Structuring Images

CS1316: Representing Structure and Behavior

Story

- Structuring images into scenes
  - Version 1: Representing *linearity* through elements order.
    - Animation through rendering and data structure tweaking
  - Version 2: Representing *layering* through order.
  - Version 3: Allowing both in a single list
    - Introducing subclasses and superclasses
      - Including *abstract* classes
    - Passing a turtle along for processing.
  - Version 4: Creating trees of images
    - Making the branches *do* something

Building a Scene

- Computer graphics professionals work at two levels:
  - They define individual characters and effects on characters in terms of pixels.
  - But then most of their work is in terms of the *scene*: Combinations of images (characters, effects on characters).
- To describe scenes, they often use *linked lists* and *trees* in order to assemble the pieces.

Use an array?

```
> Picture [] myarray = new Picture[5];
> myarray[0]=new Picture(FileChooser.getMediaPath("katie.jpg"));
> myarray[1]=new Picture(FileChooser.getMediaPath("barbara.jpg"));
> myarray[2]=new Picture(FileChooser.getMediaPath("flower1.jpg"));
> myarray[3]=new Picture(FileChooser.getMediaPath("flower2.jpg"));
> myarray[4]=new Picture(FileChooser.getMediaPath("butterfly.jpg"));
> Picture background = new Picture(400,400)
> for (int i = 0; i < 5; i++)
  {myarray[i].scale(0.5).compose(background,i*10,i*10);}
> background.show();
```

Yeah, we could. But:

- Inflexible
- Hard to insert, delete.

Using a linked list

- Okay, so we'll use a linked list.
- But what should the ordering represent?
  - Version 1: Linearity
    - The order that things get drawn left-to-right.
  - Version 2: Layering
    - The order that things get drawn bottom-to-top

Version 1: PositionedSceneElement

```
> PositionedSceneElement tree1 = new PositionedSceneElement(new
    Picture(FileChooser.getMediaPath("tree-blue.jpg")));
> PositionedSceneElement tree2 = new PositionedSceneElement(new
    Picture(FileChooser.getMediaPath("tree-blue.jpg")));
> PositionedSceneElement tree3 = new PositionedSceneElement(new
    Picture(FileChooser.getMediaPath("tree-blue.jpg")));
> PositionedSceneElement doggy = new PositionedSceneElement(new
    Picture(FileChooser.getMediaPath("dog-blue.jpg")));
> PositionedSceneElement house = new PositionedSceneElement(new
    Picture(FileChooser.getMediaPath("house-blue.jpg")));
> Picture bg = new Picture(FileChooser.getMediaPath("jungle.jpg"));
> tree1.setNext(tree2); tree2.setNext(tree3); tree3.setNext(doggy);
    doggy.setNext(house);
> tree1.drawFromMeOn(bg);
> bg.show();
```

In this example, using chromakey to compose..just for the fun of it.

What this looks like:

Slightly different ordering: Put the doggy between tree2 and tree3

```
> tree3.setNext(house); tree2.setNext(doggy); doggy.setNext(tree3);
> bg = new Picture(FileChooser.getMediaPath("jungle.jpg"));
> tree1.drawFromMeOn(bg);
> bg.show();
```

Yes, we can put multiple statements in one line.

Slightly different picture

PositionedSceneElement

```
public class PositionedSceneElement {

 /**
  * the picture that this element holds
  **/
 private Picture myPic;

 /**
  * the next element in the list
  **/
 private PositionedSceneElement next;
```

Pretty darn similar to our music linked lists!

Constructor

```
/**
  * Make a new element with a picture as input, and
  * next as null.
  * @param heldPic Picture for element to hold
  **/
 public PositionedSceneElement(Picture heldPic){
   myPic = heldPic;
   next = null;
 }
```

Linked list methods

```
/**
  * Methods to set and get next elements
  * @param nextOne next element in list
  **/
 public void setNext(PositionedSceneElement nextOne){
   this.next = nextOne;
 }

 public PositionedSceneElement getNext(){
   return this.next;
 }
```

Again, darn similar!

Traverse the list

Traversing the list in order to draw the scene is called *rendering* the scene: Realizing the picture described by the data structure.

```
/**
 * Method to draw from this node on in the list, using
   bluescreen.
 * Each new element has it's lower-left corner at the
   lower-right
 * of the previous node. Starts drawing from left-
   bottom
 * @param bg Picture to draw drawing on
 **/
public void drawFromMeOn(Picture bg) {
  PositionedSceneElement current;
  int currentX=0, currentY = bg.getHeight()-1;

  current = this;
  while (current != null)
  {
    current.drawMeOn(bg,currentX, currentY);
    currentX = currentX +
    current.getPicture().getWidth();
    current = current.getNext();
  }
}
```

Core of the Traversal

```
current = this;
while (current != null)
{
  //Treat the next two lines as "blah blah blah"
  current.drawMeOn(bg,currentX, currentY);
  currentX = currentX +
            current.getPicture().getWidth();

  current = current.getNext();
}
```

Drawing the individual element

```
/**
  * Method to draw from this picture, using bluescreen.
  * @param bg Picture to draw drawing on
  * @param left x position to draw from
  * @param bottom y position to draw from
  **/

 private void drawMeOn(Picture bg, int left, int bottom) {
   // Bluescreen takes an upper left corner
   this.getPicture().bluescreen(bg,left,
                          bottom-this.getPicture().getHeight());
 }
```

Generalizing

- Reconsider these lines:

```
> tree3.setNext(house);
tree2.setNext(doggy);
doggy.setNext(tree3);
```

- This is actually a general case of:
  - Removing the doggy from the list
  - Inserting it after tree2

Removing the doggy

```
> tree1.setNext(tree2);
    tree2.setNext(tree3);
    tree3.setNext(doggy);
    doggy.setNext(house);
> tree1.remove(doggy);
> tree1.drawFromMeOn(bg);
```

Putting the mutt back

```
> bg = new
    Picture(FileChooser.getMe
    diaPath("jungle.jpg"));
> tree1.insertAfter(doggy);
> tree1.drawFromMeOn(bg);
```

Removing an element from the list

Note: How would you remove the *first* element from the list?

```
/** Method to remove node from list, fixing links
     appropriately.
 * @param node element to remove from list.
 **/
public void remove(PositionedSceneElement node){
  if (node==this)
  {
    System.out.println("I can't remove the first node from
    the list.");
    return;
  };

  PositionedSceneElement current = this;
  // While there are more nodes to consider
  while (current.getNext() != null)
  {
    if (current.getNext() == node){
      // Simply make node's next be this next
      current.setNext(node.getNext());
      // Make this node point to nothing
      node.setNext(null);
      return;
    }
    current = current.getNext();
  }
}
```

Error checking and printing

```
/** Method to remove node from list, fixing links
   appropriately.
 * @param node element to remove from list.
 **/
public void remove(PositionedSceneElement node){
  if (node==this)
  {
    System.out.println("I can't remove the first node
  from the list.");
    return;
  };
```

The Removal Loop

```
PositionedSceneElement current = this;
// While there are more nodes to consider
while (current.getNext() != null)
{ // Is this it?
  if (current.getNext() == node){
    // Simply make node's next be this next
    current.setNext(node.getNext());
    // Make this node point to nothing
    node.setNext(null);
    return;
  }
  current = current.getNext(); // If not, keep searching
}
```

We're checking getNext() because we need to stop the step *before*.

insertAfter

Think about what's involved in creating insertBefore()…

```
/**
 * Insert the input node after this
   node.
 * @param node element to insert
   after this.
 **/
public void
  insertAfter(PositionedSceneElement
  node){
  // Save what "this" currently points at
  PositionedSceneElement oldNext =
  this.getNext();
  this.setNext(node);
  node.setNext(oldNext);
}
```

Animation = (Changing a structure + rendering) * n

- We can use what we just did to create *animation.*
- Rather than think about animation as "a series of frames,"
- Think about it as:
  - Repeatedly:
    - Change a data structure
    - *Render* (draw while traversing) the data structure to create a frame

AnimatedPositionedScene

```
public class AnimatedPositionedScene {

  /**
   * A FrameSequence for storing the frames
   **/
  FrameSequence frames;

  /**
   * We'll need to keep track
   * of the elements of the scene
   **/
  PositionedSceneElement tree1, tree2, tree3, house, doggy, doggyflip;
```

Setting up the animation

```
public void setUp(){
  frames = new FrameSequence("D:/Temp/");

  Picture p = null; // Use this to fill elements

  p = new Picture(FileChooser.getMediaPath("tree-
    blue.jpg"));
  tree1 = new PositionedSceneElement(p);

  p = new Picture(FileChooser.getMediaPath("tree-
    blue.jpg"));
  tree2 = new PositionedSceneElement(p);

  p = new Picture(FileChooser.getMediaPath("tree-
    blue.jpg"));
  tree3 = new PositionedSceneElement(p);

  p = new
    Picture(FileChooser.getMediaPath("house-
    blue.jpg"));
  house = new PositionedSceneElement(p);

  p = new Picture(FileChooser.getMediaPath("dog-
    blue.jpg"));
  doggy = new PositionedSceneElement(p);
  doggyflip = new PositionedSceneElement(p.flip());
}
```

Render the first frame

```
public void make(){
  frames.show();

  // First frame
  Picture bg = new
   Picture(FileChooser.getMediaPath("jungle.jpg"));
  tree1.setNext(doggy); doggy.setNext(tree2);
   tree2.setNext(tree3);
  tree3.setNext(house);
  tree1.drawFromMeOn(bg);
  frames.addFrame(bg);
```

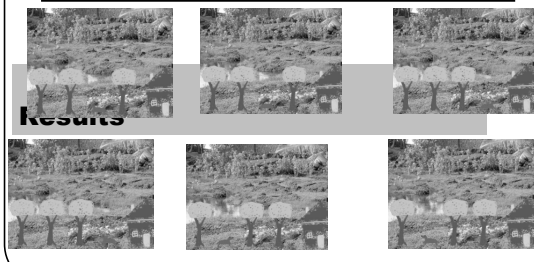
Render the doggy moving right

```
// Dog moving right
  bg = new Picture(FileChooser.getMediaPath("jungle.jpg"));
  tree1.remove(doggy);
  tree2.insertAfter(doggy);
  tree1.drawFromMeOn(bg);
  frames.addFrame(bg);

  bg = new Picture(FileChooser.getMediaPath("jungle.jpg"));
  tree1.remove(doggy);
  tree3.insertAfter(doggy);
  tree1.drawFromMeOn(bg);
  frames.addFrame(bg);

  bg = new Picture(FileChooser.getMediaPath("jungle.jpg"));
  tree1.remove(doggy);
  house.insertAfter(doggy);
  tree1.drawFromMeOn(bg);
  frames.addFrame(bg);
```

Moving left

```
//Dog moving left
  bg = new Picture(FileChooser.getMediaPath("jungle.jpg"));
  tree1.remove(doggy);
  house.insertAfter(doggyflip);
  tree1.drawFromMeOn(bg);
  frames.addFrame(bg);

  bg = new Picture(FileChooser.getMediaPath("jungle.jpg"));
  tree1.remove(doggyflip);
  tree3.insertAfter(doggyflip);
  tree1.drawFromMeOn(bg);
  frames.addFrame(bg);

  bg = new Picture(FileChooser.getMediaPath("jungle.jpg"));
  tree1.remove(doggyflip);
  tree2.insertAfter(doggyflip);
  tree1.drawFromMeOn(bg);
  frames.addFrame(bg);

  bg = new Picture(FileChooser.getMediaPath("jungle.jpg"));
  tree1.remove(doggyflip);
  tree1.insertAfter(doggyflip);
  tree1.drawFromMeOn(bg);
  frames.addFrame(bg);

}
```

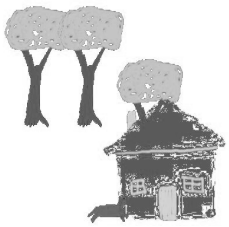
Results



Version 2: Layering

```
> Picture bg = new Picture(400,400);
> LayeredSceneElement tree1 = new LayeredSceneElement(
new Picture(FileChooser.getMediaPath("tree-blue.jpg")),10,10);
> LayeredSceneElement tree2 = new LayeredSceneElement(
new Picture(FileChooser.getMediaPath("tree-blue.jpg")),100,10);
> LayeredSceneElement tree3 = new LayeredSceneElement(
new Picture(FileChooser.getMediaPath("tree-blue.jpg")),200,100);
> LayeredSceneElement house = new LayeredSceneElement(
new Picture(FileChooser.getMediaPath("house-blue.jpg")),175,175);
> LayeredSceneElement doggy = new LayeredSceneElement(
new Picture(FileChooser.getMediaPath("dog-blue.jpg")),150,325);
> tree1.setNext(tree2); tree2.setNext(tree3); tree3.setNext(doggy);
    doggy.setNext(house);
> tree1.drawFromMeOn(bg);
> bg.show();
```

First version of Layered Scene

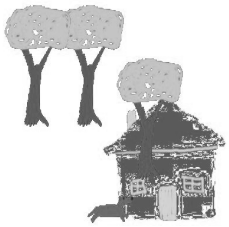


Reordering the layering

```
> house.setNext(doggy);
  doggy.setNext(tree3);
  tree3.setNext(tree2);
  tree2.setNext(tree1);
> tree1.setNext(null);
> bg = new Picture(400,400);
> house.drawFromMeOn(bg);
> bg.show();
```

Basically, we're reversing the list

Reordered (relayered) scene



Think about what's involved in creating a *method* to *reverse()* a list…

What's the difference?

- If we were in PowerPoint or Visio, you'd say that we changed the *layering*.
  - "Bring to front"
  - "Send to back"
  - "Bring forward"
  - "Send backward"

These commands are actually changing the *ordering* of the layers in the *list* of things to be redrawn.

- Change the ordering in the list.
- *Render* the scene
- Now it's a different layering!

LayeredSceneElement

```
public class LayeredSceneElement {

  /**
   * the picture that this element holds
   **/
  private Picture myPic;

  /**
   * the next element in the list
   **/
  private LayeredSceneElement next;

  /**
   * The coordinates for this element
   **/
  private int x, y;
```

Constructor

```
 /**
   * Make a new element with a picture as input, and
   * next as null, to be drawn at given x,y
   * @param heldPic Picture for element to hold
   * @param xpos x position desired for element
   * @param ypos y position desired for element
   **/
 public LayeredSceneElement(Picture heldPic, int xpos, int ypos){
   myPic = heldPic;
   next = null;
   x = xpos;
   y = ypos;
 }
```

Linked List methods (We can sort of assume these now, right?)

```
/**
 * Methods to set and get next elements
 * @param nextOne next element in list
 **/
public void setNext(LayeredSceneElement nextOne){
  this.next = nextOne;
}

public LayeredSceneElement getNext(){
  return this.next;
}
```

Traversing

```
/**
 * Method to draw from this node on in the list, using
    bluescreen.
 * Each new element has it's lower-left corner at the
    lower-right
 * of the previous node. Starts drawing from left-bottom
 * @param bg Picture to draw drawing on
 **/
public void drawFromMeOn(Picture bg) {
  LayeredSceneElement current;

  current = this;
  while (current != null)
  {
    current.drawMeOn(bg);
    current = current.getNext();
  }
}

/**
 * Method to draw from this picture, using bluescreen.
 * @param bg Picture to draw drawing on
 **/

private void drawMeOn(Picture bg) {
  this.getPicture().bluescreen(bg,x,y);
}
```

Linked list traversals are all the same

```
current = this;
 while (current != null)
 {
   current.drawMeOn(bg);
   current = current.getNext();
 }
```

Doing a reverse()

```
/**
 * Reverse the list starting at this,
 * and return the last element of the list.
 * The last element becomes the FIRST element
 * of the list, and THIS points to null.
 **/
  public LayeredSceneElement reverse() {
  LayeredSceneElement reversed, temp;

  // Handle the first node outside the loop
  reversed = this.last();
  this.remove(reversed);

  while (this.getNext() != null)
  {
    temp = this.last();
    this.remove(temp);
    reversed.add(temp);
  };

  // Now put the head of the old list on the end of
  // the reversed list.
  reversed.add(this);

  // At this point, reversed
  // is the head of the list
  return reversed;
}
```

Getting the last()

```
/**
 * Return the last element in the list
 **/
  public LayeredSceneElement last() {
  LayeredSceneElement current;

  current = this;
  while (current.getNext() != null)
  {
    current = current.getNext();
  };
  return current;
}
```

Basically, it's a complete traversal

Adding to the end

```
/**
 * Add the input node after the last node in this list.
 * @param node element to insert after this.
 **/
public void add(LayeredSceneElement node){
  this.last().insertAfter(node);
}
```

Pretty easy, huh? Find the last(), and insertAfter()

Does it work?

```
> Picture bg = new Picture(400,400);
> LayeredSceneElement tree1 = new LayeredSceneElement(
new Picture(FileChooser.getMediaPath("tree-blue.jpg")),10,10);
> LayeredSceneElement tree2 = new LayeredSceneElement(
new Picture(FileChooser.getMediaPath("tree-blue.jpg")),10,10);
> LayeredSceneElement house = new LayeredSceneElement(
new Picture(FileChooser.getMediaPath("house-blue.jpg")),10,10);
> tree1.setNext(tree2); tree2.setNext(house);
> LayeredSceneElement rev = tree1.reverse();
> rev.drawFromMeOn(bg);
> bg.show();
> // Hard to tell from the layering—let's check another way
> rev == house
true
> rev == tree1
false
```

Let's add this up then…

```
while (this.getNext() != null)
  {
    temp = this.last();
    this.remove(temp);
    reversed.add(temp);
  };
```

So how *expensive* is this loop?

- We go through this loop *once* for each element in the list.
- For each node, we find the *last()* (which is *another* traversal)
- And when we *add()*, we know that we do *another last()* which is *another* traversal

Total cost: For each of the *n* nodes, reversing takes two traversals (*2n*) => O(n*2n) => O(n^2)

There is a better way…

Version 3: A List with Both

- Problem 1: Why should we have *only* layered scene elements *or* positioned scene elements?
- Can we have both?
  - SURE! If *each* element knows how to draw itself!
  - But they took different parameters!
    - Layered got their (x,y) passed in.
  - It works if we always pass in a *turtle* that's set to the right place to draw if it's positioned (and let the layered ones do whatever they want!)
- Problem 2: Why is there so much duplicated code?
  - Why do only layered elements know last() and add()?

Using Superclasses

- What we really want is to define a class *SceneElement*
  - That knows most of being a picture element.
  - It would be an *abstract* class because we don't actually mean to ever create instances of *THAT* class.
- Then create *subclasses*: *SceneElementPositioned* and *SceneElementLayered*
  - We'd actually use these.

Class Structure

Abstract Class *SceneElement*

It **knows** its Picture myPic and its next (SceneElement).

It **knows how** to get/set next, to reverse() and insertAfter(), and to drawFromMeOn().

It **defines** drawWith(turtle), but leaves it for its subclasses do complete.

An abstract class defines structure and behavior that subclasses will inherit.

Class Structure

An abstract class defines structure and behavior that subclasses will inherit.

Abstract Class *SceneElement*

It **knows** its Picture myPic and its next.

It **knows how** to get/set next, to reverse() and insertAfter(), and to drawFromMeOn() and drawWith(turtle)

The subclasses ***inherit*** data and methods from superclass.

We say that the subclasses ***extend*** the superclass.

Class *SceneElementLayered*

It **knows** its position (x,y).

It **knows how** to drawWith(turtle) by moving to (x,y) then dropping.

Class *SceneElementPositioned*

It **knows how** to drawWith(turtle)

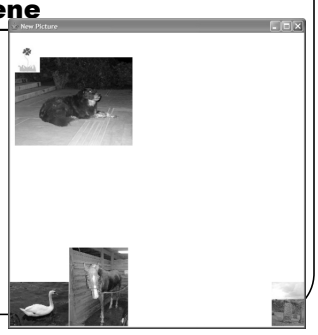
Using the new structure

```
public class MultiElementScene {

 public static void main(String[] args){

  // We'll use this for filling the nodes
  Picture p = null;

  p = new Picture(FileChooser.getMediaPath("swan.jpg"));
  SceneElement node1 = new SceneElementPositioned(p.scale(0.25));
  p = new Picture(FileChooser.getMediaPath("horse.jpg"));
  SceneElement node2 = new SceneElementPositioned(p.scale(0.25));
  p = new Picture(FileChooser.getMediaPath("dog.jpg"));
  SceneElement node3 = new
   SceneElementLayered(p.scale(0.5),10,50);
  p = new Picture(FileChooser.getMediaPath("flower1.jpg"));
  SceneElement node4 = new
   SceneElementLayered(p.scale(0.5),10,30);
  p = new Picture(FileChooser.getMediaPath("graves.jpg"));
  SceneElement node5 = new SceneElementPositioned(p.scale(0.25));
```

Rendering the scene

```
  node1.setNext(node2); node2.setNext(node3);
  node3.setNext(node4); node4.setNext(node5);

  // Now, let's see it!
  Picture bg = new Picture(600,600);
  node1.drawFromMeOn(bg);
  bg.show();
 }
}
```

Rendered scene



SceneElement

```
/**
 * An element that knows how to draw itself in a scene with a turtle
 **/
public abstract class SceneElement{

 /**
  * the picture that this element holds
  **/
 protected Picture myPic;

 /**
  * the next element in the list -- any SceneElement
  **/
 protected SceneElement next;
```

Linked List methods in SceneList

```
/**
  * Methods to set and get next elements
  * @param nextOne next element in list
  **/
 public void setNext(SceneElement nextOne){
   this.next = nextOne;
 }

 public SceneElement getNext(){
   return this.next;
 }
```

By declaring everything to be *SceneElement*, it can be any *kind* (subclass) of *SceneElement*.

drawFromMeOn()

```
/**
  * Method to draw from this node on in the list.
  * For positioned elements, compute locations.
  * Each new element has it's lower-left corner at the lower-
    right
  * of the previous node. Starts drawing from left-bottom
  * @param bg Picture to draw drawing on
  **/
public void drawFromMeOn(Picture bg) {
  SceneElement current;

  // Start the X at the left
  // Start the Y along the bottom
  int currentX=0, currentY = bg.getHeight()-1;

  Turtle pen = new Turtle(bg);
  pen.setPenDown(false); // Pick the pen up

  current = this;
  while (current != null)
  { // Position the turtle for the next positioned element
    pen.moveTo(currentX,currentY-
      current.getPicture().getHeight());
    pen.setHeading(0);

    current.drawWith(pen);
    currentX = currentX + current.getPicture().getWidth();

    current = current.getNext();
  }
}
```

But SceneElements can't drawWith()

```
/*
 * Use the given turtle to draw oneself
 * @param t the Turtle to draw with
 **/
public abstract void drawWith(Turtle t);
// No body in the superclass
```

SceneElementLayered

```
public class SceneElementLayered extends SceneElement {

 /**
  * The coordinates for this element
  **/
 private int x, y;

 /**
  * Make a new element with a picture as input, and
  * next as null, to be drawn at given x,y
  * @param heldPic Picture for element to hold
  * @param xpos x position desired for element
  * @param ypos y position desired for element
  **/
 public SceneElementLayered(Picture heldPic, int xpos, int ypos){
   myPic = heldPic;
   next = null;
   x = xpos;
   y = ypos;
 }
```

SceneElementLayered drawWith()

```
/**
 * Method to draw from this picture.
 * @param pen Turtle to draw with
 **/

public void drawWith(Turtle pen) {
  // We just ignore the pen's position
  pen.moveTo(x,y);
  pen.drop(this.getPicture());
}
```

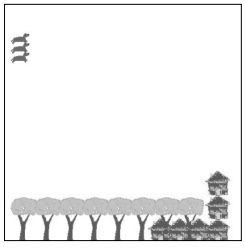
SceneElementPositioned

```
public class SceneElementPositioned
      extends SceneElement {

 /**
  * Make a new element with a picture as
    input, and
  * next as null.
  * @param heldPic Picture for element to
    hold
  **/
 public SceneElementPositioned(Picture
    heldPic){
   myPic = heldPic;
   next = null;
 }

 /**
  * Method to draw from this picture.
  * @param pen Turtle to use for drawing
  **/
 public void drawWith(Turtle pen) {
   pen.drop(this.getPicture());
 }
}
```

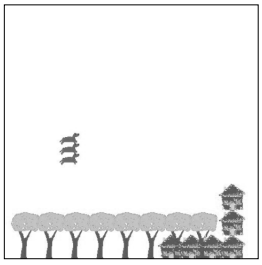
Version 4: Trees for defining scenes

- Not everything in a scene is a single list.
  - Think about a pack of fierce doggies, er, wolves attacking the quiet village in the forest.
  - Real scenes *cluster*.
- Is it the responsibility of the elements to know about layering and position?
  - Is that the right place to put that *know how*?
- How do we structure *operations* to perform to sets of nodes?
  - For example, moving a set of them at once?

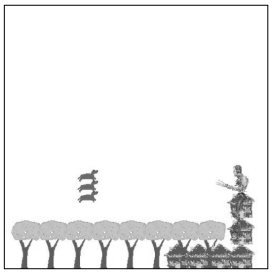
The Attack of the Nasty Wolvies



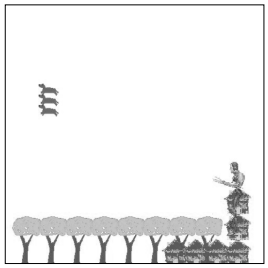
Closer…



Then the Hero Appears!



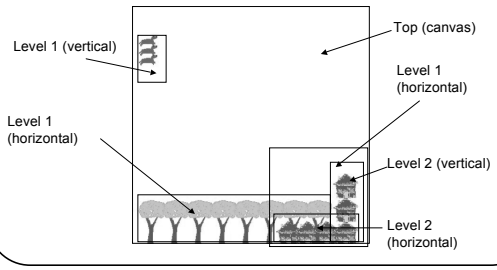
And the Wolvies retreat



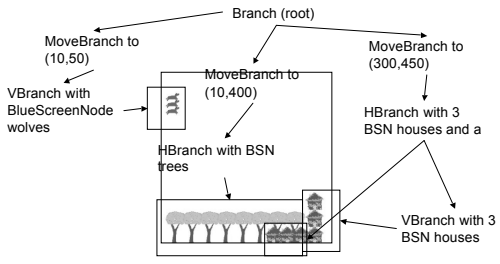
What's underlying this

- This scene is described by a *tree*
  - Each picture is a *BlueScreenNode* in this tree.
  - Groups of pictures are organized in *HBranch* or *VBranch* (Horizontal or Vertical branches)
  - The root of the tree is just a *Branch.*
  - The branches are positioned using a *MoveBranch*.

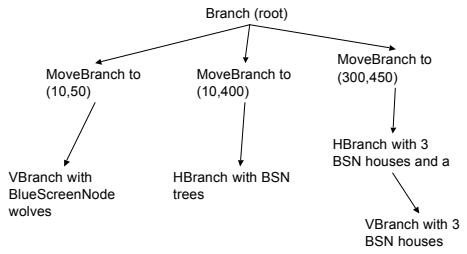
Nesting is a tree relationship!



Labeling the Pieces



It's a Tree (of instances!)



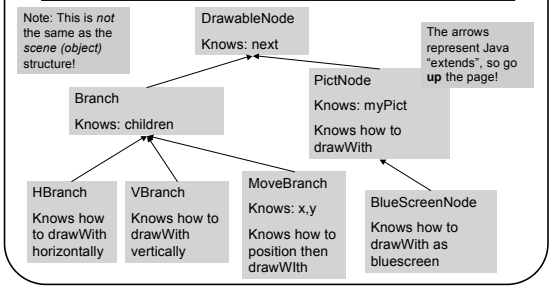
The Class Structure (another tree) 1. nodes

- *DrawableNode* **knows** only *next*, but **knows how** to do everything that our picture linked lists do (insertAfter, remove, last, drawOn(picture)).
  - Everything else is a subclass of that.
- *PictNode* **knows** it's *Picture myPict* and **knows how** to drawWith(turtle) (by dropping a picture)
- *BlueScreenNode* **knows how** to drawWith(turtle) by using bluescreen.

The Class Structure (another tree) 2. branches

- *Branch* **knows** its *children*—a linked list of other nodes to draw. It **knows how** to drawWith by:
  - (1) telling all its children to draw.
  - (2) then telling its next to draw.
- A **HBranch** draws its children by spacing them out horizontally.
- A **VBranch** draws its children by spacing them out vertically.

The Class Structure Diagram



Using these Classes: When doggies go bad!

```
public class WolfAttackMovie {
  /**
   * The root of the scene data structure
   **/
  Branch sceneRoot;

  /**
   * FrameSequence where the animation
   * is created
   **/
  FrameSequence frames;

  /**
   * The nodes we need to track between methods
   **/
  MoveBranch wolfentry, wolfretreat, hero;
```

These are the nodes that change *during* the animation, so must be available outside the local method context

Setting up the pieces

```
/**
  * Set up all the pieces of the tree.
  **/
 public void setUp(){
  Picture wolf = new Picture(FileChooser.getMediaPath("dog-
    blue.jpg"));
  Picture house = new Picture(FileChooser.getMediaPath("house-
    blue.jpg"));
  Picture tree = new Picture(FileChooser.getMediaPath("tree-
    blue.jpg"));
  Picture monster = new
    Picture(FileChooser.getMediaPath("monster-face3.jpg"));
```

Making a Forest

```
//Make the forest
  MoveBranch forest = new MoveBranch(10,400); // forest
   on the bottom
  HBranch trees = new HBranch(50); // Spaced out 50
   pixels between
  BlueScreenNode treenode;
  for (int i=0; i < 8; i++) // insert 8 trees
  {treenode = new BlueScreenNode(tree.scale(0.5));
    trees.addChild(treenode);}
  forest.addChild(trees);
```

Make attacking wolves

```
// Make the cluster of attacking "wolves"
  wolfentry = new MoveBranch(10,50); // starting position
  VBranch wolves = new VBranch(20); // space out by 20 pixels
   between
  BlueScreenNode wolf1 = new BlueScreenNode(wolf.scale(0.5));
  BlueScreenNode wolf2 = new BlueScreenNode(wolf.scale(0.5));
  BlueScreenNode wolf3 = new BlueScreenNode(wolf.scale(0.5));
  wolves.addChild(wolf1);wolves.addChild(wolf2);
   wolves.addChild(wolf3);
  wolfentry.addChild(wolves);
```

Make retreating wolves

```
// Make the cluster of retreating "wolves"
  wolfretreat = new MoveBranch(400,50); // starting position
  wolves = new VBranch(20); // space them out by 20 pixels
   between
  wolf1 = new BlueScreenNode(wolf.scale(0.5).flip());
  wolf2 = new BlueScreenNode(wolf.scale(0.5).flip());
  wolf3 = new BlueScreenNode(wolf.scale(0.5).flip());
  wolves.addChild(wolf1);wolves.addChild(wolf2);
   wolves.addChild(wolf3);
  wolfretreat.addChild(wolves);
```

It takes a Village…

```
// Make the village
  MoveBranch village = new MoveBranch(300,450); // Village on bottom
  HBranch hhouses = new HBranch(40); // Houses are 40 pixels apart
   across
  BlueScreenNode house1 = new BlueScreenNode(house.scale(0.25));
  BlueScreenNode house2 = new BlueScreenNode(house.scale(0.25));
  BlueScreenNode house3 = new BlueScreenNode(house.scale(0.25));
  VBranch vhouses = new VBranch(-50); // Houses move UP, 50 pixels
   apart
  BlueScreenNode house4 = new BlueScreenNode(house.scale(0.25));
  BlueScreenNode house5 = new BlueScreenNode(house.scale(0.25));
  BlueScreenNode house6 = new BlueScreenNode(house.scale(0.25));
  vhouses.addChild(house4); vhouses.addChild(house5);
   vhouses.addChild(house6);
  hhouses.addChild(house1); hhouses.addChild(house2);
   hhouses.addChild(house3);
  hhouses.addChild(vhouses); // Yes, a VBranch can be a child of an
   HBranch!
  village.addChild(hhouses);
```

Making the village's hero

```
// Make the monster
  hero = new MoveBranch(400,300);
  BlueScreenNode heronode = new
BlueScreenNode(monster.scale(0.75).fli
p());
  hero.addChild(heronode);
```

Assembling the Scene

```
//Assemble the base scene
  sceneRoot = new Branch();
  sceneRoot.addChild(forest);
  sceneRoot.addChild(village);
  sceneRoot.addChild(wolfentry);
}
```

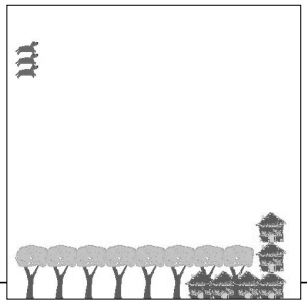
Want the forest on top of the village? Put the village in BEFORE the forest! Then it will get rendered first

Where's the wolfretreat and monster? They'll get inserted into the scene in the *middle* of the movie

Trying out one scene: Very important for testing!

```
/**
  * Render just the first scene
  **/
 public void renderScene() {
   Picture bg = new Picture(500,500);
   sceneRoot.drawOn(bg);
   bg.show();
 }
```

Okay that works



Rendering the whole movie

```
/**
  * Render the whole animation
  **/
 public void renderAnimation() {
   frames = new FrameSequence("D:/Temp/");
   frames.show();
   Picture bg;
```

Wolvies attack! (for 25 frames)

```
// First, the nasty wolvies come closer to the poor village
   // Cue the scary music
   for (int i=0; i<25; i++)
   {
     // Render the frame
     bg = new Picture(500,500);
     sceneRoot.drawOn(bg);
     frames.addFrame(bg);

     // Tweak the data structure
     wolfentry.moveTo(wolfentry.getXPos()+5,wolfentry.getYPos()+10);
   }
```

Inch-by-inch, er, 5-pixels by 10 pixels, they creep closer.

Our hero arrives! (In frame 26)

```
// Now, our hero arrives!
   this.root().addChild(hero);
   // Render the frame
   bg = new Picture(500,500);
   sceneRoot.drawOn(bg);
   frames.addFrame(bg);
```

Exit the threatening wolves, enter the retreating wolves

```
// Remove the wolves entering, and insert the wolves
   retreating
   this.root().children.remove(wolfentry);
   this.root().addChild(wolfretreat);
   // Make sure that they retreat from the same place that
   they were at
   wolfretreat.moveTo(wolfentry.getXPos(),
   wolfentry.getYPos());
   // Render the frame
   bg = new Picture(500,500);
   sceneRoot.drawOn(bg);
   frames.addFrame(bg);
```

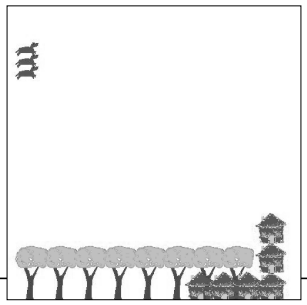
The wolves retreat (more quickly)

```
// Now, the cowardly wolves hightail it out of there!
   // Cue the triumphant music
   for (int i=0; i<10; i++)
   {
     // Render the frame
     bg = new Picture(500,500);
     sceneRoot.drawOn(bg);
     frames.addFrame(bg);

     // Tweak the data structure
     wolfretreat.moveTo(wolfretreat.getXPos()-10,
                        wolfretreat.getYPos()-20);
   }
 }
```

Making the Movie

```
Welcome to DrJava.
> WolfAttackMovie wam = new WolfAttackMovie();
    wam.setUp(); wam.renderScene();

> wam.renderAnimation();

There are no frames to show yet.  When you add a frame it
    will be shown

> wam.replay();
```

The Completed Movie



Okay, how'd we do that?

- This part is important!
- Remember: You have to do this for your animation with sound!
  - You need to understand how this actually works!
  - And, by the way, there's a lot of important Java in here!

DrawableNode: The root of the class structure

```
/**
 * Stuff that all nodes and branches in the
 * scene tree know.
 **/
abstract public class DrawableNode {
  /**
   * The next branch/node/whatever to process
   **/
  public DrawableNode next;

  /**
   * Constructor for DrawableNode just sets
   * next to null
   **/
  public DrawableNode(){
    next = null;
  }
```

DrawableNodes know how to be (chained into) linked lists

```
/**
   * Methods to set and get next elements
   * @param nextOne next element in list
   **/
  public void setNext(DrawableNode nextOne){
    this.next = nextOne;
  }

  public DrawableNode getNext(){
    return this.next;
  }
```

DrawableNodes know how to draw themselves (and list)

```
/**
 * Use the given turtle to draw oneself
 * @param t the Turtle to draw with
 **/
abstract public void drawWith(Turtle t);
// No body in the superclass

/**
 * Draw on the given picture
 **/
public void drawOn(Picture bg){
  Turtle t = new Turtle(bg);
  t.setPenDown(false);
  this.drawWith(t);
}
```

An *abstract method* is one that superclasses **MUST** *override*—they have to provide their *own* implementation of it.

DrawableNodes know all that linked list stuff

```
/** Method to remove node from list, fixing links appropriately.
 * @param node element to remove from list.
 **/
public void remove(DrawableNode node){
…
/**
 * Insert the input node after this node.
 * @param node element to insert after this.
 **/
public void insertAfter(DrawableNode node){
…

/**
 * Return the last element in the list
 **/
 public DrawableNode last() {
…

/**
 * Add the input node after the last node in this list.
 * @param node element to insert after this.
 **/
public void add(DrawableNode node){
  this.last().insertAfter(node);
}
```

PictNode is a kind of DrawableNode

```
/*
 * PictNode is a class representing a drawn picture
 * node in a scene tree.
 **/
public class PictNode extends DrawableNode {
  /**
   * The picture I'm associated with
   **/
  Picture myPict;
```

To construct a PictNode, first, construct a DrawableNode

```
/*
  * Make me with this picture
  * @param pict the Picture I'm associated with
  **/
 public PictNode(Picture pict){
   super(); // Call superclass constructor
   myPict = pict;
 }
```

If you want to call the superclass's constructor, you must do it *first*.

How PictNodes drawWith

```
/*
  * Use the given turtle to draw oneself
  * @param pen the Turtle to draw with
  **/
 public void drawWith(Turtle pen){
   pen.drop(myPict);
 }
```

BlueScreenNodes know nothing new

```
/*
 * BlueScreenNode is a PictNode that composes the
 * picture using the bluescreen() method in Picture
 **/
public class BlueScreenNode extends PictNode {

  /*
   * Construct does nothing fancy
   **/
  public BlueScreenNode(Picture p){
    super(p); // Call superclass constructor
  }
```

BlueScreenNodes draw differently

```
/*
  * Use the given turtle to draw oneself
  * Get the turtle's picture, then bluescreen onto it
  * @param pen the Turtle to draw with
  **/
 public void drawWith(Turtle pen){
   Picture bg = pen.getPicture();
   myPict.bluescreen(bg, pen.getXPos(),
   pen.getYPos());
 }
```

Branches add children

```
public class Branch extends DrawableNode {
  /*
   * A list of children to draw
   */
  public DrawableNode children;

  /*
   * Construct a branch with children and
   * next as null
   **/
  public Branch(){
    super(); // Call superclass constructor
    children = null;
  }
```

But because they're DrawableNodes, too, they still know how to be linked lists.
They reference things in *two* directions—as children and as next.

Hence, they *branch*.
Hence, a *tree*.

Adding children to a Branch

```
/**
  * Method to add nodes to children
  **/
 public void addChild(DrawableNode child){
   if (children != null)
   {children.add(child);}
     else
     {children = child;}
 }
```

Drawing a Branch

```
/*
  * Ask all our children to draw,
  * then let next draw.
  * @param pen Turtle to draw with
  **/
 public void drawWith(Turtle pen){
   DrawableNode current = children;

   // Tell the children to draw
   while (current != null){
     current.drawWith(pen);
     current = current.getNext();
   }

   // Tell my next to draw
   if (this.getNext() != null)
   {this.getNext().drawWith(pen);}
 }
```

HBranch: Horizontal Branches

```
public class HBranch extends Branch {

  /**
   * Horizontal gap between children
   **/
  int gap;

  /*
   * Construct a branch with children and
   * next as null
   **/
  public HBranch(int spacing){
    super(); // Call superclass constructor
    gap = spacing;
  }
```

HBranch draws horizontal children

```
/*
  * Ask all our children to draw,
  * then let next draw.
  * @param pen Turtle to draw with
  **/
 public void drawWith(Turtle pen){
   DrawableNode current = children;

   // Have my children draw
   while (current != null){
     current.drawWith(pen);
     pen.moveTo(pen.getXPos()+gap,pen.getYPos());
     current = current.getNext();
   }

   // Have my next draw
   if (this.getNext() != null)
   {this.getNext().drawWith(pen);}
 }
```

Just draws at a different position

VBranch is *exactly* the same, but vertically

```
public void drawWith(Turtle pen){
  DrawableNode current = children;

  // Have my children draw
  while (current != null){
    current.drawWith(pen);
    pen.moveTo(pen.getXPos(),pen.getYPos()+gap);
    current = current.getNext();
  }

  // Have my next draw
  if (this.getNext() != null)
  {this.getNext().drawWith(pen);}
}
```

MoveBranch is different

```
public class MoveBranch extends Branch {

 /**
  * Position where to draw at
  **/
 int x,y;

 /**
  * Construct a branch with children and
  * next as null
  **/
 public MoveBranch(int x, int y){
   super(); // Call superclass constructor
   this.x = x;
   this.y = y;
 }
```

MoveBranch accessors, to make them movable

```
/**
  * Accessors
  **/
 public int getXPos() {return this.x;}
 public int getYPos() {return this.y;}
 public void moveTo(int x, int y){
   this.x = x; this.y = y;}
```

MoveBranch passes the buck on drawing

```
/*
  * Set the location, then draw
  * @param pen Turtle to draw with
  **/
 public void drawWith(Turtle pen){
   pen.moveTo(this.x,this.y);
   super.drawWith(pen); // Do a normal branch
   now
 }
```

Doing the Branches…backwards

- What if you processed *next* <u>*before*</u> the children?
- What if you did the move *after* you did the superclass drawing?
- What would the scene look like?
- **Different kinds of tree traversals…**

Representing Structure and Behavior

- Think about trees
  - Branches represent structure
  - HBranch, VBranch, and MoveBranch represent structure *and* behavior
- Think about objects
  - They *know* things, and they *know how* to do things.
  - They represent structure and behavior.
- Sophisticated programs represent both.
- ***The line between data and programs is very thin…***