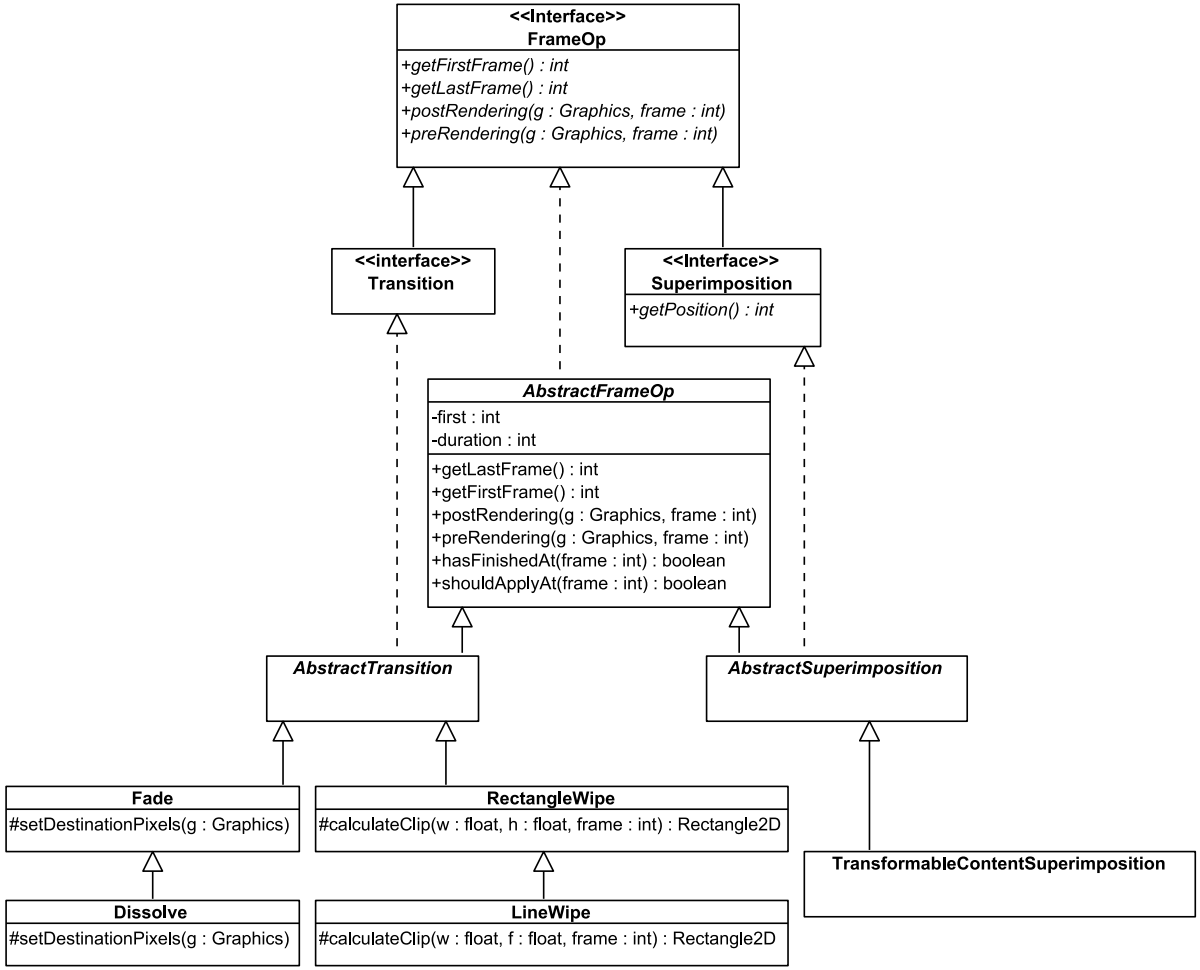
Alternative 3

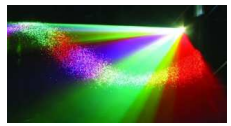


At the level of the interface, it does not distinguish between transitions and superimpositions. Hence, it is not type-safe.



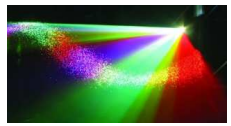
Alternative 4





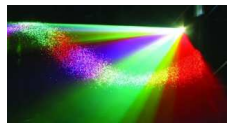
FrameOp

```
package visual.dynamic.sampled;

import java.awt.Graphics;

public interface FrameOp
{
  public abstract int getFirstFrame();

  public abstract int getLastFrame();

  public abstract void postRendering(Graphics g, int frame);

  public abstract void preRendering(Graphics g, int frame);
}
```



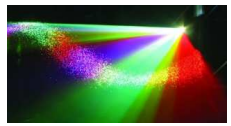
Superimposition

```
package visual.dynamic.sampled;

public interface Superimposition extends FrameOp
{
  public abstract int getPosition();
}
```



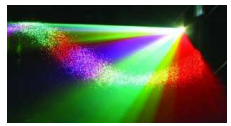
Transition

```
package visual.dynamic.sampled;

public interface Transition extends FrameOp
{
}
```



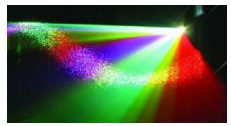
AbstractFrameOp – Structure

```
package visual.dynamic.sampled;

import java.awt.Graphics;

public abstract class      AbstractFrameOp
                implements FrameOp
{
  protected int              duration, first;

  public AbstractFrameOp(int first, int duration)
  {
    this.first    = first;
    this.duration = 0;
    if (duration > 0) this.duration = duration;
  }
}
```



AbstractFrameOp – Methods

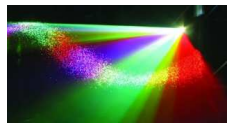
```
public int getFirstFrame()
{
  return first;
}

public int getLastFrame()
{
  return first+duration;
}

protected boolean hasFinishedAt(int frame)
{
  return (frame >= (first+duration-1));
}

protected boolean shouldApplyAt(int frame)
{
  return ((frame >= first) && (frame <= (first+duration-1)));
}
```



Screen – Transitions and Superimpositions

```
// Attributes used for transitions
private IntervalIndexedCollection<Transition> transitions;
```

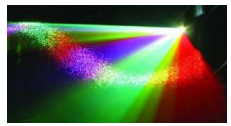
```
// Attributes used for superimpositions
private IntervalIndexedCollection<Superimposition> superimpositions;
```



Screen – Managing Transitions

```
public void addTransition(Transition t)
{
  transitions.add(t, t.getFirstFrame(), t.getLastFrame());
}
```

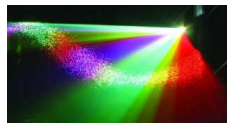
```
public Iterator<Transition> getTransitions()
{
  Iterator<Transition>    result;

  result = null;
  if (frameNumber >= 0) result=transitions.get(frameNumber);

  return result;
}
```



Screen – Managing Superimpositions

```
public void addSuperimposition(Superimposition si)
{
  superimpositions.add(si, si.getFirstFrame(), si.getLastFrame());
}
```

```
public Iterator<Superimposition> getSuperimpositions()
{
  Iterator<Superimposition>    result;

  result = null;
  if (frameNumber >= 0) result=superimpositions.get(frameNumber);

  return result;
}
```



Enhancing the `VisualizationView`

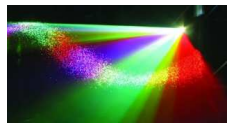
- Recall:

  `VisualizationView` objects delegate their rendering responsibilities to a `VisualizationRenderer`.

  A `VisualizationRenderer` object can be decorated to add functionality.

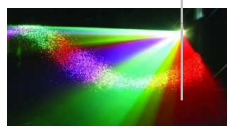
- What We Need Now:

  A decorator called a `ScreenRenderer`.



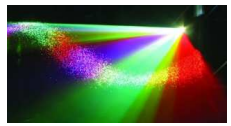
ScreenRenderer – Pre-Rendering

```
public void preRendering(Graphics          g,
    Visualization     model,
    VisualizationView view)
{
  Graphics2D                 g2;
  int                        frameNumber;
  Screen                     smodel;
  Iterator<Transition>       transitions;
  Iterator<Superimposition>  superimpositions;

  g2 = (Graphics2D)g;
  oldComposite = g2.getComposite();
  view.setDoubleBuffered(true);

  // Get information from the model
  smodel           = (Screen)model;
  transitions      = smodel.getTransitions();
  superimpositions = smodel.getSuperimpositions();
  frameNumber      = smodel.getFrameNumber();

  // Apply the transitions
  if (transitions != null)
  {
    while (transitions.hasNext())
    {
      transitions.next().preRendering(g, frameNumber);
    }
  }
```



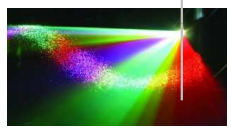
ScreenRenderer – Pre-Rendering (cont.)

```
    // Apply the superimpositions
    if (superimpositions != null)
    {
      while (superimpositions.hasNext())
      {
        superimpositions.next().preRendering(g, frameNumber);
      }
    }
  }
```



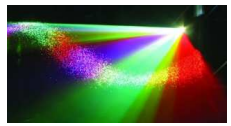
ScreenRenderer – Post-Rendering

```
public void postRendering(Graphics          g,
    Visualization     model,
    VisualizationView view)
{
  Graphics2D                g2;
  int                       frameNumber;
  Screen                    smodel;
  Iterator<Transition>      transitions;
  Iterator<Superimposition> superimpositions;

  g2 = (Graphics2D)g;
  g2.setComposite(oldComposite);
  view.setDoubleBuffered(true);

  // Get information from the model
  smodel           = (Screen)model;
  frameNumber      = smodel.getFrameNumber();
  transitions      = smodel.getTransitions();
  superimpositions = smodel.getSuperimpositions();

  // Apply the transitions
  if (transitions != null)
  {
    while (transitions.hasNext())
    {
      transitions.next().postRendering(g, frameNumber);
    }
  }
```



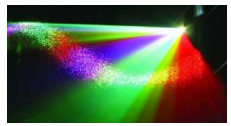
ScreenRenderer – Post-Rendering (cont.)

```
    // Apply the superimpositions
    if (superimpositions != null)
    {
      while (superimpositions.hasNext())
      {
        superimpositions.next().postRendering(g, frameNumber);
      }
    }
  }
```



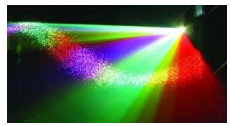
ScreenRenderer – Rendering

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



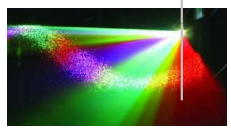
Screen – createDefaultView()

```
protected VisualizationView createDefaultView()
{
  ScreenRenderer    renderer;

  renderer = new ScreenRenderer(new PlainVisualizationRenderer());

  return new VisualizationView(this, renderer);
}
```



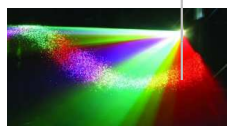
PIP - Sampled Content

```
import javax.swing.*;

import app.*;
import visual.*;
import io.ResourceFinder;
import visual.dynamic.sampled.*;
import visual.statik.*;
import visual.statik.sampled.*;

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

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

  public void init()
  {
    ResourceFinder finder = ResourceFinder.createInstance(new resources.Marker());

    Screen screen1 = new Screen(4);
    screen1.setRepeating(true);
```



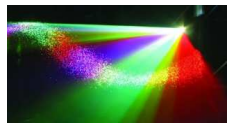
PIP - Sampled Content (cont.)

```
VisualizationView view1 = screen1.getView();
view1.setRenderer(new ScaledVisualizationRenderer(view1.getRenderer(),
    200.0, 200.0));
view1.setBounds(200,10,100,100);
view1.setSize(100,100);

String[] names = finder.loadResourceNames("solidclock.txt");
ContentFactory factory = new ContentFactory(finder);

SimpleContent[] frames1 = factory.createContents(names, 4);
for (int i=0; i<frames1.length; i++)
{
  screen1.add(frames1[i]);
}

Screen screen2 = new Screen(24);
screen2.setRepeating(true);

VisualizationView view2 = screen2.getView();
view2.setBounds(0,0,320,240);

names = finder.loadResourceNames("scribble.txt");
SimpleContent[] frames2 = factory.createContents(names, 4);

for (int i=0; i<frames2.length; i++)
{
```

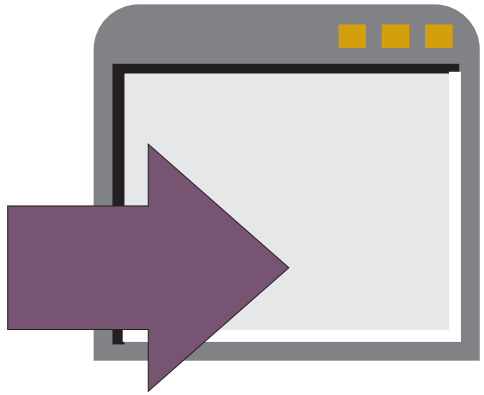


PIP - Sampled Content (cont.)

```
      screen2.add(frames2[i]);
    }

    // The content pane
    JPanel contentPane = (JPanel)getContentPane();
    contentPane.add(view1);
    contentPane.add(view2);

    screen2.start();
    screen1.start();
  }
}
```

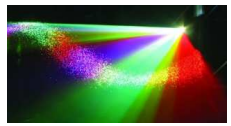


PIP – Demonstration (Sampled)



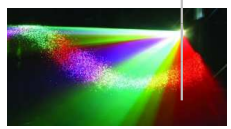
In `examples/chapter`:

```
java -cp multimedia2.jar:examples.jar DualScreenPIPDemo -Xmx256m
```



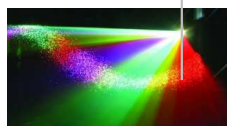
PIP - Sampled and Described Content

```
import javax.swing.*;

import app.*;
import visual.*;
import io.ResourceFinder;
import visual.dynamic.sampled.Screen;
import visual.statik.SimpleContent;
import visual.statik.sampled.ContentFactory;

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

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

  public void init()
  {
    ResourceFinder   finder;

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



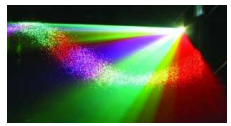
PIP - Sampled and Described Content (cont.)

```
Screen screen1 = new Screen(20);
screen1.setRepeating(true);

VisualizationView view1 = screen1.getView();
view1.setBounds(0,0,320,240);

String[] names = finder.loadResourceNames("scribble.txt");
ContentFactory factory = new ContentFactory(finder);

SimpleContent[] frames1 = factory.createContents(names, 4);
for (int i=0; i<frames1.length; i++)
{
  screen1.add(frames1[i]);
}

Screen screen2 = new Screen();
screen2.setRepeating(true);

VisualizationView view2 = screen2.getView();
view2.setRenderer(new ScaledVisualizationRenderer(view2.getRenderer(),
    200.0, 200.0));
view2.setBounds(200,10,75,75);
view2.setSize(75,75);

MovingRectangle mr = new MovingRectangle();
SimpleContent[] frames2 = mr.getFrames();
```

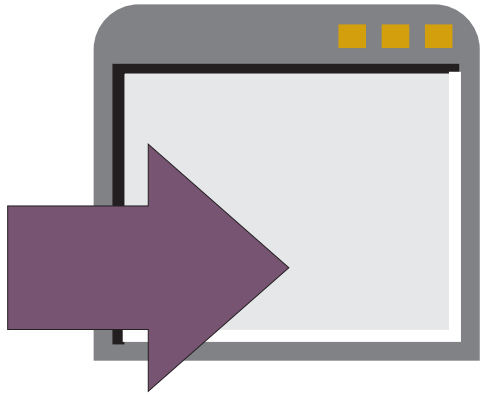


PIP - Sampled and Described Content (cont.)

```
    for (int i=0; i<frames2.length; i++)
    {
      screen2.add(frames2[i]);
    }

    // The content pane
    JPanel contentPane = (JPanel)getContentPane();
    contentPane.add(view2);
    contentPane.add(view1);

    screen1.start();
    screen2.start();
  }
}
```



PIP – Demonstration (Sampled and Described)



In `examples/chapter`:

```
java -cp multimedia2.jar:examples.jar ScreenPIPDemo -Xmx256m
```