The HmiAnimation package

The hmi/animation package is meant for 3D animation purposes.

1.1 Overview

The hmi.animation package defines classes for animation of objects and humanoids inside a virtual environment. It deals only with abstract descriptions of such environments, but not, for instance, the implementation of visualization virtual environments. As a consequence, it is necessary to combine this package either with simple 2D graphics, or with more complicated 3D graphics. The basic entity defines in this package is the VObject. ("Virtual and/or Visual Object"). Such VObjects have a defined name, they define some set of *attributes* and they define a limited set of physical attributes, like position and orientation in 3D space. Moreover, VObjects are arranged in a hierarchical scene-graph like arrangement, where VObjects are built up from smaller parts. There can be a direct *parent-child relationship* between two VObjects, or some VObject V can be considered a *part* of some other VObject P if it is *recursively* a child of P. This scene-graph takes into account the positioning and orientation (and possibly scaling) of child parts, relative to the position and orientation of their parent VObject. The relative position, called the *translation*, the relative *orientation*, and the relative *scaling* together define the *local transform* of a VObject.

See the javadoc.

1.2 XML encoding of VObjects

1.3 Interpolators

Animation, Physic simulation, and 3D rendering

2.1 Scenegraphs and Physics

Animation is usually based upon VJoints, arranged into tree shaped scenegraphs. This provided the interface between, on the one hand, rendering systems like those from the hmi/graphics packages and, on the other hand, animation systems based upon interpolators, procedural animation, or physics based simulation.

Our animation structure is such that it supports splitting the rendering and animating processes into two separate threads. To allow this (and in the future possibly to allow multiple remote visualizations of the animation process running on a server), we separate the rendering and animation VJoint scene graphs (see Figure 2.1). The local transform of the nodes under animation root is coupled to the render scene at its initialization. This coupling is many-to-one: one or more animation VJoints can steer a single render node. For example: a render node containing a deformable mesh for a humanoid is steered by a tree of animation VJoints representing the skeleton that steers the mesh. The animation tree is coupled to the render tree using the VJoint rotation/translation buffers: setting the rotation or translation of a animation tree VJoint directly sets the rotation in the corresponding render node. If animation and rendering occurs in different threads, access to the animation tree should be synchronized. This prevents the render thread reading partial information from the animation scene while it is still being updated by the animation thread. Typically the animation processes use copies of parts of the animation tree locally in time consuming animation processes and then uses a synchronized copy action to copy the resulting VJoint transformations to the animation scene. The render thread uses a synchronized action to calculate the global render node transformations from rotations in the animation tree (see also Figure 2.2).

For a typical linking scenario see this scenegraph linking example

2.1.1 Example: animation algorithm in Elckerlyc

The AnimationPlayerManager manages the animation proces in Elckerlyc. It contains an AnimationPlayer for each animated virtual human. It uses 3 copies of the joint tree of this humanoid: cur, prev, next, representing the joint rotations at t, $t-1$ and $t+h$ respectively. vjAni represents the root VJoint of the human in the animation joint tree. Animation is executed as follows:

1. run all animation players, in each animation player:
   (a) copy cur to prev, next to cur
   (b) run all procedural animations on next
   (c) using prev, next and cur, calculate and apply the forces and torques generated by the kinematic motion to the human's physical body
2. do a physics step
3. copy the physical body rotations to cur
4. copy the rotations of curr and its children to vjAni (and its children)

2.1.2 Synchronisation

Acces to the VJoints in the Animation root should be synchronized to the animation sync:

```
synchronized(AnimationSync.getSync())
{
do something with VJoints in animation root
}
```

Access to objects controlled by physical simulation should be synchronized to the physics sync:

```
synchronized(PhysicsSync.getSync())
{
access physical objects
}
```

2.2 Procedural Animation

2.2.1 XML format

...
Note that some symbols need to be XML encoded: > as > , < as < , && as && etc.

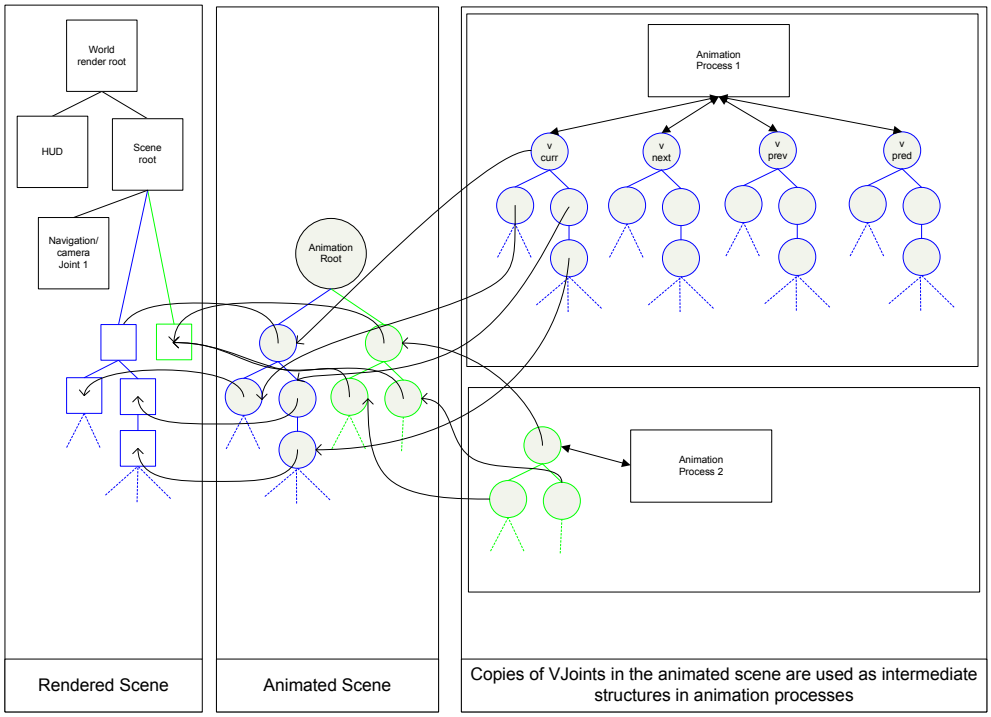

Figure 2.1: Typical scenegraph layout

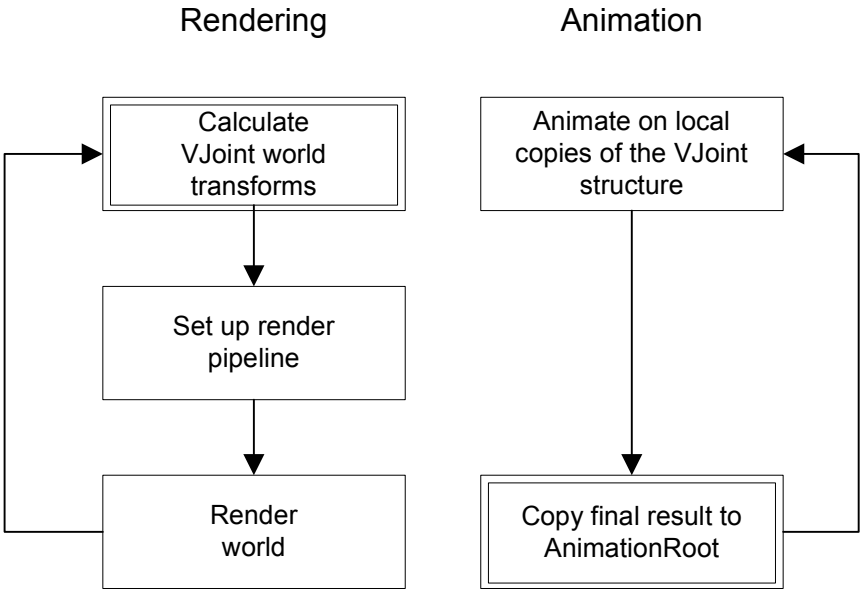

Figure 2.2: Animation and rendering loops running in different threads. The processes with double lines are synchronized to each other.

JEP examples and custom macros

Function of time ($0 \leq t \leq 1$) and float parameters (as constructed using <Parameter>).

If-statement

`if(<condition>,<expression_true>,<expression_false>)`

Evaluates the condition and executes 'expression_true' if its true, expression_false otherwise.

Random

`rand()`

Generates a random number between 0 and 1.

Hermite-spline

`hermite(time,v0,vn, p0,...,pn)`

Constructs a uniform Hermite spline, time distance between the points is 1.
time: time to execute the spline at, typically t*n to interpolate smoothly from p0 to pn as t ranges from 0 to 1.

v0: speed at p0
vn: speed at pn
pi: position at i

TCB-spline

`tcbspline(time,v0,vn, p0,t0, pi,ti,tensi,conti,biasi,...,pn,tn)`

Constructs a non-uniform TCB spline.
time:time to execute the spline at $t0 \leq time \leq tn$
v0: start speed
vn: end speed
pi: point i
ti: time stamp i, $ti - 1 < ti < ti + 1$
tensi: tensity i, $-1 < tensity < 1$, default $= 0$
conti: continuity i, $-1 < conti < 1$, default $= 0$
bias: bias i, $-1 < bias < 1$, default $= 0$