Construction Types

There are many different types of construction that are available in different menus and provide different features. To distinguish between the different types, there are different keys for the `type` property. Constructions in the station menus use:

- "`STREET_STATION`" for passenger street stations as constructions.
- "`STREET_STATION_CARGO`" for cargo street stations as constructions.
- "`RAIL_STATION`" for passenger train stations.
- "`RAIL_STATION_CARGO`" for cargo train stations.
- "`HARBOR`" for passenger harbors.
- "`HARBOR_CARGO`" for cargo harbors.
- "`AIRPORT`" for passenger airports.
- "`AIRPORT_CARGO`" for cargo airports.

Construction in the depot menus use:

- "`STREET_DEPOT`" for street depots (bus, truck and tram).
- "`RAIL_DEPOT`" for train depots.
- "`WATER_DEPOT`" for shipyards.

Industry constructions use the "`INDUSTRY`" type.

Assets in the asset menu use:

- "`ASSET_DEFAULT`" for free assets.
- "`ASSET_TRACK`" for assets that snap along tracks.

Buildings that are generated by towns use "`TOWN_BUILDING`".

"`STREET_CONSTRUCTION`" is used for street constructions like roundabouts and highway intersections.

Stations

Stations are the constructions where people and cargo items can board and leave vehicles. Like the transport network provider in single meshes, constructions can have streets and tracks too. Therefore there are three blocks in the result:

- `result.edgeLists` as described in the construction basics.
- `result.runways` that are similar to the ones in the transport network provider but on edges from the `edgeLists` instead of the model lanes.
- `result.terminalGroups` that combines terminals from one or more models with the vehicle edges from the `edgeLists`.
- `result.stations` that assigns terminal groups to independent substations in the construction.

Runways

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 `edgeLists` of the construction 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.

```
result.runways = {
  {
    edges = { 0, },
    node = 0,
    type = "LANDING",
  },
  {
    edges = { 1, },
    node = 3,
    type = "TAKEOFF",
  },
},
```

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.

Terminal Groups

For constructions, terminals from the models can be grouped together. This is not neccessary, but it might be helpful e.g. when several models are used to form a platform of a train station.

The `result.terminalGroups` list may contain zero or more blocks:

```
result.terminalGroups = {
  {
    terminals = { {5, 0 } },
    vehicleNodeOverride = 42
  },
  ...
}
```

Each block in the list contains two properties:

- `terminals` is a list of terminal references in models that should be grouped together. The list items are two value pairs with the id of the model in `result.models` as first number and the number of the terminal inside this model as second number.

- `vehicleNodeOverride` is the id of the node in the `result.edgeLists` that should replace the `vehicleNode` values of the terminals included in this terminal group.

Please note, that it is currently not possible to select the lane of streets that are defined by edges in the construction for the vehicle node. By default, the vehicleNode will be placed on the first lane that is most likely a sidewalk.

Once a `result.terminalGroups` block is defined, non grouped terminals are ignored.

Assignment of Terminals to Stations

To distinguish between the passenger and cargo part of a station, the `result.stations` list is used. It contains a list of sub stations each with their own properties:

- `terminals` is a list of terminalGroup ids that are fetched in the order of the groups in `result.terminalGroups`.
- `tag` is an integer that is used to match a station to the station with the same tag before the update of the construction. If some orders change, this would lead to broken lines otherwise.

```
result.stations = {
  {
    terminals = { 0,1 },
    tag = 0
  },
  {
    terminals = { 2,3 },
    tag = 1
  },
  ...
}
```

The same is done for terminals of different carrier types.

Depots

Depots are the places where the vehicles can be bought by the player. The construction files for depots lay in `res/construction/depot/`. There are no additional property blocks for depots. Instead, a used model has a `vehicleDepot` block. The `updateFn` does not contain any special params except the custom parameters of the construction.

Depot Models

To be used as a depot, a model can have an additional metadata block called `vehicleDepot`.

```
vehicleDepot = {
  capacity = 5,         --unused
  carrier = "RAIL",
```

```
  inNodes = { 0, },
  outNodes = { 0, },
},
```

The block features three properties that are currently in use:

- `carrier` is the carrier key to identify the type of depot. It may be either “ROAD”, “TRAM”, “RAIL”, “WATER” or “AIR”.
- `inNodes` is a list of nodes from the transportNetworkProvider that are used as entry points for vehicles that enter the depot.
- `outNodes` is a list of nodes from the transportNetworkProvider that are used as exit points for vehicles that leave the depot.

Events

Depots support some animation events:

- `open` animation is used to open the main doors.
- `close` animation is used to close the main doors.

Industries

Industries are used for cargo production, conversion and consumption. Their `.con` files can be found in `res/construction/industry/`. They have some special properties as well as some specifics regarding the `updateFn`.

Industry Properties

Beside the general properties like the `description` block, industries have two additional property blocks:

```
  ...
  soundConfig = {
    soundSet = { name = "chemical_plant" },
    effects = {
      select =  { "selected_industry_chemicalplant3.wav" }
    }
  },
  placementParams = {
    buildOrder = 14,
    initWeight = 1,
    tags = { "INPUT_OIL", },
    distanceWeights = {
      INPUT_PLASTIC = 1,
      INPUT_OIL = -2,
    },
  },
```

```
...
```

The `soundConfig` block contains two things:

- `soundSet` is a block that has a reference to a soundset relative to `res/config/sound_set`.
- `effects` is another block for additional sound effects. Currently there is the `select` sound effect that is played whenever the player clicks on the industry.

The other block, `placementParams` is used for the automated distribution of industries over the map. The properties are used to contrain the distribution on dependance of previously built industries. There are:

- `buildOrder` is the order in which the industries are generated. Usually industries without inputs have low values and are built first.
- `initWeight` is used to define the quantity ratio relative to the other industries.
- `tags` is a list of string keys that are assigned to this industry.
- `distanceWeights` is a list of tag keys with assigned weights. If another industry has a tag with negative assigned weight, it is unlikely that the current industry is placed near it. With positive weights, it's more likely that they pull the new industry near them.

UpdateFn Parameters

The available parameters for industries are:

- `paramX` and `paramY` are from the keyboard controls. See the basics to find out more about them.
- `seed` is a number that can be used for randomization purposes. It increases whenever a construction is set.
- `state` is a data struct containing several cached informations about assets and tracks.
- `year` is the current year.
- All custom parameters that were set in other industry constructions before and the custom parameters of the current asset construction.

In some circumstances, e.g. when an industry is upgraded to a new production level or is influenced by mission scripts, it receives some additional parameters:

- `upgrade` is true, when the industry is upgraded or downgraded.
- `input` might contain an input value for the rule (see below).
- `output` might contain an output value for the rule.
- `capacity` might contain a production capacity value for the rule.
- `stocks` might contain a list of stocks.
- `commercialCapacity` might be a capacity for commercial needs of residents.
- `industrialCapacity` might be a capacity for jobs of residents.

It is recommended to support these additional parameters to allow mission or game scripts to adjust the production and consumption of industries. The `industryutil.lua` in `res/scripts/` contains a possible implementation in the `addIndustryData` function.

UpdateFn Return Properties

Beside the common return properties like `models` and `groundFaces`, there are several additional blocks to let a construction consume/produce and store cargo items, you need to specify stocks and stock rules.

Stocks

A stock entry defines for a cargo type whether it's ordered/shipped and on which edges it can be piled up.

```
result.models = { }

-- Add some edges for the stock piles, they can be distributed over an
arbitrary number of models.
-- In this example, each model contains exactly one edge, and we add for
each cargo type one model.
for i = 1, 5 do
  result.models[#result.models + 1] = { id =
"industry/common/stock_lane_8m.mdl", transf = { ... } }
end

-- Specify the necessary cargo types and which edges are used for stock
piling
result.stocks = {
  { cargoType = "PLANKS",   edges = { { 0, 0 } }, moreCapacity = 100 },
  { cargoType = "STEEL",    edges = { { 1, 0 } },
},
```

`cargoType` can be any type defined in `base_config.lua`. `edges` stores a list of edges, where the cargo can be piled up. It consists of one or more pairs of indices. The first index refers to a model from `result.models`, the second index to the particular edge in the edge list of that model. The size of the stock is calculated by the length and widths of each edge. For every 4 m², there is space for one cargo item. If the stock should have more capacity, it can be added with the `moreCapacity` property.

Rules

So far, the construction knows which cargo types it can receive/send and where it should store them. But it knows nothing about how many cargo items it can produce/consume per year and how the required items relate to the produced items. This is specified by the stock rules. Following code shows a rule that refers to the above example for the stock lists:

```
result.rule = {
  input = { { 4, 0 }, { 1, 1 } },
  output = { MACHINES = 2, },
  capacity = 50
```

```
},
```

The number of times that the rule can be processed per year is limited by `capacity`. Each entry of `input` is a pair of numbers. The first number is the quantity of items, the second is the index of the stock from which these items are taken. `output` is a list of produced cargo items per cargo type. The rule processes 4 items of stock 0 and 1 item of stock 1 to two items of type MACHINES.

Examples

There are constructions that only produce cargo items, but don't consume anything. For example, the coal mine.

```
result.stocks = { }
result.rule = { input = { { } }, output = { COAL = 1 }, capacity = 200 }
```

Or the steel mill that consumes two of each iron ore and coal to produce one item of steel:

```
result.stocks = {
  { cargoType = "IRON_ORE", edges = { ... } },
  { cargoType = "COAL", edges = { ... } }
}
result.rule = { input = { { 2, 2 } }, output = { STEEL = 1 }, capacity = 200
}
```

Events

Industries support some animation events:

- `forever` animation is used for roof ventilators or other elements that should animate permanently. It is looped forever and has no fixed length.
- `producing` animation is triggered if a industry is producing.

Assets

Assets are generic constructions that are mainly used for decorative purposes. They may contain functional elements like edges though.

Depending on the used `type` they have a slightly different placement behavior. If set to “ASSET_DEFAULT” they can be placed freely. If set to “ASSET_TRACK”, they snap along nearby tracks by default. It can be suppressed by pressing SHIFT while clicking.

Assets have some additional properties as well as parameters for the updateFn. They do not have additional properties for the result data struct.

Asset Properties

The additional properties of assets are:

```
...
buildMode = "MULTI",
skipCollision = true,
autoRemovable = false,
categories = { "misc" },
snapping = {
  rail = true,
  road = true,
  water = false
},
...
```

The `buildMode` property specifies, whether one can build one (“`SINGLE`”) or more constructions (“`MULTI`”) until the construction menu is closed. It's also possible to build constructions with a brush (“`BRUSH`”).

To ignore collider information and enable the intersection of models, it is possible to skip collision by setting `skipCollision` to true.

When assets should be removed when they are in the way for other constructions (both player built and game built ones), the `autoRemovable` property can be set to true.

To provide additional filter possibilities, it is possible to assign an asset construction to one or more categories. The string keys are added to the `categories` list.

To allow snapping along tracks, streets or the water surface, the appropriate flags can be set to true in the `snapping` property.

UpdateFn Parameters

The available parameters for assets are:

- `paramX` and `paramY` are from the keyboard controls. See the basics to find out more about them.
- `seed` is a number that can be used for randomization purposes. It increases whenever a construction is set.
- `state` is a data struct containing several cached informations about assets and tracks.
- `year` is the current year.
- All custom parameters that were set in other asset constructions before and the custom parameters of the current asset construction.

Town Buildings

Town buildings are automatically constructed by towns depending on the growth and era. Town

buildings have some additional properties and some parameters for their updateFn function.

Town Building Properties

The first additional property is soundConfig:

```
...
soundConfig = {
  soundSet = { name = "town_building" }
},
...
```

It contains a block soundSet that has a reference to a soundset relative to res/config/sound_set.

To inform the game for which purposes the house is suitable, there is a townBuildingParams block:

```
...
townBuildingParams = {
  landUseType = "COMMERCIAL",
  parcelSize = { 1, 1 },
  level = 1,
},
...
```

This block has three properties:

- landUseType is the type of building. It may either be “RESIDENTIAL”, “COMMERCIAL” or “INDUSTRIAL”.
- parcelSize is the size of the building in 8 meter steps. { 1, 2 } would be a building that has an 8 meter section along the street and is 16 meters long away from the street.
- level is the level of the building. The higher the value is, the wealthier and large the building is. The minimum level is 1, the maximum level is 4.

UpdateFn Parameters

The updateFn parameters for town buildings are:

- capacity is the capacity that should be set for the town building.
- cargoTypes is a list of cargo typ keys that should be accepted by this town building.
- depth is the size of the parcel away from the street front.
- width is the size of the parcel along the street front.
- seed is a number that can be used for randomization purposes. It increases whenever a construction is set.
- state is a data struct containing several cached informations about assets and tracks.

UpdateFn Return Properties

The returned data struct of town buildings should have the following properties that are described in the construction basics:

- `bulldozeCost`
- `cost`
- `groundFaces`
- `models`
- `terrainAlignmentLists`

In addition, there are several custom properties described below.

Person Capacity

The `personCapacity` block defines the number of residents that can be covered by this building in the specific type of building:

```
result.personCapacity = {
  capacity = 2,
  type = "INDUSTRIAL"
},
```

It contains two properties:

- `capacity` is the number of residents.
- `type` is the type of the capacity. “`RESIDENTIAL`” results in people living in the building, “`COMMERCIAL`” results in people go shopping there, “`INDUSTRIAL`” results in the people working there.

Stocks and Rules

Buildings may request cargo items. Therefore they have `stocks` and `rule` blocks too:

```
result.stocks = {
  {
    cargoType = "MACHINES",
    edges = {},
    moreCapacity = 100,
    type = "RECEIVING"
  },
...
},
result.rule = {
  capacity = 1,
  consumptionFactor = 1.2,
  input = { { 1 } },
  output = {}
```

```
},
```

The details can be found at the industry explanations above.

Scaffold

The scaffolding is shown while the building is constructed. To define its outline, there is a `scaffold` property:

```
result.scaffold = {
  buildingFace = { { { 7.68415, 4.89475, 0 }, { 7.67744, 15.68718, 0 }, {
0.80814, 15.69474, 0 }, ... } },
  height = -1
},
```

The `buildingFace` is the bottom polygon around the house. The scaffolding is placed along the edges of this polygon with a slight offset so that it does not collide with the walls. The x and y coordinates are relative to the construction origin, z coordinates are ignored and automatically calculated depending on the terrain. The `height` value is used to offset the top height of the scaffolding below the roof.

Events

Town buildings support some animation events:

- `forever` animation is used for roof ventilators or other elements that should animate permanently. It is looped forever and has no fixed length.

Street Constructions / Track Constructions

Street constructions can be used to provide street templates like larger intersections or roundabouts. To provide additional filter possibilities, it is possible to assign an asset construction to one or more categories. The string keys are added to the `categories` list property.

Their `upgradeFn` has only a small set of parameters:

- `paramX` and `paramY` are from the keyboard controls. See the basics to find out more about them.
- `seed` is a number that can be used for randomization purposes. It increases whenever a construction is set.
- `state` is a data struct containing several cached informations about assets and tracks.
- `year` is the current year.
- All custom parameters that were set in other street constructions before and the custom parameters of the current street construction.

The constructions may either use models or not. See below for information about pure street and track construction templates.

Track constructions work similar but for railway. They appear in the track construction tab in the rail menu. Without mods, this tab is not visible.



A track construction example is available for download.

Build free streets and tracks

It is possible to build streets and tracks without creating an actual construction. This way one can build predefined complex street and track arrangements, which are not locked by a construction. They can be modified or removed as streets and tracks built with the standard building tools by the player at a later time.

The example below demonstrates how a segment of street can be built without creating a construction:

```
function data()
return {
  type = "ASSET_DEFAULT",

  updateFn = function(params)
    local result = { }

    result.models = { } -- key 'models' is required, but MUST be empty

    result.edgeLists = {
      {
        type = "STREET",
        params = { type = "standard/town_small_new.lua" },
        edges = {
          { { -20, 0, 0}, { 20, 0, 0 } },
          { { 0, 0, 0 }, { 20, 0, 0 } },
          { { 0, 0, 0}, { 20, 0, 0 } },
          { { 20, 0, 0 }, { 20, 0, 0 } }
        },
        snapNodes = { 0, 2 },
        freeNodes = { 0, 2 }
      },
    }

    return result
  end
}
end
```

To build free streets and tracks it requires:

- the tag `models` should be set, but must be empty
- only `edgeLists` and `edgeObjects` are allowed
- all edges should be “free”

To “free” an edge, their nodes should be “free”. Free nodes are either snap nodes (defined in `snapNodes`) or nodes defined in `freeNodes`. They can be both at the same time.

It is allowed to “free” some nodes/edges of a regular construction. They wont be owned by the construction, thus if it gets removed, the “free” edges and their nodes remain.

Construction Basics

Modular Constructions

From:
https://wiki.transportfever2.com/ - **Transport Fever 2 Wiki**

Permanent link:
https://wiki.transportfever2.com/doku.php?id=modding:constructiontypes

Last update: **2025/04/08 09:50**