Model Tools User Manual

Overview

Greetings! Before we dive into how to actually get a model into Spark from Maya/3dsmax/Modo/whatever, we should take a look at the format, how it works, and what its capabilities are. Rather than simply diving in head first, this will help clear up a lot of confusion later on.

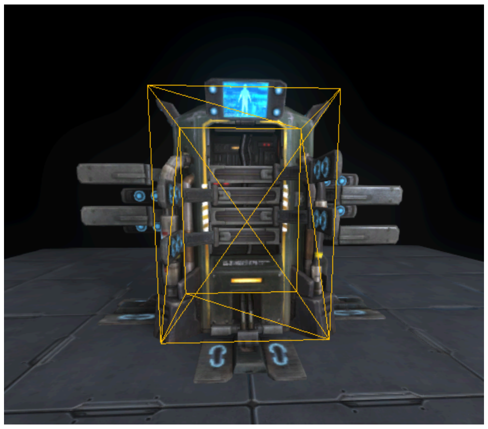
A model in Spark is comprised of one geometry layer -- that is, the vertices and triangles that you see. These vertices can be moved/deformed using bones. Pretty standard stuff, right? Models can also have attachment points and cameras within them. Attachment points are for specifying a position on a model, for stuff like babblers and keeping track of where exactly the bullet effects come out of a gun. Cameras are useful for creating player view models. Both attachment points and cameras can be parented to bones so they animate with the model, or left static.

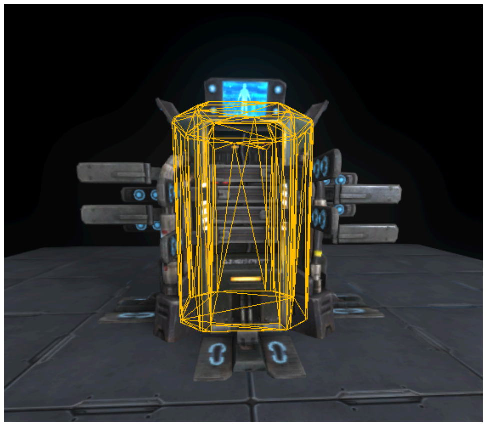
Models can only be animated using bones. No other deformation methods your 3d package might support (eg. blendshapes from maya) will carry-over to Spark.

Physics

Collision Reps and Solids

The physics system is where things get interesting. A spark model can have many different “collision reps” which are different ways for the model to be interacted with. For example, the “damage” collision rep -- used for calculating bullet hits -- will likely have a lot of collision solids, making for a very finely detailed volume that closely resembles the visual mesh, whereas the “move” collision rep will be very simple, using very few shapes so player movement will be unimpeded by small protrusions and bumps.


“Move” collision for armory (simple box)

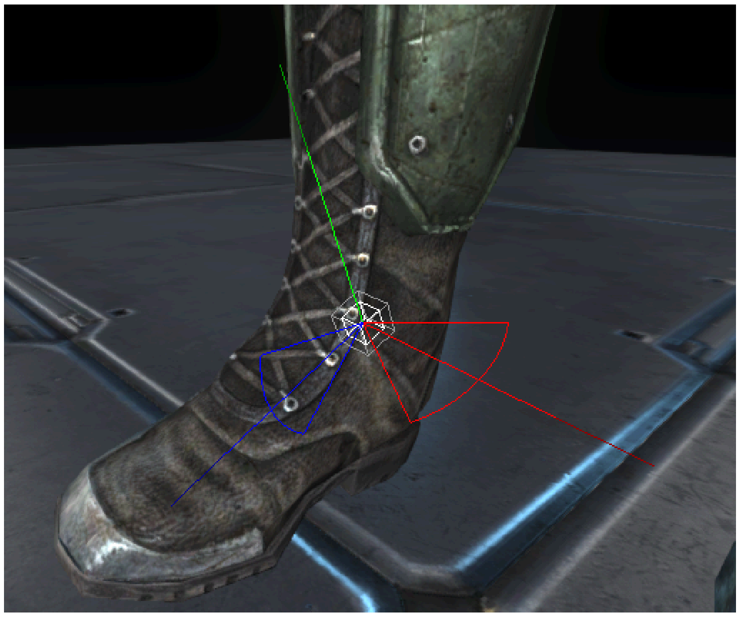

“Damage” collision for armory (more detailed)

There is also a collision rep called “default”. This is always present if any physics are present. The default collision rep acts as a fallback for any reps that aren’t specified. For example, the skulk model only has a default collision rep. This means that when you shoot a skulk, it looks for the damage rep, doesn’t find it, so it instead uses the default.
A special thing about the default collision rep, is it has double-duty as not only a fallback for all other reps, but this rep is also the one whose role is reversed whenever an entity is simulated, or “ragdolled”. This is what makes player models able to flop about after death, or structures able to be pushed around after being destroyed. They are no longer being moved by the game, but are instead having their movements simulated by the physics engine.
Ordinarily, the bones in a model determine the position of the physics objects. However, in simulation mode, it’s reversed: the collision solids of the DEFAULT rep drive the bone positions, and then the other collision reps will continue to inherit their positions from their parent bones, as usual. This introduces a limitation, however: the default collision rep may ONLY have 1 solid per bone if it is planned to be simulated at some point. If the model will never be simulated (eg it is just a static prop), then it does not matter how many solids are in the default collision rep.

Quick recap, because I know that’s a lot to take in at once: collision solids belong to collision reps. These are the things you collide with. Collision reps are the different ways in which the model can be interacted with, eg movement, hitboxes, simulation, etc.

Joints

Joints are special objects that hold collision solids together during simulation. Since it has to do with simulation, this means that joints are only ever useful in the *default* collision rep. Technically, they *can* be declared in the other reps, but it’s a complete waste of time as they’re never ever used.



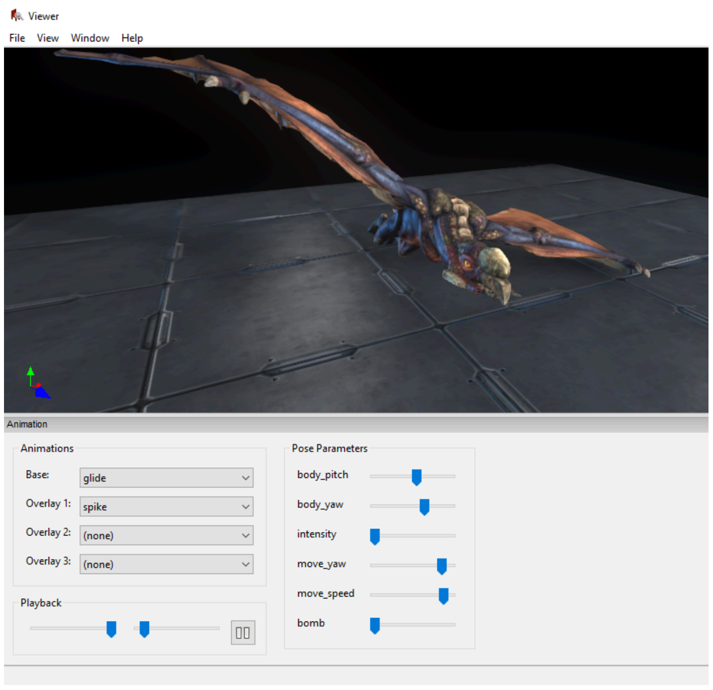
A joint specifies two collision solids, and a range of freedom between them -- 3 rotation axes. For example, a joint that only allows rotation around one axis would behave like a hinge, like your knees, whereas a joint that allows rotation around multiple axes would act more like a ball-and-socket type of joint, like your shoulder.
Without joints, there is nothing holding together the pieces of a model during simulation -- which might very well be exactly the behavior you’re after, especially if the model you’re working on is

mechanical. For organic objects, however, this makes the models appear to stretch out horribly, and just doesn't look nice at all.

Collision Pairs

You will rarely, if ever, need to worry about collision pairs because they are mostly handled automatically, but I'll go over them briefly for consistencies-sake.
Most of the time, if you've created your collision solids properly, you're inevitably going to have some overlap between the solids. Solids will normally try to repel themselves out of each other. If you have two adjacent solids overlapping, and connected with a joint, this causes extremely bad jittering, if not full out model explosions, as the solids repel each other, and are forced back together by the joint. Collision pairs allow you to specify pairs of collision solids that will not interact collide with each other. In the case of the example above, the solids will still be restricted by the joint angles, so the solids won't be able to pass through each other, but will no longer create a ridiculous explosion of movement.
As mentioned above, the process of finding collision pairs is automated during the compiling process. The compiler looks for collision solids that are interpenetrating in the rest pose, and disables collisions between them. However, there are some circumstances where you may want to specify other disabled pairs, or even override the automatic disabling to force a pair enabled (for example, if you want a structure to explode when it is rag-dolled, and two solids aren't jointed by a joint). More details on the manual overrides below.

Animations



Base: glide
Overlay 1: spike
Overlay 2: (none)
Overlay 3: (none)

Spark has a powerful animation system which allows you to do everything from playing a simple animation, to creating complex sequences built by blending together many different sub-animations. For example, the lerk flying animation not only makes the lerk flap its wings, but also allows the game to specify which direction the lerk is looking, which direction it is flying, is it attacking at the same time as well, and is it twitching because it's being shot? Rather than having to animate every single combination of these variables, Spark allows you to combine many smaller animations into more complex sequences.

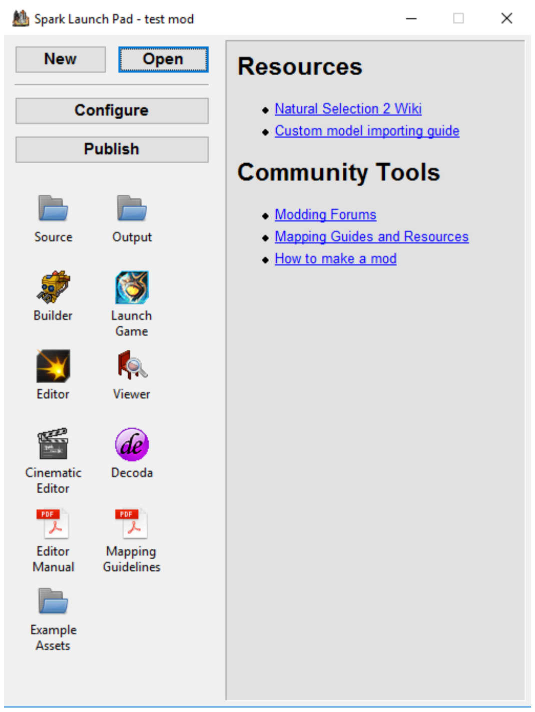
To create complex sequences like this, the artist animated the individual pieces of the animations, then combined them in the compilation process. For example, the artist created several different animations of the lerk-glide for different bearings, and the game is smart enough to blend between them based on the “move_yaw” slider.

Animations also can contain time markers or “frame tags” embedded in the animation data. This is useful for marking specific moments in an animation. For example, the marine axe view-model’s attack animation has two markers named “hit” located right at the moment when the axe blade is swinging in front of the player.

One last thing I’d like to point out about animations is that they will massively inflate the file size of a model. The model format, however, provides a way to reuse the animations of an existing model. For example, the shadow and kodiak skins for skulks use the animations from the base skulk model. This of course requires that the two models both have an identical bone structure, and that the names of bones are the same. You obviously can’t tell a marine model to use skulk animations… that just simply won’t work, and on the off-chance it doesn’t present an error message, the result will be a fate worse than death for the poor marine.

Getting Started

The first step in getting custom content into NS2 is to set up a mod. This is done using Launchpad, a program found in your NS2 directory. All of the modding tools can be accessed directly from the ns2 directory, however it is better to use a properly structured mod, as this provides a way for you to make changes to NS2 without cluttering your NS2 directory.



After creating a new mod, you should see shortcuts to all the modding programs inside launchpad, as well as shortcuts to the source and output folders.

Builder and Viewer are going to be the two programs we use most heavily when importing custom models and animations into NS2.

Builder looks through the files and folders in the “source” directory, and “builds” the files it finds. For example, builder will look for PSD image files and convert them to DDS image files to be used in-game.

Viewer allows you to view your converted model in-game without having to actually launch the game.

To get a model from your 3d package of choice into Spark requires you follow these steps:
- Export your model as either Collada (*.dae) or Filmbox (*.fbx).
- Create a *.model_compile file in the corresponding source directory.
- Run Builder, which will look for *.model_compile files, and convert those into *.model files and place them in the corresponding “output” directory.

The model_compile file format

The model_compile file is a plain text file that tells the compiler exactly what files it needs and what needs to be done with them to generate the final model file. The file is composed of a series of “directives” and “parameters” following. For example, a very simple prop with no collisions and no animations would look something like this:

```
geometry “myAwesomeProp 1.fbx”
```

This is the “geometry” directive, which accepts only a single parameter -- the file name -- which tells the compiler to look for a file named “myAwesomeProp 1.fbx”, and use that as the *visual geometry* of the file -- that is, what you can see. This will result in a un-animated (static) model that can’t be interacted with, only looked at.
NOTE: Any file names declared in the model compile are expected to be included in the same directory as the model_compile file.
Note that the parser that reads this file uses whitespace (spaces) to denote the breaks between terms, but by surrounding text with double quotes, you can use spaces in names. Notice the double-quotes surrounding the filename. These are only necessary because the filename itself contains spaces. If the filename did not contain spaces, the double quotes would be optional, though still recommended for clarity-sake. If we tried to write this without the double quotes, the parser would see
-geometry
-myAwesomeProp
-1.fbx
and would complain that “1.fbx” is not a valid directive. By surrounding it in quotes, we tell the parser that the space is part of the file name.
Also worth pointing out is that the parser also supports both C and C++ style commenting.

```
geometry “myAwesomeProp 1.fbx” //hey look, I’m a comment!
physics “myAwesomeProp_phys.fbx” /* I am also a comment, in fact I am a
multi-line comment! */
```

Directives listing

Here is a list of all the directives that can be used in the model_compile file, and what they do.

Required Directives

geometry [filename]

Example:

```
geometry mygeo.fbx
```

Animation-Related Directives

animation [name] [filename] [speed [number]] [loop] [relative_to_start] [from [time] to [time]]
(there are many other ways to use this directive, this is a simple (ish) version. See the Advanced Animation section for more information)
Start with the keyword "animation". The next term is the name of the animation, followed by the filename to read the animation data from. Those are the only required terms, the following are optional.
speed - default 1.0. A multiplier for animation speed (eg 2.0 makes it twice as fast)
loop - makes the animation loop. Omit this term for non-looping animations.
relative_to_start - makes this animation's bone transforms relative to their starting position. Used for creating a sub-animation that will be combined into a larger animation later.
from time **to** time - "from" keyword, followed by either a frame number or frame-tag name (3dsmax users only), followed by "to" keyword, followed by another frame number or frame-tag name. This specifies the frame range within the animation file to use for our animation. It's standard practice to have many animations in the same file, but at different positions on the timeline. This lets us select the exact frame range to sample.

Example:

```
animation "run" "run1.fbx" loop from 38 to 96
```

animation_model [filename]
Tells the compiler to include a reference to the specified file in the compiled model, so the game engine will look for that model to use for animations instead. The filepath is expected to be the spark-relative model (ie relative to the ns2 root or mod "output" directory).

Example:

```
    animation_model "models/alien/gorge/gorge.model"
```

animation_node
(this one is very complex, see the Advanced Animation section)

disable_compression
Highly recommended you do not alter these settings. Included for completeness only.
Spark uses an animation compression system to save game memory, resulting in very slight reduction in visual quality. Compression is automatically disabled for viewmodels (model_compile filename ends with "_view.model_compile"), so there aren't many cases left where this option should be used.

frame_tag [name] [filename] [framenumber] (FBX only)

Creates a frame tag at the specified frame *in the source animation file*. The FBX file format does not support frame tags exported from 3dsmax. The way animations are handled in spark is it keeps track of all the animation files that the compiler will need. Then, it samples the specified frame ranges of those files, and copies that to a new animation. The frame tags you are specifying with this directive are applied to the source files, *not* the individual animations that result after sampling.
To clarify: the frame number you specify must be a non-negative whole-number, and it will correspond with the frame number that appears in the timeline of the file you are exporting from.

linear_max_error
Highly recommended you do not alter these settings. Included for completeness only.
A tolerance value used by the animation compression system.

quat_max_error
Highly recommended you do not alter these settings. Included for completeness only.
A tolerance value used by the animation compression system.

Physics-Related Directives

collisions [on/off] [solid 1 name] [solid 2 name]
This allows you to specify if two solids should or should not collide. Note that if two solids already interpenetrate in the rest state, you don't need to disable them -- the compiler already does this automatically. This option is used to disable solids that aren't automatically, or to force-enable collisions that *are* disabled automatically.

Example
```
collisions off "face1" "palm"
```

physics [filename]
Specifies one and only one collision rep -- the default rep -- from the specified filename.

Example
```
     physics "chair_01_phys.fbx"
```

physics {
[rep name 1] [filename 1]
…
[rep name n][filename n]
}
Specifies multiple collision reps from multiple files. Note that if none of these are named "default", the compiler will use the visual geometry to automatically generate one. It's always best to explicitly create it, however. Also, **YOU MAY ONLY DECLARE "physics" ONCE!**

Example

```
    physics {
    default “chair_phys_default.fbx”
    move “chair_phys_move.fbx”
    damage “chair_damage.fbx” }
```

Note that the compiler really doesn’t care about new-lines. You could put all this on one line if you really wanted to. It’s much easier to look at if you keep things formatted nicely of course.

joint [null name] [solid 1 name] [solid 2 name] [limits]
(**FBX-only**. Collada (DAE) users must use the reactor tools present in 3dsmax 2009)
Tells the compiler to treat an empty/null/locator object as a joint.
In maya, max, or whatever application you choose to use, if you export a null object (sometimes also referred to as an empty or locator) with the file loaded via “geometry”, you can declare these objects to be joints.
null name - the name of the object to use as a joint.
solid 1/2 name - name of the solids that this joint constrains
limits - 3 numbers, the amount of freedom each axis (x, y, and z) has, in degrees.

Example

```
    joint mainToRear1 mainBody rear1 30 45 10
```

Note: high values will cause the joint to be unstable. Try not to go above 45. This is actually 90 degrees of freedom, as this sets the range from -45 to 45.

Miscellaneous Directives

attach_point [name of object]
This tells the compiler to treat an empty/null/locator object as an attachment point. Attachment points can be parented to bones or left static.

Example

```
    attach_point “babbler_attach_1”
```

scale [numeric value greater than zero]
This applies a scale to the model during compilation. If your model is twice as big as it needs to be, it’s much easier to set the scale here to 0.5 than it would be to re-scale the entire scene in your 3d app, and re-export.

Example

```
    scale 0.375
```

bone_order_override

```
{
	[Bone Index] [Bone Name]
	...
}
```
When a model is compiled, the bones are sorted based on hierarchy such that no parent has a higher index than any of their children. There can be many solutions that satisfy this requirement, and bones can be sorted differently based on how the export software ordered them, even though their names may not have changed at all. This is really only an issue when attempting to export a new model using an existing bone structure. For example, the shadow fade is in FBX format, while the other earlier fade models are in DAE format. This causes the bones to be sorted differently, leading to very odd results when inheriting the animations of the bones, since they are stored internally by index, not by name. This directive allows you to force the bone ordering to conform to an existing structure -- provided that it doesn't break the hierarchy requirement. Also, the list of bone indices cannot contain duplicates, must be consecutively numbered (ie no missing numbers in the final assortment), and must start at index 0. Note that this is with regard to the final assortment of bones -- they can be specified in the directive in any order (eg 3,0,2,1 would be valid if there are only 4 bones).

material_path_replace [stringToSearchFor] [stringToReplaceWith]
Specifies a string pairing to find and replace in the material path name, shortly before writing. For example, maya users will probably find it useful to replace "sourceimages" with "alien/shell" if they are modifying the shell. You can use as many of these directives as you want. The replacements are performed not just once, but for every instance of 'stringToSearchFor' that is found, and in the order that they are declared in the model_compile file.

flip_bitangent
BLENDER ONLY! Inverts the exported bitangent vector, making the model compatible with tangent-space normal maps exported with a -Y swizzle. (Long story short, without this step, you'd need to invert the green channel of your normal maps to make it look right).

Physics workflow

We recommend you use the FBX file format over the DAE file format wherever possible. We've continued to support DAE for backwards compatibility. The DAE workflow requires that physics are exported using a special set of tools only available in 3dsmax 2009, called “reactor” tools. This provides a convenient toolset for specifying which meshes are collision solids, and for creating joints and limits.

Unfortunately, we don't have such a luxury with the FBX format, but we DO have more compatibility with a LOT more programs. That alone is worth the tradeoff.

So, how does one get collision solids and joints working with the FBX format? Under the new format, any FBX files that are reference via the “physics” directive in the model_compile file are used, like before. However, unlike it's Collada (dae) counterpart, the FBX compiler will look for ALL visible meshes in the exported file, and use each one as an individual collision solid, keeping in mind which ones are parented to which bones. It will also load and keep track of all the null/locator/empty/whatever-your-app-calls-them objects, so they can be converted to joints via the model_compile command.

At the end of the day, the only functionality we're really trading off is the ability to visualize the joint constraint angles in a 3d viewport.

Advanced Animation

For the most part, the description of animations in the Animation-Related Directives section will be more than sufficient for most models. For example, an animated prop for a level will generally not need any more advanced techniques.
Creating playable entities such as an alien class or a marine model, however, *do* require more advanced controls for handling the animations.

Terminology

Note to skim-readers, the following 4 terms are NOT directly related to the model_compile directives!!!

Bear with me, this is where things get a little bit hairy. There are a few terms that mean different things depending on the context they're used in.
Sequence - the highest level of the animation system, this is what we'd traditionally call an "animation". Examples of this are "idle", "run", "attack", etc. A sequence takes into account all the various parameters needed to blend together all the underlying animations, for example it will know what angle a player's head should be looking, and it will know which specific animations to blend or layer together to make that happen. Sequences are declared in the model_compile file using the "animation" keyword. Confusing, I know.
Animation Node - Think of animation nodes as the "pieces" of the animation that fit together to form the complete sequences. For example, an animation node named "#turn" might contain the animation data and the blending parameters needed to show an alien turned left, middle, and right, depending on the slider position. Note that the hashtag '#' in front of the name isn't required -- it's just a naming convention. Animation nodes are declared explicitly (ie named and available for re-use) in the model_compile file with the animation_node keyword, or they can be defined implicitly (ie one off, only needed once) with the animation keyword.
Blend Parameter - These are all the values that can affect the final look of a model's animation in game. For example, many models in ns2, especially alien ones, have an "intensity" value, that ranges from 0 to 1 that determines how much an alien will be shaking/flinching, to be used while it's taking damage. The "intensity" is a blend parameter. Other examples include "body_yaw" and "move_speed", which changed based on body orientation, and movement speed, respectively. Blend parameters are declared in the model_compile file when declaring a "blend"-type animation or animation node.
Animation - This is a structure that holds all of the actual bone data -- the absolute lowest level of the animation system. These are all the pieces that are pulled together via animation nodes, and are also the animations that are created by the artist in their 3d program. Examples of animations would be a lerk looking to the left, or looking to the right, or sitting still and breathing. Not all of them will necessarily have names, as they are defined by a frame range and a file name only.

Alright, we've got the terminology out of the way, so let's take a look at how all these pieces fit together -- and don't worry if it doesn't make sense just yet, things should start clicking a little bit later.

To start, let's go back to simple example that was used in the Animation-Related Directives section, for the "animation" directive.

```
animation "run" "run1.fbx" loop from 38 to 96
```

Unlike the other directives in the file, this one is really in two main parts: First, we have the word "animation" followed by the name "run". That's the sequence. Everything after these two words is actually part of an animation_node definition. Remember, don't confuse an "animation_node" structure with the "animation_node" directive. These are two slightly different things. Using the "animation_node" directive does indeed create an animation_node, but it also names it. In fact, we can substitute the word "animation" for "animation_node" and it is a valid statement.

```
animation_node "#run" "run1.fbx" loop from 38 to 96
```

This is doing the same thing as the example above, except instead of creating a sequence and associating its animation_node with a new unnamed animation_node, it is creating an animation_node, and naming it "#run". (Note the '#' prefix. It's just a naming convention, it's not required.)

There are 3 types of animation_nodes: animation, blend, and layer. In both examples, what we see is an animation-type animation_node. This means the node refers to a single animation, "run1.fbx" sampled from frames 38 to 96, set to loop. Instead of an animation-type node, let's see what a blend-type node looks like.

Blend-type animaton_node

```
animation_node "#aim"
blend "body_pitch" -45 45
{
"lerk_turns.dae" from "lerk_look_down_POSE" to "lerk_look_down_POSE" loop relative_to_start
"lerk_turns.dae" from "lerk_idle_SFRAME" to "lerk_idle_SFRAME" loop relative_to_start
        "lerk_turns.dae" from "lerk_look_up_POSE" to "lerk_look_up_POSE"  loop
relative_to_start
}
```

Here, we have a new animation_node being explicitly defined so it can be used again later, named "#aim". Rather than following with a filename, which would make this an animation-type animation_node, we follow it with the keyword "blend", and the parameter of this blend we'll call "body_pitch", and that will be between -45 and 45. Now, at this point, we can optionally add the keyword "**wrap**" right after the last parameter value. This is useful for movement animations, where -- if you refer to the example model_compile files -- you'll see it's normal to define a north, east, south, and west animation. We would use the keyword "wrap" in that case if the starting value, 0, equates to the ending value, 360, which in that case it does, but in this case it does not, so we omit it. The next section is surrounded with curly braces { } and is a listing of more animation_nodes. In this case, we only use animation-type animation_nodes, but they can be filled with blend or layer-type nodes as well.

Layer-type animation_node

(assume that "#turn" and "#flinch" have already been defined)

```
      animation "idle"
            layer
            {
                  "lerk_idle.dae" loop
                  #aim
                  #turn
                  #flinch
            }
```

This code was taken directly from "assets\modelsrc\alien\lerk\lerk.model_compile", cleaned up a little bit.

Here, we see a new sequence named "idle", and instead of using a filename as the next term, which would begin the definition of a new animation-type animation_node, we use the keyword "layer", followed immediately by an opening curly brace '{' (required). Here, we list 4 animation_nodes. The first one should appear familiar, it's implicitly defining a new animation_node using "lerk_idle.dae", setting it to loop, presumably using the entire timeline of that file, rather than a small section, as there are no "to" or "from" keywords used. Following that are 3 animation_node names of animation_nodes that have already been explicitly defined at an earlier point in the file.

So here, we're combining the "lerk_idle" animation, which is just the lerk sitting on the ground, breathing, with the #aim node, which allows the "body_pitch" parameter to determine how up or down the lerk is looking, combined with the #turn node, which allows the lerk to look left and right, combined with the #flinch node, which will make the lerk flinch when they're being damaged.

Here is the form to follow for declaring sequences and animations in the model_compile file.

Sequence definition

```
      animation [sequence name] [animation node definition]
```

Explicit animation node definition

```
     animation_node [animation_node name] [animation node
definition]
```

Animation node definition (animation-type) (must follow either sequence definition or explicit animation node definition, cannot be started independently)

```
     [file name to sample]
```

(the following are all optional, and in no particular order, but no more than once each).

`-[speed] [multiplier value]`*- multiplies animation frame rate by value provided.*

`-[relative_to_start]` *- transforms all bones to be relative to their starting transformations.*

`-[from] [frame_number/tag name] [to] [frame_number/tag name]` *- crop animation timeframe to this duration.*

`-[loop]` *- makes the animation loop. Shocking, isn't it?*

Animation node definition (layer-type) (must follow either sequence definition or explicit animation node definition, cannot be started independently)

```
     [layer] [{]
          [animation_node name OR animation_node definition]
```

(repeat for as many nodes as you desire)

```
     [}]
```

Animation node definition (blend-type) (must follow either sequence definition or explicit animation node definition, cannot be started independently)

```
     [blend] [blend_parameter name] [minimum value] [maximum value]
     [wrap] - optional
     [{]
[animation_node name OR animation_node definition]
```

(repeat for as many nodes as you desire. The blend will evenly space out the nodes and their respective blend parameter values.)

```
     [}]
```

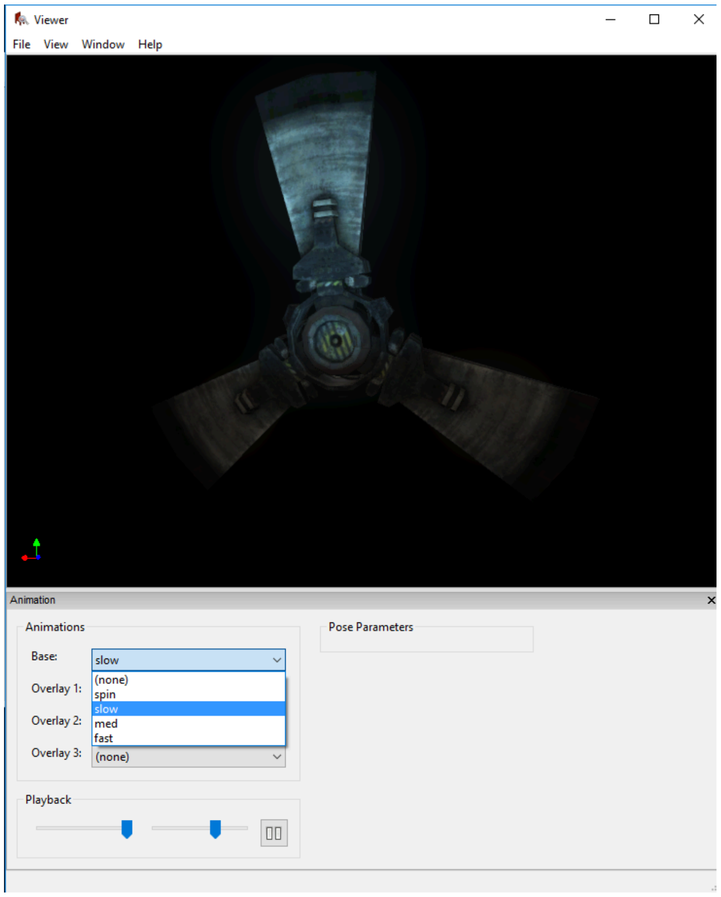
I hope this has at least shed some light on the animation system. If you wish to study the animation system further, I suggest you open up one of the alien models in the spark viewer, and examine the corresponding model_compile in notepad (or notepad++!).

Animation Graph editor

There is one final step to getting your model animated, and playing back in-game. It's not enough to simply have the sequences set up correctly in the model_compile file, we still need to hook these up with an "animation_graph" file.

The way the two files work together is when you load an animated model into a level with the Spark Editor, it will look for a ".animation_graph" file with the same name. Otherwise, the animations will not play in game.

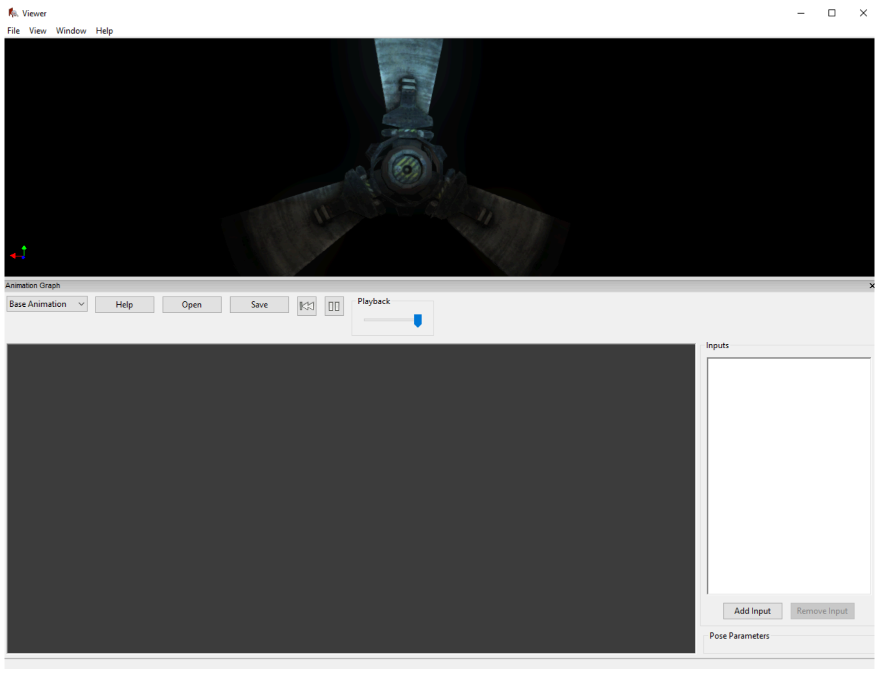
Animation_graph files are created with the Spark Viewer program. Open "ns2/models/props/refinery/refinery_fan_blade_01.model" with viewer. You may have to hide the ground plane with the View > Ground option.



If you choose Window > Animation, you will be able to view all the sequences contained within the model file you've loaded. Here, we can see the fan has 4 animations: spin, slow, med, and fast.

Close the animation panel for now. Let's create an animation graph for this fan.

Choose Window > Animation Graph to open the animation graph editor. Arrange the windows however you want, and you should now see something like this:



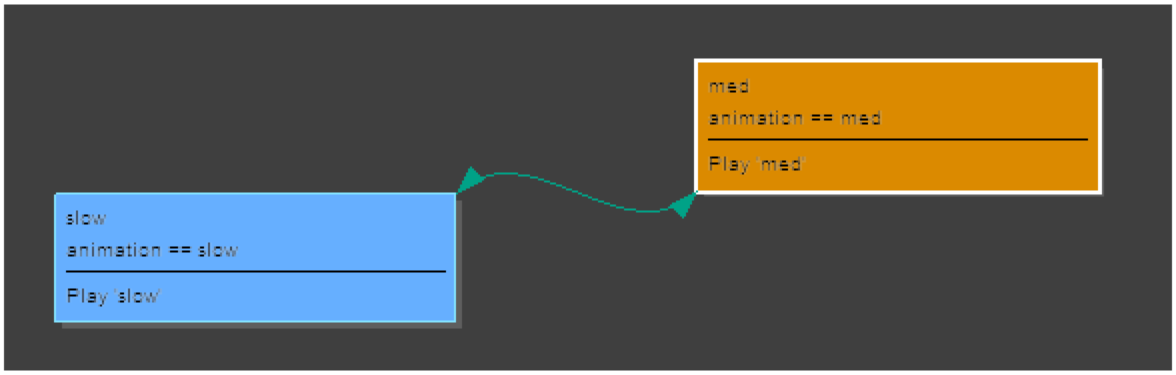
Choose “add input”, set the name to “animation” (without quotes), set the type to “enumeration” and for the enumeration, enter “slow, med, fast” (without quotes), and press OK.

What this has done is create a variable named “animation” that we can reference within the node graph to determine how the fan behaves. Note that “animation” is a special name that is changed automatically depending on the “Animation” field in the prop_dynamic object in the editor. For example, if we enter “fast” in the prop_dynamic object in the map editor, it will automatically alter this variable’s value to match it. There is only one other input type that is automatically modified without modding in this way, the “powered” input. This input is a boolean type, and is set to “true” when a power node in a location is built, and set to “false” when a location’s power node is unbuilt or destroyed. You can see an example of this in Repair on Tram. (“ns2/models/props/refinery/refinery_tram_repairarm.animation_graph”)

Now, let’s add a new node to the graph, do this by right clicking in the grey area.
Double click the node to open its properties menu.
We’ll label it “slow”, just for clarity’s sake -- it really makes no difference. Double-click “<New Condition>” and type “animation == slow” (without quotes). That’s not a typo, we use two ‘=’ signs to test for equality.
Then, select “Play Animation” as the action type, and choose “slow” from the animation menu. Click OK.

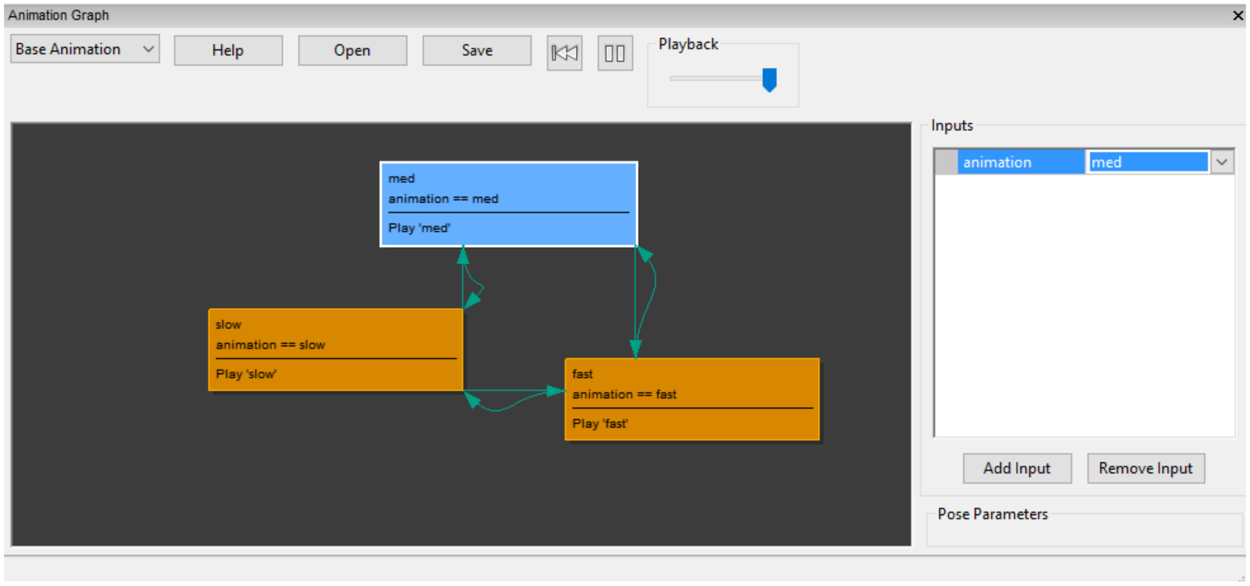
Right click the node you've just created, and choose "Set as Start Node", and click the rewind button. You should now see the fan blade slowly rotating in the viewer.

Create a node for "med" now, just the same as you did for the slow node. Now, we can connect the two nodes by Shift-click and dragging from one to the other. Control flows in the direction of the arrow connecting two nodes, but two nodes *can* be connected to each other in both directions simultaneously. Do this now.



Now, you can change the "animation" input in the drop down box to "med", and the fan speed will change! This is because the active node, the one highlighted in blue, was "slow", and it is searching through its connections for another node it can jump to. The only connection *from* "slow" is to "med", and since "med" requires that "animation" equals "med", it was unable to make the jump until you changed the value to "med". Once it makes the jump, it cannot jump back until the input value is changed back to "slow".

Try creating the node for "fast", and connecting it up to the others.



animation	med

The fan is now able to jump freely between fast, med, and slow speeds as you change the "animation" parameter.

If you make a connection or node that you do not want, you can select them and delete them. For connections, you have to move your mouse before the change appears.

You can save your work with the “save” button of course… just make sure you use the same name as the model file it is supposed to be used in conjunction with, otherwise it won’t work. Also, be sure to save often, as this program is prone to crashing.

For more complex examples of animation_graphs, see the repair arm mentioned above, or any of the alien or marine graphs.

Blender Workflow

The models exported by the blender plugin are identical to those exported by FBX or DAE methods, so the workflow is very similar as well.

The key difference between the workflows is that, while the FBX and DAE methods function using these intermediate file formats, the Blender exporter works by feeding the .blend file directly into the builder! Contained *within* each .blend, however, are facilities that mimic the file structures you setup in the old pipeline.

For example, for a typical static mesh in the FBX pipeline, you'll have 3-4 files: 1 geometry FBX file, 1-2 physics FBX files, and 1 model_compile file. To create an identical model using the new Blender pipeline, you'll have the single .blend file, but within it you might have 3 different "scenes" specified, 1 for geometry, 2 for physics, and 1 "text-block" -- that is, a self-contained text file stored in the .blend file -- acting as the model_compile file.

model_compile differences

With the FBX and DAE pipelines, you specify geometry and physics *filenames*. With the blender pipeline, instead of specifying a filename, you specify a *group name*, followed by a *scene name*. Both are required. Here, Blender scenes are analogous to the filenames in the old pipeline, and you can think of groups as being analogous to the selection when you would export from your 3d package.

Example:

```
    geometry "vis_group" "scene1"
```

Animations in Blender

For **`animation`** and **`animation_node`** directives, where we would have used a filename in the old pipeline, we instead use a blender scene name, and ONLY a scene name. We do not use groups with animations. Keep in mind that whichever scene you set as an animation MUST contain an armature object that uses the SAME DATA as the armature used in your "geometry" scene. Note, this means the object's "data" -- that is, the "armature" data-type it references, must be the same armature data referenced by the armature object in the "geometry" scene -- not just a duplicate that happens to look the same.

Also worth noting: you can place "timeline markers" in your animation scenes, and reference those in the same way you would reference "frame tags" in the DAE/FBX pipeline (from 3dsmax).

Example:

```
animation “my_cool_anim” “animation_scene” from anim_start to
anim_finish
```

This will create a new animation, called “my_cool anim” using the armature’s animation found in the scene called “animation_scene”, and using only the frames between the markers named “anim_start” and “anim_finish”.

Physics in Blender

For simple, static objects, it will suffice to just have your collision objects in the physics group and scene. The exporter will automatically run a convex hull generator on them.

For animated objects, there’s only one additional step: parenting the collision solids to bones. This is NOT the same as parenting to the *armature* and using the bones as deformers. The solids need to be parented to the *bones*, and the armature should not deform the meshes.

Also, keep in mind the one-solid-per-bone rule for the default collision rep if it is possible your model will be “ragdolled” at some point. Otherwise don’t worry about it.

If you ARE making your model “ragdoll-able”, then you will almost certainly need joints. This is quite simple to do in Blender. Create joints by creating “empty”-type objects -- I recommend using the “arrows” draw-type, as this shows you which axis is which. You can make these objects be treated as rigid body constraints by going to the object’s physics tab, and click the “Rigid Body Constraint” button. If you get an error saying “No Rigid Body WOrld to add Rigid Body Constraint to”, you can easily fix this by selecting one of your collision solids, and in the same category as the empty, click the “Rigid Body”, button. This makes the mesh act like a rigid body if you play on the timeline -- which doesn’t affect the export to spark, but does let you preview how the physics might interact.

Once your empty is set to a rigid body constraint, a number of options will appear. Only the following options are taken into consideration by the exporter:

- “Type”
- “Enabled”
- “Disable Collisions”
- “Object 1/2”

Spark supports angular constraints only, whereas Blender’s constraints are more generalized. Therefore, only the following “types” of joints are exportable:

- “Fixed” -- gives zero degrees of freedom on all axes.
- “Point” -- gives an unlimited freedom around all axes.
- “Hinge” -- gives a specified amount of freedom around the z-axis only.
- “Generic” -- gives a specified amount of freedom around any or all 3 axes. (Also has some linear constraint settings, which are of course ignored).

The “Enabled” setting on a joint can be disabled to make the exporter treat it as though the object didn’t even exist. There will be no connection between solids with a disabled joint.

The “Disable Collisions” setting means that collisions between the joint’s two solids is guaranteed to be disabled. However, turning off this switch does not guarantee that the two solids will collide, it simply means collisions aren’t forced off. To force collisions off, you’ll need to use a model_compile directive. Use the two boxes labeled “Object 1” and “Object 2” to specify the two collision objects. Note that the joint will be skipped if there aren’t two *valid* objects provided (eg they have to be solids, not just random blender objects.)

Attachment Points and Cameras

Attachment points are empty-type objects that are added to both the geometry group and scene, and also declared in the model_compile file, just like in the DAE and FBX workflows. Cameras work the same way too, but as usual do not have a model_compile declaration. Keep in mind for both attachment points and cameras: parenting them to an armature is NOT the same thing as parenting to a bone. Using an armature as a parent will have the same effect as leaving it unparented.

Multiple Models from a Single Blend

Sometimes, models are broken up into different pieces to make it more flexible for a mapper to use. For example,a catwalk prop might consist of a floor, a railing, an end-piece, and perhaps many variations of these pieces. Rather than creating a separate blend file for every single piece, you can utilize the “model_compile_list” text-block.

Here’s how it works: when the compile scripts are run, they first search for a text-block called “model_compile_list”. If it can’t be found, the script defaults to looking for the “model_compile” text-block, and proceeds from there exporting one model for the blend. If it *does* find a text-block named “model_compile_list”, it will move through it, making note of the output model name specified, followed by the name of the model_compile text-block used to compile that model. It continues searching through the file until no more entries are found.

Example:

```
    “catwalk_01.model” compile_cat1
    “catwalk_02.model” compile_cat2
    “catwalk_01_90.model” compile_cat1_curve90
```

This will result in 3 model files being created:

- catwalk_01.model, compiled based on the settings found in the “compile_cat1” text-block
- catwalk_02.model, compiled based on the settings found in the “compile_cat2” text-block
- catwalk_01_90.model, compiled based on the settings found in the “compile_cat1_curve90” text-block.

Note: Builder is designed to work with a 1:1 relationship between input files and output files. To work around this, builder will always report a failed build, BUT the error text will tell you the truth. Builder will always report a successful build when the “rebuild” option is selected, and no error text will be visible. This is only an issue if the model actually *doesn’t* compile properly, as you will have no indications otherwise. Long story short: read the text output of builder to determine whether a blend built successfully or not -- don’t trust the standard error output!

Alternate Origin

Sometimes, it can be convenient to create a model off-origin in Blender -- for example if you were creating a prop matching up precisely with geometry imported from the editor, unlikely to be working directly over the world origin in such a case -- and have the exporter re-orient the world such that your workflow doesn’t need to change, but the model is still exported such that it is reasonably well-centered on the world origin.

You can use the “alternate_origin” directive in a “model_compile” text-block to tell the exporter to re-orient the world such that the object specified is the new origin. This will not only work for translation, but also rotation and scaling. For example, if your “alternate origin” object has a scale of 10, and a rotation of 45 degrees, the model that gets built will be one tenth the scale it is in blender, and will be rotated -45 degrees to compensate.

Textures/Materials

Blender will automatically attempt to name the Spark-material to be referenced in the file, based on the names of the texture maps used in the Blender-material. Usually, it will just grab the albedo map, and figure out a texture path from there by changing the file extension to “.material”, and call it a day. If you would like full control over the Spark-material name that each Blender-material is converted into, you’re in luck! To explicitly tell Blender the path its corresponding Spark-material is to have, create a custom property in the *material* (not the object, the material), name it “SparkMaterial” (no quotes, and that IS case sensitive), and change the value to the path of the material. For example “materials/refinery/tram_floor_clean_01.material”.
Note: At the time of writing this, Blender won’t allow you to type in a string directly into the custom property when you first make it. You have to click the property’s “Edit” button, and in there you can type in a string -- don’t use quotes.

Miscellaneous Topics

Tangent Space

The new FBX compiler in Spark will attempt to use the exported tangents/bitangents from the files it is given. This means that if your normal map looks correct in your 3d app, it should look correct in Spark. The DAE compiler, however, will always re-generate tangents regardless of what data was available. This generation is NOT equivalent to Mikkt-space. Normal maps will likely appear incorrect or warped. DAE compilation is really only provided as a backwards compatible method; it will not be updated in the future. We recommend you use the FBX format for models.

Cameras

Cameras exported with the FBX geometry file will automatically be imported and compiled with the scene.

Maya Users

If you're having trouble compiling an FBX file, ensure you've deleted all the non-deformer history on your meshes. This causes problems with the FBX exporter, and could potentially crash the program. For example, if you delete a polygon on your mesh and don't delete the "deleteComponent" history node, the maya fbx exporter won't export the FBX correctly, and the model compiler won't know what to do with it.