AniTMT Hackers guide I – Animation System

Contents

1 Introduction

This Program is still in an early stage but we are pleased about any kind of feetback. We are very sorry about the few comments in the source yet.

For a reference to code files we only use the filename of the header file (`.hpp`) even if we also talk about the implementation (`.cpp`).

2 Program structure

The Mother of the object hierarchy is an object of the class Animation `(anitmt.hpp)`. It contains an object for the global options, a list of script objects, a list of scenes and a list of files that only have to be copied. The global options my be defined as command line parameter or within ini files `(globals.hpp)`. The virtual base class for script objects is declared in `script.hpp` and the derived class for adl scripts – the only animation description language already implemented – is situated in `adl.hpp`. The scene objects `(scene.hpp)` may contain a list of comonents that may contain a list of functions that may contain a list of subfunctions `(ani_base.hpp)`. All these objects are derived from a class called `Linked_Valtree` that is again derived from Valtree `(valtree.hpp)`. The `Valtree` class provides this tree behaviour where each node has a list of child `Valtrees` and a list of properties that have a name a general value type for scalars, vectors or strings `(vals.hpp)`. With the aid of a list of property solvers each Valtree provides the method `solve_property(name)` to solve for one property. Each `Linked_Valtree` node within this object hierarchy additionally knows his previous and next defined node. This is used by functions that solve properties with the help of neighbour `Linked_Valtrees` (see chapter 4).

All functions with its subfunctions are derived from the classes defined in `ani_base.cpp`. One example is the move function for objects `(object.hpp)`. The general subfuntion

class Subfunc_Move_object provides the functions to make it possible to solve for the properties that are common for all track elements. In order to calculate the resulting position it declares some pure virtual functions that have to be implemented by each specialized track like a straight element (straight.hpp) or a circle (circle.hpp). These files are very important to improve AniTMT. Each child type has to be inserted in one function of the parent. To add a new function you have to insert it into Component::get_function() in ani_base.cpp. If you want to add a new track type for the move function you have to add it in Move_Object::get_subfunction() in object.cpp. Then you may copy the source for the straight track element and modify it. If you would like to add property solver for a continuously accelerated system you may use the add_accelerated_solver() function.

Finally the results of each Component has to be written into files. This is scene type and component type dependant. That is why each scene provides scene adapters for each component type (scalar, vector or object). These adapters provide component type specific interfaces to set the result and scene specific parts that detect components in the scene description files and write the result in the correct file.

3 Sequence of Execution

The program starts in main.cpp. It only creates an object of the main Animation class (anitmt.hpp) and calls the start method with the command line parameters as argument. This function orders to interprete the arguments, to read the files, to solve the animation, to write down the results and to copy the needed files.

At first all scripts and non moving scenes defined by command line parameters or ini files are added. Afterwards the program reads the script files and adds more scenes with the whole object tree of included components, functions and subfunctions. Then there go some function calls through the whole object tree to solve for undefined properties (see chapter 4). Another function call follows for each frame to calculate the non-moving scenes and write them to files. Afterwards all other files needed for the rendering process are copied to the animation directory.

4 The AniTMT Solve System

Valtree is the base class of all elements in the main object hierarchy. It provides a mechanism to solve for any property of an element. Therefore property solver may be added that know all the possibilities to solve for a specific property. The solvers are stored in a std::multimap called Valtree::property_solve. During the solve process a map of search_infos stores which properties are locked because they are already in the search path and wheather the property has to try to solve itself again. Therefore it stores an id that is increased everytime a property is solved. If the id didn't change since the last trial a property doesn't try to solve itself again. You can find some simple property solvers in scalar.cpp. They all start with SCA_Solve_\dots.

In the object tree of the scenes every element is even derived from `Linked_Valtree`. This class provides a solve system that isn't limited to one element. This is for example very useful for animation functions with concatenated tracks. Two atteched track elements should have the same time and position between them. Moreover it could be useful to have the same direction or even speed. The `Linked_Valtree` provides therefor a solve system that works on different priorities. On each priority level an element may pass some properties to it's neighbours which will take it if they couldn't solve for the corresponding property. The `set_priority()` function increases the number of priority levels if nessessary. Each element that wants to take part in this communication has to overload the virtual function `send_command()`. You can find a interesting one in `object.cpp` but the basics may also be seen in `scalar.cpp`.

Additionally `Linked_Valtree`s provide functions to set properties by default. This is also done in different levels. To make use of this an element needs an implementation of `allow_default()`. The most intresting one should also be in `object.cpp`.

The function that controles all these priority leves is called `Animation::solve()` and resides in `anitmt.cpp`.

Finally the funtion `Animtion::calc()` in `anitmt.cpp` starts the execution of `calc_res()` of all elements that has to calculate the final positions and values for each frame. In `scalar.cpp` you can find an implementation of `Linear_Change_Scalar::calc_res()` that calculates a linear interpolation. In `object.cpp` is a very general implementation `Subfunc_Move_Object::calc_res()` that provides a track independant calculation of accelerated movement and rotation so that all concrete track types only have to implement some special functions to get the real position and directions. There is a short implementation of the five functions `get_pos()`, `get_front()`, `get_up()`, `get_end_up()` and `get_first_up` (default up) in `staight.cpp`.

5 File Reference

The following tables give a brief information about all Objects of `anitmt-calc`. The corresponding file with the `.hpp` extension is the declaration and the file with the `.cpp` extension is implementation.

Files of AniTMT:

Object without extension	Description
`main`	`main()` function that only calls `Animation::start()`
`anitmt`	Main object `Animation` with initial calls
`globals`	Class for global animation options defined by the user
`copy`	Class for files that only have to be copied
`script`	Base class that handles animation script files
`adl`	Derived Class that handles `adl` scripts
`valtree`	This class is used for nodes in a object hierarchy where every node owns a list of child nodes and a list of properties
`scene`	Base class that handles source scenes; all scenes are the root of a `valtree` object hierarchy
`povscene`	Derived Class that handles POV-Ray scenes
`ani_base`	Basic object hierarchy within a scene (`Component`, `Function`, `Subfunction`)
`scalar`	Scalar functions (`change { linear and accelerated }`)
`object`	Object functions (only `move` at the moment) and the basic subfunction that provides track independant accelerated movement and rotation
`straight`	Subfunction for `move` for straight track elements
`circle`	Circular track elements for `move`
`accel`	This is very useful for subfunctions to provide accelerated systems like: `length`, `startspeed`, `endspeed`, `acceleration` and `duration`
`utils`	Templates to delete all elements of lists or maps
`p_aninfo`	Parser info to provide access from expressions to values of the internal animation structure like references to other expressions

Independant Utilities of AniTMT with an expession Parser:

Object without extension	Description
`error`	My basic error class with or without position information
`vals`	Basetype values::Valtype that can be one of scalar, vector or string and may be calculated with all common operators.
`mystream`	Character and word streams (Tokenstreams)
`parser`	Provides a Parser that can read any expressions with scalars, vectors and strings in any word stream
`parsinfo`	Some basic parser infos that may be inserted in parser objects to provide constants or functions like sin, etc...
`files`	File classes for word streams and precompiled files with `#include` directives that may be changed while copying
`constant`	Konstants like *pi* and *e*
`vector3`	Vector library of Roberto Javier Peon (LGPL)
`vector`	Puts the vector library into namespace `vect`
`matrix`	Basic matrix library especially for rotations
`ini`	Read and write functions for ini files with sections and options