Timing Framework

by Chet Haase

Beyond the Built-in Timers

- Key functionality lacking in core timers for typical animation requirements
- The Basics
  - duration
    - nothing lasts forever (besides acne)
  - elapsed time:
    - have the system tell you how much animation has elapsed
  - repeat:
    - repeating, reversing animations common
- Advanced
  - many other requirements for typical animations, like non-linear interpolation

Timing Framework

- http://timingframework.dev.java.net
  - Project in development for the last two+ years
- Core concepts:
  - Cycle: basic animation loop
    - duration, resolution
  - Envelope: contains one or more Cycles
    - number of cycles, start delay, repeat behavior, end behavior
  - TimingTarget: callback target
    - begin, end, repeat, timingEvent(fraction)
  - Animator:
    - Cycle and envelope properties, one or more TimingTargets
- Not Swing *animation* engine
  - *Timing* engine on which animations can be more easily built

The Basics: Sample Code

```
class MyTarget implements TimingTarget {
      public void begin() {…}
      public void end() {…}
      public void repeat() {…}
      public void timingEvent(float fraction) {…}
}
TimingTarget target = new MyTarget();

// animate once for 5 seconds, then stop
Animator singleRun = new Animator(5000, target);
singleRun.start();

// animate for 5 cycles of 2 secs, reversing each time
Animator oscillator = new Animator(2000, 5,
                                   RepeatBehavior.REVERSE,
                                   target);
oscillator.start();
```

Demo: BasicRace

BasicRace: Sample Code

```
public class BasicRace extends TimingTargetAdapter
          implements ActionListener {
  // Starts/stops timer based on Go/Stop action events
  public void actionPerformed(ActionEvent ae) {
    if (ae.getActionCommand().equals("Go")) {
      timer = new Animator(RACE_TIME, this);
      timer.start();
    } else if (ae.getActionCommand().equals(“Stop”)) {
      timer.stop();
    }
  }
  // Callback: Linearly interpolate car position according
  // to fraction of animation elapsed thus far
  public void timingEvent(float fraction) {
    current.x = (int)(start.x + (end.x-start.x) * fraction);
    current.y = (int)(start.y + (end.y-start.y) * fraction);
    track.setCarPosition(current);
  }
}
```

Advanced Concepts

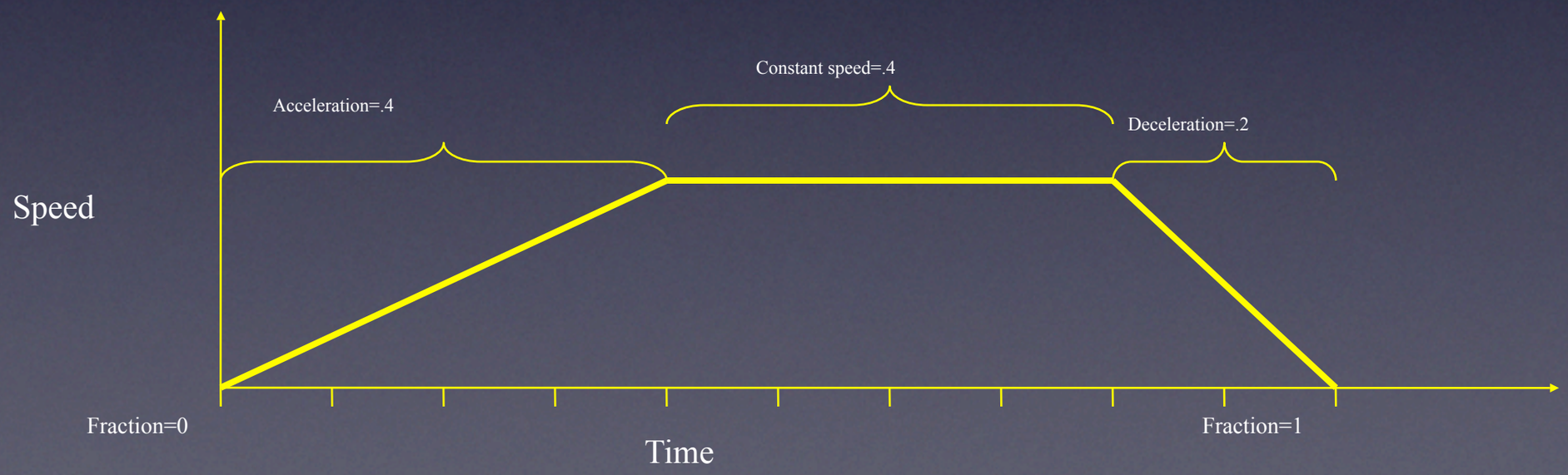
- Non-Linear Interpolation
  - More realistic motion
- Property Setters
  - TimingTargets that animate JavaBean properties

Non-Linear Interpolation

- When was the last time you saw someone gliding smoothly along the street?
  - Not counting bad movie camera effects
- We live in a non-linear world
  - Gravity, acceleration, deceleration, friction
    - ... as well as tripping, stumbling, falling, crashing, settling
- ... so our eyes expect to see non-linear movement
- Linear movement results in bad animations
  - Looks unnatural
  - Emphasizes rendering artifacts
    - Easy to track mistakes and hiccups when we are tracking linear movement

Acceleration/Deceleration

- Easiest approach for simple situations
  - setAcceleration(float)
  - setDeceleration(float)
- Fraction of cycle speeding up, slowing down



Interpolator

```
public Interface Interpolator {
    public float interpolate(float fraction);
}
```

- Set on Animator

  Animator.setInterpolator(Interpolator)

- Pre-defined implementations
  - LinearInterpolator
  - DiscreteInterpolator
  - SplineInterpolator
- Or build your own

PropertySetter

- Animate JavaBean properties of Objects
  - "location" of a button, "bounds" of a label, …
- Works for any property name ("prop") that has related setter ("setProp")
  - Component's size, foreground, location, …
- Custom components or delegators when no appropriate property exists
  - e.g., opacity, rotation, scale
- PropertySetter implements TimingTarget

Example: PropertySetter

```
// Move button between ‘from’ and ‘to’ in
// 2 seconds
Point from = (50, 50);
Point to = (100, 150);
PropertySetter ps = new PropertySetter(
    button, “location”, from, to);
Animator mover = new Animator(2000, ps);
mover.start();

// Same thing, only easier
PropertySetter.
    createAnimator(2000,button,“location”,
                   from,to).start();
```

SetterRace

- Easier means of moving car than in BasicRace
- No need to handle timingEvent()

```
Animator racer =
    PropertySetter.createAnimator(
        RACE_TIME, track, "carPosition",
        start, end);
```

Triggers

- Wrappers around event listeners
- Automatically start animations based on GUI events and other animations
  - component action events
  - button enter/exit
  - focus
  - animation start/stop/repeat
  - …

Demo: Triggers

Triggers: The Code

```
ActionTrigger.addTrigger(triggerButton, action.getAnimator());
FocusTrigger.addTrigger(triggerButton,
                focus.getAnimator(), FocusTriggerEvent.IN);
MouseTrigger.addTrigger(triggerButton,
                armed.getAnimator(), MouseTriggerEvent.PRESS);
MouseTrigger.addTrigger(triggerButton,
                over.getAnimator(), MouseTriggerEvent.ENTER);
TimingTrigger.addTrigger(action.getAnimator(),
                timing.getAnimator(), TimingTriggerEvent.STOP);
```

But Wait, There's More!

- Multi-Step Animations
  - Support more complex animations
    - Key frames
      - times, values, interpolation for multiple intervals
- More properties in Animator
  - initialFraction, direction, EndBehavior

Demo: FinalRace

Timing Framework Summary

- Everything is possible with built-in timers
  - But much easier with Timing Framework
- API relatively stable, v 1.0 soon
- Animated Transitions library
  - Not published yet, available by the time "Filthy Rich Clients" is released (early May)
- Swing Animation feature for JDK 7
- Project site: http://timingframework.dev.java.net

Q&A

chet.haase@sun.com