Raw bridge data

In Transport Fever 2, bridge are constructed by UpdateFn, the game supplies bridgeutil.makeDefaultUpdateFn to generate an updateFn but you can ignore it and make you own bridge form.

params

The params are in the following structure:

Code

```
1. {
2. pillarHeights = { ... },
3. pillarLength = ..,
4. pillarWidth = ..,
5. railingIntervals = {
6. {
7. hasPillar = { false, true },
8. lanes = {
9. { offset = .., type = .. }
10. },
11. length = 26.934680938721
12. },
13. ...
14. },
15. railingWidth = 4,
16. state = {
17. models = { ... }
18. }
19. }
```

Display More

If you want to make your customized updateFn for bridge, the important information are `pillarHeights`, `railingIntervals`, `railingWidth`

The `state.models` table stores some mdl size information, which maybe useful

result

The result table requires two elements:

Code

```
1. result = {
2. railingModels = ...,
3. pillarModels = ...
4. }
```

basic model elememt

A model element is expected in the following structure, we call it an **M** in the following text

Code

```
1. {
2. id = "..." // mdl path
3. transf = { ... } // coordinate transformation
4. }
```

pillarModels

`pillarHeights` describes the pillars of different heights requires by the game, so you need to give the answer in `pillarModels` for each height

`pillarModels` can be obtained by mapping `pillarHeights` subElements

Code

```
1. for i = 1, #pillarHeights do
2. pillarModels[i] = fx(pillarHeights[i], railingIntervals, railingWidth)
3. end
```

The internal structure of each pillarModels[i] element is expected as a 2-order table

Code

```
1. function fx(height, railingIntervals, railingWidth)
2. local result = fz(height, fy(...))
3. return result
```

4. end

in which

`fy(z, ...)` returns a list of M arranged in horizontal direction, according to `railingIntervals` and `railingWidth`, `z` is the Z-axis displacement given by `fz` (only 1 M in case of a column pillar, or many in case of a wide pillar. nothing is case no pillar is needed)

`fz(...)` calculats the number and type of .mdl needed to stack the pillar in Z-axis and gives `fy(...)` needed information

Attention:

The heights given in `pillarHeights` are heights of the top of pillars

Assertion

#pillarModels == #pillarHeights

railingModels

`railingModels` describes the arrangement of railing models in forward direction, as X-axis, in each case, the railing many be divided into several segements

`railingModels` can be obtained by mapping `railingIntervals` subElements

Code

1. for i = 1, #railingIntervals do
2. railingModels[i] = fs(railingIntervals[i], railingWidth)
3. end

for each segment, like the `pillarModels`, the `railingModels[i]` are expected to be a 2-order table

Code

1. function fs(interval, railingWidth)
2. local result = fx(interval.length, fy(...))

3. return result
4. end

in which

`fy(x, offset, ...)` returns a list of M arranged in horizontal direction, according to `offset` and `railingWidth`, x is the X-axis displacement given by `fx`, the `offset` is `interval.lanes.offset`, a list of each track/street arranged on Y-axis

`fx(...)` calculats the number and type of .mdl needed to arrange the railing X-axis and gives `fy(...)` needed information, based on the `length` given by the `interval`

```
The x = 0 point is aligned to the start of the track/street and the x = length point is the end of end.
```

Assertion

#railingModels == #railingHeights

Other requirements

If you are suffering from strange rotation of model on curves, try to set its pivot on x = 0, and the rest part of model on the positive x side

Attention:

For if the left and right railings are on the same mdl file, use X-aixs flipping rather than Z-aixs rotation by 180°, since the game will flip the normals of flipped models automatically.

Code example

(from Ventabren viaduc)

Code

1. updateFn = function(params)
2. local result = {
3. railingModels = {},
4. pillarModels = {}
5. }
6. for i, height in ipairs(params.pillarHeights) do

```
local colHeight = height - 10.2
local nSeg = math.ceil(colHeight / 6)
local rs = {
{
{
id = "bridge/ventabren/pillar_top.mdl",
transf = {1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, height, 1}
}
}
}
for s = 1, nSeg do
table.insert(
rs,
{
{
id = "bridge/ventabren/pillar_btm.mdl",
transf = {1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, -10.2 - (s - 1) * 6 + height, 1}
}
}
)
end
table.insert(result.pillarModels, rs)
end
for i, interval in ipairs(params.railingIntervals) do
local nSeg = math.floor((interval.length) / 6)
if nSeg < 1 then nSeg = 1 end
local lSeg = interval.length / nSeg
local xScale = lSeg / 5.8
local minOffset = interval.lanes[1].offset
local maxOffset = interval.lanes[#interval.lanes].offset
local width = maxOffset - minOffset
local nPart = math.floor(width / 5)
local wPart = width / nPart
local yScale = wPart / 5
local set = function(n)
local partName = n == 1 and "start" or (n == nSeg and "end" or "start")
local x = (n - 1) * lSeg
local set = {
{
id = "bridge/ventabren/railing_" .. partName .. "_side.mdl",
transf = {xScale, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, x, minOffset - 2, 0, 1}
},
{
id = "bridge/ventabren/railing_" .. partName .. "_side_2.mdl",
transf = {xScale, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, x, maxOffset + 2, 0, 1}
}
}
for k = 1, nPart do
table.insert(set,
{
id = "bridge/ventabren/railing_" .. partName .. "_rep.mdl",
transf = {xScale, 0, 0, 0, 0, yScale, 0, 0, 0, 0, 1, 0, x, minOffset + (k - 1) * wPart, 0, 1}
}
)
end
```

```
    return set
end
local rs = {}
for s = 1, nSeg do
table.insert(rs, set(s))
end
table.insert(result.railingModels, rs)
end
return result
end
```

Display More