Cheloniidae Turtle Graphics

Spencer Tipping

Contents

1 Introduction **1**

2 Extensions **1**
2.1 State Stack 2
2.2 Coordinate Models 2
2.2.1 Spherical Coordinates 2
2.2.2 Cylindrical Coordinates 3
2.2.3 Orthogonal Planar Coordinates 3

3 Command Index **3**
3.1 Basic Commands 4
3.2 3D Commands 4
3.3 Accessors 5
3.4 Advanced Turtle Commands 6

1 Introduction

Cheloniidae turtle graphics is a capable, easy-to-use library for Java. It provides many advanced features such as three-dimensional space support under three different coordinate models, multiple turtles, antialiased and depth-adjusted rendering, highly customizable axes and grids, and perspective projection, all without losing the simplicity and elegance of pure vector graphics.

This library originated as a drop-in replacement for the Galapagos turtle software, used primarily for educational purposes. As such, there are two sets of commands: One set uses the same nomenclature and functionality as the original Galapagos library, and the other uses my nomenclature and supports the extensions listed previously. For more details, see section 3.

2 Extensions

As mentioned earlier, Cheloniidae provides several extensions to the traditional turtle model.

2.1 State Stack

Each turtle maintains a stack of *states*. A state is simply a turtle's location and heading. Under normal operation, a turtle has only one state, which is modified with the normal movement and turn commands. However, a turtle can also "remember" its old states and jump back to them by pushing and popping states from its internal state stack.

This is useful for building a fractal tree, for instance. The included example file `tree.java` implements this concept using the turtle stack. The premise is this: A tree consists of a line forwards and two more trees spaced at some angle apart. Here is some simple code to implement this idea:

```
public static void tree (Turtle t, int recursionLevel) {
  t.move (10);

  if (recursionLevel > 0) {
    t.pushTurtleState ();
    t.turn (-5);
    tree (t, recursionLevel - 1);
    t.popTurtleState ();
    t.pushTurtleState ();
    t.turn (5);
    tree (t, recursionLevel - 1);
    t.popTurtleState ();
  }
}
```

The state stack works very well with recursive algorithms, as demonstrated here. The number of state entries that can be stored at a time is limited only by Java's heap size, however it is an error to pop an empty stack.

2.2 Coordinate Models

Cheloniidae seamlessly integrates two-dimensional and three-dimensional drawing. By default, a turtle draws only on the plane and the model is for all purposes two-dimensional. However, by changing the turtle's ϕ heading, the turtle begins to change its z position.

Depending on the drawing, different coordinate models will make sense. For example, drawing a sphere such as those in the included example `spheres.java` is most easily achieved using spherical coordinates. However, spherical coordinates make it very difficult to draw a spiral. There are similar tasks that are well-suited to other models, so three separate coordinate models are provided. Others may be written for Cheloniidae without very much work, especially by inheriting from the `Turtle` class.

2.2.1 Spherical Coordinates

Spherical coordinates allow the turtle's default plane to be bent into a cone whose depth angle is determined by ϕ. By default, $\phi = 0$, so the plane is flat. Mathematically, the distance vector $\langle dx, dy, dz \rangle$ moved by the turtle for a move of distance d is computed this way:

$$dx = d\cos\theta\cos\phi$$
$$dy = d\sin\theta\cos\phi$$
$$dz = d\sin\phi$$

2.2.2 Cylindrical Coordinates

Cylindrical coordinates allow the turtle's drawing plane to be rotated about the Y axis. The degree of rotation is determined by ϕ, which is zero by default. Mathematically, cylindrical coordinates are implemented as follows (see section 2.2.1 for notation):

$$dx = d\cos\theta\cos\phi$$
$$dy = d\sin\theta$$
$$dz = d\cos\theta\sin\phi$$

2.2.3 Orthogonal Planar Coordinates

The idea behing orthogonal planar coordinates is that the plane in which the turtle's local θ coordinate operates can be rotated arbitrarily. By default, its normal vector is oriented along the z axis; however, ϕ and ξ provide y-axis rotation and relative x-axis rotation, respectively, in a spherical setting. Mathematically, orthogonal planar coordinates are implemented as follows (see section 2.2.1 for notation):

$$dx = d\cdot[\cos\theta\cos\phi + \sin\theta\sin\phi\sin\xi]$$
$$dy = d\sin\theta\cos\xi$$
$$dz = d\cdot[\cos\theta\sin\phi - \sin\theta\cos\phi\sin\xi]$$

3 Command Index

This section lists all of the commands supported by the Turtle class in Cheloniidae. All of the examples assume the following definitions:

```
import cheloniidae.*;

public class test {
  public static void main (String[] args) {
    TurtleDrawingWindow w = new TurtleDrawingWindow ();
    Turtle t = new Turtle ();
    w.add (t);

    // Any example would be legal here.

    w.setVisible (true);
  }
}
```

3.1 Basic Commands

These commands allow the turtle to create basic shapes. If only these commands are used, then the turtle will remain in a two-dimensional plane; thus we postpone the three-dimensional details to section 3.2.

`move` Moves the turtle a given distance along its heading, drawing a line if the pen is down. If the distance is negative, then the turtle moves backwards.

Example: `t.move(100);`

`moveTo` Moves the turtle to a specific location, drawing a line if the pen is down. The turtle's heading is not taken into account and not changed by this operation.

Example: `t.moveTo(50, 10);`

`jump` Moves the turtle a given distance along its heading without drawing a line. If the distance is negative, then the turtle moves backwards.

Example: `t.jump(100);`

`jumpTo` Moves the turtle to a specific location without drawing a line. Like `moveTo`, the turtle's heading is not considered for this operation.

Example: `t.jumpTo(10, 10);`

`turn` Adds an angle to the turtle's heading. The turtle does not draw anything when it is turned. All headings are measured in degrees.

Example: `t.turn(90);`

`setPenColor` Sets the color of future lines drawn by the turtle. The color may be translucent, in which case the drawn lines will also be translucent.

Example: `t.setPenColor(java.awt.Color.BLUE);`

`setPenIsDown` Determines whether the turtle will draw lines when moved. By default, this is true.

Example: `t.setPenIsDown(true);`

3.2 3D Commands

These commands allow the turtle to travel anywhere in 3D space and change coordinate models.

`moveTo` In addition to the two-dimensional version, `moveTo` may also take a `z` parameter.

Example: `t.moveTo(0, 3, 10);`

`jumpTo` The same is true of the `jumpTo` command.

Example: `t.jumpTo(10, 28, 1);`

`turnTheta` Rotates the turtle within its immediate plane or cone. The exact behavior of this command is determined by the coordinate model used. (See section 2.2.) This command is an alias for the `turn` command. (See section 3.1.)

Example: `t.turnTheta(-90);`

`turnPhi` Adjusts the turtle's ϕ heading. The exact behavior of the ϕ heading depends on the coordinate model. (See section 2.2.)

Example: `t.turnPhi(45);`

`turnXi` Adjusts the turtle's ξ heading. This is meaningful only if the turtle is using the orthogonal-planar coordinate model. (See sections 2.2.3 and 2.2.)

Example: `t.turnXi(30);`

`setPolarAxisModel` Determines the roles of the turtle's 3D heading coordinates, θ, ϕ, and ξ. See section 2.2 for a mathematical description of these roles. Valid settings are `Turtle.Z_SPHERICAL`, `Turtle.Y_CYLINDRICAL`, and `Turtle.ORTHOGONAL_PLANAR`.

Example: `t.setPolarAxisModel(Turtle.Z_SPHERICAL);`

3.3 Accessors

The turtle provides accessors to its position, heading, and several other fields. Accessors follow a standard naming convention; a field named `namingConvention`, for example, would have accessors `getNamingConvention` and `setNamingConvention`.

- X, Y, Z: `double`
- headingTheta, headingPhi, headingXi: `double`
- bodyColor: `java.awt.Color`
- penColor: `java.awt.Color`
- penIsDown: `boolean`
- penSize: `double`
- delayPerMove: `int`
- visible: `boolean`
- polarAxisModel: `int` – see section 3.2 for a list of valid values.

3.4 Advanced Turtle Commands

These commands are not normally used for simple scenes. They are provided for more complex scenes such as fractal formations. Others are simply more esoteric commands that one doesn't normally use.

`pushTurtleState` Each turtle maintains a stack of states (see section 2.1). This command causes the turtle to push its current state onto the stack for later recovery. Its current state is not altered by this command.

Example: `t.pushTurtleState();`

`popTurtleState` This command causes the turtle to pop the last pushed state from the stack and restore its state. Its current state is replaced by the state from the top of the stack.

Example: `t.popTurtleState();`

`replicate` Returns a duplicate of the current turtle and adds it to the current `TurtleDrawingWindow`. The result is that code such as this:

```
Turtle t1 = t.replicate ();
t1.move (100);
```

is not only legal but does what it seems like it should.

`setBodyColor` Each turtle is drawn in its position on the screen. Setting the body color determines what color it is drawn in. Translucency is enabled for this color setting.

Example: `t.setBodyColor(java.awt.Color.BLACK);`

`setPenSize` Sets the width of the lines that the turtle draws. 0.5 is approximately the minimum, and there is no maximum.

Example: `t.setPenSize(0.5);`

`setVisible` Determines whether the turtle itself is drawn. If this is false, then only the lines drawn by the turtle are visible, but the turtle is hidden. Setting this value to false may improve rendering speed if rendering is performed while the `TurtleDrawingWindow` is visible.

Example: `t.setVisible(false);`

`setDelayPerMove` Each turtle has a default delay amount per move. This is so that the drawing process can be observed step-by-step. The number of milliseconds to wait per move may be changed using this parameter. If set to zero, then the turtle draws as fast as possible. Note that this parameter doesn't make any difference if the turtle is drawing on a window before the window is shown.

Example: `t.setDelayPerMove(0);`