SMF documentation

SMF v0.99, docs v1
by Sindre Hauge Larsen
2017 - 2020

The SMF system for GameMaker Studio 2 allows you to breathe life into your models, using a skeletal hierarchy and dual quaternion vertex skinning for animation. This document will first and foremost give an overview on how to use the system, as well as go into some detail for the ones of you who are particularly interested.

The system is built upon the foundations of the SMF system, but it has been expanded and optimized greatly, and the bloat has been taken out. **The model tool's only purpose is now to make it easy for anyone to import and animate their models.** Materials and the basic level editor have been removed for now, and collision buffers are now their own separate system.

The system does not support any standard animation formats, but instead uses its own open-source format. It is as such aimed at developers who do not have much experience animating models from before, but who'd still like an easy way to draw animated models in their GameMaker Studio 2 games.

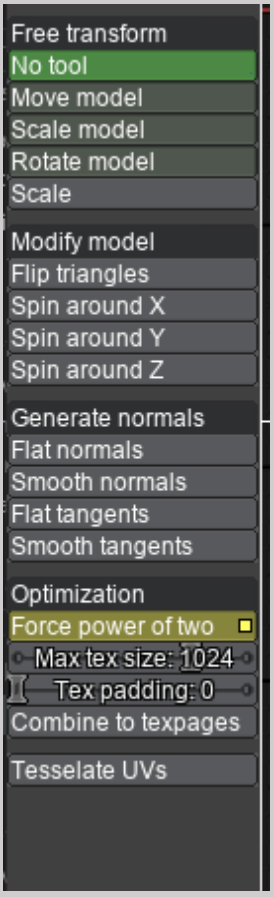
Contents

Animating a model in the Model Tool

Load model

When you open the Model Tool, you start in the Model editor tab. Load your model by pressing “Load model”, all the way to the right of the screen. This supports the following formats:

- OBJ: Loads the OBJ file as one sub-model per material. It will also look for an MTL file with the same name, as well as any referenced textures.
- ZIP: Unpacks the ZIP in a local folder, loads the contents, and deletes the local folder. The contents of the ZIP must be of the supported formats.
- SMF v7: This is the same format as used by the previous version of the model tool. This format is still in use.
- SMF v1: This is the format produced by the very first few versions of the model tool. This version of the format is the latest version still supported by the importer for GMS 1.4. Support for this format was added for backwards compatibility.



Edit model

There are a couple of simple tools to edit your model on the left of the screen in the model tab. Here you can find tools to move, scale and rotate your model, as well as the possibility to generate smooth and flat normals and tangents. Tangents will be encoded into the colour attribute of the model.

Once your model has loaded, its submodels will show up in a list on the right side of the screen. Here you can reorganize the order of the submodels, change their texture indices, merge, delete or hide them. If there are too many models to fit on screen, you can scroll through them with your mouse. Right clicking a selected model will give you some more options.

“Combine to texpages” will put the textures of the currently selected models onto texture pages, and will combine their vertex buffers. This is useful for reducing the number of draw calls, and could speed up your game.

Rigging

A rig is in a sense the skeleton of the model. When creating the rig, you create it in what's called the "bind pose". The bind pose is the resting pose of your model, where it has not been altered by the rig yet. In order to make rigging and skinning easier, it's often useful to make humanoid characters in a T pose with arms straight out for the bind pose.

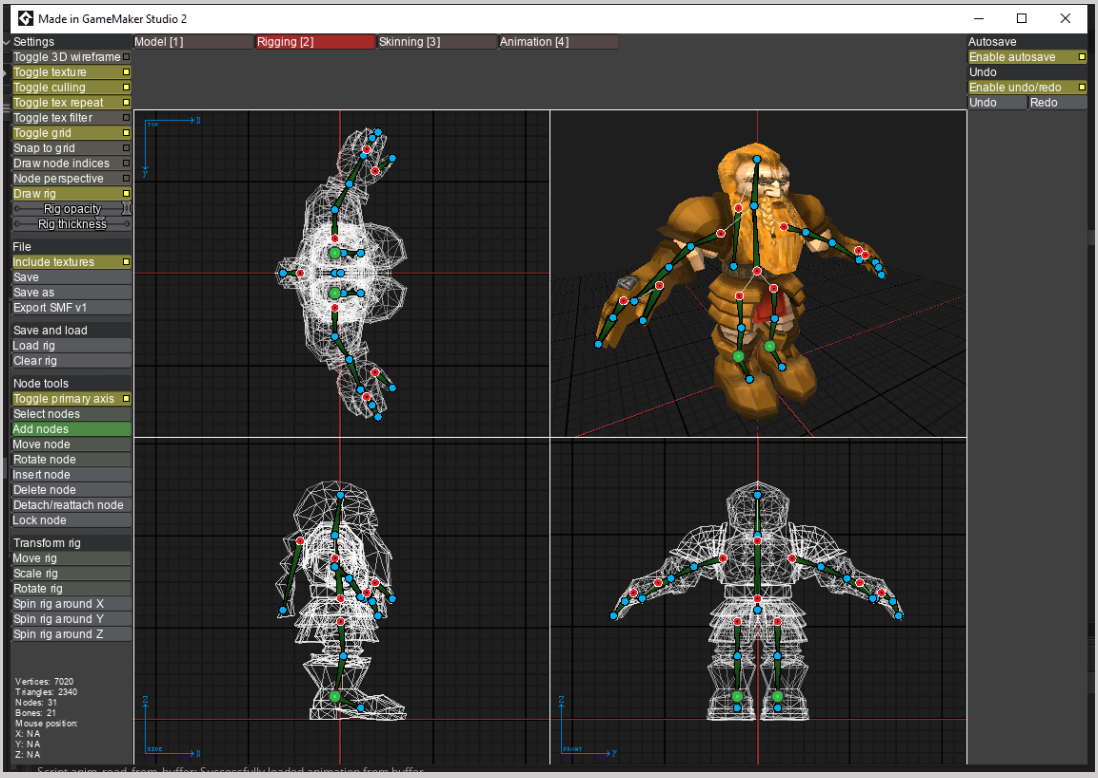
When you open the Rigging tab, the default selected tool is "Add nodes". Click anywhere in any of the 2D views to add nodes. When adding new nodes, they will create a bone between the previously selected node and the new node. You can move previously created nodes by clicking on them twice, or by using the "Move node" tool.

Important: Nodes can be rotated around their own axes. The orientation of the node is represented as red, green and blue arrows at the tip of the node. Click the node with the "Rotate node" tool active, and move the mouse right or left to rotate the node around its own axis. Red denotes the node's up-axis, green the to-axis, and red the side axis. Rotating the bones to sensible directions becomes useful once you start animating the rig.

Not all nodes represent bones. The first node has no parent, and thus cannot represent a bone. Also, you can manually detach a node from its parent by clicking "Detach/reattach node" to make it not represent a bone. Only bones need to be sent to the shader, and this is a useful tool to reduce the number of variables that need to pass on to the GPU.

You can toggle what's called the "Primary axis". This is the first axis to rotate around when doing inverse kinematics. If you do not edit this, the program will make an educated guess, so this is not necessarily something you need to pay attention to. IK is done by first rotating around the primary axis, and then a perpendicular secondary axis. Setting the primary axis manually will change how the IK behaves.

A node can be "locked" in place. The button for locking a node is present in both the rigging and animation tabs. A locked node will attempt to stay in the same position as the rest of the model moves. This is particularly useful for making sure feet stay on the ground.

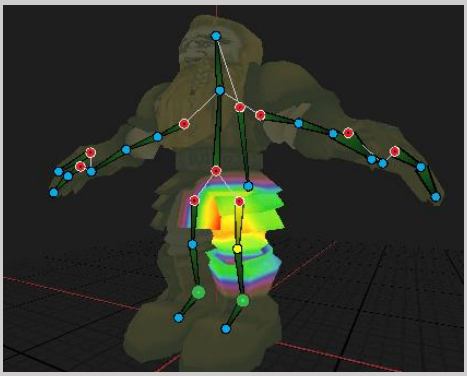


Skinning

When skinning a model, you tell the vertices how they should follow the rig. Each vertex can follow up to four bones.

There are two ways to skin a model, and you'll typically want to use a combination of both.

- **Autoskinning**: Automatic skinning loops through all the vertices of your model and compares the distance to the bones in the rig. The four nearest bones are added to the vertex' bone list. In order to figure out how much the bone should influence the vertex, it raises the distance to the bone to a power. You can select the power in a slider. The inverse of the result defines the weighting of that bone. Once it has the weights of all four bones, it normalizes them so that the sum of weights equals 1. You can also select an additional weighting influencer called "Normal weight", which takes normals into account when weighting bones. This makes bones that are on the back side of a vertex be weighted more. If the bones of the legs are very close, for example, it makes it more likely that vertices are weighted more towards the bone that is inside the leg than the bone of the other leg.
- **Manual skinning**: You can select bones by clicking the node at the tip of the bone. When you have a bone selected, you can see how the vertices are weighted to this bone. You can paint weights directly onto the model in the perspective view, using one of the following tools:



    1. Set paint – Makes weights smoothly gravitate towards the selected paint weight
    2. Additive paint – Adds paint weights to the model, gradually increasing the influence of the selected bone
    3. Subtractive paint – Gradually reduces the influence of the selected bone
    4. Autoskin paint – Lets you autoskin only parts of the model. This tool does not care which bone is currently selected, it will loop through all bones and figure out their influence based on the distance.
- The following sliders will affect the skin paint tools:
    - Paint radius: The radius of the brush. The actual value is a bit obscure, but you can think of it as a relative scale where 0 is just the triangle under the mouse, and 20 is "a large area".
    - Paint weight: The weight of the brush.

A typical workflow is to start off by autoskinning the entire model, followed by vertex painting to fine-tune the skinning.

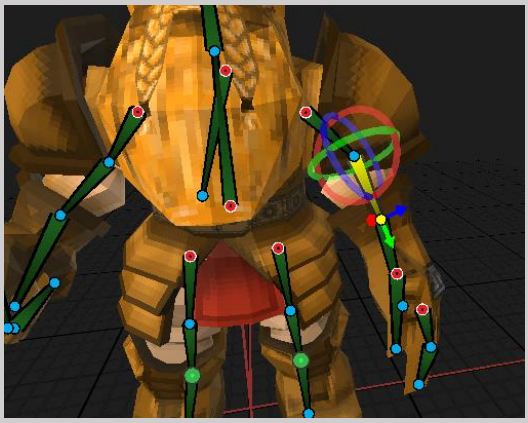
Animating the model

Once in the Animation tab, you can finally make your model move. Animations consist of a list of keyframes, where each keyframe stores the orientation of each bone at that given time. You can see the animation timer at the top of the screen, just below the tab buttons. Create an empty animation by clicking “New animation” on the right toolbar. To add more keyframes, simply double-click somewhere on the animation timer.

To the left of the timeline, you can select one of three tools that will affect how new frames are created: “Bind” will add blank keyframes to your animation, “Insrt” will attempt to interpolate between neighbouring keyframes, and “Copy” will copy the currently selected keyframe.

When editing a keyframe, you have several different tools:

- Drag node: This is the default selected tool. Click a node in any view and drag it with the mouse.
- Move node IK: Similar to “Drag node”, but this one will perform inverse kinematics if both the selected node and its parent represent bones.
- Rotate node local: In order to rotate a bone, select the bone you’d like to rotate, and drag one of the three rotation rings. The orientation of the rings depends on the rotation of the node in the rig. This is why it’s useful to rotate bones to sensible orientations while rigging. If you’d like more info on this, go back to the “Rigging” tutorial.
- Rotate node global: Lets you rotate the node around world x, y and z.



You can set the following properties of an animation:

- Play time: Changes how fast the animation is played in the editor. This does not automatically affect the speed of the animation in game, but you can read this value and change the animation speed yourself.
- Enable looping: This affects how the animation is sampled towards the end of the animation. If looping is enabled, it will interpolate between the last and the first keyframe. If it is disabled, it will only show the last keyframe towards the end of the animation.
- Interpolation: There are two options:
    - Linear: Linearly interpolate between the nearest two frames. There’s practically no speedup to gain from selecting this option, so only use it if you’d like “stiff” animations.
    - Quadratic: Quadratic interpolation between the three nearest keyframes. The extremes will be rounded a bit, meaning that the animation will “cut short” large movements, but the animations will look smooth.
- Frame multiplier: When creating samples in real time, the samples aren’t actually created directly from the keyframes. Instead, a bunch of samples are pre-computed using the animation settings, and will be linearly interpolated between. The number of pre-computed samples is (number of keyframes) x (frame multiplier). For an animation with 3 keyframes and a frame multiplier of 10, it will generate 30 pre-computed samples. The more pre-computed samples, the smoother the animation will look, at the cost of memory. You can see the times of precomputed samples as blue lines in the animation timeline.

Using SMF models in a game

Load the model

To load your model into the game, you must save the model as SMF v7, and import it into the included files in your project. Once ingame, you can load the model with the following function:

```
model = smf_model_load(fname);
```

For your game to work in HTML5, you need to load the SMF files as buffers using buffer_load_async and loading the model from the buffer like this:

```
model = smf_model_load_from_buffer(buffer);
```

"Model" will now be a handle containing the model's vertex buffers, its textures, rig and animations. Submitting it using smf_model_submit at this point will simply draw a static model. Note that the SMF format cannot be drawn without a shader, since its format differs from GMs default format.

Animating the model

There are several ways to animate an SMF model. The old way, which was used in previous versions of SMF, was for the user to keep track of time, generating samples and interpolating between them. A lot of this has now been automated into what's called an "animation instance". The old method is still available, but this tutorial and its demos will focus on the new method.

An animation instance is a resource that can be created and referenced. It is an array, and as such, it will automatically be garbage collected by GMS2 once it's no longer needed. The animation instance will keep track of time for you, and will smoothly interpolate between animations, unless you tell it otherwise. Playing an animation using animation instances requires a minimal amount of code. Once a model has been loaded, this is all the code that is needed to create an animation instance and start an animation:

```
inst = smf_instance_create(model);
var animName = "Idle"; //The name of the animation, as defined in the model tool
var animSpeed = .01; //The normalized speed of the animation
var lerpSpeed = 1; /* The speed at which the animation instance will interpolate from the previous animation to the new one. Since this is the first animation that is played, this can be set to 1, which means it will happen instantly. */
var resetTimer = true; /* Whether or not to start the animation from the beginning, or continue where the previous one left off */
smf_instance_play_animation(inst, animName, animSpeed, lerpSpeed, resetTimer); /* Start the animation */
```

The animation instance also needs to be updated once each step. Here you can define the time step for each update. For a typical game you'll want to set this to 1:

```
var timeStep = 1; //Increase the timer by 1 each step
smf_instance_step(mainInst, timeStep); //Update the animation instance
```

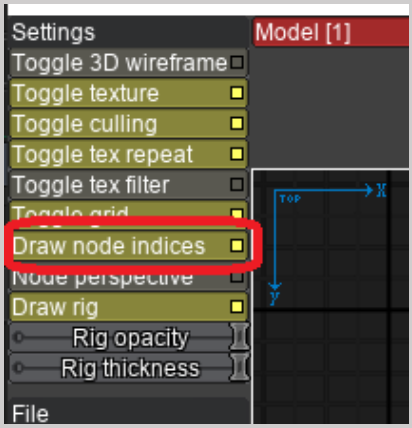
And finally, drawing the animation only requires setting a compatible shader, and drawing the instance:

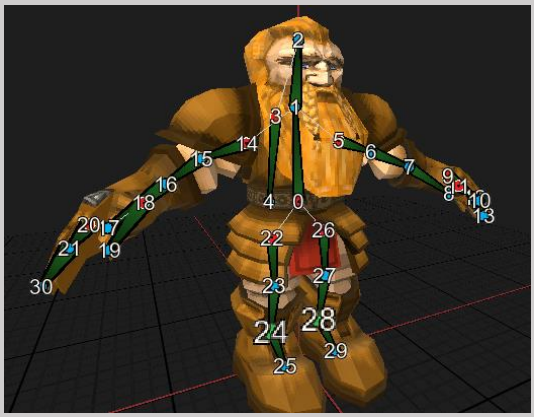
```
shader_set(sh_smf_animate); //Set a shader compatible with the SMF format
smf_instance_draw(mainInst); //Draw the instance (this also handles animation)
shader_reset(); //Reset the shader
```

Advanced animation usage

Animations can be edited in real time. This is useful for things like head turning and foot placement. Transforming a node will also move and/or rotate its descendants, making sure bones always stay connected.

When editing an animation, you need to supply the index of the node you'd like to rotate. This index can be found by enabling "Draw node indices" in the upper left corner of the model tool.





The real-time animation editing scripts can be sorted from fastest to slowest in the following way:

- smf_instance_node_rotate_x/y/z is the absolute fastest way of editing an animation. When rotating around one of the cardinal axes, a lot of factors cancel out when multiplying dual quaternions, resulting in faster calculation.
- smf_instance_yaw/pitch/roll/rotate_axis is slightly slower, since no factors cancel out. These allow you to rotate bones around its own axes and arbitrary axes.
- smf_instance_node_drag lets you move nodes around. If the node is detached, the node and its children are just moved, otherwise it will rotate towards the given position.
- smf_instance_node_move_ik_fast works similarly to the previous one, except that if both the given node and its parent are bones, it will perform a fast inverse kinematic operation to try to reach the given position. This IK algorithm works best for small adjustments like foot positions and the like, and may twist bones into weird positions if moved too far.
- smf_instance_node_move_ik is more stable than the previous function, and can be used for larger movements. The algorithm rotates the nodes around two perpendicular axes to find the new orientation of the given node and its parent. The primary rotation axis is created automatically using an educated guess, but can manually be edited in the rigging tab of the Model Tool.

Editing an animation must always be done *after* updating the animation instance with smf_instance_step.

You can also combine different animation instances. There are two ways to do this:

- smf_instance_lerp will interpolate between two instances. The instances can play different animations, but must use the same rig structure. Note that the node positions will be correct at both endpoints, but bones may detach during the interpolation if there's a large difference between the endpoints. This is usually not a problem, but good to be aware of.
- smf_instance_splice_branch will copy over the given node and all of its children from one instance to another, and modify it so that it is attached to the rig of the destination instance. This is a powerful tool that for example will let you play separate animations for the upper and lower bod|y of a character.

The specifics

Ready to go down and dirty into the details? Okay, feel free to! Here I will attempt to detail the things that go on behind the scenes to make skeletal animation possible.

The model container

The rig and its animations are stored in the SMF model container that can be created with smf_model_create. The model container contains the following entries:

- **mBuff**: An array containing the buffers of the SMF model. An SMF model can consist of many sub models, and each of them can have their own texture.
- **vBuff**: An array containing the model's vertex buffers.
- **Visible**: An array that says whether or not each given submodel is visible
- **TexPack**: An array containing the textures each vertex buffer should use. Note that including textures in an SMF file works well for proto-typing, but for a finished project, all textures should be included in the project. GMS2 uses 4x the amount of memory for externally loaded textures.
- **Rig**: The index of a rig (a rig is a resource in its own right, detailed later in the document)
- **Animations:** An array containing the model's animation indices
- **SampleStrips:** An array containing strips of pre-calculated samples for each animation

In addition, it contains the following auxiliary entries, which are used when needed:

- **SubRigs**: A partially functioning feature that is not complete yet. The rig can be split up into smaller rigs of up to 16 bones, and the format of the model simplified to work better on mobile and HTML5. Improvements to this system are planned.
- **SubRigIndex**: An array containing the subrig index of each submodel
- **Partitioned**: Whether or not the model has been partitioned into smaller subrigs. Partitioned models can not be loaded back into the Model Tool.
- **Compatibility**: Whether or not the format of the model has been changed to the compatibility format. Compatibility models can not be loaded back into the Model Tool.
- **AnimationMap**: A ds_map mapping the name and indices of the animations in the animation array

The rig

A rig is a ds_list that details the node hierarchy, positions and orientations of the bind pose. It contains the following entries:

- **NodeList**: The node list contains the following info for each node:
    - **WorldDQ:** The node's object-space orientation, stored as a dual quaternion.
    - **LocalDQ:** The node's orientation in relation to the parent's orientation.
    - **Parent:** The index of this node's parent.
    - **IsBone:** Whether this node represents a bone.
    - **Length:** The length of the bone. This is 0 if the node is not a bone.
    - **WorldDQConjugate:** The conjugate of WorldDQ
    - **LocalDQConjugate:** The conjugate of LocalDQ
- **BindMap**: A ds_list that maps node indices to bone indices. Nodes that aren't bones are mapped to -1. This is updated automatically any time you add or remove bones.
- **BoneNum**: This is the number of bones in the rig. Note, not the number of nodes, but the number of actual bones.

The animation

An animation is a ds_list containing the following entries:

- **Name:** The name of the animation
- **Loop:** Whether the animation should loop
- **Interpolation:** The type of interpolation that should be performed. The following constants can be used:
    - **eAnimInterpolation.Keyframe**: No interpolation is performed, the sample of the nearest keyframe is returned
    - **eAnimInterpolation.Linear:** Linear interpolation between keyframes.
    - **eAnimInterpolation.Quadratic:** Quadratic interpolation between keyframes
- **PlayTime**: The time it takes to complete the animation, in milliseconds. This does not automatically affect the animation speed, but can be used to manually change the speed of the animation.
- **SampleFrameMultiplier**: A positive integer that indicates how many samples should be pre-computed. The resulting number of frames equals (number of keyframes) x (frame multiplier).
- **KeyframeGrid**: A ds_grid containing the keyframes and their timing. For a given keyframe, the time of the keyframe is keyframeGrid[# 0, keyframe]. The keyframe itself is found at keyframeGrid[# 1, keyframe]. The keyframe is an array containing dual quaternions that describe the change in local orientation from bind pose to keyframe.

The animation instance

An animation instance is an array containing the following entries:

- **ModelInd**: The instance's model index
- **Rig**: The instance's rig index
- **CurrAnim**: The animation that is currently playing. Note that this is not the actual animation index, but its index in the modelInd's animation array.
- **AnimSpeed**: How fast the animation should be playing
- **Timer**: The current time of the instance. This loops around and is always between 0 and 1.
- **Lerp**: A value storing how far we've interpolated between two animations
- **LerpSpeed**: How fast to interpolate between two animations
- **Sample**: The sample of the instance. This is an array containing info on how each bone has moved from bind to current pose. When drawing an animated model, the sample tells the shader how to move each vertex.
- **PrevSample**: The sample of the previous animation. Useful for interpolating between animations.
- **Smooth**: Whether to enable interpolation between samples. Not interpolating is faster but may look a bit "choppy".
- **FastSampling**: Whether fast sampling is enabled. When it is enabled, there will be no interpolation between samples. Instead, samples are returned straight from the sample strip. This takes zero toll on the CPU, but the samples returned this way cannot be edited, otherwise the samplestrip itself gets altered as well.
- **BackupSample**: When enabling fast sampling, the instance's own sample is saved as a backup sample, so that we can easily switch back without having to create a new sample from scratch.
- **NewAnim**: When switching animations, this stores the index of the new animation.

The samplestrips

Generating a sample from an animation is a relatively slow process. It is fast enough to be done in real time, but at the risk of reducing the performance of the game. The SMF system tries to circumvent this by pre-computing a bunch of samples and linearly interpolating between them. This results in super-fast animations at the cost of precision.

If you need a higher precision, there are two ways to do this:

1. Increase the SampleFrameMultiplier. This will make the system generate more samples per keyframe. If the frame multiplier is 10, it will pre-compute 10 frames for each keyframe. This increases memory usage.
2. Normalize your samples. You can use a script called smf_sample_create_normalized, which creates a sample and then normalizes all the dual quaternions within it. This is a bit more resource intensive.

In short, weigh the pros and cons and figure out what solution is best for your models.

It is, of course, still possible to generate new samples directly from an animation with smf_animation_generate_sample, but I recommend avoiding this.

Thank you

Developing this tool has taught me a lot. Countless hours have been spent trying to find ways to improve and optimize each and every piece of code. There are still a few things that need to be done before the project can be considered complete, but it is getting close.

If you have any trouble or questions, or would like to share something you've made with this system, please do so at the GameMaker Community topic at the following link:

https://forum.yoyogames.com/index.php?threads/19806

Sindre Hauge Larsen