# Animations

Many models gain a lot when they are not static but have some moving parts. Be it a door, a window, a revolving beacon, a windmill rotor or a retracting landing gear, they offer a whole new world of options.

And here´s how they are written.

### 1. Introduction, or "What is an animation?"

Basically, an animation is about moving, resizing or turning parts relative to others. A "part" in this respect can be a mesh or an entire group; those have only minor handling differences. In order to define an animation, there are two questions to be answered: "When?" and "Where?" - the game needs to know when the part to be animated has to be where.

### 2. Keyframes and transformation matrices

There are two basic options to explain this to the game. The easiest to wrap ones head around is the keyframe; the matrix is a bit more powerful but a little encoded. Let us examine a keyframe first.

Code

```
1. {time = 0, rot = { 0,0,0}, transl = {0,0,0} },
```

This line answers the previously posed questions of "when" and "where". "Time" defines the time elapsed since the animations start, its unit are milliseconds. "rot" defines the parts rotation about the Z, Y and X axes in degrees, and "transl" tells the game the parts shift along the X, Y and Z axes. Of interest here is that the sequence differs between "rot" and "transl"; why this is the case would probably be a question to ask UG. I cannot answer it. So let us live with it for now.

A transformation matrix looks like this:

Code

```
1. transf = {1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, },
```

This is a bit less obvious, but let us look at it anyways. Coded here are the rotation and scale of a part by virtue of the sine and cosine of the respective angles, influenced by the scale about each axis. Calculating such a matrix is best left to the electronics; the lexicon contains a calculator that I find extremely helpful.. One thing is worth noting, as it is easy to edit manually: the last four figures define the parts position along the XYZ axes and the general scale.

### 3. Movement

One single keyframe does not induce a movement, of course. Movement is the change of relative postion over time, therefore we need at least two keyframes or matrices to get our part going.

Let us start with an easy animation. If we have a part that needs to be pushed one meter upwards in one second, the keyframes to achieve this will look like this:

Code

```
1. {time = 0, rot = { 0,0,0}, transl = {0,0,0} },
2. {time = 1000, rot = { 0,0,0}, transl = {0,0,1} },
```

At time 0, the part is lying in its original, pristine state at its original position. Then it will immediately move upwards so it reaches the position 0,0,1 = one meter up the Z axis at time 1000ms = 1 second.

And from this, everything else can be deduced and rather complex animations defined. Just add as many keyframes as needed, answering the original question of "when" and "where" for as many points in time as needed.

If we want to move the part along a square pattern, one meter up, one meter left, one meter down and then one meter right again, each within one second, it would look like this:

Code

```
1. {time = 0, rot = { 0, 0, 0}, transl = {0, 0, 0} },
2. {time = 1000, rot = { 0, 0, 0}, transl = {0, 0, 1} },
3. {time = 2000, rot = { 0, 0, 0}, transl = {0,-1, 1} },
4. {time = 3000, rot = { 0, 0, 0}, transl = {0, -1 0} },
5. {time = 4000, rot = { 0, 0, 0}, transl = {0, 0, 0} },
```

Note that the shift as defined in "transl" always refers to the parts original position, not its previous one: there is no need to keep track of shifts, sum those up manually or chastise oneself with anything else - it´s really simple...

### 4. Integrating the animation to the .mdl

The animation is linked to a part in the .mdl. Keeping with the above simple animation, it will look like this:

Code

```
1. {
2. animations = {
3. forever = {
```

```
4.  params = {
5.  keyframes = {
6.  {time = 0, rot = { 0,0,0}, transl = {0,0,0} },
7.  {time = 1000, rot = { 0,0,0}, transl = {0,0,1} },
8.  },
9.  origin = { 0, 0, 0, },
10. },
11. type = "KEYFRAME",
12. },
13. },
14. materials = { "Material.mtl", },
15. mesh = "Mesh to be animated.msh",
16. transf = { 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 5,5, 5, 1, },
17. },
```

Display More

Let us look at this in detail. The two curled brackets that enclose the definition of a part in the .mdl are already known and present here as well, so the block is traditionally opened with a {. "animations={" opens another block that (logically) contains the definitions of the animations the part is required to perform. "forever={" is the trigger that fires the animation; I´ll come back to this later - for now, let us just note that this is where the game is told how and when to start the animation. "params ={" and "keyframes = {" tell the game to expect keyframes, this has to be written down just like this.

And then, we find something we remember from earlier: the two keyframes that move the part up one meter within one second. After these keyframes, the block "keyframes" is closed with a }, "origin=..." tells the game to start at the parts original position (just leave it like this with 0,0,0), and another } closes the "params=" block. "type = "KEYFRAMES" is a required reminder to the game so it remembers it has just read keyframes - there are other options I´ll discuss later. Finally, } closes the trigger and another } the "animations=" block.

The 3 lines "material", "mesh" and "transf" are not new, they are common to all parts called in the .mdl.

Finally, a last } closes the block calling for the mesh, and that´s it.

**5. The triggers**

I have briefly mentioned it and shall now expand it a little: an animation requires a trigger so the game knows when to initiate it. A beacon may well flash forever, a landing gear on the other hand should keep its position after retraction and not merrily move out, in, out and in again. Here´s how this is done.

| **Trigger** | **fires when...** |
|---|---|
| forever | the part is in the game and for as long as it is there. |
| open | when a door is opened |

| Trigger | fires when... |
|---|---|
| close | when a door closes |
| drive | when a model is driving. The animations speed is proportional to the vehicles speed. |
| close_doors | when all doors of the model are closed |
| open_doors | when all doors of the model are opened |
| open_doors_left OR _right | when the doors on the respective side are to open |
| close_doors_left OR right | when the doors on the respective side are to close |
| close_wheels | to retract an aircraft landing gear |
| open_wheels | to extend an aircraft landing gear |

Those triggers are entered where it said "forever" in the above example.

**6. Animations using an .ani file**

Long lists of keyframes tend to cause the .mdl to become rather long and unkempt. It is possible to relocate the animation into a file instead if some cleanliness is desired. Cleanliness is Fordliness after all. The respective block in the .mdl looks a little different in this case:

Code

```
1. animations = {
2. close_wheels = {
3. params = {
4. id = "Animation.ani",
5. },
6. type = "FILE_REF",
7. },
8. },
```

...is what precedes the 3 lines "material", "mesh" and "transf" in this case. Here, we see an animation intended to retract a landing gear: it is triggered by "close_wheels". The block "params=" does not contain the keyframes in this case but refers to a file under "id=" that the game expects to be in the directory res\models\animation. Here, it´s called "Animation.ani", it can be named freely though. Let the name be unique within the game to avoid cross contaminations.

And now, let us turn our attention towards the mentioned .ani file.

Code

```
1. function data()
2. return {
3. times = { 0, 1000, },
```

```
4. transfs = {
5. { 1, 0, 0, 0, 0, 0, -1, 0, 0, 1, 0, 0, 0, 0, 0, 1, },
6. { 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, },
7. },
8. }
9. end
```

Here, we find the mentioned transformation matrices. "times" lists a chain of points in time since the animation started, as usual in milliseconds. This answers the question of "When?". To tell the game "Where?", the matrices are worked off one by one, the first at the first time, the second at the second, and so on. Now, if we remember that the last four figures in the matrix are the XYZ coordinates and the scale, we immediately see that this .ani does exactly the same as the two initial keyframes: it pushes the part up by 1 meter within 1 second.

**7. Turning a part**

This is only a little more complex: if a part is to be turned, the movement has to be divided into appropriate chunks. The keyframes:

Code

```
1. {time = 0, rot = { 0,0,0}, transl = {0,0,0} },
2. {time = 2000, rot = { 0,180,0}, transl = {0,0,0} },
3. {time = 4000, rot = { 0,0,0}, transl = {0,0,0} },
```

...will **not** work: they just tell the game that the part is to be turned by 180° about the Y axis at time 2000, but do not specify a direction of rotation and therefore will not result in an agreeable animation.

So here´s what we do instead: we chop up the animation into a few steps.

Code

```
1. {time = 0, rot = { 0,0,0}, transl = {0,0,0} },
2. {time = 1000, rot = { 0,90,0}, transl = {0,0,0} },
3. {time = 2000, rot = { 0,180,0}, transl = {0,0,0} },
4. {time = 3000, rot = { 0,270,0}, transl = {0,0,0} },
5. {time = 4000, rot = { 0,360,0}, transl = {0,0,0} },
```

This will indeed and smoothly turn the part by 360° about the Y axis within 4 seconds.

Here, something else needs to be minded. TPF2 internally transfers keyframes into matrices, resulting in weird shifts in scale when the part travels between two keyframes: the part will pulsate a little, which is obviously unwanted. This can only be reduced, not eliminated unfortunately: the animation can be chopped up even further, thereby forcing the game to display the part properly as often as possible. In the above example, I´d recommend not placing a keyframe with a 90° turn every 1000ms, but use many more, moving the part by 9° every 100ms for example. This will obviously greatly increase the number of keyframes but result in a much smoother animation.

**8. Useful little tidbits**

There are situations in which a part can stand more than one animation. For example, a landing gear typically does not only retract but also extend, a door does not only close but also open. Also, for example the propeller on the beautiful Schienenzeppelin will run at speed when the vehicle is moving and run in idle when at the station. No problem at all: you can place more than one animation upon a part. In the latter example, writing one "forever" and one "drive" animation will result in the propeller speed proportional to the vehicles speed when it drives on the tracks; when it is standing at the station, the "drive" part is not used but "forever" is still active and that part of the animation still shows.

Other situations require parts to move a bit, then hold their position and then continue moving. For example a landing gear door that opens to allow the wheels out or in and then closes again would be animated something like this:

Code

1. {time = 0, rot = { 0,0,0}, transl = {0,0,0} },
2. {time = 1000, rot = { 0,0,90}, transl = {0,0,0} },
3. {time = 5000, rot = { 0,0,90}, transl = {0,0,0} },
4. {time = 6000, rot = { 0,0,0}, transl = {0,0,0} },

The door will then open from 0 to 1000ms, hold its position from 1000 to 5000ms (that time slot is then mirrored in the legs animation and used to fold it in), and then closes again from 5000 to 6000ms.

There are heaps of options - just try for Yourself now that You have read the basics!