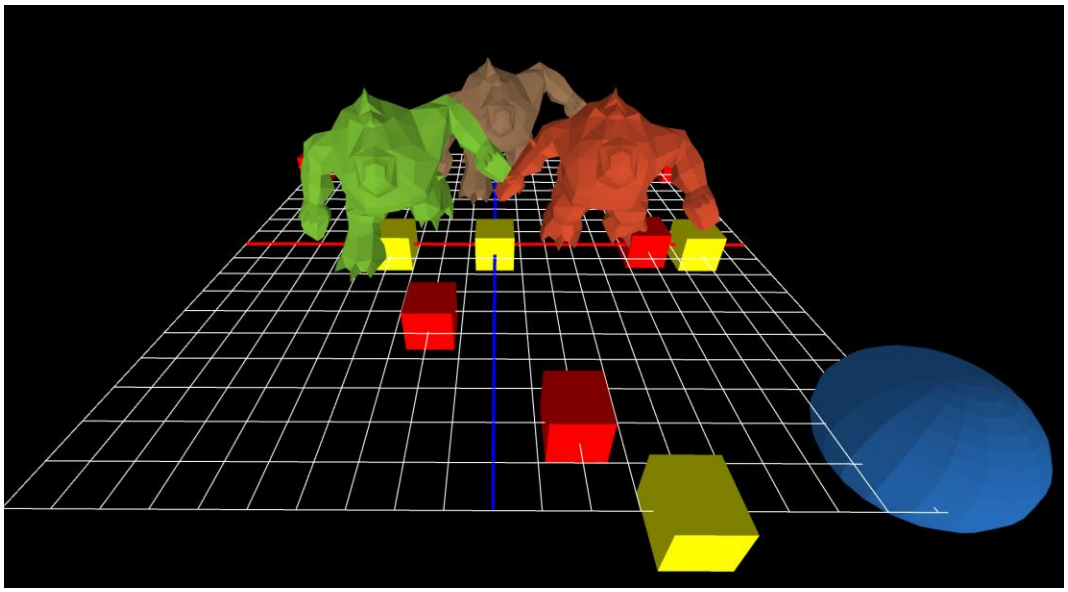
Project 6 – Animation

Overview

In this assignment you are going to load and process a few animations that all revolve around the same core technique: linear interpolation. You will load some animations from external files, create some simple animations yourself in code, and then update/draw all of them in a single scene.



Description

The first thing you are going to animation is the translation of an object.

Animation is typically set up as a series of **Key Frames** that contain the data being animated, or changed over time. What's IN a key frame? The answer to that, like many things, is "it depends". Generally speaking, though, a key frame would have 2 things:

Time – When does this key frame happen?
Data – What information is being stored at this point in time? This data could be a lot of different things, but generally the type of data we associated with rendering:

Transformation values:
x/y/z coordinates
rotation values (rotation around x/y/z axes)
scale data (some amount of scale along the, you guessed it, x/y/z axes)

Color – r/g/b/a values

There could also be values associated with some AMOUNT of something

- how much of some color to use (0-100% perhaps)

- the intensity of a reflection of light (0-100% most likely)
- the angle of something relative to something else

And really any other sort of values that you want to change over a period of time. For this assignment we are going to focus only position data—x/y/z coordinates.

Creating a class to store such data would be trivial:

```
class KeyFrame
{
   float time; // Non-optional component of a key frame

   /*customized for whatever you want*/
   float x, y, z; // Position, the most commonly used data
   float rotationX, rotationY, rotationZ;
   float scaleX, scaleY, scaleZ;
   float r, g, b, a;
   // And anything else you might want or need…
};
```

A class for animation, then, might be little more than a collection of those key frames:

```
class Animation
{
   ArrayList<KeyFrame> keyFrames;
};
```

Having the animation data is a start, but you would need some other object to USE that data in some way. This type of object is common called an **animator**, or an **interpolator**.

```
class Interpolator
{
   Animation animationData;
   /* and functionality to update/interpolate the data */
};
```

The purpose of this object is to translate the data stored in key frames and, based on some time value (typically some ratio of 0-1.0, indicating how far along the animation has progressed) and calculate some new data to use in the current frame. The relationship between those objects might look like this:

Animation

```
class Animation
{
   KeyFrame data
}
```

Animator / Interpolator

```
class Interpolator
{
   float currentTime
   Animation data
}
```

Object to use that data (what we really care about)

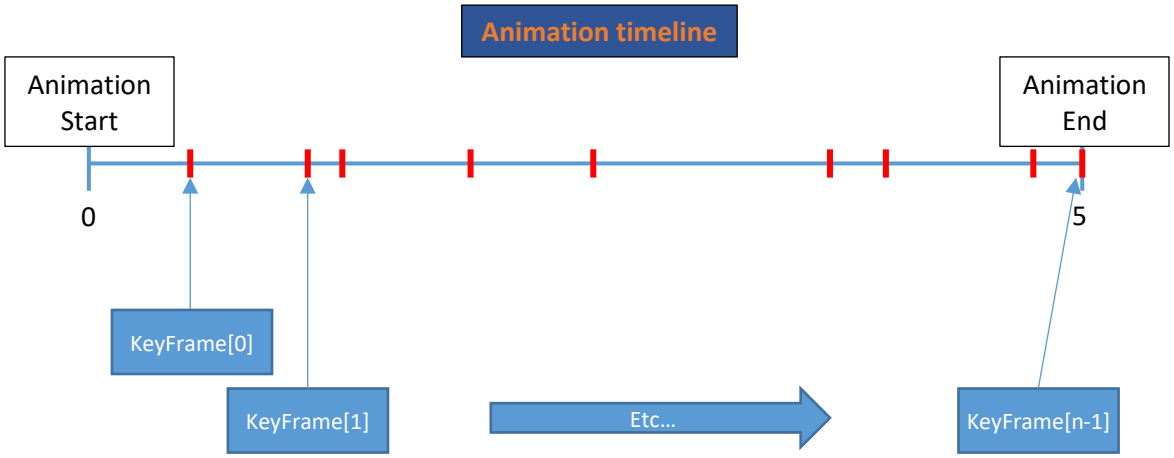
```
class SomeObject
{
   Interpolator data
   Update()
   { // Change data based
     // on time, key frames, etc
   }

   Draw()
   { // Draw based on data}
}
```

Timelines

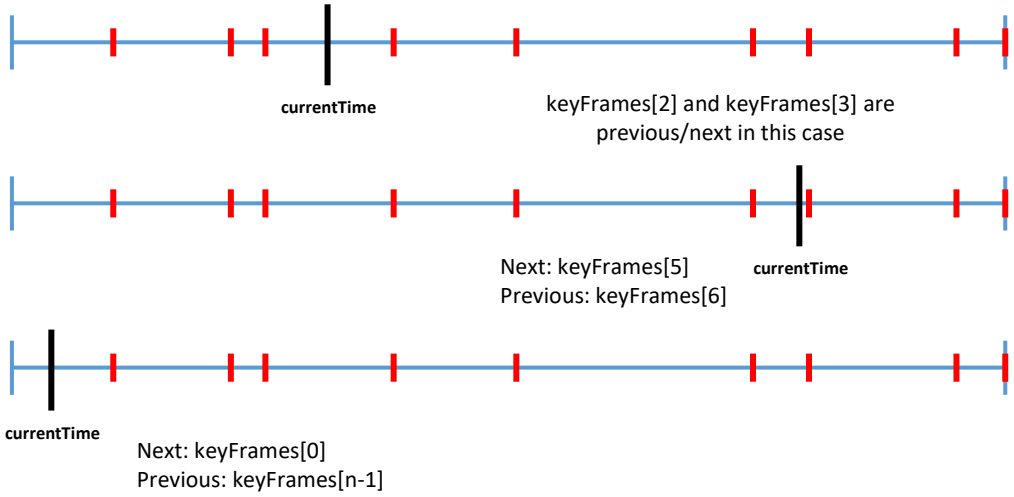
Animations play out over time. It's important to be able to get a sense of how all of that information is structured, and for that we use a **timeline**.

As an example, let's look at an animation that is 5 seconds in duration, and has 9 key frames. The timeline might look something like this:



As an animation plays, you will always have a **current time**. Whether the animation was 2 seconds, 0.2 seconds, or 2 minutes in duration, an animation starts, **current time** increments as the program updates, and that current time variable is used as the starting point for determining what values should be used on that particular frame of the application.

Examples of Current Time in an Animation

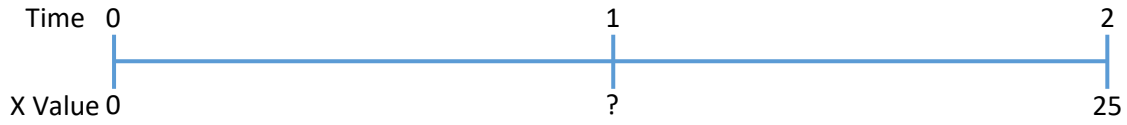


Consider this example:

You have an object at the origin (0, 0, 0). Over 2 seconds, you want that object to move 25 units along the X-axis to a final position of (25, 0, 0). The key frames for that scenario might look like this:

KeyFrame[0]	KeyFrame[1]
Time: 0 Position: 0, 0, 0	Time: 2 Position: 25, 0, 0

Plotting those numbers out might give us something like this (Y and Z omitted for this example):



Assuming a constant rate of change, what would the X coordinate be at the 1 second mark? This is an easy example, but the formula for determining that value can be applied to any sort of scenario. This process is called **linear interpolation**.

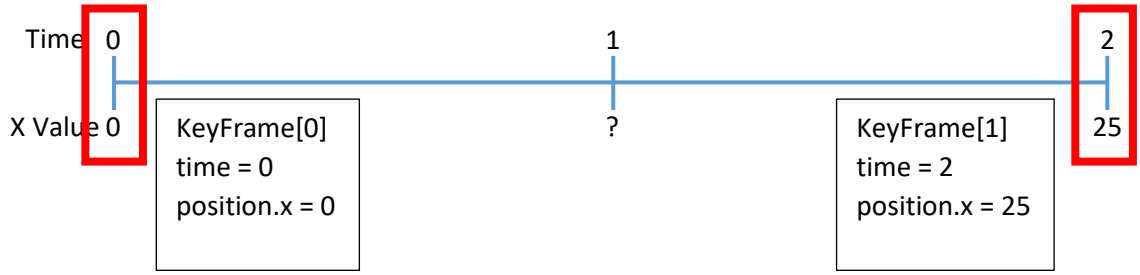
Linear Interpolation

The basic idea behind this process is that you have 3 pieces of information, which can be used to determine a 4th. You need:

Start Value: (X value of 0, in the previous example)
End Value: (X: 25)
Ratio: How much we want to change from the start value (typically a floating point value from 0-1)

In the case of animation, often (not always!) the start/end values are set up in advance—loaded from a file, created during load time, etc. and then stored in one or more key frames.

So in this example, the start and end values are just two key frames:



Time Ratio

Calculating a ratio for time involves using 3 pieces of information (this sounds familiar…) to calculate a 4th. You will need:

A start value (the time of the previous key frame)

At any point in this animation, the **current time** will be between two key frames, and only two. One of these key frames will be the **next** key frame, and another will be the **previous** frame.

Those two frames are what you will use to determine the data of the current rendered frame (i.e. what data will you use to draw something to the screen).

The algorithm

1. Find out which two key frames you're dealing with. What's the next key frame? What's the previous key frame?
2. Determine the ratio of current time to next/previous times. This ratio will ALWAYS be between 0 and 1.0f. If you ever get a ratio that is beyond this range, something is wrong (and you will definitely be able to tell by looking at your model)
3. Using that ratio, use linear interpolation to determine a current set of data from the data contained in the two key frames.

Things to worry about

You might notice that in previous examples, there was no key frame at time 0. This will be significant in this assignment (it creates an **edge case** you have to specifically code for). When your animation is just starting (or at the beginning of a new loop of the animation), the "next" frame will be frame[0], while the "previous" frame will be the last key frame. You will want to use the DATA from the previous key frame, but the time you want to use for that frame is just 0, as the animation is starting over.

Manually creating an animation

For parts of this project, you are going to create an animation in code. You will have to:

1. Create an Animation object to store the key frames
2. Create the key frames, setting their time and data values
3. Once finished, store this Animation in an Interpolator object
4. Use that Interpolator the same way you would as if the animation had been loaded from a file (it's just data, after all—the interpolator doesn't care where you got it from!)

Important Classes and Variables

Animation, **KeyFrame** – The main storage containers for animation data
Interpolator – An abstract base class for creating new interpolators
ShapeInterpolator – A class for implementing interpolation and creation of an entire PShape every frame
PositionInterpolator – Very similar to its ShapeInterpolator sibling, this will store only a single PVector as the result of an animation

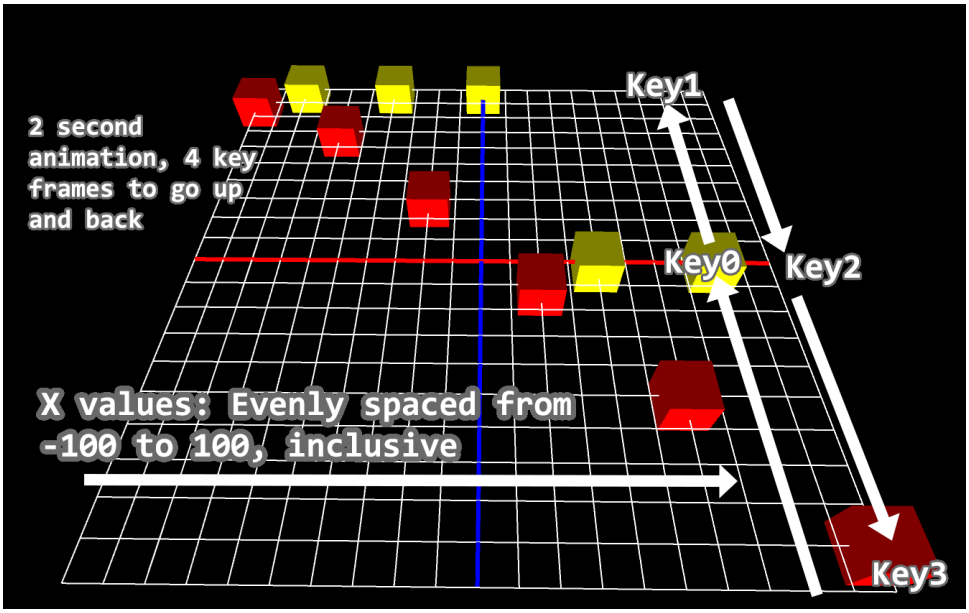
playbackSpeed – Control the rate of all animating objects by changing this variable in draw().

Tasks

So that's a lot of background information, what do you need to actually DO in this assignment?

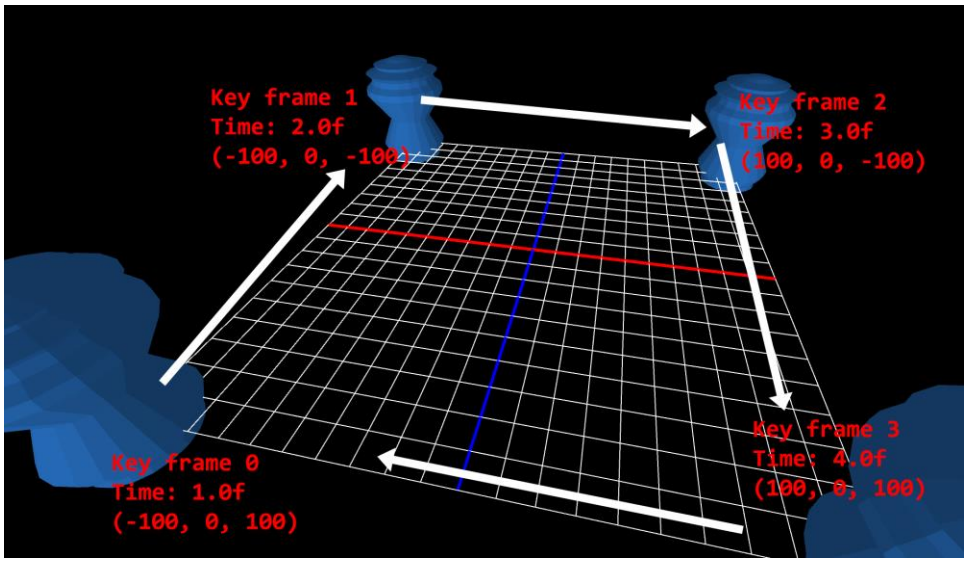
Note: The grid shown in the example runs from -100 to +100 in both the X and Z axes.

1. Create 11 animations to move from end to end of the grid:



Every other animation is going to be set to snap to key frames instead of interpolating between them.

2. Create an animation around the grid, to move the sphere-like object shown below



3. Load in the shape of the spheroid, from the file "sphere.txt". Interpolate between its frames, and draw it at the location of the previously-created animation.

4. Load in the shape of the monster, from the file “monster.txt”. Create 3 instances of an interpolator: One playing forward, one playing backward, and one playing forward but snapping to key frames instead of using linear interpolation.

Animation Files

The data you will be loading from a file is in a fairly simple format. The files are in plain text (which is generally sub-optimal for size/performance reasons, but works fine for this project) and has the following structure:

The number of key frames in the animation
The number of data points (vertices) for each frame in the animation

For each key frame:

A time – When does this frame occur?
A number of lines equal to (NumberOfDataPoints) from earlier in file – each line contains an X/Y/Z value, the position of a vertex in the mesh

To load these files you can use the class BufferedReader, which is a Processing class.

```
24
2262
0.416667
22.583 -11.2495 16.151
21.0444 -14.7182 14.4866
21.9732 -12.3142 13.4917
25.5208 -17.2282 8.76546
```

The first few lines of an animation file

These values will be stored in an Animation object, which gets returned from the ReadAnimationFromFile() function.

Camera

The example shown uses a simple orbit camera, identical to one you wrote in a previous assignment. You can reuse your own camera from that assignment, or implement another one here.

Drawing Meshes

You won’t be drawing PShape objects that are created only once; instead, you will be creating a new PShape every frame, for every object. This will be the same process as in a previous assignment. Using the createShape() function, beginShape(TRIANGLES) and endShape(), and filling in vertices using the vertex() function of the shape you created.

Rubric

Item	Description	Maximum Points
Grid	Free points! (if you already did this from a previous assignment)	10
Camera	Implement an orbit camera similar to a previous assignment (also free points if you've already done it)	10
Monster animation – plays forward	Monster animation plays forward (a positive time value sent to the Update() function)	30
Monster animation – plays backward	Monster animation plays backward (a negative time value sent to the Update() function)	30
Monster Animation – plays forward without interpolation, but snapping to key frames		30
Procedural animation – 6 moving boxes with linear interpolation		30
Procedural animation – 5 moving boxes with no interpolation, but snapping to key frames		30
Spheroid animation	Sphere animation loaded from file, plays and loops properly	15
Procedural animation – spheroid moves around grid		15
	Total	200