Java game programming

2D Graphics and animation

2010

Fayolle Pierre-Alain

Plan

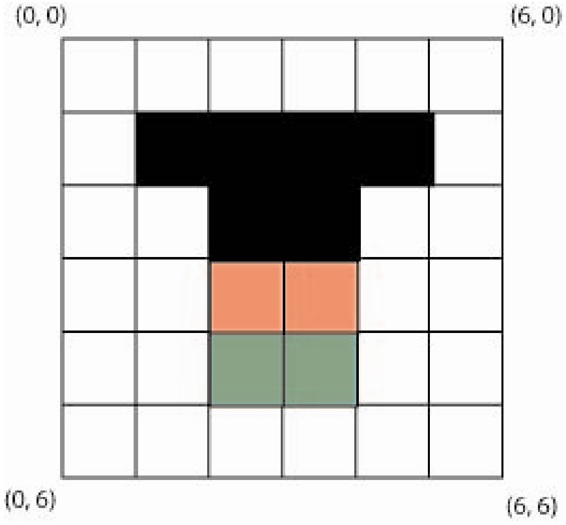
- Basic remainder on graphics hardware
- Window application / applet / full-screen application
- 2D graphics (text, shape)
- Images (type, loading and displaying)
- Animation
- Active rendering
- Double buffering, page flipping and buffer strategy
- Simple effects

Graphics hardware

- 2 parts: monitor and graphics hardware
- Video card:
  - Store the screen content in its memory
  - Has functions for modifying its memory and pushing its content to the monitor
- Monitor displays what it is told to by the graphics card

Screen layout

- The screen is a 2D array of pixels
- A pixel (derived from picture element) is a single point of light displayed by the monitor
- The screen’s origin is located at the top left corner, its width and height define the screen resolution
- The screen resolution is hardware dependent
- Any location in screen is accessed by its coordinates (x, y)



Pixel color, bit depth and refresh rate

- Pixel Color:
  - Screens use RGB (Red – Green – Blue) color model to control color
  - Intensity of Red, Green and Blue are combined to make a color for display
  - Ex: Yellow = Green + Blue (i.e. (0.0, 1.0, 1.0) in RGB coordinates)
- Bit depth:
  - Num of colors a monitor can display depends on the bit depth
  - Examples of common bit depth:
    - 8-bit → 256 (= 2^8) colors selected from a color palette
    - 15-bit (5 bits / color) → 32,768 (=2^15) colors
    - 16-bit (5 for R,B, 6 for G) → 65,536 colors (human eye is more sensitive to green)
    - 24-bit (8 bits / color) → 16,777,216 colors (human eye can see about 10 million colors)
    - 32-bit (8 bits / color, 8 bits for padding) fits into a word on 32-bit computer
- Refresh rate:
  - Num of times per second that the monitor is redrawn based on the video card memory

2D Graphics with Java

- In Java when a Component (e.g. JFrame, Applet) is displayed, AWT called the Component’s paint method
- To force AWT to call paint(), call the method repaint()
- Paint events are sent by AWT in a separate thread (you can use wait and notify if you want to be notified when the painting is finished)

```
public Class AComponent extends
SomeComponent {

 < …..>

 public void run() {
     // do something
     repaint(); // force a call to paint
 }

 public void paint(Graphics g) {
     // do painting here

  }

}
```

Graphics (and Graphics2D) object

- Graphics is an abstract base class for all graphics contexts
- It allows to draw onto components (on various devices: screen, printer)
- Graphics2D extends Graphics and provide more sophisticated control over geometry, coordinate transformations, color management
- Both Graphics and Graphics2D propose several methods for drawing text, lines, rectangles, ovals, polygons, images and so on
- (Affine) Transformations can be applied through an instance of the class AffineTransform
- Affine transformation means transformation mapping 2D coordinate to 2D while keeping collinearity (i.e. keep alignment of points) and ratios of distance (i.e. a point in the middle of 2 points is still in the middle after transformation)
  - Example: rotation, translation, dilations
- Check the Java API doc for classes Graphics, Graphics2D and AffineTransform

Full-screen exclusive mode

- Introduced in Java API 1.4
- Allows the programmer to suspend the windowing system so that drawing can be done directly to the screen
- Traditional GUI program:
  - AWT responsible for propagating paint events from the OS through the event dispatch thread
  - By calling AWT’s Component.paint method when appropriate
  - Application limited to the size and bit depth of the screen
- Full-screen mode:
  - Painting is done actively by the program itself
  - Program can control bit depth and size (display mode)
  - Advanced techniques like page flipping and stereo buffering (system with separate set of frames for each eye)

Switching to full-screen mode

- To invoke full-screen graphics and change graphics mode several objects are needed:
  - A Window object (for example JFrame)
  - A DisplayMode object to specify what graphics mode to change to
  - A GraphicsDevice object to inspect display properties and change graphic modes
- See sample code for switching to full-screen mode

```
JFrame win = new JFrame();
DisplayMode dm = new DisplayMode(800, 600,
16, 75);

GraphicsEnvironment env =
GraphicsEnvironment.getLocalGraphicsEnvironment();
GraphicsDevice gd = env.getDefaultScreenDevice();

gd.setFullScreenWindow(win);
gd.setDisplayMode(dm);
```

Switching to full-screen mode

- Some points to notice:
  - setDisplayMode() throws an IllegalArgumentException and an UnsupportedOperationException so the call should be within try ….. Catch
  - Restoring to the original display mode is done by:
    - gd.setFullScreenWindow(null); // where gd is an instance of a GraphicsDevice

Example: displaying text in full-screen

- Demo + look at source code

Some comments

- Setting the screen to full-screen mode is within a try … finally block
- Even if something happens during setting the full-screen mode or changing the display, then the original display mode will be recovered
- The text displayed is anti-aliased, i.e. the pixels are blurred on the edges to make the text looks smooth
- Antialiasing is obtained by setting appropriate rendering hint before drawing
  - Done by calling the method setRenderingHint of the class Graphics2D
  - The Graphics object passed to paint() is casted to a Graphics2D (paint () takes in face a Graphics2D object as argument since Java 2)

Display mode

- Finite list of display modes that can be used in full-screen mode
- Good practice to allow the user to select a list of possible display modes and allow the first matching the list of display modes available
- In Code:

Display mode

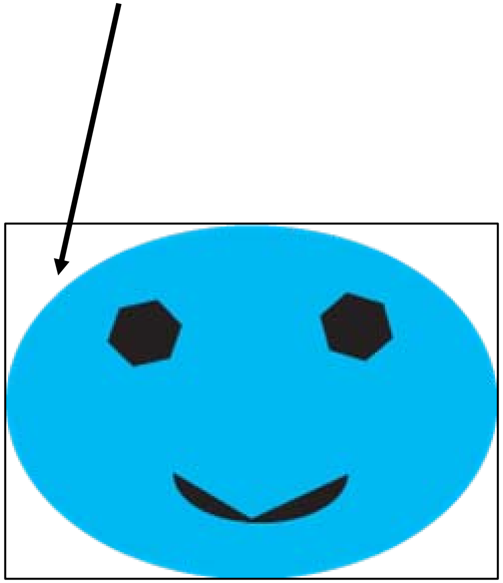
```
public DisplayMode findCompatDm(DisplayMode[] dm) {
  DisplayMode[] allowdm = graphicsdevice.getDisplayModes();
  for(int i=0; i<dm.length; i++) {
    for(int j=0; j<allowdm.length; j++) {
       if (dmMatch(dm[i], allowdm[j])) {
           return dm[i];
       }
    }
  }
}

public boolean dmMatch(DisplayMode a, DisplayMode b) {
  if (a.getWidth() != b.getWidth() || a.getHeigth() != b.getHeight()) {
    return false;
  }
  if (a.getBitDepth() != b.getBitDepth()) {
    return false;
  }
  if (a.getRefreshRate() != DisplayMode.REFRESH_RATE_UNKNOWN &&
      b.getRefreshRate() != DisplayMode.REFRESH_RATE_UNKNOWN &&
      a.getRefreshRate() != b.getRefreshRate()) {
     return false;
   }
  return true;
}
```

Images

- Opaque (Fast):
  - Every pixel is visible
- Transparent (Fast):
  - Every pixel is either visible or not
- Translucent (Slow):
  - Every visible pixel can be partially visible (obtained by blending the pixel color and the background color)

Is this pixel transparent or is its color white ?



Formats

- Images can be vector or raster
  - Vector
    - Image described geometrically
    - Example: SVG, EPS
  - Raster
    - Images is described as an array of pixels (like the screen)
- Java API is not supporting vector type of images
  - But Apache project:
    - http://xml.apache.org/batik
- Java API is however supporting various formats of raster images such as GIF, PNG, JPEG

Raster image formats

- GIF: 8-bit color. Opaque or transparent.
- PNG: Any bit depth. Opaque, transparent, or translucent.
- JPEG: 24-bit depth. Compressed format. Opaque only.
- Possible to export from vector to raster (rasterization)
- Software to create: Gimp, Inkscape

Loading images with Java

- Method 1: get an Image using Toolkit’s getImage()
  - Note: if you are developing an applet and not an application, then getImage() is a method of the class Applet
- The default toolkit can be accessed by the static method of Toolkit: getDefaultToolkit()
- getImage() starts another thread to load the image
  - If you display the image before it is finished loading, then only part of the image will appear

```
Toolkit tk = Toolkit.getDefaultToolkit();
Image im = tk.getImage(filename);
```

Loading images 2

- Possible solution is to use MediaTracker as follow:

```
Image[] images;
void loadImages() {
  images = new Image[3];
  MediaTracker tracker = new MediaTracker(this);

  for(int i=0; i<3; i++) {
    images[i] = getImage(getCodeBase(),"image"+i+".gif");
    tracker.addImage(images[i], 0);
  }
  try {
    // Start downloading images. Wait until they are loaded.
    tracker.waitForAll();
  } catch (InterruptedException e) {}
 }
```

Loading images 3

- But better way is to use the ImageIcon class in swing package
  - Note: it loads an image using the Toolkit and waits for it to finish loading before return

```
ImageIcon icon = new ImageIcon(filename);
Image im = icon.getImage();
```

```
Image im = new ImageIcon(filename).getImage();
```

Example: Loading / displaying an image

- Demo and look at source code

Hardware accelerated images

- Hardware accelerated images are stored in video memory rather than system memory
- They can be copied faster to the screen
- Java tries to create hardware accelerated images for the images loaded by Toolkit’s getImage()
- It is possible to force an image to be hardware accelerated by using the VolatileImage interface
- Points to keep in mind:
  - Only opaque and transparent images can be accelerated; not translucent
  - Hardware accelerated images are not supported on all systems
  - An image whose contents constantly changes will not be hardware accelerated

Animation

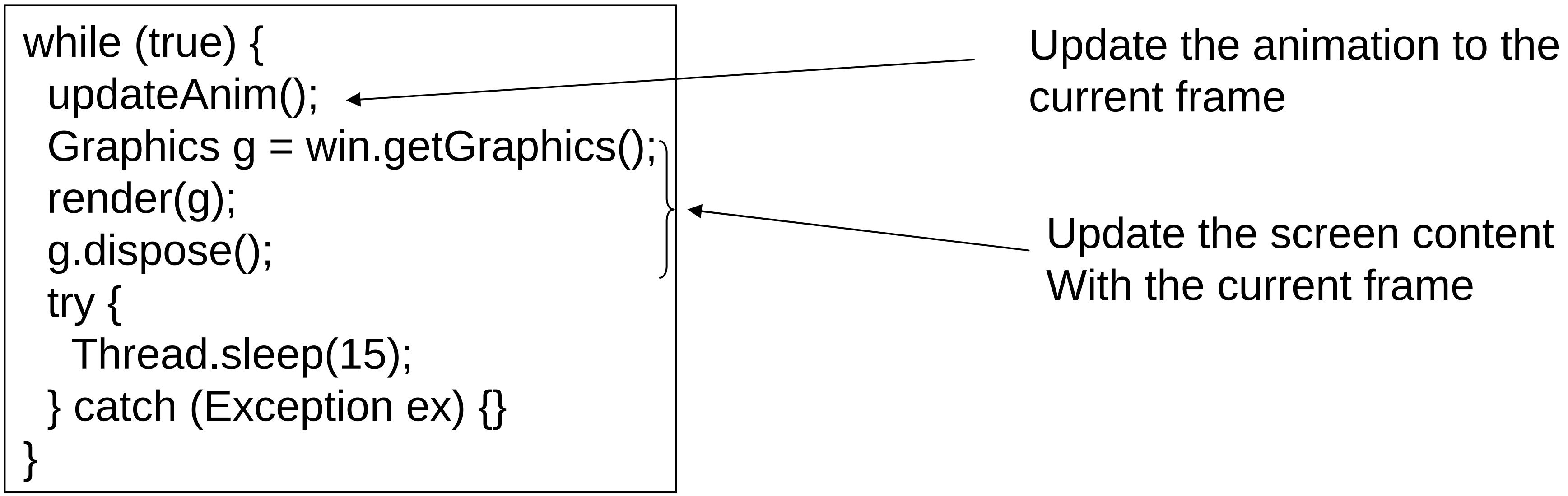
- An animation is a sequence of images
- Each image is displayed for a brief amount of time
- Each image in an animation if sometimes referred as “frame”
- Animation loop: loop that updates the animation and displays the current frame on the screen
  - Update the animation
  - Draw the current frame on the screen
  - Sleep for some time
  - Go back to the first step

Active rendering

- In GUI applications, the question of when to paint is handled by the OS
- The OS sends a paint event to AWT, then AWT figures out what needs to be painted, creates a Graphics object and call paint() with that object
- This is called *passive rendering*
- This incurs lots of overhead and we have no control over when paint() is really called
- For applications requiring performance it is better to use *active rendering*
- Drawing is done directly to the screen in the main thread

```
Graphics g = win.getGraphics();
render(g);
g.dispose();
```

Animation loop sample code



```
while (true) {
  updateAnim();
  Graphics g = win.getGraphics();
  render(g);
  g.dispose();
  try {
    Thread.sleep(15);
  } catch (Exception ex) {}
}
```

Update the animation to the
current frame

Update the screen content
With the current frame

Some remarks:

- Do not put code in the paint() routine, instead put in your own method, like render.

In window application, paint() can call render(), and full-screen, render() will go in the rendering loop.

- Turn off all paint events dispatched by the OS, by using the method setIgnoreRepaint(boolean) (this method is in the class Component).

Example: animation of the previous image

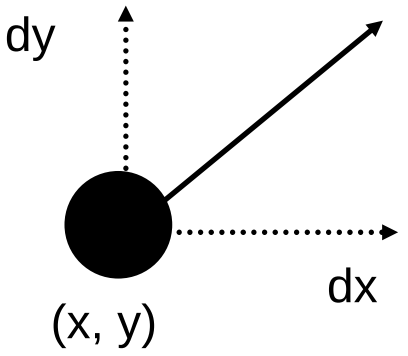
- Demo and looking at code
- AnimTest:
  - Contains the animation loop:
    - Update the animation
    - Render the current frame
- Anim:
  - Contains an animation as a sequence of images (frames)
  - Allows to retrieve the current frame in the animation
- FullScreen:
  - Set the window to full-screen mode
  - Give handles to the full-screen window

Sprites

- 2D sprites are small bitmap graphics moving independently within the screen
- A sprite is make of two component:
  - The animation (as seen previously) that animates the object locally
  - Something that makes the object look within the screen
- In a 2D game, sprite corresponding to the hero would be controlled by the keys while bad guys would be controlled by the computer program
- In the following we will make the previous image moves within the screen and bounces on the screen’s border

Sprite

- A sprite is defined by its current location, as well as its movement in space
- To keep the sprite movement constant (independent of the frame rate) we will use its speed instead of movement
- Movement is obtained by:
  - dx = vx * dt
  - dy = vy * dt
  - Where dt is the elapsed time



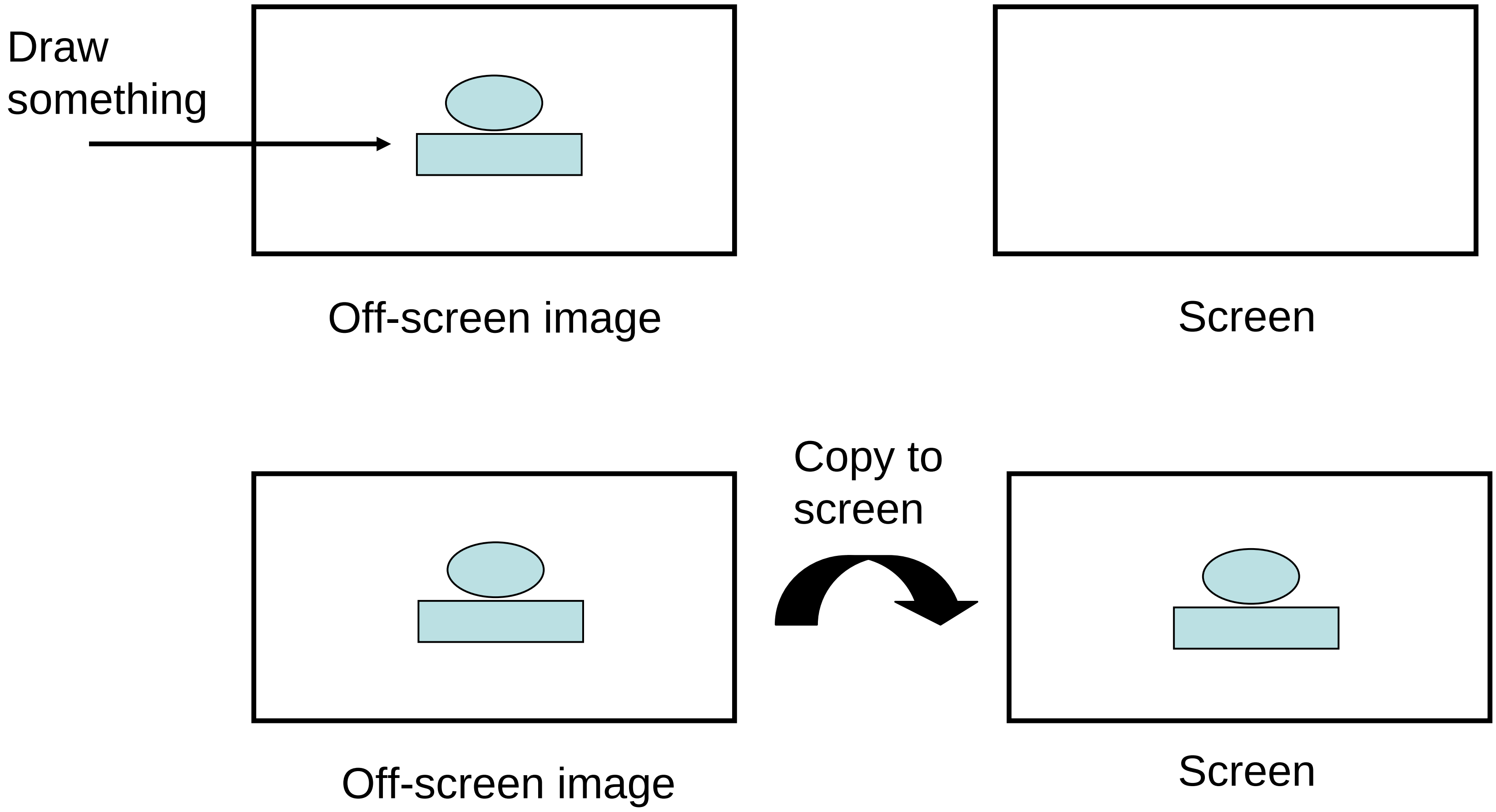
Example: a face bouncing on the screen

- Demo and looking at the code
- The demo has some problem: the screen is flickering

Double buffering

- In the previous demonstration, you could notice that the animation flickered
- The reason is that the image is drawn directly on the screen, then drawn over by the background, then the updated image is drawn again
- To avoid this, there is a technique called: double-buffering
- Double-buffering works as follow:
  - Create an off-screen image (back buffer)
  - Draw to that image using the image’s Graphics object
  - Call drawImage with the target window’s Graphics object and the off-screen image

Illustration of double-buffering



Draw something

Off-screen image

Screen

Copy to screen

Off-screen image

Screen

Page flipping

- Double-buffering requires to copy the content of the video memory to the screen
- There is a faster technique where only a pointer to a zone in video memory is copied. This is called *page flipping*
- Graphics cards have the notion of the video pointer (an address in video memory)
- The pointer indicates where the graphics card should look for the contents to be displayed during the next refresh style
- This pointer can be manipulated for some OS and graphic cards

Illustration of page flipping

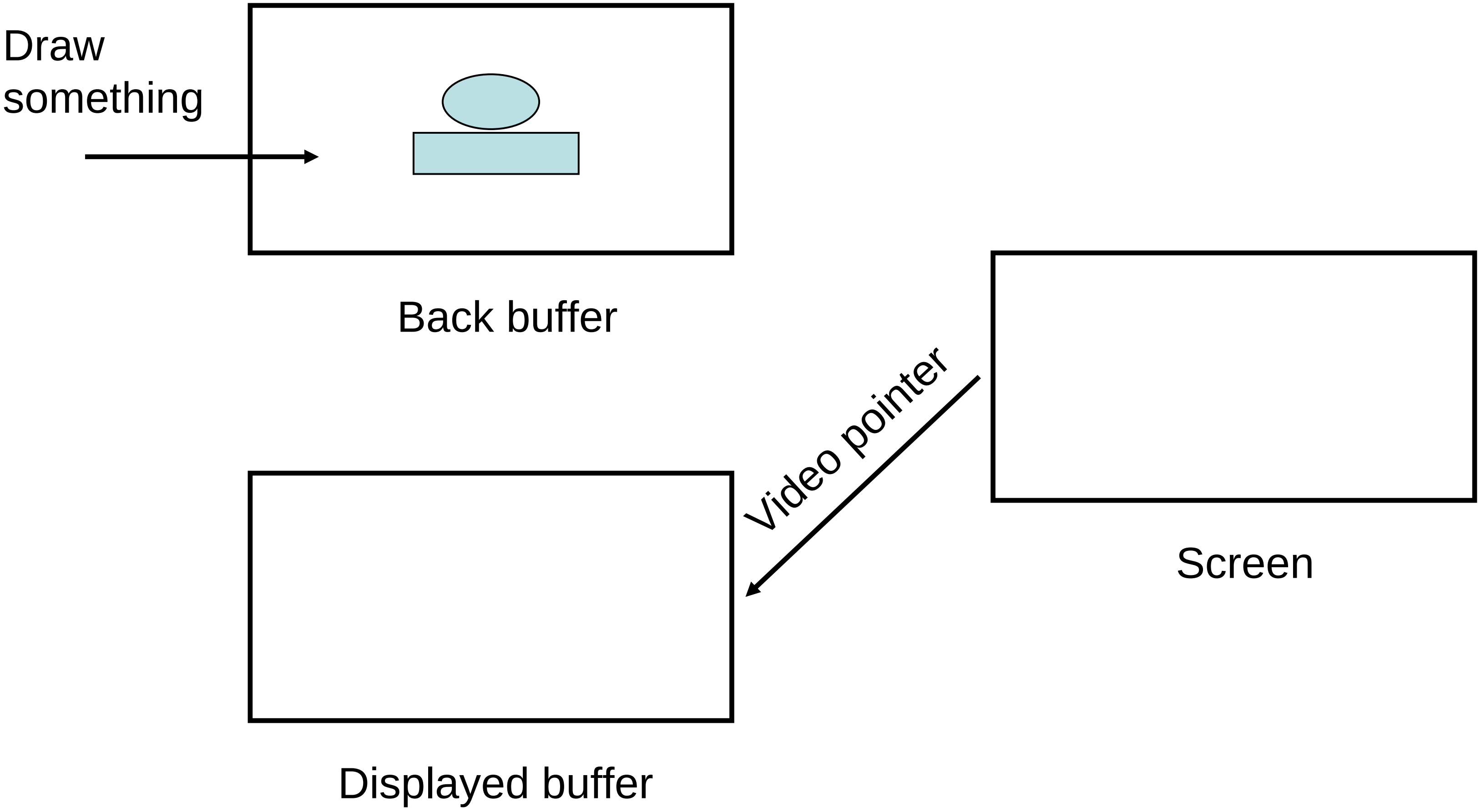
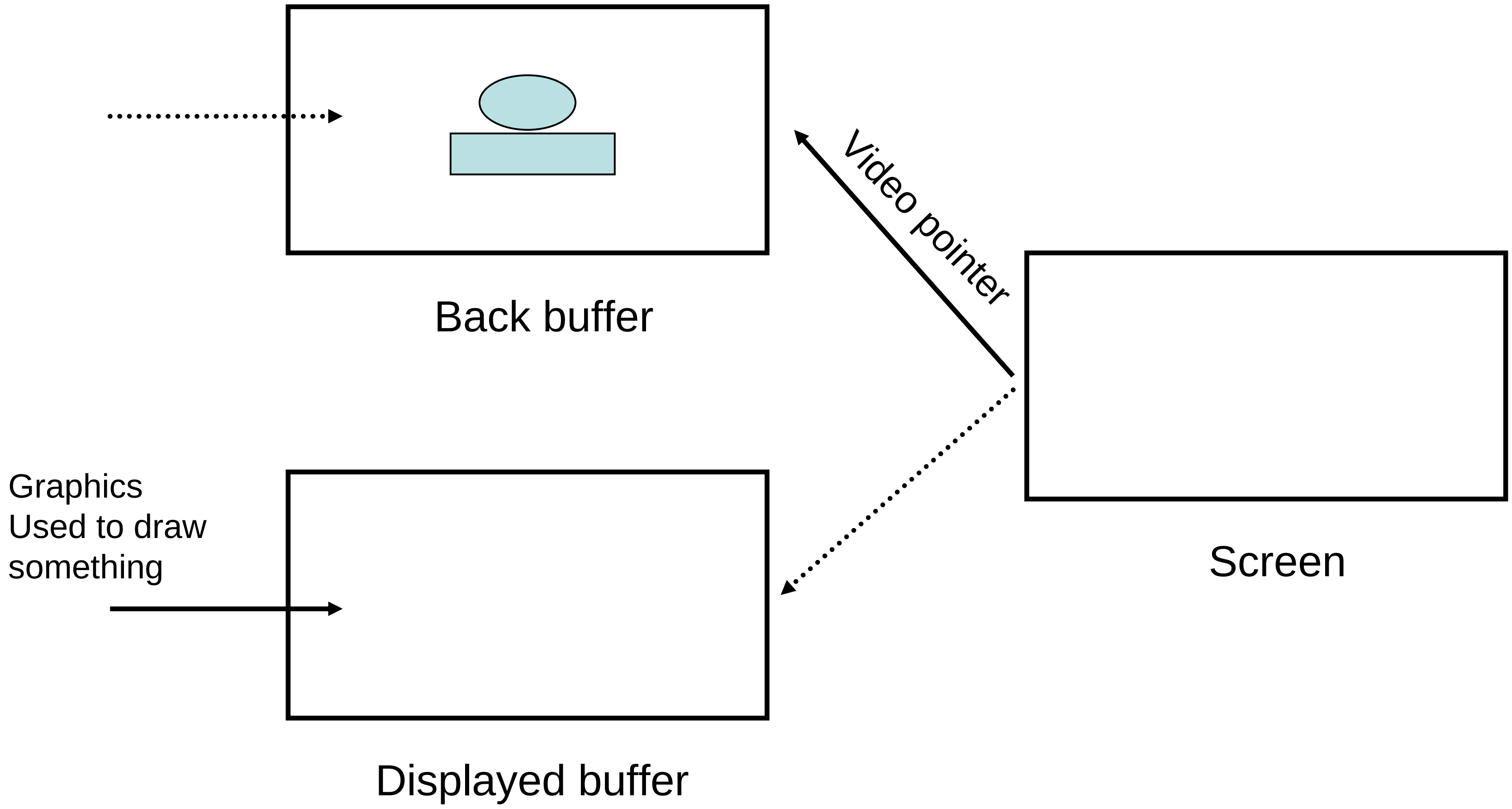


Illustration of page flipping



Back buffer

Video pointer

Screen

Graphics
Used to draw
something

Displayed buffer

BufferStrategy

- In Java (2 and above) you do not have to worry about these low level details to exploit double-buffering or page flipping
- The class java.awt.image.BufferStrategy has been added for the convenience of dealing with this
- A BufferStrategy has two important methods:
  - getDrawGraphics(): return a Graphics object for the drawing surface
  - show(): makes the next buffer visible by copying the memory (double-buffering) or changing the display pointer (page flipping)

Sample code using a buffer strategy

```
BufferStrategy strategy;
while (!done) {
  Graphics g = strategy.getDrawGraphics();
  render(g);
  g.dispose();
  strategy.show();
}
```

Example: a sprite bouncing on the borders

- Code and demo
- Same as the previous example but using BufferStrategy instead
- FullScreen has been modified to incorporate a BufferStrategy
- The test class has been updated to call getDrawGraphics() and show() from the BufferStrategy

Adding simple effects

- It is possible to use the class AffineTransform to add simple effect (rotation, scaling, etc) to the objects
- To do that you can create an empty transformation and compose it with rotation, translation:

  -
    ```
    AffineTransform a = new AffineTransform();
    a.setTranslation(translationx, translationy);
    a.rotate(Math.PI / 20.0);
    ```
- Transforming images does not use hardware acceleration so be careful when using such effects

Summary

- Graphics hardware
- Graphics and Graphics2D classes
- Full-screen exclusive mode
- Loading and displaying images
- Animation loop
- Active rendering
- Sprites
- Double buffering, page flipping and buffer strategy