Animation

- Java 3D provides a very powerful and easy to use animation facility
- It is based on two classes
  - Alpha
  - Interpolator
- Animations are run in separate threads
  - This is handled by the Alpha and Interpolator classes so the programmer doesn't have to worry about it

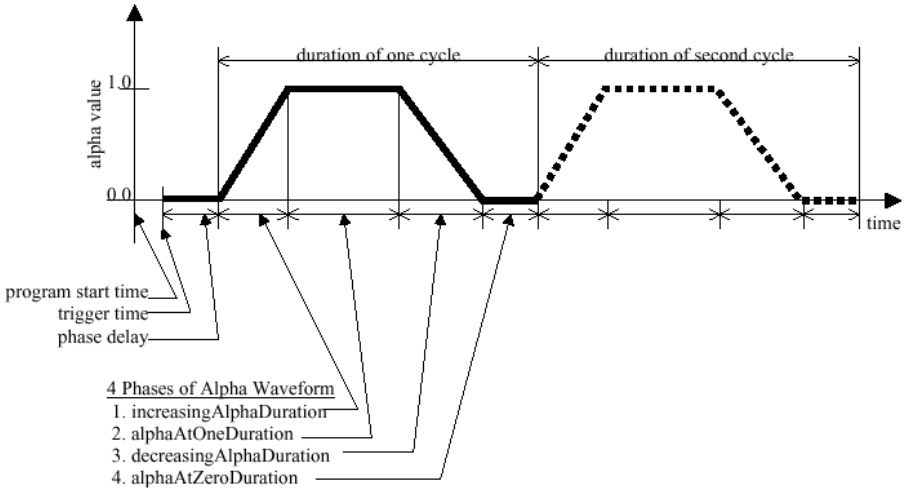
The Alpha and Interpolator of Java 3D

- Alpha controls timings
  - The start times for all alpha objects are the same
    - This is the system start time
    - All events are therefore synchronised even if they are instigated by different alpha objects
- Interpolator defines behaviours
  - Interpolators are awakened by certain events
  - The processing of these events constitutes the animation

Alpha

- An alpha object generates a value between 0 & 1
  - This value is determined by the current time and the parameters used to construct the alpha object
- Plotting alpha over time gives an *alpha waveform*
- There are four phases to the waveform
  - Increasing Alpha
  - Alpha at One
  - Decreasing Alpha
  - Alpha at Zero

The Alpha Waveform



Setting up an Alpha I

`Alpha()`

Constructs an Alpha object with

`mode = INCREASING_ENABLE,`
`loopCount = -1,`
`increasingAlphaDuration = 1000,`
all other parameters = 0 except `triggerTime`
which is set to the start time of the program

Setting up an Alpha II

```
Alpha(int loopCount, int mode,
          long triggerTime,
          long phaseDelayDuration,
          long increasingAlphaDuration,
          long increasingAlphaRampDuration,
          long alphaAtOneDuration,
          long decreasingAlphaDuration,
          long decreasingAlphaRampDuration,
          long alphaAtZeroDuration)
```

A negative `loopCount` means cycle indefinitely

All times and durations are specified in milliseconds

Setting up an Alpha III

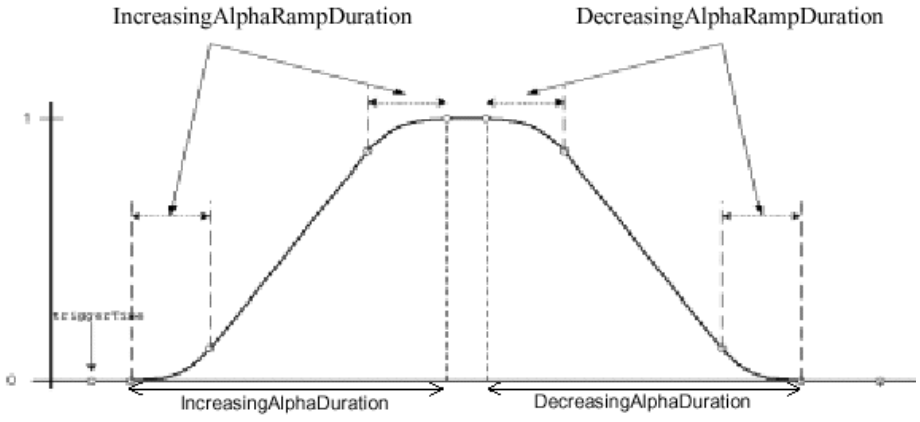
- Mode is one of the following -
  - INCREASING_ENABLE

    Only the Increasing Alpha and Alpha at One phases should be used
  - DECREASING_ENABLE

    Only the Decreasing Alpha and Alpha at Zero phases should be used
  - INCREASING_ENABLE | DECREASING ENABLE

    All four phases should be used
- Ramp durations smooth the alpha waveform
  - During the ramp duration alpha changes gradually
  - Ramp durations are applied at the start **and** end of a phase

Ramped Alpha Waveform



Interpolator

Interpolator extends Behavior

Interpolator class	used to	target object type
`ColorInterpolator`	change the diffuse color of an object(s)	Material
`PathInterpolator`[10]	*abstract class*	*TransformGroup*
`PositionInterpolator`	change the position of an object(s)	TransformGroup
`RotationInterpolator`	change the rotation (orientation) of an object(s)	TransformGroup
`ScaleInterpolator`	change the size of an object(s)	TransformGroup
`SwitchValueInterpolator`	choose one of (switch) among a collection of objects	Switch
`TransparencyInterpolator`	change the transparency of an object(s)	TransparencyAttributes

Interpolator Axes

- By default all interpolators work on the Y axis
- In order to get them to work on other axes we need to specify a transform that converts from the Y axis to the new axis.
- For example, to allow an interpolator to work on an X axis we need to supply a transform that rotates about Z by -90
  - Note that if you rotate an axis system in this way the Y axis lines up with what was previously the X axis
  - We can use
    ```
    myTransform3D.rotZ(3*Math.PI/2.0d);
    ```
- For it to work on a Z axis , we need to rotate about X by +90
  - We can use
    ```
    myTransform3D.rotX(Math.PI/2.0d);
    ```

Translational Interpolators

`PositionInterpolator(Alpha alpha, TransformGroup target)`

Constructs a trivial position interpolator with a specified target

Default axis of translation is X, startPosition is 0.0f and endPosition is 1.0f

`PositionInterpolator(Alpha alpha, TransformGroup target, Transform3D axisOfTranslation, float startPosition, float endPosition)`

Constructs a new position interpolator that varies the target TransformGroup's translational component from startPosition to endPosition along the specified axis of translation

The axisOfTranslation is the identity matrix if the Y axis is required

Rotational Interpolators

`RotationInterpolator(Alpha alpha, TransformGroup target)`

Constructs a rotation interpolator with a specified target, the axisOfRotation set to the Y axis, a minimum angle of 0.0f and a maximum angle of 2*pi radians

`RotationInterpolator(Alpha alpha, TransformGroup target, Transform3D axisOfRotation, float minimumAngle, float maximumAngle)`

Constructs a new rotation interpolator that varies the target transform node's rotational component from minimumAngle to maximumAngle about an axis defined by axisOfRotation

The axisOfRotation is the identity matrix if the Y axis is required

Scale Interpolators

`ScaleInterpolator(Alpha alpha, TransformGroup target)`

Constructs a scale interpolator that varies its target TransformGroup node between the two specified alpha values using the specified alpha, an identity matrix, a minimum scale = 0.1f and a maximum scale = 1.0f

```
ScaleInterpolator(Alpha alpha, TransformGroup target,
Transform3D axisOfScale, float minimumScale,
float maximumScale)
```

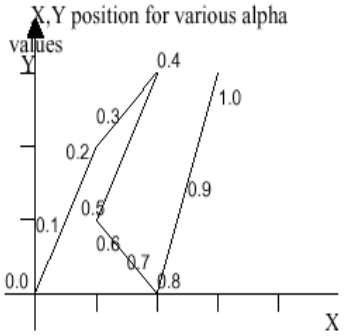
Constructs a new scaleInterpolator object that varies its target TransformGroup node's scale component between two scale values (minimumScale and maximumScale)

Path Interpolators

- Path interpolator objects store a set of values called *knots*
  - These are used two at a time during interpolation
  - The alpha value determines which two knot values to use
- Knot values lie in the range of 0.0 to 1.0
  - These correspond to the range of values of the alpha object
  - The first knot must have a value of 0.0
  - The last knot must have a value of 1.0
  - Other knots must be stored in increasing order
- Knot values correspond to the minimum and maximum values used in the various interpolations
  - One parameter value is specified for each knot value
  - The knot with the largest value less than or equal to the alpha value and the subsequent knot are used in the interpolation

Knot Values

knot	knot value	position(x, y, z)
0	0.0	(0.0, 0.0, 0.0)
1	0.2	(1.0, 2.0, 0.0)
2	0.4	(2.0, 3.0, 0.0)
3	0.5	(1.0, 1.0, 0.0)
4	0.8	(2.0, 0.0, 0.0)
5	1.0	(3.0, 3.0, 0.0)



alpha value (a)	knots used
0.0	0
0.0 < a < 0.2	0, 1
0.2	1
0.2 < a < 0.4	1, 2
0.4	2
0.4 < a < 0.5	2, 3
0.5	3
0.5 < a < 0.8	3, 4
0.8	4
0.8 < a < 1.0	4, 5
1.0	5

Path Interpolator Types

- Java 3D provides path interpolator classes for -
  - position interpolation
  - rotation interpolation
  - position and rotation interpolation
  - position, rotation, and scale interpolation.
- The target of a path interpolator object is a TransformGroup object which changes the position, orientation and/or scale of its children

Path Interpolator Constructors

```
PositionPathInterpolator(Alpha alpha, TransformGroup
  target, Transform3D axis, float[] knots, Point3f[]
  positions)

RotationPathInterpolator(Alpha alpha, TransformGroup
  target, Transform3D axis, float[] knots, Quat4f[]
  quaternions)

RotPosPathInterpolator(Alpha alpha, TransformGroup
  target, Transform3D axis, float[] knots, Quat4f[]
  quaternions, Point3f[] positions)

RotPosScalePathInterpolator(Alpha alpha, TransformGroup
  target, Transform3D axis, float[] knots, Quat4f[]
  quaternions, Point3f[] positions, float[] scales)
```

Interpolating Rotations

- Order is important when dealing with two or more rotations
- One consequence of this is that rotation interpolations can look unnatural
  - Foley and van Dam example
    - A rotation of 90° about z followed by a rotation of 90° about y has the effect of rotating by 120° about (1,1,1)
    - A rotation of 30° about z followed by 30° about y has the effect of rotating by 42° about (1,0.3,1) not by 40° about (1,1,1) as we would like for a nice interpolation
- Quaternions provide a solution to this

Quaternions

- A quaternion is a quadruplet (a,b,c,d) in which the first element, a, is a scalar and the other three are treated as a vector (b,c,d)
- Quaternions have some interesting properties
  - Take the unit quaternion $(a + bi + cj + dk)$ where *1*, *i*, *j* and *k* are the basis vectors (1,0,0,0), (0,1,0,0), (0,0,1,0) and (0,0,0,1)
    Then $a^2 + b^2 + c^2 + d^2 = 1$ $\quad i^2 = j^2 = k^2 = -1$
    $ij = k = -ji \quad jk = i = -kj \quad ki = j = -ik \quad ijk = -1$

Quaternion Algebra

Quaternion addition is the same as vector addition

We define multiplication of quaternions as follows

Let $p = (p_1, p_2, p_3, p_4)$ and $q = (q_1, q_2, q_3, q_4)$

Then $p*q = (p_1q_1 - p_2q_2 - p_3q_3 - p_4q_4,$
$p_1q_2 + p_2q_1 + p_3q_4 - p_4q_3,$
$p_1q_3 - p_2q_4 + p_3q_1 + p_4q_2,$
$p_1q_4 + p_2q_3 - p_3q_2 + p_4q_1)$

Note that $p*q =/= q*p$

Quaternion Rotations

A rotation of *ø* about a unit vector *(b,c,d)* can be represented by the quaternion

$$\cos ø/2 + b \sin ø/2\, i + c \sin ø/2\, j + d \sin ø/2\, k$$

Successive rotations correspond to multiplication of quaternions

Note that the order of the multiplications is important due to the non-commutativity of quaternion multiplication

Quaternion Problems

- Every rotation can be represented by 2 quaternions
  - Rotating by an angle *ø* about an axis *v* is the same as rotating about *-v* by *-ø*
  - In interpolating from one quaternion to another we select the shortest route
- Rotations of whole circles
  - Rotating by 360° in an animation is not the same as rotating by 0° but their quaternions are the same
- Quaternion interpolation is isotropic
  - Bad for camera rotations - tilting all over the place

Colour Interpolators

Colour interpolators modify the diffuse colour of target material objects by linearly interpolating between a pair of specified colours

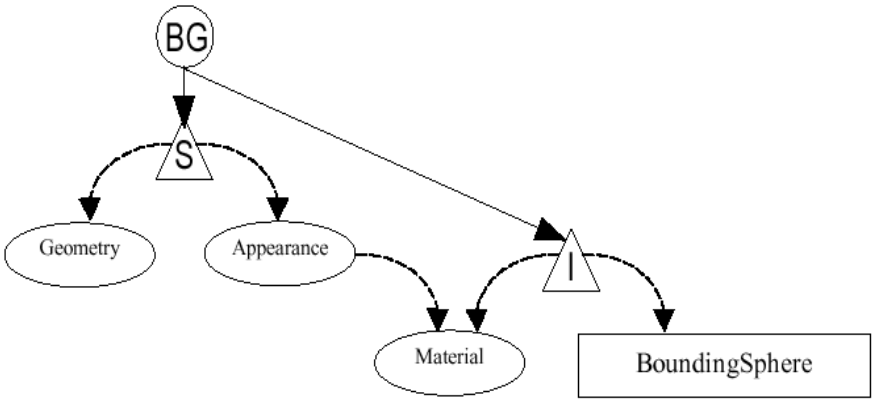
`ColorInterpolator(Alpha alpha, Material target)`

Constructs a default colour interpolator with a specified target, a starting colour of black, an ending colour of white

`ColorInterpolator(Alpha alpha, Material target, Color3f startColor,Color3f endColor)`

Constructs a colour interpolator with the specified target, starting colour and ending colour

Linking a Colour Interpolator



Morphs

- Morphs permit the geometry of objects to be changed in animations
- Morphs are not interpolators, nor even extensions of the `Behavior` class
  - The `Morph` class extends `Node`
- `Morph` manages transitions between frames
  - It does not do any animation itself
  - `MorphBehaviors` do that
- `MorphBehaviors` are added to Morphs

Using Morphs

- Each key frame has an object defined by a geometry array associated with it
- The number of vertices in the arrays must be the same for all frames because the vertices are what will be manipulated during morphing
- An array of weights must be provided which is used to form a weighted average of all of the geometry arrays (1 weight/array)

Morph Weights

- The weights in a `Morph` must sum to 1.0
- An interpolator varies the weights according to alpha during the morphing process
- The result is that different versions of the geometry of the object being morphed become prominent at different times
- In between the key frames hybrids of the geometry are produced

Morph Example

```
// Create 4 geometry arrays of the same size
GeometryArray[] geomArray = newGeometryArray[4];
// Fill these 4 geometry arrays with data
// One for each of 4 key frames
…
// Create a morph with all of the arrays
Morph flowerMorph = new Morph(geomArray);
// Make sure we can change the weights
flowerMorph.setCapability(Morph.ALLOW_WEIGHTS_WRITE);
// Create at alpha
Alpha alpha = new Alpha(-1,5000);
MorphBehavior flowerBehav =
      new MorphBehavior(flowerMorph, Alpha, 4);
flowerBehav.setSchedulingBounds(new BoundingSphere());
scene.addChild(flowerMorph); scene.addChild(flowerBehav);
```