# Isopod Labs Animation System: Technical Design Document

## Overview

This document describes the new animation system to be created for the next Isopod Labs game.

## Goals

We wish to create an animation system on a par with, or exceeding, the capabilities of the most advanced third-party animation systems available today. The advantages of developing this system in-house rather than using an existing off-the-shelf implementation are:

1. Deeper integration with our existing technology, affording a consistent and seamless pipeline for all our assets.
2. Complete access to, and familiarity with the source code, which will let us use the animation system in vastly more inventive ways.
3. Complete extendibility for innovating past generic solutions.
4. Ability to tailor the system to our specific needs.
5. Avoidance of struggling against a generic solution which is an imperfect fit for our needs.

## Pipeline

We have an existing extendible asset pipeline, which we use for all in-game assets, including menus, models, particles, and configuration data. New data structures can be easily defined for output, and actual serialization, endianness, external dependencies, data duplication, and so on are automatically handled. We shall refer to this as the ‘ix’ system.

The ix system handles a large number standard file types, for bitmaps, sounds, and so on, but it also has three ‘native’ file types, with the extensions ix, ib and ibz. An ix file is a human-readable (and editable) text file which describes the data in the form of ix-script. An ib file is a raw binary file which, once loaded, is binary-compatible with data read from ix script. An ibz file is simply an ib file that has gone through the gzip compressor on output.

While we use the ix system for our existing, very minimalist animation system, we can use its extendibility to define the more complex structures required for our new animation system.

### Stage 1 - MAX

We shall extend our 3DSMAX export plugin to extract raw animation data from all potential sources of animation. This includes:

* animation material parameters
* visibility information
* node-type specific parameters, such as camera, lighting
* vertex shells for morph targets
* triggered events, for sounds, arbitrary instantiation of entities, custom events, etc

In addition, some meta-data about the animation itself will be needed, such as:

* cut points – where instantaneous changes are required
* match points – where multiple blended animations are forced to coincide
* data to be passed up to the next stage of the pipeline – such as compression quality information

We propose using MAX’s animation note-track to store animation meta-data and events. These can be stored in raw ix script format for maximum flexibility. If time permits, or at a later time, we will be able to develop a MAX plugin to add native ix-script tracks which could then use the existing entity editor.

Often, simple animations can be associated directly with a scene node and will be exported alongside the model using the general export function in MAX. For more complex animating entities, such as characters, experience tells us we will need to support a separate animation-only export. In any case where multiple animations can be applied to a single model, we shall require that a separate MAX file is used to define the animation, and that it be exported as a separate animation file.

The animation in a MAX file can span multiple nodes, and so an animation file can be considered an ‘animation hierarchy’ which describes the animating values of all of the hierarchically defined nodes in the MAX file.

At this stage, the animation will be raw and uncompressed. Each animating value will be output at each defined time interval (typically 30Hz), and the value will be stored in a plain uncompressed format – for instance, a quaternion to represent an orientation would be stored as four floating point numbers.

We defer the compression to a later stage to avoid premature loss of data, and to allow experiments to vary the compression quality levels without going through MAX. It is also expected that later stages of the pipeline will contribute information that might affect how the data should be compressed.

The animation data from MAX will usually be output as an ib file, though ix might be used on occasion as a debugging aid, since it is human-readable.

### Stage 2 - Editor

In the past, we have composited the compressed versions of the simple animations produced from stage 1 programmatically. Animation blends, for example, were set up using code at the point where the effect was required. Modern animation systems allow complex animation blends to be defined in a separate editor. This puts more power into the hands of the designer or animator, and allows complex effects to be previewed instantly.

We already have the framework of an editor for the ix system. It is able to browse any data that can be handled by ix, and also preview common uses such as particles and menus.

We shall add a module to this editor which is designed to allow graphical creation and manipulation of an animation blend hierarchy, and previewing of the result. The majority of the user-interface would not need to be specific to animation, and so this work can be leveraged across a multiple of similar hierarchical data layouts. For example, the menu system could be similarly visualized.

A simple example of the kind of interface we envision is given below on the left. The boxes can be individually dragged across a large scrollable area, and the connections are generated by dragging a line out from the connection points (shown as black boxes in the image) to the inputs.

Input 1

Input2

blend

run

walk

Input 1 Input2

blend

The second style, given on the right, might be more appropriate to describe a ‘containment’ type of relationship. We will make this achievable by dragging the child boxes inside the parent.

We anticipate these animation blends could become quite large, and be difficult to visualize in their entirety. In order to manage this complexity we shall define a grouping mechanism whereby arbitrarily large sub-trees can be encapsulated inside boxes that can then be shrunk down and drawn more simply. The insides of the box can be edited by zooming into them.

Similarly, macros can be defined by creating a re-usable box that encapsulates an arbitrary number of animation components.

An important component of the editor is the ability to preview animation systems that are created within in, as if they were running in the game. It will be possible to preview on the pc or target hardware (console).

The editor will have the option of exporting the data directly to in-game data – which might be useful for artists who want to test their work directly in the game. The usual procedure, however, would be to simply save the editor data and rely on stage 3 to create in-game data.

### Stage 3 – isocmd

The third pipeline stage uses isocmd – our all-purpose ix command-line tool. This stage is responsible for stripping out editor-specific information (such as user-interface locations of the boxes), compressing the animation data, and merging the data with any external files that are requested (e.g. if, say, a sound file is triggered from an animation, if may be desired to embed the sound data within the animation file).

We shall also rearrange the data for fast access to the data, potentially optimizing it differently for the separate platforms.

Typically this stage is triggered by our build process from within visual studio. We currently have levels, menus, particles, etc, automatically built this way, and adding animations to the process would be simple and natural. One advantage to building in this way is that we can use the configuration settings of visual studio to automatically export the data for the platform we are currently developing for.

## Compression

Compression is most complex component of the export pipeline, and happens during the last possible stage in order to preserve data fidelity.

The flexibility of the run-time system we propose to build will allow for arbitrary combinations of arbitrary compression schemes, so we can minimize any risk involved in this choice. By implementing the overall animation scheme before tackling the compression, we can start game development with uncompressed data, and seamlessly switch in the compression when it becomes robust enough.

Based on research into the findings of others, we propose to use non-uniform rational B-splines to approximate the data. In addition, the data and time values used for the splines will be quantized in order to save space. By taking the quantization into account when spline-fitting, potential quantization artifacts can be minimized.

## Data layout

### Stream Data

I propose making the lowest data level a byte stream as follows:

The first byte will represent either a time offset (in 60ths or 30ths of a second), or an escape code (i.e. if the byte is greater than, say, 0xf0, it will indicate some special data to follow).

If the byte is a time offset (i.e. < 0xf0), then the following data will represent a key value in a compressed format. These keys are knot values for the NURB evaluation, and so the runtime must maintain the last three keys at any one time.

If the byte is an escape code (i.e. > 0xf0), then the following data will represent some settings for the animation stream. Using escape codes will allow us to embed time-specific data, but will also allow us to embed rarely needed information on an as-needed basis, as opposed to adding fixed data to stream headers.

An example of such data is quantisation information (basically a scale to be applied to the key data). We will be able to change the quantization settings along the animation stream.

Another example is ‘match-points’ – where two animations that are being combined should be forced to coincide. Each match-point will have an id that must correspond to an id in the other animation(s).

Embedding match-points in the stream makes sense because the streams must be parsed up to these times anyway.

### Stream Header

Each stream begins with a stream header. This is used to match animating parameters, and describes the type and format.

When streams from different animations are blended, they must be matched up by id, and also type. The main use of type is where multiple streams must combine to create one end result, such as bone animation, which may require separate rotation, translation, and scale.

struct AnimationStream {

crc32 id;

uint8 type;

uint8 format;

};

The byte stream follows immediately on from the stream header.

It is proposed that the animation events will be stored in a special stream with an id of, say, ‘Events’. The format field will specify that the data represents events. Basically this track will consist of time offsets followed by ix-formatted event data.

The format can specify that the stream always refers to the base-pose values, and so there would be no following data. This differentiates between un-animated nodes, and ones that should explicitly stay at the base pose.

The format can specify a fixed value, which would then directly follow the header.

### Animation Unit

An animation unit is a collection of animation streams that represents one animation as authored in, say, 3DS MAX.

struct AnimationUnit {

uint32 flags;

float length;

ISO\_openarray<AnimationStream> streams;

};

The length represents the overall duration of the animation. The flags will specify whether the animation loops, and is available for future expansion.

### Animation Commands

The remainder of the structures in an animation definition correspond to the various combining and state-machine operations that are specified in the editor. An example would be:

struct AnimationBlend {

float contribution0;

ISO\_ptr<void> anim0;

ISO\_ptr<void> anim1;

};

This specifies two animations to be blended together, in a ratio specified by contribution0. This is not a transitional blend.

Such structures are easily changed and expanded using the ix system. In reality, for instance, it might make more sense to have an n-way blend rather than a cascade of 2-way blends.

### Non-animation Data

Some information required by the animation system will actually be stored separately from the animation data. The principal example is the animation bind information, which is stored with the item to be animated. This data specifies what streams are supported, and how they should be composited and stored.

The binding information will be an array of AnimationBind structures:

struct AnimationBind {

crc32 id;

uint8 type; //matrix, float, int, bool

uint8 unused; //I'll think of something

uint16 parent;

};

The id is used to match with the animation streams. The type specified the type of final destination. If it is a matrix, for example, then the system will know to combine any rotation, translation and scale streams of this id together into a matrix.

The parent field is used when the type is a matrix to indicate which (earlier) matrix this one is relative to. Special values might be used to composite the matrix as a root-animation.

## Run Time Component

### Loading

Since the animation system uses the ix system, loading is done automatically by referencing the data you want to use.

The animation can be embedded inside larger file units, in which case they will be automatically preloaded when the larger file loads (such as a level).

We shall almost certainly choose to embed all the animations for a particular character in one file, along with its mesh and configuration data.

### Starting an animation

A certain amount of run-time information must be kept for each animation command, as well as for each animation unit (possibly) and stream (definitely). We have the choice of expanding the data into run-time classes, or of taking the menu-system approach of traversing the data at run-time and maintaining a separate memory buffer for dynamic data. The benefit of the former is simplicity. The benefit of the latter is that we need fewer dynamic memory allocations.

Where two or more animations must be blended together, their streams must first be matched up. Unpaired streams will typically be blended with a base-pose, or used directly.

The final resultant animation hierarchy must be paired up with the animating item’s data using the AnimationBind structures. The resultant data will usually be an array of root-relative matrices, though the binding information could be set up to create world-space matrices just by changing one parent field.

The run-time structures are straightforward and will change as programming progresses, so I shall not list them here.

### Multiprocessing

Animation is an ideal candidate for multi-processing. Each animation hierarchy is composed of smaller animation units that can be evaluated completely separately. The results of these evaluations can then be blended together.