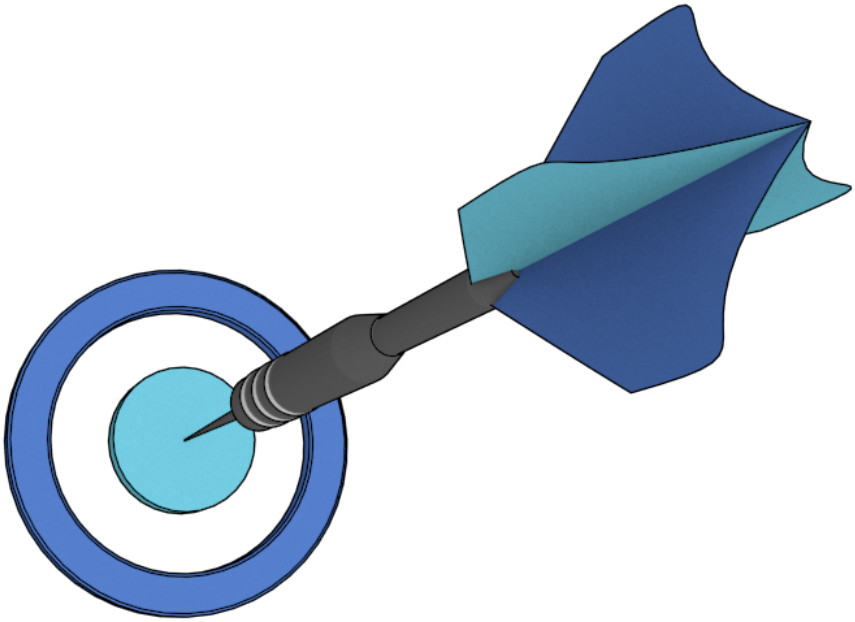
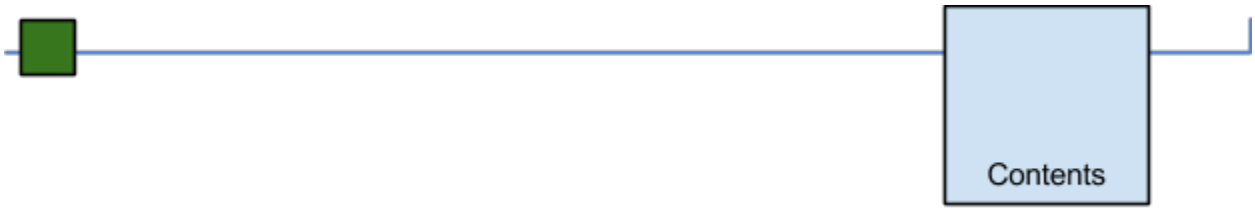
Ranger Animation

version 1.0.0

William R. DeVore





Universal Tween Engine

Animation in **Ranger** is provided by the Universal Tween Engine (UTE). The port to Dart is called TweenEngine by Xavier Guzman.

A fair amount of the document is going to be talking about how **Ranger** uses UTE to animate objects. But before we can do that we need to talk about UTE (btw, a good starting reference is [TweenEngine's Github](#).

UTE

You can view UTE from a conceptually high level in four parts: **TweenManager**, **Tween**, **TweenAccessor** and **Tweenable**.

The TweenManager is handles your tweens, routes timing to them and manages their lifetime.

You use Tween to create animations and hand them over to a TweenManager. In **Ranger**'s case the TweenManager is contained within **Ranger**'s TweenAnimation.

The animations you create have either implemented the Tweenable interface or created a TweenAccessor wrapped around some independent object you want to animate. TweenAccessor allows you to animate any object. The object has no idea it is being manipulated by UTE. On the other hand an object that implemented the Tweenable interface is acutely aware that it will be manipulated by UTE.

TweenAccessor

A good example of a TweenAccessor (other than **Ranger**'s TweenAnimation) can be seen in the level 4 template. The SVG icon is animated using an accessor called **RotateAnimationAccessor**

```
class RotateAnimationAccessor extends UTE.TweenAccessor<Ranger.SpriteImage> {
  static const int ROTATE = 50;

  RotateAnimationAccessor();

  int getValues(Ranger.SpriteImage target, UTE.Tween tween, int tweenType, List<num>
returnValues) {
    switch (tweenType) {
      case ROTATE:
        returnValues[0] = target.rotationInDegrees;
        return 1;
    }
```

```
    return 0;
  }

  void setValues(Ranger.SpriteImage target, UTE.Tween tween, int tweenType, List<num>
newValues) {
    switch (tweenType) {
      case ROTATE:
        target.rotationByDegrees = newValues[0];
        target.dirty = true;
        break;
    }
  }
}
```

Defining the accessor isn't enough you still need to register an instance of it with UTE

```
layer._rotateArrowAni = new RotateAnimationAccessor();
...
UTE.Tween.registerAccessor(Ranger.SpriteImage, _rotateArrowAni);
```

Once registered you can begin creating animations

```
      UTE.Tween tw = new UTE.Tween.to(_arrowSprite, RotateAnimationAccessor.ROTATE,
0.25)
        ..targetRelative = [-180.0]
        ..easing = UTE.Linear.INOUT
        ..callback = _rotationComplete
        ..callbackTriggers = UTE.TweenCallback.COMPLETE // We only need the complete
signal.
        ..userData = _arrowSprite; // optional, but can be handy.
      app.animations.add(tw); // Tween starts when added
```

UTE will cross match _arrowSprite with the registered accessor and begin animating it using the RotateAnimationAccessor that "wraps" your object. Shown on the last line, the Tween created will be given to TweenAnimation's internal TweenManager for handling. You can create as many TweenManagers as you want but TweenAnimation already defines one and it is freely available to use wherever and whenever necessary.

Tweenable

If the intent of creating an object is with the sole purpose of it being animated by UTE then implementing the Tweenable interface is what you do.

A good example of a Tweenable is level 5's slide-out panel TestsDialog. This is a class that directly modifies the CSS properties of an HTML DIV element

```
  int getTweenableValues(UTE.Tween tween, int tweenType, List<num> returnValues) {
    switch(tweenType) {
```

```
      case X:
        int pos = content.style.left.indexOf("p");
        returnValues[0] = double.parse(content.style.left.substring(0, pos));
        return 1;
    }

    return 0;
  }

  void setTweenableValues(UTE.Tween tween, int tweenType, List<num> newValues) {
    switch(tweenType) {
      case X:
        content.style.left = "${newValues[0]}px";
        break;
    }
  }
```

The Tweenable's getter reads the DIV's content.style.left property and the setter modifies the property based on the current tween value.

Because TestsDialog is a Tweenable you don't need to register it as you would an accessor, you simply pass it to a Tween creator method and then pass the created tween to TweenAnimation's handy dandy TweenManager (app.animations.add(tw)).

```
    Ranger.Application app = Ranger.Application.instance;

    UTE.Tween tw = new UTE.Tween.to(this, X, 0.5);
    tw..targetRelative = [-_panelWidth.toDouble()]
      ..easing = UTE.Cubic.OUT
      ..callback = _tweenCallbackHandler
      ..callbackTriggers = UTE.TweenCallback.COMPLETE;
      app.animations.add(tw);
```

Callbacks

Sometimes you want to react when your tween animation finishes/completes. With your new tween you can assign a TweenCallbackHandler

```
typedef void TweenCallbackHandler(int type, BaseTween source);
```

and define trigger filters

```
      ..callback = _tweenCallbackHandler
      ..callbackTriggers = UTE.TweenCallback.COMPLETE;
```

In the case of the TestsDialog class we want to "hide" the element once it has shuffled out of view

```
void _tweenCallbackHandler(int type, UTE.BaseTween source) {
  switch(type) {
    case UTE.TweenCallback.COMPLETE:
      _transitioning = false;
      if (!isShowing) {
        content.style.visibility = "hidden";
      }
      break;
  }
}
```

Timelines

UTE also provides Timelines for both Sequential or Parallel animation Tweens.

Sequence

RangerRocket has an example of a sequential tween. First the object is scaled “up” then “down” and finally rotated.

```
  UTE.Timeline seq = new UTE.Timeline.sequence();

  UTE.Tween scaleUp = app.animations.scaleTo(
      _trianglePolyNode,
      0.5,
      200.0, 200.0,
      UTE.Elastic.OUT,
      Ranger.TweenAnimation.SCALE_XY,
      null, Ranger.TweenAnimation.NONE,
      false);
  seq.push(scaleUp);

  UTE.Tween scaleDown = app.animations.scaleTo(
      _trianglePolyNode,
      1.0,
      100.0, 100.0,
      UTE.Linear.INOUT,
      Ranger.TweenAnimation.SCALE_XY,
      null, Ranger.TweenAnimation.NONE,
      false);
  seq.push(scaleDown);

  UTE.Tween rotate = app.animations.rotateBy(
      _trianglePolyNode,
      0.25,
      5.0,
      UTE.Cubic.INOUT,
      null,
      false);
  seq.push(rotate);

  seq.start();
```

Notice at the end I don't call start() with a parameter. Again, by using TweenAnimation's creator methods the tweens have already been added to the TweenManager. So all that is required is to call *start* on your tween.

Parallel

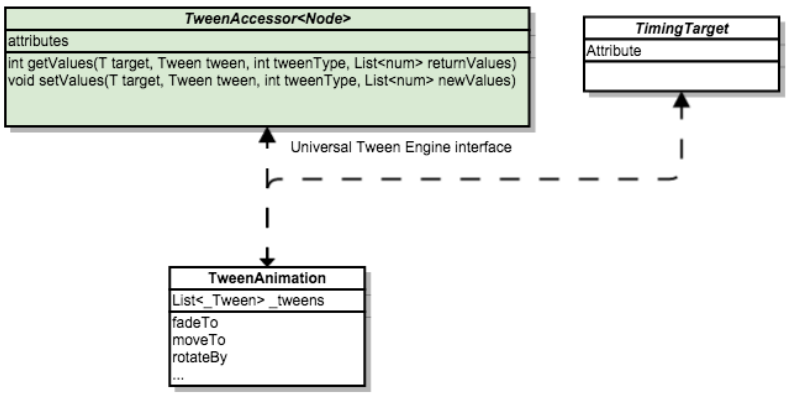
Ranger has an example of a parallel tween in the TransitionRotateAndZoom class. The transition simultaneously rotates and scales the incoming scene but not after an optional pause. I chose to use a containing sequence but I could have just as easily added a pause: par.pushPause(...)

```
    UTE.Timeline par = new UTE.Timeline.parallel();

    par.push(app.animations.rotateBy(inScene,
                                     duration / 1.5,
                                     720.0,
                                     UTE.Sine.OUT, null, false));

    par.push(app.animations.scaleTo(inScene,
                                     duration / 1.2,
                                     1.0, 1.0,
                                     UTE.Sine.OUT,
                                     TweenAnimation.SCALE_XY,
                                     null, TweenAnimation.MULTIPLY, false));

    UTE.Timeline seq = new UTE.Timeline.sequence();
    if (pauseFor > 0.0)
      seq.pushPause(pauseFor);
    seq.push(par);
    seq..push(app.animations.callFunc(0.0, _finishCallFunc, null, false))
       ..start();
```

Notice that I also push a *callFunc* tween at the end. All transitions require that *finish*(...) is called when the transition is has completed. The callFunc allows us to tack on a "task/callback". The callback does nothing more than properly call *finish*.

TweenAnimation

Ranger has one major TweenAccessor called TweenAnimation. It sole purpose is to provide an interface similar to the original Cocos2d-js animation system. You are not require to use it, but it can save you some effort if you need simple one-off type animations.



Bonus
TweenAnimation also provides a default TweenManager that can be used by any Tween you created “manually” outside of TweenAnimation. You don’t need to create a separate TweenManager of your own just use TweenAnimation’s.

TweenAnimation implements the TimingTarget interface which makes it possible for scheduling with the Scheduler.

You don’t need to schedule it because it is done automatically by the Application class during bootstrap phase.

To use it simply call one of the methods that creates a tween for you. Here is an example of rotating one of the spinners. “False” is passed as the last parameter to allow us to set the repeat count.

```
    Ranger.SpriteImage si = new Ranger.SpriteImage.withElement(spinner);

    UTE.Tween rot = app.animations.rotateBy(
        si,
        1.5,
        -360.0,
        UTE.Linear.INOUT, null, false);
    rot..repeat(UTE.Tween.INFINITY, 0.0)
       ..start();
```

Once a tween is created you can’t change certain parameters. In this case the repeat count must be set *before* the tween is started.

By using the TweenAnimation’s method your tween is automatically passed to the TweenManager.

Doubling up
You shouldn't call start() with the TweenAnimation's tween manager otherwise your animation will play twice as fast which isn't what you want.

TweenAnimation has quite a few tween creator methods for creating one-off like animations. Be sure to peruse the class and see which ones work for your game. At very least they act as examples of creating your own Tweens.

End.