

Objectives:

- Create a new class intended to be based upon .mesh files with (several) associated animations.
- Enable these objects to smoothly blend between two animations (e.g. we're in a walk animation, the player hits 'C' and we smoothly transition to a crouch animation over 0.5 seconds)

Tasks:

- (4 points) Create a EngineActorSkel class.
 - It should inherit from EngineActor.
 - You'll find it useful to store a pointer to the entity's SkeletonInstance (get at this through the entity).
- (5 points) Create a nested class called **AnimationQueueEntry**, (define it within EngineActorSkel) for holding info about each (playing) animation. This should probably store, at a minimum:
 - The animation name (e.g. "Walk") which corresponds to the "action" in the .skeleton file.
 - The animationState pointer for this animation.
 - An indication of whether this animation should be looped or not.
 - A play state variable (which can store these values: **easeIn**, **playing**, **easeOut**, **stopped**).
 - A speed value (1.0 = normal, 0.5 = half-speed, 2.0 = double speed)
 - A transitionCurTime, and transitionTotalTime (floats) attribute – see step 4b and the notes.)
- (1 point) Make a vector of AnimationQueueEntry's in the EngineActorSkel class.
- Create the following methods of EngineActorSkel:
 - (7 points) **startAnimation**: Should...
 - The user should pass you any info you need to start the animation (probably the name, looping-type, speed, and a transitionTime value)
 - create a new AnimationQueueEntry object and add it to our vector.
 - This method should start to transition all playing animations to the stopped state (see Notes below)
 - Note: Use animState->setEnabled(false) for all animations, even if they are meant to be looped – we'll do this manually in our update method.
 - (12 points) **update** (note: this is a re-define of the stub function from EngineActor)
 - Cycle through all active animations, doing the following...
 - Transition animations from "ease-in" to "playing" state (if applicable)
 - Transition animations from "ease-out" to "stopped" state (if applicable)
 - If an animation ends, check the loop state. If set to true, "rewind" the time position to 0. Else, set its play state to "stopped"
 - Add time to all playing (or transitioning) animations.
 - If the animation is transition (in or out), add time to its curTransitionTime.
 - Disable and remove an "stopped" animations.
 - If the animation is transitioning in or out, adjust its weight using an "S-curve" (see the notes below)
 - (1 point) **getNumActiveAnimations**: Returns the number of active animations (i.e. the size of our vector of AnimationQueueEntry's)
 - (1 point) **getActiveAnimationNames**: Fill in a vector (passed by reference) of strings.
 - (1 point) **getActiveAnimationWeights**: Fill in a vector (passed by reference) of floats.
- (3 points) Make a new method (of the EngineLevel class) for creating and adding (to the map of EngineActor's) a new EngineActorSkel.

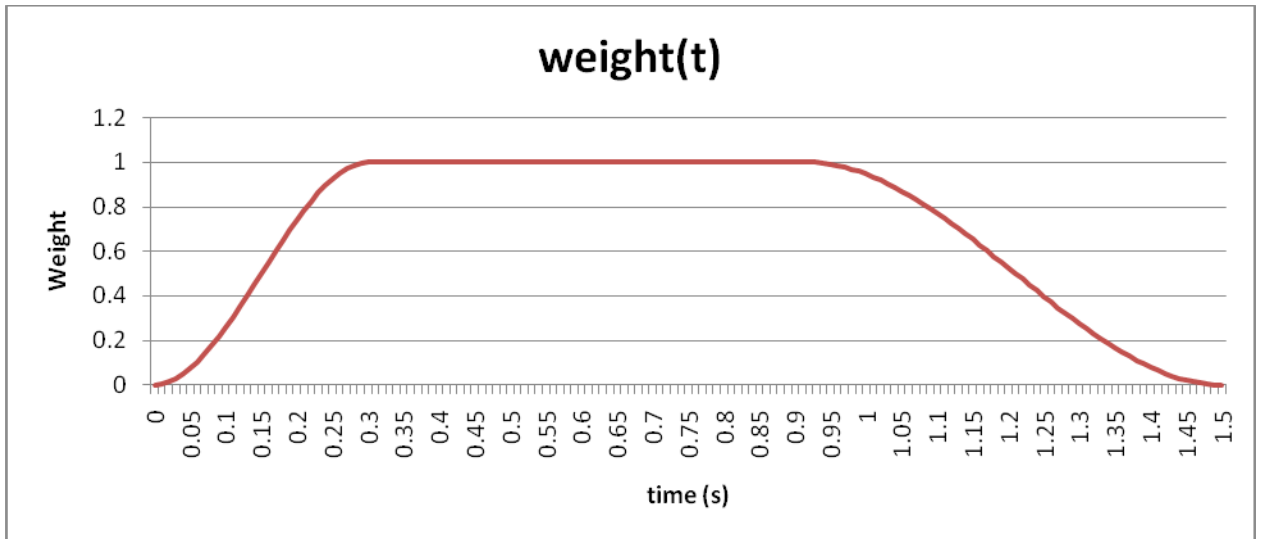
Notes:

In the update method, we want the weight (set though `animState->setWeight(x)`) of the animation to *gradually* increase as the animation is transitioning in. It should remain at 1.0 while the animation is playing. When the animation is set to the ending state, it should *gradually* decrease to 0.

Let's say we have an animation that:

- begins to transition at $t=0.0$
- reaches full weight at $t=0.3$
- starts to end at $t=0.92$
- Reaches 0 weight at $t=1.5$

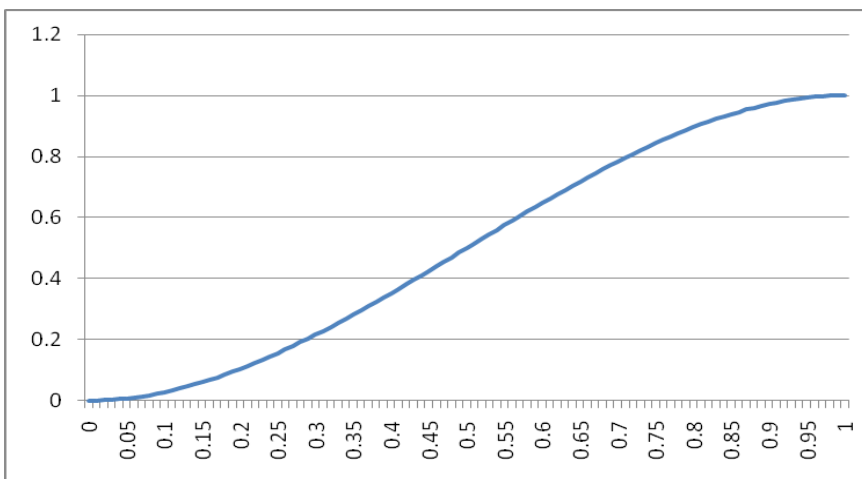
We can graph the weight of this animation like:



One way to get this relatively smooth (and cheap to compute) effect is with the “S-curve” function:

$$f(x) = 3x^2 - 2x^3$$

Where x is in the range 0-1 and $f(x)$ ranges from 0-1. We can plot this function as a graph too:



The only trick is expressing your time values as a *percentage* ($0.0 = \text{transition start}$, $1.0 = \text{transition end}$) and handling the inverted case when we're transition out.