Animator and Animation Generator User's Guide

Matthew T. Miguel

Jason Tsao

Vikram Balakrishnan

Contents

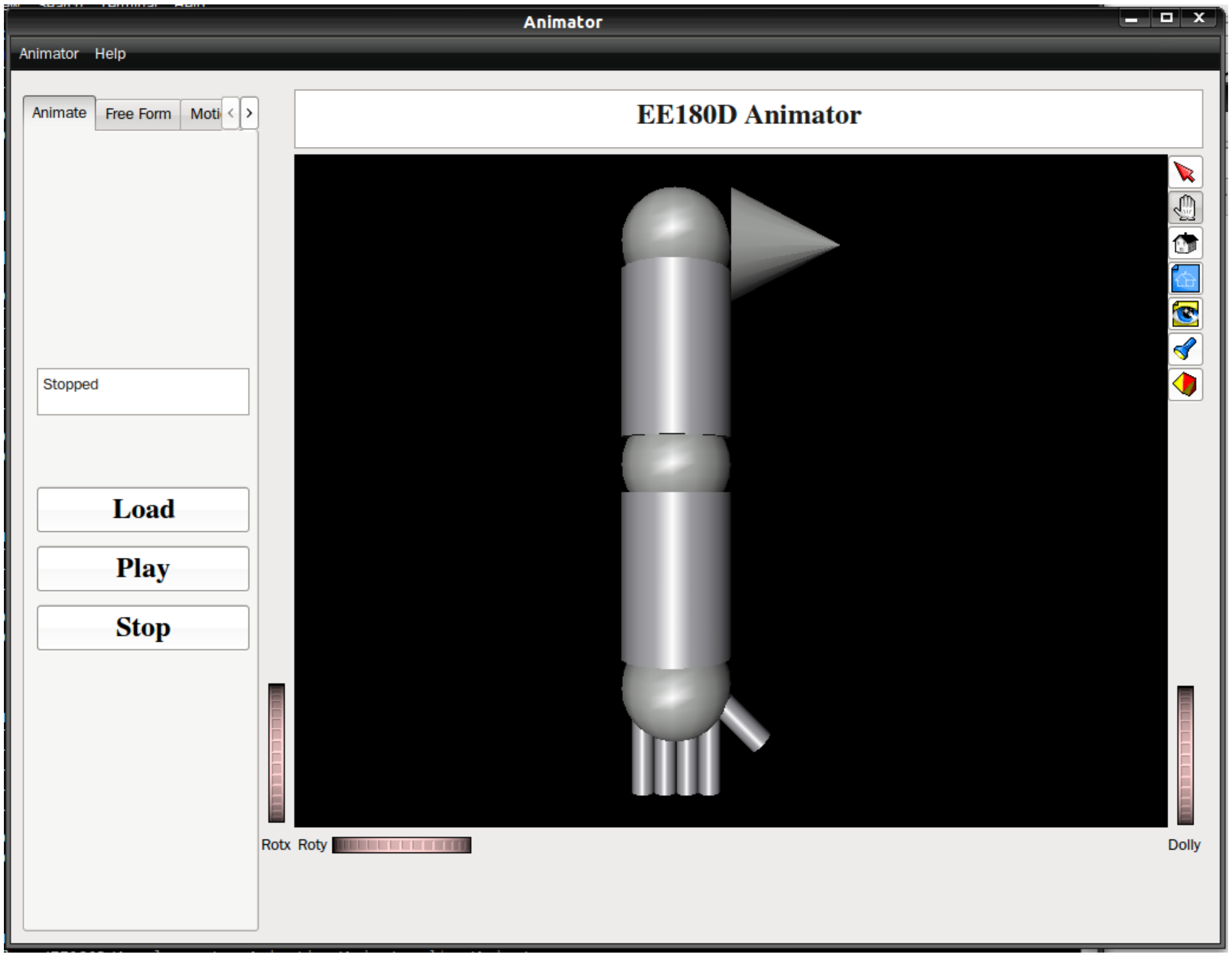
Overall Process

The Animator and the Animation Generator are designed to work with the classification toolbox provided for EE180D.

Overall Procedure with current toolbox:

1) Accelerometer data is collected
2) Data is merged and possibly labeled using the Classifier Toolbox
3) Window Size and Increment are set in the Classifier Toolbox
4) Merged/Labeled data is tested using the Testing Window of the Classifier Toolbox
5) A classification results file is exported from the Testing Window after testing is complete
6) The Animation Generator takes merged (optionally labeled) data and the classification export file to perform statistical calculations defined in the listStatistics.m file of the Animation Generator. The results are stored in a statistics file.
7) The motions are modeled in the Animator using the FreeForm and MotionForm tabs. Once modeled, the calculation equations are exported. These equations can be used to create a Motion Definition file that the Animation Generator can use.
8) The Animation Generator takes the statistics file and a Motion Definition file and produces an animation file which contains the instructions that the Animator needs to animate the 3D model.
9) The Animator takes the animation file and plays the animation

Animator



Introducing the Libraries

The Animator is written in C++ using the Coin3D and Qt software development libraries.

Coin3D is described as an "OpenGL based, retained mode 3D graphics rendering library." Coin3D is used to generate and render 3-dimensional objects in the Animator. Although Coin3D is great for creating 3D animations, it does not include any user interface functionality at all, and is meant to work with other software that can provide such capabilities. This is where Qt comes in.

Qt is described as a "cross-platform application and UI framework." Qt handles all user interface within our Animator and practically runs the entire application apart from the 3D rendering and animation.

In order for Coin3D and Qt to work together, a GUI binding glue library is required to interface between the two. The inventors of Coin3D have developed several such glue libraries, in particular, one called SoQt. SoQt is designed to bridge all gaps between Qt and Coin3D and allows for smooth integration of the two libraries.

Installing the Libraries

For easy development, we recommend working in a Linux environment. Our Animator was created in Ubuntu, and this is our first choice due to the ease at which the development environment can be set up. For determined Windows or Mac users, Qt and Coin3D are both designed to be cross-platform and can work on those operating systems, but it is more difficult to get all of the libraries installed correctly. If you must use Windows or Mac, and are not willing to partition your hard disk for Ubuntu, we recommend using a virtual machine to emulate Ubuntu, and to program and develop in that virtual machine. A nice free virtual machine is available through VMWare called VMWare Player. To get started on Ubuntu with VMWare player, simply download the Ubuntu disk image from the Ubuntu home page, and create a new virtual machine with VMWare player. For the rest of this document, we will assume that programming is being performed in a Linux environment.

To install all necessary libraries in Ubuntu, open a terminal and type:

```
sudo apt-get install libqt4-core libqt4-dev libcoin60 libcoin60-dev libsoqt4-20 libsoqt4-dev build-essential python git
```

That's it! The libraries are installed and ready to use (pat yourself on the back for *choosing* to use Ubuntu).

Obtaining the Animator Source Code

The next course of action is to download the Animator source code. The source code is available online at GitHub:

https://github.com/jtsao22/Accelerometer-Animation

If you use git, you can clone from this repository. If not, you can download in .zip or .tar.gz formats from the github page for our project.

The Code So Far

The relevant source files for the current Animator are located in the Animator folder of the overall project folder.

Relevant Files:

- main.cpp
  - Contains the main function, sets up the libraries, instantiates important classes, defines some GUI features, and starts the main loop
- QCoin.h/QCoin.cpp
  - General purpose Qt widget designed to encapsulate the SoQt/Coin3D functionality into a standard Qt widget
- Animator.h/Animator.cpp
  - A QCoin subclass that implements the 3D model, and controls and interacts with other major widgets
- AbstractWindow.h/AbstractWindow.cpp

    - An abstract base class for classes that represent different usages of the Animator. These different usages are implemented in the form of a tabbed system. Switching from one usage to another is a very stream-lined process, as is the creation of new usages.
- AnimateForm.h/AnimateForm.cpp
    - A subclass of AbstractWindow that is responsible for reading animation files and playing them.
- FreeForm.h/FreeForm.cpp
    - A subclass of AbstractWindow that allows the user to manipulate parameters of the model (such as angles) using a simple array of sliders.
- MotionForm.h/MotionForm.cpp
    - A subclass of AbstractWindow that allows the user to specify simple oscillatory motions by filling in the values of a table. The motions can be played and the motional information can be saved into an equation format, which can be helpful in constructing Motion Definition files in the Animation Generator.
- ColorForm.h/ColorForm.cpp
    - A subclass of AbstractWindow that allows the user to change the color/material properties of different parts of the 3D model. Users can experiment with different settings and save and load different color configurations.

Compiling and Running

To compile the source (in Ubuntu), simply navigate to the Animator folder in the terminal and type:

```
make
```

After compilation, the Animator can be run by typing

```
./Animator
```

Note: If you are adding new files to the project, you will need to update the Makefile. See the Compiling with Qt section for more information.

The 3D Model

The current 3D Model implemented in our Animator is just a human arm. The model is defined in Animator.cpp, in the function **createSceneGraph()**. To understand this code, one must first understand the basic Coin3D API. See the Further References section for a link to the Inventor Mentor, a clear, extremely helpful tutorial for the Coin/Inventor family of APIs.

Very Brief Coin3D Explanation

After reading the basic sections of the Inventor Mentor, you will understand that in Coin3D, models are represented by a directed graph of what are called scene objects. This graph is traversed in a particular order based on how the scene objects are inserted into the graph. Each scene object modifies the rendering process in some way, be it applying a translation or rotation to the "cursor", defining a camera angle, changing the colors of rendered objects, or actually rendering objects. Our createSceneGraph function defines the scene graph that represents the 3D model we want to animate.

Currently, this model is just an arm, but it is not particularly difficult to build on the scene graph we have defined to make an entire body.

Animation Engines

Dynamic motions in the Animator are implemented using Coin3D engines. An engine in this context is a scene object which can take in inputs, and produce outputs, both of which can be tied to the properties of other scene objects. This allows for dynamic, continuous modification of these properties. In other words, engines allow us to animate the 3D model. There are three types of engines used in our arm model at this time: the ElapsedTime engine, the Calculator engine, and the OneShot engine.

Elapsed Time Engine

As its name suggests, the ElapsedTime engine produces an output equal to the time elapsed since the Animator started. However, this is not completely true because the ElapsedTime engine has a speed property that determines the rate at which the engine output increases with respect to time (units per second). If the speed parameter is always set to 1, then the result is equal to the time in seconds since the program initialized. If set to 1/60, then the time will be the elapsed time in minutes. The output of this engine is used by other engines for time dependent behavior. For animation purposes, the speed can be dynamically modified to smoothly change frequency type motions.

Calculator Engine

The calculator engine is a general-purpose engine that can take in several inputs, mathematically manipulate them, and output the results. The calculator has an expression property, which is a string that relates the various outputs to the various inputs. In our Animator, all calculators have as input, the output of a common elapsed time engine. This input is called input a. The outputs of our calculators directly control the properties of the 3D model; specifically, the angles of the arm. By specifying different calculator expressions of time and applying these expressions to the angles of the arm, we are able to specify animation when such expressions depend on time (or more accurately, the parameter a). In addition to the input a, all of our calculators take in another input, b, which is relevant for soft transitioning (explained later). This parameter b comes from a OneShot engine, which is described next.

OneShot Engine

A OneShot engine is a simple engine that can be triggered. Once triggered, the OneShot engine will run for a specified duration and then stop. While it is running, an engine output called the ramp of the engine steadily rises from 0 to 1, reaching completion when the OneShot's duration expires. This output is useful for soft transitioning.

Soft Transitioning

Consider the mathematical equation:

$$\theta_0=\pi\cos(\pi t)$$

Suppose that θ_0 represents a particular angle of the arm. Clearly θ_0 will cyclically rise and fall from –π to +π at a frequency of 1 cycle per 2 seconds. Suppose that this occurs from time t=0 to t=5. After t=5, our subject decides to double the frequency at which he is oscillating this angle. Clearly our new expression will resemble

1=cos(2)

However, can we simply replace the old equation with the new equation and tell the Animator to immediately animate the new one. At time t=5, simple substitution shows that cos(θ_0)= -1, while cos(θ_1)= +1. If we simply substitute the new equation into the calculator, the animation will instantaneously (and awkwardly) jump from having an angle of –π to having an angle of +π. This is not acceptable for a good Animator, and our solution to this problem is soft transitioning. Instead of changing the calculator expression from θ= θ_0 to θ=θ_1, we change the calculator expression to

=1−0+()1

during a motion transition and trigger the OneShot engine (whose ramp output is connected to b in the equation). We have arbitrarily set the duration of the OneShot engine (the soft transitioning time) to be equal to 1 second by default. By using soft transitioning, different types of motion can be animated with fluid transmission from one to another.

To implement soft transitioning, the Animator class keeps certain maintenance variables that help to keep track of how expressions should be updated and when.

softExpr[NUM_ANGLES][2] is a 2-dimensional array of std::string objects which represent θ_0 and θ_1 (though not necessarily in that order). The first index in softExpr corresponds to the object to which the soft transitioning is being applied (e.g. an angle). The second index can be 0 or 1, with one element representing θ_0 and another representing θ_1. Which element represents which expression is determined by another variable, softTracker.

softTracker[NUM_ANGLES][2] is a 2-dimensional array of integers that keeps track of different soft transition parameters. The first index in softTracker corresponds to the object to which the soft transitioning is being applied (e.g. an angle). The second index refers to different types of information to keep track of. softTracker[X][0] can be 0 or 1 and specifies which element in softExpr holds the current expression (θ_1) for angle X. Typically, whenever the expression for an angle changes, this value of softTracker toggles from 0 to 1 or 1 to 0. The other array element, softTracker[X][1] can be 0, 1, or 2, and is used to ensure that soft transitioning does not occur for angles that have retained the same expression since the last soft transition. When the expression for angle X is modified, the softTracker[X][1] variable is set to 2, and softExpr[X] is updated. When soft transitioning is triggered, softTracker[X][1] decreases from 2 to 1. If soft transitioning is triggered once again without the expression for angle X being modified, then softTracker[X][1] decreases from 1 to 0, softExpr[X] is modified, so that both elements contain the same (current) expression. In addition, the calculator expression is modified to purely reflect the current expression without any reference to soft transitioning. This prevents the angle from displaying any semblance of soft transitioning until the expression is once again modified. If soft transitioning is once again triggered without modification to the expression of angle X, then nothing happens to angle X.

The GUI Side

Our Animator has been designed to include some user-friendly interfaces for manipulation of the 3D model. Several tabs are used to perform different functions on the Animator. Each such tab is derived from the AbstractWindow class and can exclusively control the Animator when switched to. In addition to tabs, there are menus (albeit somewhat limited) at the top of the GUI, and potential for other control widgets to sit at a space towards the bottom of the GUI. In order to understand how all of these widgets interact with one another and with the Coin3D renderer, you need to understand the basics of Qt. Many tutorials are available online for getting started with Qt, and as with Coin3D, there is thorough documentation at the home page for the library.

Very Brief Qt Explanation

A widget is a GUI component that can be placed on the screen and contributes to user interaction. Widgets can be buttons, sliders, tables, text boxes, tabs, … just about anything you see in a GUI. Classically, widgets interacted with one another and with other parts of the application through the use of call-backs. A call-back is a portion of code that a widget can execute when a particular event occurs. In C++, this basically means a function call. The designers of Qt decided to implement inter-widget communication in a slightly different way, using what they call signals and slots. A slot is a special function designated as such in the definition of a Qt derived object. A signal is basically just a function declaration, i.e. a name and a list of arguments. Signals have no function bodies. Using a Qt function called connect, any signal of any object may be connected to any slot of any object provided that the signal and the slot have matching argument types. Signals can be emitted by using the Qt keyword **emit**. Instead of a function body associated with the signal being executed, all slots connected to the emitted signal are executed.

Compiling with Qt

It may seem strange to you that I mentioned Qt keywords, when we have been programming in pure C++ this whole time. How is it that Qt can define new keywords? One of the unique traits of Qt is that provides special preprocessor instructions that convert special Qt-specific keywords to normal C++ code during compilation. The consequence of this is that the compilation procedure needs to follow certain guidelines for Qt to work properly. First, all source files need to be registered in a Qt project file.

This can be done by navigating to the source folder in the terminal and typing

```
qmake   -project
```

By default, this adds all *.h and *.cpp files in the folder to the project, and names the project after the folder containing the source code. The project file is a *.pro file, which can be edited like a normal text file. In order to make Qt work with Coin3D, the following line must be added to the project file.

```
LIBS +=    -lCoin    -lSoQt
```

Once this is done, you may type

```
qmake
```

to generate a Makefile. This Makefile can be used in conjunction with the make command to compile your project.

Defining Signals and Slots

If you would like to define your own signals and slots for your own class, you will need to have your class inherit from the object QObject. In addition, you will need to add the line

```
Q_OBJECT
```

in your class definition.

You may then define signals using "signals:" as if it were a member access specifier (e.g. public, private, protected). To define slots, use "public slots:" as if it were a member access specifier.

Animator Usage Tabs

The tabs currently included in our Animator are AnimateForm, FreeForm, MotionForm, and ColorForm. AnimateForm reads in animation files and plays them. FreeForm allows direct manipulation of arm angles using sliders. MotionForm allows oscillatory motions to be created by specifying different values in a table. It animates these motions, and can export the calculator expressions corresponding to these motions. ColorForm allows the user to create, save, and load different color schemes for the arm.

Each of these different usages derive from the AbstractWindow class, which defines a standard interface for managing these usages. Any subclass of AbstractWindow must define at least 5 functions.

The first two are their constructors and destructors. The constructor should simply call the AbstractWindow constructor, passing in the name of the usage as it will appear in the GUI as the string argument to the AbstractWindow constructor. A destructor should be defined, even if it does nothing.

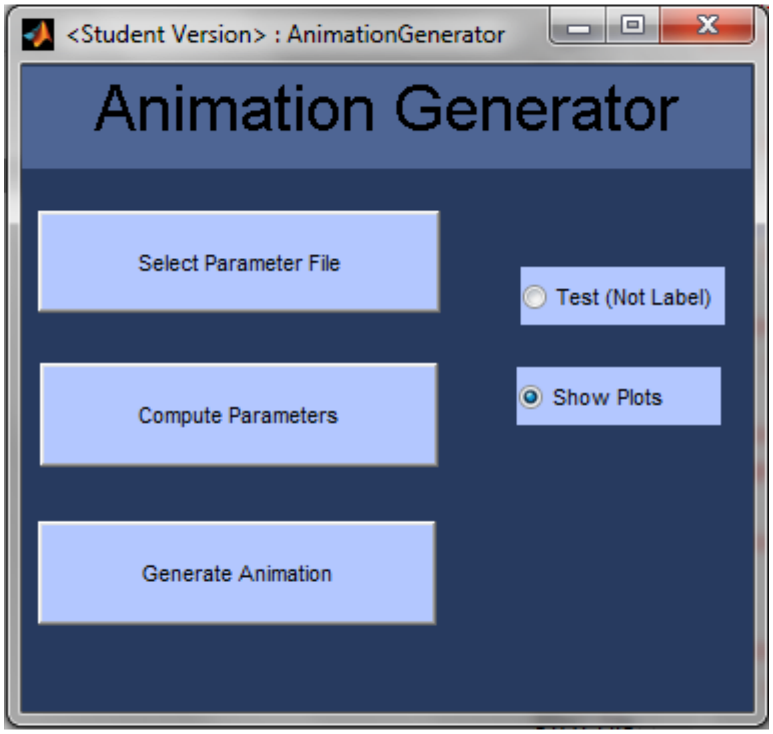
The next two functions are the transitionTo() and transitionFrom() functions. These functions are automatically called via polymorphism whenever the associated usage is switched to or switched away from in the tab list. These functions are necessary because multiple usage tabs try to control the 3D model, and only the active one should be able to at any given time. These two functions can make and break connections that would interfere with other usage tabs as needed.

The last function that needs to be defined by a usage is the createWindow() function. This function should instantiate the "window" QWidget variable and populate it with subwidgets that will appear in the tab.

Creating Additional Usage Tabs

To streamline the process of creating additional usages for the Animator, we have included a python script called tabMaker.py. If you run this script, it will prompt you with a series of questions regarding the type of usage you would like to add, generate skeleton *.cpp and *.h files for your new tab, integrate them into the Qt project, update the Makefile, and provide further instructions on how to complete the integration (adding 2 lines of code at the top of Animator.cpp). All you need to do is define member variables/functions, the transitionTo(), transitionFrom(), and createWindow() functions and you are done! Additionally, you may need to further define the destructor to delete dynamically allocated data.

Animation Generator



Basic Overview

The Animation Generator is implemented in Matlab and is designed to work directly with the EE180D Classifier Toolbox (which is also programmed in Matlab).

Relevant Files

- AnimationGenerator.m/AnimationGenerator.fig
    - Simple GUI designed to work with the other scripts in the Animation Generator
- computeParameters.m
    - Calculates statistical data (e.g. frequency) from merged and possibly labeled accelerometer data, as well as classification export data. The number and type of statistical calculations performed is defined in the listStatistics.m file.
- generateAnimation.m

    - Takes the statistical calculations from computeParameters, as well as a motion definition file (e.g. defineArmMotions.m) and produces a *.anm file that can be played with the AnimateForm tab in the Animator.
- computeStatistics.m
    - Computes all of the statistics listed in listStatistics for a single window. Each type of statistic listed in listStatistics should have a calculation algorithm defined in computeStatistics.m.
- listStatistics.m
    - Lists all types of statistics calculated in the Animation Generator as well as default values and minimum and maximum values. If a statistic is calculated beyond the specified minimum and maximum value, it is automatically set to the default value. A window with an unknown classification also has its statistics set to the default value.
- defineArmMotions.m
    - A motion definition file for use with the arm 3D model currently implemented in the animator.

Statistical Calculations

The Animation Generator takes raw accelerometer data, partitions it in time to many windows (which may overlap), and performs statistical calculations for each window. The number of windows that will exist depends on the length of the merged data file and the increment size. The increment size along with the window size are set using a parameter file (*.para). Such a file can be created using the Classifier Toolbox. Default values for window size and increment are 16 seconds and 1 second respectively. Each statistic is defined in listStatistics.m and in computeStatistics.m. The name, default value, minimum value, and maximum value for each statistic is defined in listStatistics.m. The algorithm for computing the statistic, given the window data and the classification state of the window is defined in computeStatistics.m. The script computeParameters.m coordinates the statistics calculation process, and the result of that script is a statistics file that contains all of the calculated statistics for all windows.

Classification Boundaries

One of the drawbacks of using window sizes that are larger than the increment size is that data from neighboring classes of motion can often spill into a window centered on a particular class of motion. This is particularly an issue if the window size is large. This spilling over can drastically affect the statistics calculations and give undesirable results in the final animation such as frequency spikes at class transitions. To attack this problem, we have implemented an algorithm that will automatically truncate a window from its standard size to the maximum size smaller than the standard size (in either direction) that contains points of the same classification. That is, windows will never cross classification boundaries. If the center of a window is labeled or classified with a particular class, than each point in the window that is passed to the statistics calculator will be of the same label or class. If a label file is specified, then the actual precise labels are used to implement this functionality. If no label file is passed, then the toolbox classification boundaries between windows of different classes are used instead.

Animation Files

Animation files have the extension *.anm, and contain instructions that can be interpreted by the Animator. Animation files represent the sole line of communication between the Animation Generator/Classification Toolbox and the Animator. Each line in an animation file performs a different action in the Animator, with the exception of comments and blank lines. Comments are denoted by "//" at the start of a line. Besides comments and blank lines, all other lines in an animation file perform some action in the Animator. Specifically, they modify Animator parameters. To further explain the animation file format, we should first define static and dynamic parameters.

Static Parameters

Static parameters are Animator parameters that are set through special instructions in an animation file. When a static parameter is set to a particular value, it remains at that value until it is set to another value in the animator file. Static Parameter lines do not correspond to any animation time. They are processed immediately, and the next line of the animation file is then read. Examples of static parameters include calculator engine expressions and the Animator step size. Lines that modify static parameters begin with a command string and cannot begin with numbers.

Dynamic Parameters

Dynamic parameters are Animator parameters that are expected to change frequently enough that they are assigned to each and every Animator window. Each dynamic parameter required by the Animator is listed in a predetermined order, and separated by white space on the same line. No command string identifies a dynamic parameter line; the parameters are simply listed immediately. One dynamic parameter line takes one Animation window. That is, upon reaching a dynamic parameter line, the Animator will perform the Animation using those parameters for one Animation window before moving to the next line in the animator file. If you would like to specify that a dynamic parameter has not changed from its value in the last window, you may replace the numerical value of the dynamic parameter with a tilde ~.

Motion Definition Files

Motion Definition files provide the Animation Generator with the information it needs to correctly produce animation files. This includes the step size (typically equal to the window increment) that the Animator should follow, the number of classes with distinct animations, the number of animation objects (e.g. angles), the number of dynamic parameters that should be passed at each line, the algorithms for calculating each dynamic parameter given the statistics calculations, and the algorithms for generating static parameter expressions for each object as necessary. Algorithms in Matlab are defined through the use of function handles.

Defining New Motions

To define new motions, one must modify the Animator source code, account for the new static and dynamic parameters required to fully describe the motion, and specify a motion definition file that matches the Animator's parsing algorithm. One must make sure that the motion definition file accurately calculates parameters based on statistical calculations of the merged sensor data as well as classifications from the Classifier Toolbox.

Detailed Usage

Modifying the 3D Model

Full understanding of how to modify the 3D model and fully integrate it with the rest of the Animator requires understanding of C++ and comfort with the Coin3D API and with Qt.

Changing the Scene Graph

Let's say that animating an arm is no longer interesting to you, and that you would now much rather animate a leg – or better yet an entire body. Where should you start? The first thing that you would need to do would be to create the scene graph that represents the 3D model.

The Separator Scene Object

Recall from the Inventor Mentor reference that 3D images are produced when scene graphs are traversed, with each scene object modifying the traversal state in some way that ultimately produces a 3D image. There is a special type of scene object that is used to make sub-scene-graphs modular. This is called a Separator scene object. The function of the Separator is to ensure that the traversal state after the Separator and its child nodes have been traversed is identical to the traversal state before the Separator was traversed. This means that all transformations, rotations, color changes, and any other modification to the traversal state are effectively isolated from the greater scene graph beyond the separator. Of course images rendered by the children of a Separator are still rendered during the traversal, but no internal state variables that effect future renderings will be detected. Thus, an arbitrarily complex scene graph can be represented solely by a Separator object.

What is the significance of this? Our entire 3D arm is contained in a Separator object. As such, by inserting this Separator object into your own scene graph, you will effectively be able to use our arm in your 3D model without any real work on your part.

This concept can be applied to each body part of the human body. Arms, legs, torsos, heads, etc… can all be designed by themselves, encapsulated using Separators, and easily inserted into an overall scene graph that can itself represent an entire body.

The actual methods to create these individual body parts require basic understanding of the Coin API, which can be obtained through reading the Inventor Mentor and the Coin documentation.

It has been our coding convention to define the primary Separator scene objects used in the scene graph using pointers with are member variables in the Animator class. If you would like to continue to follow our convention, you may define the primary Separator scene objects in the Animator class definition in Animator.h, and implement the sub scene graphs in the createSceneGraph() function in Animator.cpp

New Animation Objects

From the previous section, you are now capable of creating your own 3D models. But how can you control their motion? Most motions in the human body can be expressed using a set of rotations. For example, in our arm, we had 6 major rotations that accounted for the full range of motion of the arm and wrist. To specify a rotation, you must translate the "cursor" to a point on the desired axis of rotation (the elbow joint for example). Then you must specify a vector parallel to the axis of rotation and an angle in radians. The rotation follows the right hand rule for positive angles. The placement of the rotation scene object in the scene graph can be visualized by rotating the "cursor" according to the specifications given by the rotation scene object. A rotation in the +Z direction by an amount of π/2 corresponds to rotating the cursor's internal +X-axis direction to where its +Y-axis direction used to be, and it's +Y-axis direction now points where it's –X-axis direction used to be. Future translations in the "+X direction" will be affected by this rotation.

The key to controlling these rotations is to connect the angle fields of each of these rotations to engines. Calculator engines seem to be particularly useful for this because they are general-purpose. Using Calculator engines, ElapsedTime engines, and OneShot engines, these angles can be controlled by arbitrary functions of time with arbitrary speed, and fluid transition from expression to expression.

While we have been focusing on rotations up until this point, it is actually possible to tie an Engine's output to any field of any scene object. A field is a numerical property of a scene object. All fields implement certain member functions that allow them to be treated in a standard way.

It has been our convention to define pointers to the primary scene objects relevant to Animation, (and usually attached to a Calculator engine) as member variables in the Animator class. If you would like to follow our convention, then you can define additional pointers for the primary animation objects in Animator.h. Typically each one of these is given its own calculator engine for simplicity. The calculator engines and the animation objects are stored in arrays of the same size and are tied together by corresponding index in the createSceneGraph() function in Animator.cpp. To tie a Coin3D field to an engine output, use the connectFrom(SoEngineOuput *) member function of Coin3D fields, which takes a pointer to an engine output as an argument. As the engine output updates, the fields that it is connected to will also automatically update to match the new value of the output.

By carefully placing sceneobjects into the scenegraph and properly tying certain fields to engines, you can create methods for complex and detailed control of minute aspects of your 3D model.

For our arm, we had 6 angles connected to 6 calculator engines. If you create a more complex 3D model, such as an entire body, it is likely that you will need to define many more calculator engines to connect to different scene objects within your graph. Fortunately for you, we have included a standard interface that allows for the modification of the expressions of such calculator engines (and automatically triggers soft transitioning). This standard interface is easily adaptable for use with a larger number of calculator engines. Currently, the number of calculator engines and primary Animation Objects is determined by a preprocessor definition NUM_ANGLES in Animator.h. By changing this value, and using the angle and angleCalc arrays of the Animator class to store your animation objects and your calculator engines respectively, you can continue to use our standard interface with your new animation objects.

Standard Interface for Animator Object

These public member functions can be used with the Animator class to modify important animation settings and variables. For usage tabs, the Animator object can be accessed using the inherited protected pointer: gfx, which should point to the Animator object being used by the application. Since you are free to modify/improve the source code, you can easily add your own functions to improve the Animator object interface as well.

float getAngle(int)

This function returns the angular value for the rotation object stored whose array index is specified by the integer argument to the function.

double getTimeSpeed()

This function returns the value of the speed field in the internal ElapsedTime engine

void setAngleExpr(int, std::string)

This function sets the expression for the calculator engine with index specified by the integer argument in this function to the string argument to this function. This automatically handles soft transitioning management, which will be observed the next time soft transition is triggered. (If soft transition is not triggered, then an instantaneous jump will occur due to the expression change. If soft transitioning eventually does occur, then the Animation will revert to the old expression and softly transition into the new one. This is usually undesirable, so we recommend that triggerSoft() is always called after a sequence of setAngleExpr calls.

void setTimeSpeed(double)

This function sets the speed parameter of the ElapsedTime engine used to control the calculator engines.

void enableTime(bool)

If the Boolean parameter is true(non-zero), then this function enables the connection between the animation objects and their respective calculator engines. If the Boolean parameter is false (zero), then this function disables the connection between the animation objects and their respective calculator engines.

void resetTime()

This function resets the value of the ElapsedTime engine to zero.

void setSoftTime(double)

This function sets the duration of soft time transitioning to be equal to the double parameter.

void triggerSoft()

This function triggers soft transitioning.

Usage Tabs

A usage tab is a tabbed widget that implements a specific purpose with regard to the Animator. If you expand the Animator to produce different Animations, you will likely need to modify the current usage tabs that are configured for arm motion. This basically boils down to creating the right number of widgets, placing them nicely, and connecting them to fulfill the desired behavior. For example, if you expand our arm Animator into a full body animator, you will likely have much more than 6 angles to control, and you will likely need to add many more sliders to the FreeForm usage tab to account for the additional angles. (In actuality, this will automatically be done if you increase the NUM_ANGLES preprocessor value, but for the sake of having an example, pretend that you might need to modify this). Connections between widgets can be accomplished through connecting various signals and slots, as explained at the Qt documentation website. Widgets can modify or control the animation by using the standard interface for the Animator object or additional functions that you can define in the Animator class.

Creating new usage tabs has been made significantly easier due to the use of an included python script: tabMaker.py.

To run this script, type

```
./tabMaker.py
```

If you get a message that says permission denied, you may need to change the executable bit in the file permissions. This can be done doing

```
chmod u+x tabMaker.py
```

followed by

```
./tabMaker.py
```

You will be prompted to enter the C++ class name for your new usage tab, the label name (as it will appear in the GUI), and whether or not you will need to define your own signals/slots (if you are not sure, just say yes).

After this, the script will generate a basic header and source file for your usage tab, will import these files into your Qt *.pro file, run qmake, and provide further instructions to complete integration.

You will be required to modify a few lines in Animator.cpp, mostly in the defineWindows() function, which should be at the top of Animator.cpp. Simply increase the windows count by updating num_windows, and insert your usage tab into the windows array in the same way as all the other usage tabs are inserted in the file. Also, add a preprocessor statement to include your header file, and you are done with Animator.cpp.

The only step left is to modify your header and source file to do what you want. All 5 functions that are required are defined with empty bodies that you should fill in (along with helpful comments). Define any variables you need in the header file, being sure to de-allocate any dynamically allocated variables. Use standard Qt commands to add widgets to your tab. The size of your tab is fixed to a particular part of the screen, so you need to be mindful of space when adding additional widgets.

Changing the Animation File Protocol/Format

Animation files represent communication between the Classifier Toolbox/Animation Generator and the Animator. Apart from the definition of static and dynamic parameters in the Animation Generator section of this User Guide, there are no real restrictions on the format of the animation files. It is up to you to optimize the format according to your needs. Specification of the format includes:

- Identification of types of static and dynamic parameters
- Selection of command words and syntax for static parameters
- Ordering of dynamic parameters

Changing the Parser

To adapt the Animator parser to different animation file formats, one must modify the parse() function in AnimateForm.cpp. The parse function takes in a single QString argument called line, which represents the current line in the animation file being processed. By copying your own code in next to the examples already present, or by modifying the current code, you can adapt the parser to any purpose.

Static parameter command names are typically checked for with the Qt startsWith() function. The rest of a static parameter line or a dynamic parameter line can be further parsed by using regular expressions. Regular expressions are a powerful way of searching for and manipulating textual patterns. Regular expressions can be used to pull specific arguments out of a command, which can then be used to set various animation parameters. To modify some parameters, you may find it necessary to add additional public access functions or modification functions to the Animator class. The use of regular expressions for parsing allows for an extremely flexible way of parsing animation files on the Animator side. The code is also organized such that adding or modifying the parser is relatively easy. For more information on Qt regular expressions, see the Qt documentation page on regular expressions.

Regular Expression Tester

The power of regular expressions comes with a price: it has a steep learning curve. It would be wonderful if before we compiled, we were able to test out a regular expression and see if it actually matches what we want it to match. To help in the extension of the parser, we have developed a regular expression tester called RegexTester, which is included in the Animator folder. To use RegexTester, type in the regular expression you want to test, the search expression that you would like to apply it to, and press the go button. If a match is found, it will appear in the Match/Capture text box. If you use parentheses in your regular expression to capture substrings, you can check those too by modifying the capture number. A capture number of 0 means the entire match, while positive integers correspond to captures in order of parentheses.

Creating New Motions

After you have defined a particular 3D model, have properly integrated it within the Animator, and specified a file protocol for the Animator parser to use, it is time to generate animations with the

Animation Generator. With the file protocol set, it is assumed that the desired dynamic and static parameters are known. The major steps involved are:

- Identify the statistics required to estimate the dynamic and static parameters
- Implement those statistics calculations in the Animation Generator if they are not already there
- Define a motion definition file to calculate all parameters
- Obtain trained/tested data from toolbox and apply Animation Generator to this data
- Take the resulting animation file and play it in the AnimateForm tab of the Animator

New Statistics

In deciding how to model your desired motion, you should come to some idea about what kind of statistics would be useful for calculating static and dynamic parameters. To implement new statistics calculations into the Animation generator, you must first declare the new statistics in the listStatistics.m file. You are required to add a line containing the name of the statistic, the default value, the minimum value, and the maximum value. The default value is placed instead of an actual calculation during errors or if the actual calculations lie beyond the minimum or maximum boundary.

After declaring your new statistic in listStatistics.m, you must define the algorithm used to calculate it in computeStatistics.m. Add a corresponding code block to the switch statement in computeStatistics.m and specify the calculation. The input variable windowData contains all of the accelerometer datapoints in the current window. If there are N sensors, then there will be 3N columns, and a large number of rows (depends on the window size). Different rows of windowData represent different points in time. Each triplet of columns starting from the first three and increasing represents the x, y, and z components of data for different sensors. The order in which the sensors are traversed in the just-described process is the same as the order in which the sensors were selected when they were merged. It is possible to find out this order by examining the “header” variable in computeParameters.m in debug mode after loading some data. The result of the statistics calculation should be placed in the variable statistics(1,i) and should be a single scalar value.

Defining a Motion Definition File

The motion definition file should define the variables that are needed by the Animation Generator to accommodate the format demanded by the Animator parser. This includes stepsize, the number of classes to be considered (excluding unknown), the number of animation objects (e.g. angles) in the Animator, and the number of dynamic parameters that should be forwarded. In addition, algorithms for computing dynamic parameters given statistics calculations are provided as an array of Matlab function handles. Also a matrix that maps classification values to calculator engine expressions for each angle should be provided in the motion definition file. The FreeForm and MotionForm tabs in the Animator can aid in the development of such expressions. Using MotionForm, one can generate any type of simple oscillatory motion by varying parameters in the grid. Each row of the grid represents a different animation angle. The variables a and b represent the minimum and maximum angle values in degrees in the oscillation for any particular angle. The p variable represents an initial phase component in degrees for the sinusoidal oscillation. The actual value of p is not as important as the differences in p between different angles. The f variable represents the frequency of oscillation for the particular angle in cycles per second (assuming speed is set to 1). More accurately, the value of speed times f gives the true frequency in cycles per second. To fine-tune an Animation, one may switch at any time from the MotionForm tab to the FreeForm tab. Upon doing so, the animation will freeze in the position it was in during switching. Angles can be manipulated at this point to obtain better values for the MotionForm grid. When satisfied with your MotionForm parameters, you may obtain the calculator expressions by using the convert button. This will produce a new *.anm file or append the expressions to an existing one, and add a single dynamic parameter line (in our case, it adds a single line with speed being set to 1). By opening this animation file and copying the expressions, one can embed these calculator expressions in a Motion Definition file so that the Animation Generator can produce this type of motion in the future.

If you would like to define your own motion equations from scratch, you need to define angle expressions as functions of the parameter a. Time dependence is achieved through this parameter a, which is the output of an ElapsedTime engine. If the speed variable of this ElapsedTime engine is set to 1, then the parameter a directly corresponds to time in seconds. The speed parameter is shared among all angles, although it is possible to assign specific Calculator engines their own ElapsedTime engines (and hence their own speeds) by changing the Animator class in Animator.h/cpp. To change the general speed parameter which is currently connect to all of our Calculator engines, use the Animator class's public member function setTimeSpeed.

Creating the Animation File

Once a motion definition file has been created, and a classification export has been obtained after testing in the Classifier Toolbox, it is possible to create the animation file. The select parameter file button in the Animation Generator allows you to select a parameter file compatible with the Classifier Toolbox. It is recommended to use the same parameter file for generating animations that was used for training and testing. To compute all statistics for the accelerometer data, press the compute parameters button of the Animation Generator. You will be prompted for either a merged .mat file or a label file, followed by a testing export file. After calculation is complete, you will be asked to select a file name in which to save the statistical data. To create the final animation file, select the generate animation button, select the statistics file produced in the previous step, and select a motion definition file. If all things were properly created, then you will be prompted to select a file name with which to save the animation file. The animation file can be played in the AnimateForm tab of the Animator.

Further References

For Coin3D development, the Inventor Mentor is a must read. Coin3D derives from another 3D development library called Open Inventor. Although none of the source code is the same, Coin3D maintained the same API as Open Inventor, and therefore to learn one is to learn the other. For more advanced features, see also the Inventor Toolmaker.

Thorough class and function information can be found at the official documentation sites for each library

Coin3D Documentation (as well as SoQt)

Qt Documentation

For help with Matlab, the best source is the Matlab's built-in help system. The second best source is Google.

Further questions can be sent to:

mmiguel6288@gmail.com

jtsao22@gmail.com

vikramb@comcast.net