Model Definition (.mdl)

Models are stored in the `.mdl` format in the `res\models\model\` folder. The format consists of geometric information for different levels of detail (LODs) and metadata. For each LOD, there is a separate hierarchy of meshes with their animations, events, and materials. In general, `.mdl` files contain a `data()`-function that returns a struct similar to the one below:

```
function data()
return {
  boundingInfo = { ... }, -- optional bounding box used e.g. for render borders
  collider = { ... },     -- optional collider for collision calculation
  lods = { ... },         -- geometric information for 3D data and textures
  metadata = { ... },     -- metadata depending on model type
  version = 1,            -- new Transport Fever 2 lod tree format
}
end
```

Bounding Box

The bounding box is a cuboid that encloses the outermost elements of the model so that all parts of the model lie within the box. It is used, for example, to decide whether an object is within the visible range of the camera or not.

```
  boundingInfo = {
    bbMax = { <+x>, <+y>, <+z>, },
    bbMin = { <-x>, <-y>, <-z>, },
  },
```

The values are the distances from model origin in positive and negative direction on all three axis.

Collider

The collider is used whenever the potential collision between models is considered. It's possible to use the mesh data for collision calculation as well as define the collision boundaries by script.

Collider from Mesh

When the mesh data should be used for collision calculation, the type “MESH” is needed:

```
  collider = {
    params = { },
    transf = { 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, },
    type = "MESH",
```

```
  },
```

Collider from script

When the boundaries should be set by script, the type “BOX”, “CYLINDER” or “POINT_CLOUD” is needed:

```
  collider = {
    params = {
      halfExtents = { 1.5, 1.5, 1.5, },
    },
    transf = { 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, },
    type = "BOX",
  },
```

The size of the volume is specified by 2 times the `halfExtents` properties for the “BOX” and “`CYLINDER`” type. It can be offsetted relative to the model origin with the `transf` parameter. The “`POINT_CLOUD`” uses a list of points that is provided in the `points` parameter. Each point has three values for the position on all three axis relative to the construction origin. The `transf` parameter is ignored.

Level of Detail

Levels of detail (LOD) are used to use simplified geometries with less tris as the distance between model and camera increases, thus saving computing power during rendering. At least one LOD is needed, but it is recommended to include more if the model has a large number of tris.

```
  lods = {
    {
      node = { ... }
      static = false,
      visibleFrom = 0,
      visibleTo = 200,
    },
    ...
  },
```

The node element is the top level of the mesh hierarchy in this LOD. The range of visibility is limited by `visibleFrom` and `visibleTo`.

Mesh hierarchy

The hierarchy is defined by nesting of nodes. The root node is the toplevel parent node.

Nodes

A node can have the following attributes:

```
  {
    children = { ... }, -- optional, contains more nodes
    transf = { 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, }, --
transformation matrix for size, ...
    animations = { ... } -- optional, for details see below
    mesh = "path/to/mesh.msh" -- optional, relative to res/models/
mesh/
    materials = { ... } -- needed if mesh is used, for details see below
    name = "mesh_01" -- optional, can be used as reference elsewhere
  }
```

Animations

The animations block contains a list of animations with eventnames as keys. Each of these animations is either defined by keyframes or file based.

Example code for a keyframe and a file based animation:

```
  animations = {
    eventname1 = {
      forward = true, -- or false for inverse keyframe playback
      params = {
        keyframes = {
          {
            rot = { 0, 0, 0, },     -- rotation around all three axis
            time = 0,               -- milliseconds since begin of animation
            transl = { 0, 0, 0, }, -- position offset for all three axis
          },
          ...
          {
            rot = { 0, 0, 0, },
            time = 1200,
            transl = { 0.75, 0.055, 0, },
          },
        },
        origin = { 0, 0, 0, }, -- offset of animation origin from node
origin
      },
      type = "KEYFRAME",
    },
    eventname2 = {
      params = {
        id = "path/to/animation.ani", -- relative to res/models/
animation/
      },
```

```
      type = "FILE_REF",
    },
  },
```

Further information on animation files can be found in the animation section.

Meshes and Materials

Nodes can contain a mesh. Meshes consist of two files, `meshname.msh` with index information and `meshname.msh.blob` with 3D model data (binary). See the documentation of msh files for further information on meshes.

For each submesh provided by the referenced `.msh` file, a material has to be referenced.

```
  materials = {
    "path/to/material.mtl", -- paths relative to res/models/material/
    "path/to/secondMaterial.mtl",
    ...
  }
```

Further details about material files and types can be found in the material documentation.

Metadata

There is large set of metadata keys to describe all the different types of models in the game. Some of the keys can be used for almost every type of model. These are described below. For specific model types, the keys are described in the relevant sections of the wiki:

- Vehicles
- Waypoints and Signals
- Landscape Assets (Trees and Rocks)
- Animals
- People

Universal Keys

These metadata keys can be applied to any or the majority of model types. Depending on the resource type, they might be mandatory, but can often be left empty to default to zero.

- `description`
- `availability`
- `cost`
- `maintenance`
- `particleSystem`
- `cameraConfig`
- `labelList`
- `transportNetworkProvider`

Description

The description provides the name and a descriptive text for the model. It is used for vehicle informations in buy menu and status windows. Static models that are not part of a construction use it for the label in the menu. In constructions, these information are not used.

```
    description = {
      name = _("Name of Model"),
      description = _("Description displayed for example in the depot menu")
    },
```

Availability

The availability defines the timespan in which the model is available. `yearFrom` and/or `yearTo` can be left out or set to zero to define an indefinite start respectively end year.

```
    availability = {
      yearFrom = 1925,
      yearTo = 1985
    },
```

Cost

To activate automatic price calculation, set the `price` to -1. When `price` is left out, it defaults to 0. The value of `priceScale` can be used to adjust the calculated price. This is a hook up point for balancing mods too.

```
    cost = {
      price = 10000,
      priceScale = 1
    },
```

Maintenance

To activate automatic maintenance calculation, set the running cost to -1. When `runningCost` or `lifespan` is left out, it defaults to 0. The value of `priceScale` can be used to adjust the calculated price. This is a hook up point for balancing mods too.

```
    maintenance = {
      runningCosts = -1,
      runningCostScale = 1,
      lifespan = 40 * 730   -- [1 unit at normal game speed corresponds to
12h, a year equals lifespan = 730]
    },
```

ParticleSystem

Particle emitters produce smoke or steam particles.

```
    particleSystem = {
      emitters = {
        {
          child = 1,
          position = { 4.3632001876831, 0, 3.8589000701904, },
          color = { 0.35, 0.35, 0.35, },
          frequency = 80,
          lifeTime = 14,
          size01 = { 0.80000001192093, 1, },
          velocity = { 0, 0, 10, },
          initialAlpha = 0.8,
          velocityDampingFactor = 2.0,
        },
        ...
      },
    },
```

A model can have zero or more particle emitters. Each of the emitters has its own block in the `emitters` list. It has several properties:

- `child` is the id of the node that should be used as origin for the positioning of the emitter.
- `position` is a vector relative to the origin of the mesh anchor.
- `color` is a color definition for r g and b values. Currently, only greyscale particles are possible!
- `frequency` is the number of particles emitted per second. Keep in mind that larger values reduce performance.
- `lifeTime` is the duration of a particle in seconds. Keep in mind that larger values reduce performance.
- `size01` is the size of the particle in meter at the beginning and end of its lifetime.
- `velocity` is the drifting speed in all three directions relative to the emitter.
- `velocityDampingFactor` is the factor by which the particle velocity, defined by the `velocity` parameter, is dampened. Default is 2.5. The `velocityDampingFactor` does not affect the wind velocity a particle has.
- `initialAlpha` is the initial opacity of the particle. Default is 1.0. 1.0 means particle is fully visible, 0.0 means particle is fully transparent.

Particles are supported for all vehicle types as well as static construction models support the particle emitters.

CameraConfig

By default vehicles, animals and people have an onboard camera that is relative to the node with the first crew seat or if no crew member is available, it is relative to the position and rotation of node 0. The `cameraConfig` allows for setting multiple custom camera positions, which can be cycled when the onboard camera is active.

```
    cameraConfig = {
      positions = {
        {
          group = 0,
          transf = transf.rotYCntTransl(math.rad(25), vec3.new(0, 0, 0),
vec3.new(-15, 0, 9)),
          fov = 90
        }
      }
    }
```

For every camera position, there is a block in the `positions` list with several parameters:

- `group` is the id of the mesh that should be used as origin for the positioning of the camera.
- `transf` is the transformation matrix that is used for the positioning and rotation relative to the origin.
- `fov` is the field of view. Larger values result in wide angle views, narrow views can be done with smaller `fov` values.

Label List

In Transport Fever 2, vehicles and constructions may display some dynamic text labels. These are defined in a `labelList` block in the model metadata which contains a `labels` list:

```
  ..
  labelList = {
    labels = {
      {
        type = "LINE_NAME",
        transf = { 0, -1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, -5.8043 , 0.1814,
2.808 , 1, },
        size = { 0.366, 0.210 },
        color = {247 / 255, 147 / 255, 33 /255},
        fitting = "CUT",
        alignment = "CENTER",
        filter = "NUMBER",
        renderMode = "EMISSIVE",
        childId = "RootNode",
      },
      ...
    },
  },
  ..
```

There are many different properties which can be used to define the labels.

To set the position of the label, use the `transf` property pointing to the coordinates where the lower left corner of the label should be. The label will be placed aligned to the X and Z axis. The coordinates are relative to the mesh that is referenced in `childID` by name.

The `size` contains a pair of two values. The first for the size in x direction, the second for the size in y direction, both relative to the `transf`. Negative values will result in no text visible. The y direction is used for the fontsize too. To use the label with more than one line of text, set `nLines` to a value larger than 1. Values below 1 are ignored.

Coloring the text is possible by using the `color` attribute. It receives a vector with three values, one for each color in the range between 0 and 1. The transparency of the text is set by `alpha`. Value 0 is invisible, value 1 is opaque, values greater than 1 might lead to artifacts. With the `alphaMode`, it is possible to define how the alphablending is done. Possible values are either `“CUTOUT”`, `“BLEND”` or `“NONE”`. In Front of opaque textures, this might be irrelevant, but on transparent faces like glass panes it might be more relevant. To get emissive text (like with LCD destination displays), set the `renderMode` to `“EMISSIVE”`, otherwise set it to `“STD”`.

The horizontal alignment of the text can be set in `alignment` with the values `“LEFT”`, `“CENTER”` and `“RIGHT”`. For the vertical alignment, set `verticalAlignment` either to `“BOTTOM”`, `“CENTER”` or `“TOP”`.

To adjust the behavior of text that is longer than the label, set `fitting` either to:

- `“NONE”` to let it overflow.
- `“CUT”` to cut excessive text.
- `“SCALE”` to scale down until it fits in the horizontal size.

The `type` property contains one of the following keys:

- `“NONE”` is applicable to all models and shows nothing.
- `“LINE_NAME”` is applicable to vehicles and shows the name of the line that the vehicle is currently on.
- `“NEXT_STOP”` is applicable to vehicles and shows the name of the next stop of the vehicle.
- `“NAME”` is applicable to any model and shows the name of the entity (vehicle, construction, ...).
- `“COMPANY_NAME”` is applicable to any model and shows the name of the company.
- `“STATION_NAME”` is applicable to any station model and shows the station name.
- `“CUSTOM”` shows some custom content based on a `labelText` property in the construction.

The `filter` property is used to filter the input from the type. It can either be set to `NONE` to not filter, `NUMBER` to filter anything that is no number or `CUSTOM` to set advanced filters in the `params` block. It contains advanced parameters to influence the text that should be displayed:

- `expr` is a regular expression that can be used to filter the input string, e.g. to only show text or some letters. If the regular expression does not match, no text is displayed. To test some regular expressions, you may use online tools like regex101.com.
- `replace` can contain a gap text that has placeholders which are replaced by the contents or parts of the string matching the regular expression above. `\\0` is replaced by the part of the string that matches the complete regular expression, `\\<number n>` is replaced by the part of the string that matches the content of the nth (...) bracket pair in the regular expression.
- `offset` can be used together with `type = “NEXT_STOP”` to display a stop further down the line.
- `relative = false` is used together with `type = “NEXT_STOP”` to start from the first station of the list (`+offset`). With `relative = true` the vehicles next stop is considered instead of the first stop.

The `font` parameter currently only supports the vanilla fonts Lato and Noto. Other fonts than the

standard ones lead to crashes.

It is recommended to test the labels with the model editor.

Transport Network Provider

Models (except vehicles) can contain lanes and terminals for vehicles, cargo and people. These are configured in the `transportNetworkProvider` metadata block of the model.

The `transportNetworkProvider` contains three different properties:

- `laneLists` for vehicle and passenger lane definitions
- `runways` for feeding in and out ships and airplanes into the fixed lanes.
- `terminals` for passenger and cargo terminals along the vehicle lanes.

```
transportNetworkProvider = {
  laneLists = {
    {
      linkable = false,
      nodes = {
        { { 0, -20, -2.1, }, { 0, 20, 0, }, 30, },
        { { 0, 0, -2.1, },   { 0, 20, 0, }, 30, },

        { { 0, 0, -2.1, },   { 0, 20, 0, }, 30, },
        { { 0, 20, -2.1, },  { 0, 20, 0, }, 30, },
      },
      speedLimit = 20,
      transportModes = { "SHIP", "SMALL_SHIP" },
    },
    ...
  },
  runways = {
    {
      edges = { 0, },
      node = 0,
      type = "LANDING",
    },
    {
      edges = { 1, },
      node = 3,
      type = "TAKEOFF",
    },
  },
  terminals = {
    {
      order = 0,
      personEdges = { 2, 3, },
      personNodes = { 5, 8, },
      vehicleNode = 2,
    },
```

```
    },
},
```

The `laneLists` is a list of blocks which have several properties each:

- `linkable` is a boolean value that decides if the lane can be targeted by the small automatically generated footpath links.
- `nodes` is a list of nodes which are used to define the edges. Every two nodes form one edge. Each node has three properties:
    1. The coordinate relative to the model origin.
    2. The tangent at the node position. Please note that the length of the vector must be equal to the length of the edge. Otherwise negative side effects like compression and stretching of vehicles may occur.
    3. The width of the edge. It is only used for the capacity calculation.
- `speedLimit` is the maximum speed on all edges defined by the nodes above.
- `transportModes` is a list of allowed modes on the edges defined by the nodes above. Possible values are “`PERSON`”, “`CARGO`”, “`CAR`”, “`BUS`”, “`TRUCK`”, “`TRAM`”, “`ELECTRIC_TRAM`”, “`TRAIN`”, “`ELECTRIC_TRAIN`”, “`AIRCRAFT`”, “`SHIP`”, “`SMALL_AIRCRAFT`”and “`SMALL_SHIP`”.

The nodes can be referenced with their index over all lists. The edge 0 is defined by node 0 and 1, edge 1 is defined by node 2 and 3, edge n is defined by node 2×n and 2×n+1.

The `runways` block contains 0 or more runway blocks which have several properties related to the entry and exit points of the freely moving planes and ships:

- `edges` is a list of edges from the `laneLists` that should count to the runway.
- `node` is the point where the vehicle is slowed down after landing or starting speedup for takeoff. It must not lay in the middle of the `edges` list.
- `type` is either “`LANDING`” or “`TAKEOFF`” depending on the type of runway.

An aircraft tries to land and slow down in front of the landing node. After passing it, it will taxi to the terminal. On departure, it will speed up and take off after passing the takeoff node. The landing/takeoff direction is given by the tangent of the first/last edge in the `edges` list. If the runway is not longSame holds for ships.

The `terminals` block contains a list of terminal definition. At a terminal, cargo items and passengers can enter and leave vehicles. Each terminal block contains the following parameters:

- `order` is used for the enumeration of terminals in the user interface. The first terminal (`order = 0`) in the first model of the station with terminals is terminal number 1, …
- `personEdges` are edges which are used as waiting zones for passengers and cargo items. There the passengers or cargo items will wait in idle mode until the vehicle arrives. The lenght and width of these lanes is used for the capacity calculation. These edges need to support the right mode (either “`PERSON`” or “`CARGO`”).
- `personNodes` are the nodes where passengers that alight of vehicles spawn in the station. From there they walk to the station exit or to a personEdge for the next vehicle to take. The invisible cargo items do the same. These nodes need to be on “`PERSON`” or “`CARGO`” edges.
- `vehicleNode` is the node where the vehicle stops. It depends on the vehicle type whether the front or middle of the vehicle exactly stops there. This node needs to be on an edge that support vehicles.

Mesh Definition (.msh/.msh.blob)