Eisenbahn Now

(document version: 2)

„Oh! Hark!“

- Lisa Mitchell

The codebase of EEP is serving us fine for over a decade now. Since the first release of Eisenbahn.exe Professional in late 2001, much functionality and many improvements, namely in the graphics area were added for the better. The source code of EEP proved to provide a solid infrastructure to build railroad simulation software on top of it.

On the other hand it turned out that it has its certain limits. To name just four:

- Track geometry has some oddities with it that popup in our bugtracker again and again. The reason for this seems to be the lack of a proper curve theory in conjunction with being somehow stubborn about the up - direction.

- For cargo transportation the Open Dynamics Engine [ODE] was utilized, but as it turned out, there is a serious problem on the very border of that kind of engines and the idiosyncratic dynamics solution, Eisenbahn.exe uses to move its trains. Namely Newton's third law gets violated at the Eisenbahn.exe's side.

- Users tend to build very huge layouts, which is what in principle can be done with a railroad simulator and is not possible with tin and plastic. But Eisenbahn.exe has certain severe limits here, since it is forced to load all of the data for a layout at once.

- Porting EEP to different platforms other than Windows, which become more and more important, is impossible. The problem with this is not that Microsoft might loose her domination in the desktop market. Death smells like this: there will be no desktops anymore; or at least they might not be available to our potential customers.

I tried to improve the codebase gradually but failed pretty soon. The idea was to slowly transform the infrastructure into something more appropriate to modern needs. But the heavy development that was made on top of that codebase also entails that the codebase gets nailed down to be unchangable.

Any effort to change something substantial suffers from the Banana-Problem, named after the little girl who claimed: 'I know how to spell Banana, I just do not know when to stop'. Meaning, if you change 'A' in the codebase, you have to change 'B'. But if you change 'B', you have to change 'C' and 'D' and so on, leading to a complete rewrite of the code which has no direct benefit other than producing bugs while trying to maintain its already available functionality.

The good news is that writing a new codebase from scratch is quite like a Wunschmaschine (wish-granting device) at the very end of a long and dangerous road [Strugatzki]. We know so much about the troubles of railroad simulation programs; we know what we wish and more important: we know what we better do not wish.

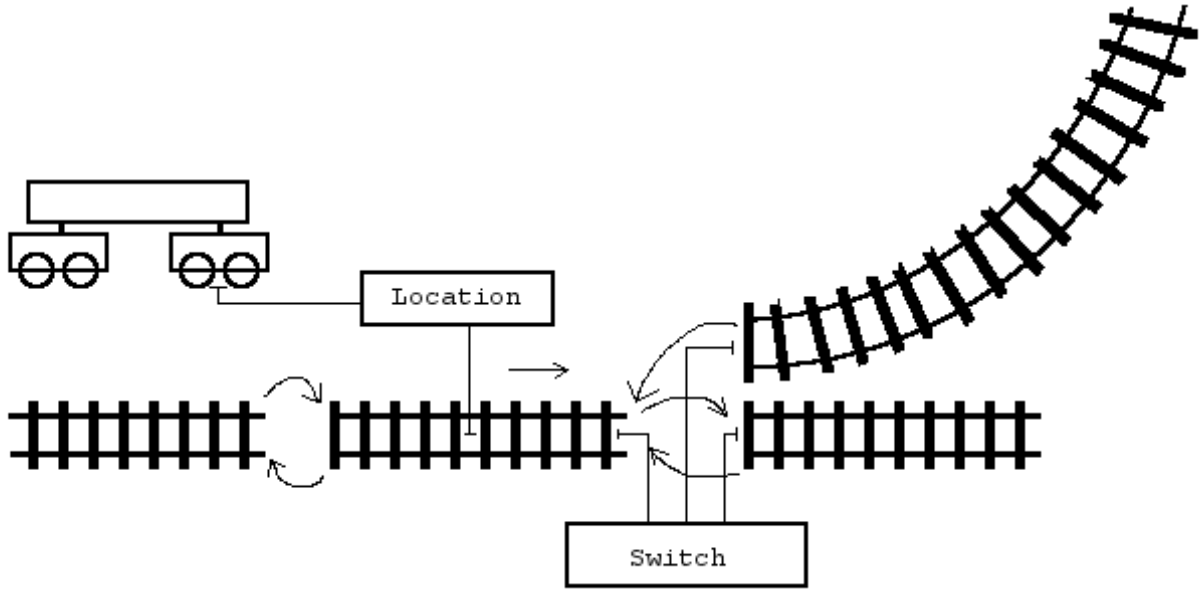
We are sufficiently prepared.

We are far out in 'the Zone' and the time is now.

For this reason I skip all the other things that want to be said and fast forward directly to a description of some of the more fundamental ideas of the new codebase:

Construction of a track system is done in the very same way as it was done with Eisenbahn.exe. The track system is constructed from relatively short track pieces that can get connected at their

ends. Differently than with Eisenbahn.exe, a track in Eisenbahn Now only can have one connected track at each end; switches are built as so called Connectors, which reconnect adjacent track ends:



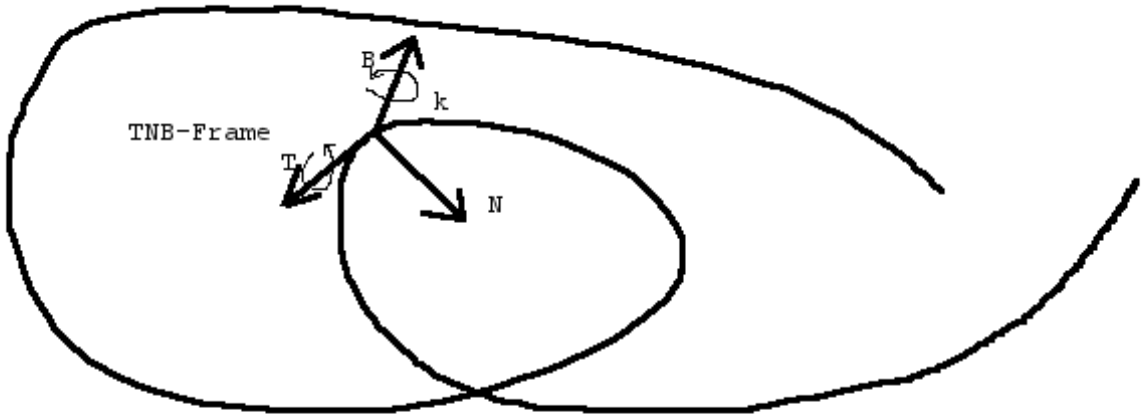
An alternative approach would have been to define tracks as running from one switch (or open end) to the next, a thing that might be called 'Line' with no branching in it [railML]. I think this would be a considerably inferior solution for us, since we have to handle detailed track geometry and so would have to stitch together curves anyway.

A so called Location keeps track of a position on the track system. It can be moved by arc length and will handle the transition between two connected tracks. On moving, it can trigger Sensors that are placed along the track or forward signalling information to the rolling stock.

The geometry of a track is given with so called Curve and Twist objects that get attached with a track. This way, arbitrary track geometries can get realized beyond what was possible with EEP, that could only handle straight lines and arc pieces. Now curve types like Helix, Clothoid or Spline become available; the only thing to watch for is that these curves have to be parametrized by their arc lengthes to keep dynamics working, but

there are mathematical methods to reparametrize curves [Peterson].

To make dynamics work like Newton thought it should, we have to first use a proper theory of curves in 3D space. Lucky us, some 160 years ago two french guys, Monsieur Frenet and Monsieur Serret, figured this all out for us [Frenet]:



Curve

Basically, they found a set of quantities: tangent, normal and binormal vectors T, N and B as well as curvature k and torsion t, which happen to be closely related during movement along the curve. Namely the TNB-Frame is constantly rotating right-handedly around B by a rotational velocity of k and around T by a rotational velocity of t and not at all around N.

In our case we have track curves with some twisted up-direction, as a track can have some camber or being a looping, etc. Fortunately, it turned out that these twists can be described by a seperate twist function that adds its derivative simply to t.

Physics engines are used for dynamics. We implement at least adapters for the Open Dynamics Engine [ODE] and the Nvidia PhysX engine [PhysX].

With these ODE-type physics engines there is some integrator that tries to solve a big system of linear equations for a bunch

of rigid bodies in each simulation step, thereby applying Newtons Laws. To make things more interesting as simply falling down, certain conditions can get enforced. For example, two bodies might get connected by a hinge like joint. This would mean, that their relative positions and orientations are somehow restricted. The important point here is, that those restrictions go into the integrator not as distances and angles but as their first derivatives, namely their velocities.

Now comes Frenet-Serret into the game: If, lets say, a Bogie moves along a track and that track has a Curve and a Twist that can answere the question above, about what the quantities T, N, B, k and t are at each point. Then! We know everything about the restrictions on its velocities. I better write them down here, since it took my poor brain a while to figure this out and I don't want to loose them:

wN : twisted normal N
wB : twisted binormal B
A : angular velocity of the bogie
V : velocity of the bogie

(1) wN*V = 0
(2) wB*V = 0
(3) T*A - t*T*V = 0
(4) N*A = 0
(5) B*A - k*T*V = 0
(6) T*V = vTarget

(1) and (2) guarantee that there will be no linear movement in the twistet normal or binormal directions. Certain limits can get specified for the maximum forces that should be applied in order to guarantee those equations. For a typical train's Bogie this would be a potentially infinite force in direction wB to eliminate all wB*V < 0, but no force to eliminate any wB*V > 0 since the Bogie might very well leave the track in its up direction.

(3) describes some screwdriver rotation around T. (4) states

that there is no rotation around N. (5) enforces the rotation around B according to the curve radius and (6) tries to maintain a certain target velocity along the track. As with (2), the force limits to hold equation (6) can get specified, which would be the actual engine tracktion or braking force for the bogie on the track. This is the reason, why the most natural way to steer a locomotive would be to specify its target velocity and a certain amount of thrust and brake to get applied if necessary in order to reach this aim.

Note that for the linear velocity the twisted TNB Frame is relevant, but the rotations still happen to go around the TNB Frame given by the curve.

There is more to this. Namely some error correction terms have to be formulated; since specifying only constraints on the velocities leaves a certain constant offset to the position completely compatible with the equations above. And as numerical computations are never exact it will grow from them if not already present within the starting situation. Another thing is what I call 'gracefull derailing': a waggon derails by first tilting around its wheels and not simply by moving aside as the track forces become unable to guarantee (1). These two problems complicate the matters quite a bit and I will write about that later and maybe move this all into an appendix to this document.

As it comes to performance it has to be said that all these hefty calculations are made on a per Bogie (not per wheel) basis, and only actually moving Bogies are concerned. Moreover, the PhysX engine provides a mechanism to distribute the calculations over several threads, what divides the calculational workload by the number of cores in the processor. And since on today's multicore desktop computers EEP is still waiting for the graphics card to finish its last frame, I am very convinced that the performance penalty we have to pay will be exactly zero.

Modules are another feature painfully missing with EEP. And this really is a pitty since building arbitray huge model sets is one core property of a virtual railroad simulation and we can not

fully exploit this.

Ideally modules would be speculatively loaded and unloaded in the background while the user moves with the camera. On module transition more than one module would be loaded and displayed at the same time. The fundamental problem of such an approach is that dynamics can not be calculated for modules that are completely not loaded. One solution would be to load the module minus any graphic data anyway and calculate them in the background, which approach would limit the maximum layout size again.

A multiplayer option could overcome this restriction by having each player in one module and his computer doing the dynamics calculations [Horstmann1].

Transition of objects from module to module has to be investigated. One solution for various problems migth be the so called 'epiphanias tracks' , that also might be used for faking incomming traffic from other modules [Horstmann1].

Signals are an important topic within any railroad. The signalling systems differ significantly from railroad company to railroad company and even more significantly from epoch to epoch. A general tendency in technology as a whole is to make everything readable and writable from a computer and leave it to the programmer to fail.

In our case this approach is not feasable. For a new codebase the question is how to design the very roots of a signalling system so that it can be used to simulate various real existing systems. It should be as simple as possible, but as it turned out, the Eisenbahn.exe solution went a little bit to far in that direction. The assumption was that a signal would be a point along a track that communicates something to a passing train.

This led to a manyfold of problems. In general a train pilot sees a signal from a distance and not only when he just passes it. Point like signal influence was always wrong: it has to be an area in which a signal has its effects.

So, as long a train's tip is in a signal area it receives the signal's message about what to do. This is especially important,

when waiting in front of a signal to go clear. With Eisenbahn.exe there had a connection between the signal and a train to be established, with all the complications of saving and restoring on loading and resolving it when somebody deletes the one or the other. In the new system there is no connection whatsoever; just a train that constantly receives the signal's messages.

Plugs N' Jacks are a generalization of the 'Kontaktpunkt' system in EEP. No abstract concept with EEP is such successful as the contact points. Not even signals. This is because it is so easy to grasp: train runs over a contact point and triggers it thereby; contact point in turn triggers various state changes in other Eisenbahn.exe entities - according to the connections, the user made.

From this simple system arbitrary complex logic can be built. In EEP a very innovative method using so called 'Steuerstrecken' and 'Schaltautos' was invented to connect various EEP entities together [maxithing]. Later, specific connections were introduced from the program side to reduce the resulting complexity a little bit; e.g. a connection between switches and signals.

Keeping this conceptional simplicity, but extending its usability is hard. However, since we build a new codebase, it has to be done; now or never. Plugs and Jacks is the one solution, I was able to come up with:

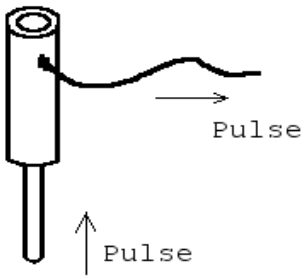
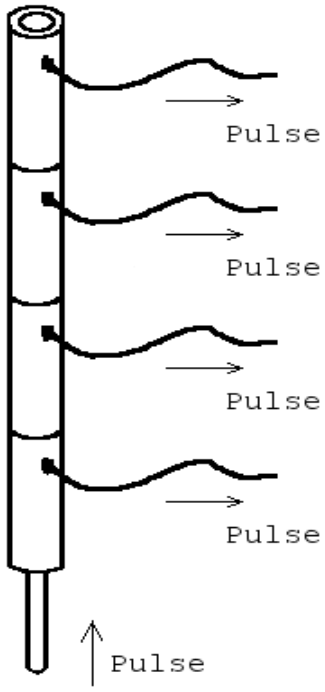


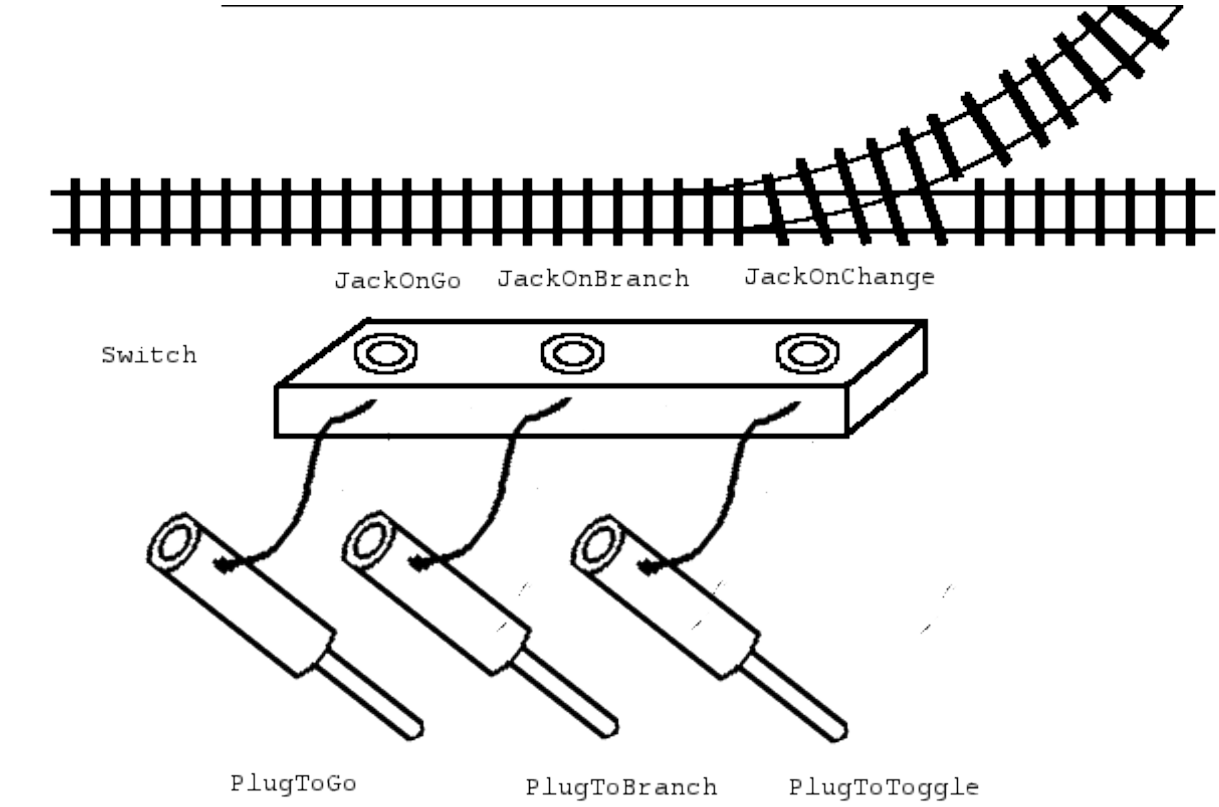
Abbildung 1: It's a Plug (with a Jack on top of it).

A Plug gets inserted into a Jack. Then it would receive the Jack's pulses. Typically the Plug is very specific; let's say it switches the light on in a building. The Jack also would be typically very specific, say it is part of a sensor to detect the brightness of the environmental light and would trigger a pulse if it falls under a certain level. Then a simple decision to plug these two sockets together would turn on the light in the appropriate moment.

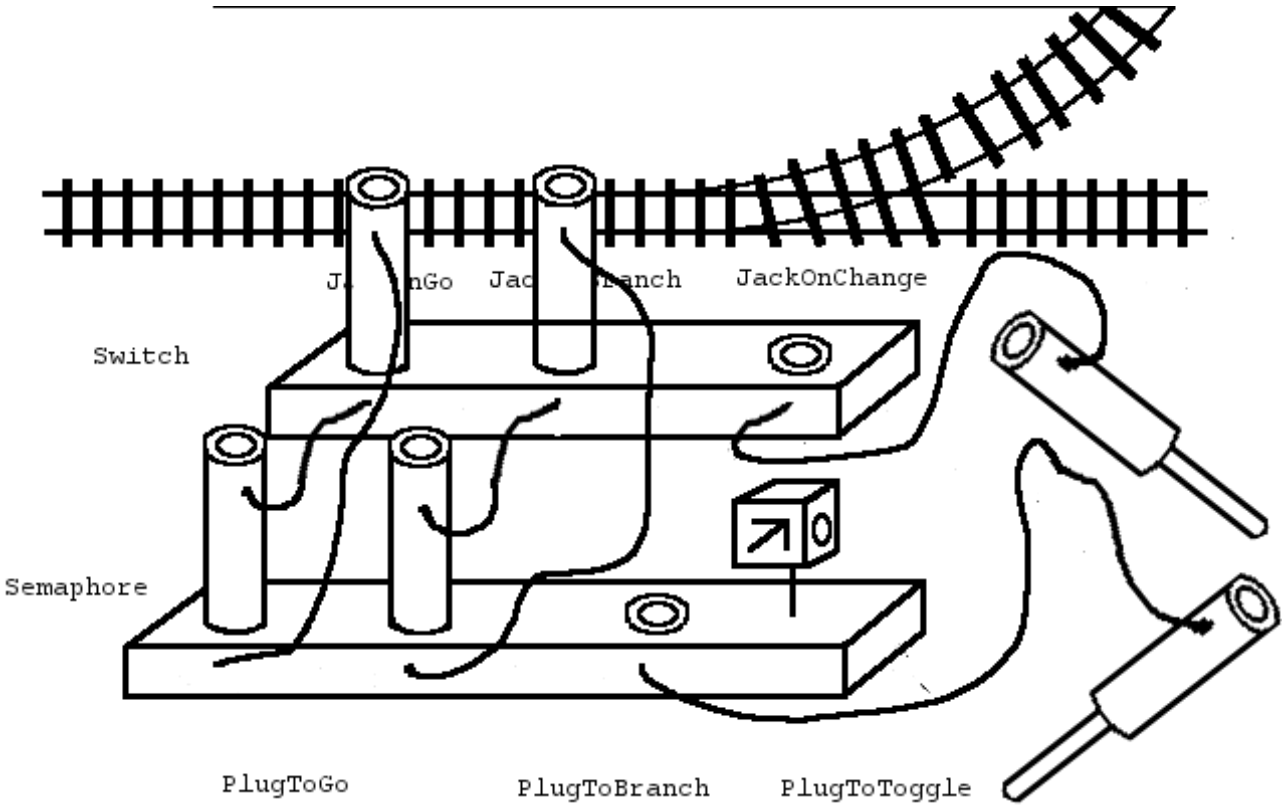
If there would be more than one lamp, we would like to not have to place several redundant sensors. For this a Plug has itself a Jack to insert further Plugs in it:



If it comes to railway stuff, we might have a switch that has two possible states to switch to: 'go', meaning the straight line and 'branch', meaning the diverting route. Now we provide Plugs N' Jacks to either trigger the switch to those settings or trigger something else if it gets setted by what cause soever:

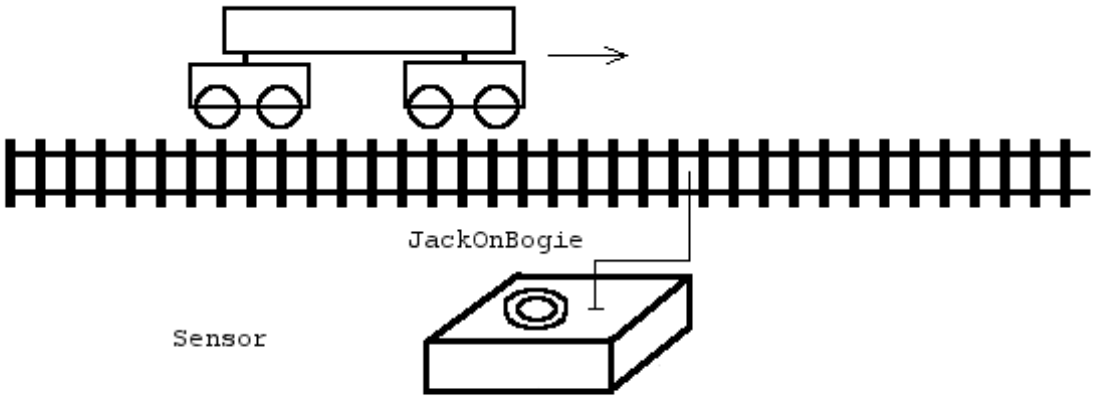


The first application of this might be to connect it with a lamp that clearly shows the state of the switch:



This way the semaphore always would show the actual status of the switch. All the Eisenbahn Now entities would offer such Plugs and Jacks to their discrete settings. Every Plug will be compatible with every Jack, giving endless combinations to try out.

Do never try to give to the 'Pulse' - which in fact is a function call - something like a parameter. A pulse is a pulse, just that. All the information comes from the fact what Plug is connected to what Jack. This is pretty much like the neurons in a human brain work [Eccles]. And I'm pretty sure: the first artificial system that will become self-aware ever - will be Eisenbahn Now!



I admit: all this could very well be achieved by using void* pointers in C++ and then endlessly fiddling around with them. But it would not be typesave. It would not be maintainable. And clearly it would not be transparent to our users and tool writers and most probably not to ourselves.

The Ultimate Asset Selector was described in [Horstmann2]. It is a gadget that would allow the user to pick an item from a very big and potentially unlimited set. The assets, e.g. Eisenbahn.exe models, would not be limited to those installed with the software

but can also be models from the shop that the user might want to buy. Even models that are not existing yet can be displayed. Pressing the select button then would not mean 'load' or 'buy' the model, but vote for it so that some designer might be compelled to build it.

Using XML for storing things wherever possible is my recommendation. The main reason for this is that it gives third party programmers a chance to develop tools for Eisenbahn.exe that base on the easy to understand format. Thus we get more content to sell from our shop with comparatively little effort. There was uttered a concern about tool programmers taking away possibly easy to program features from us. I do not regard this as a real problem, because we always have other means to stop such unwanted development. With our development we want the utmost flexibility. If this opens a hole for a rat we close it (or poison the rat). It's a question of the right order of concerns and the new codebase will not resort to a rat-driven-development paradigm.

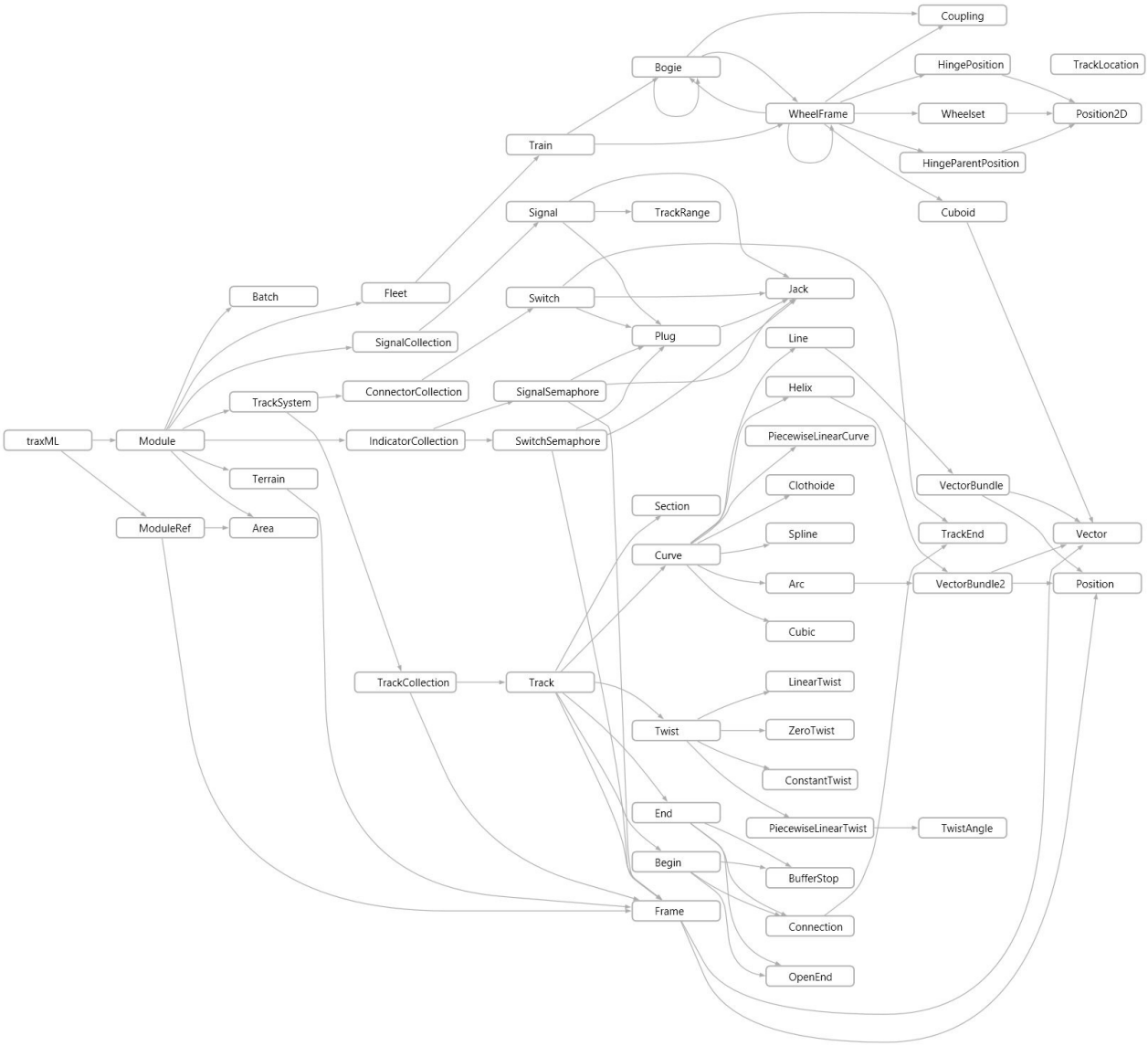
Moreover, binary format has the crucial drawback that it is not platform independent: the layout of the very basic data structures like 'float' or 'int' is different on different machines; so storing it bytewise is not compatible. This also means that our models would have to be translated to go to a different platform. In their case however, because of copy protection, an open format is not wanted anyway.

INI - Files are an alternative, frequently used with Eisenbahn.exe. This is a very simple text format that barely allows hierachical data stuctures. On the far side we have Lua - scripts, which augment the pure data with control structures and thereby data generating logic. I think XML is a quite sane solution in the middle, appropriate for most situations.

There are existing projects of 'describing the world with XML', most remarkable in our case is [railML].

XML offers features for testing like XML Schema. Not only can the sanity of the syntax of an XML file as such automatically be tested but with an *.xsd file an XML file can further get restricted in its possible syntax. This comes in very handy to catch bugs early.

The following diagram shows the XML schema definition for the Eisenbahn Now layout files in its current state:



Concerns about loading times might nevertheless get us to pick some binary format. But we would generate it from an XML file if needed and thereby poisoning the rats exactly were needed.

The command pattern [Gamma] will save us from some of our more severe technical problems with Eisenbahn.exe: each action the engine as such is able to perform (e.g. decouple a waggon from a train or applying a brake), will be programmed as a so called command. These commands know their reverse actions. Something that triggers a command to execute does not have to store information about the actual command's details. This way command provide a clean way of solving several problems in different areas at once:

- User Interface gadgets like menue items can get loaded with their commands, without opening any detail information about the system to the UI code. This way an UI system becomes decoupled from the system and stays exchangable.

- Hot Key tables would become a mapping of key-to-command. This way a user can define his own hotkey settings, since the commands are exchangable in the table.

- Macros of all kind. Since all the actions are commands, they can get cloned and stored in a list for later execution. These lists would be effectively macros as we know them from some word processors for example.

- Undo/Redo. Commands can get stored in a history, thereby providing us with undo/redo functionality, a topic which is implemented in EEP in an utterly complicated and error prone way.

- Commands can map to Lua instructions, thereby making the implementation of script functionality quite straight foreward.

There are some intricacies with commands though. E.g. do they tend to duplicate data structures, since information has to be stored somewhere, or if actions are dependend from each other the execution of a command might need information from a previously executed one (e.g. if two commands create two waggons that

subsequently should be coupled).

These problems can be overcome. The first one for example by storing system objects with the commands to hold the information. The second one by storing state with the system so that a command might not couple waggon X with waggon Y but the last two created.

The benefit will be worth the effort.

The decorator pattern [Gamma] will solve us another crucial problem. An oject is typically something like a track that starts with a logic behaviour in the system. But then it turns out that many other areas have their contributions to make. In EEP I created objects wrapping other objects, thereby providing additional functionality like 3D and 2D painting, sound, serializability, etc. These wrapper objects of course had to get stored somewhere themselves. So there was a multiplication of lists, all of which have to be kept in sync. Basically this questioned the whole idea of having functionality encapsulated in a separate library, since all the lists were best provided by the user of the library in the first place.

With Eisenbahn Now most of the lists provide decorators that allow to add functionality in the case objects are manipulated through those interfaces. E.g., if they get added or removed, some 3D models might have to be created or destroyed.

The alternative would have been to decorate objects like Track and Bogie, but this approach is somehow wrong, since most of the additional data isn't owned by these entities anyway (like a 3D model that might get applied for more than one object in the scene).

The decorator works for the library as the original class did and of course decorators decorate decorators as well, so arbitray aspects can be cascaded.

Object identification. The system inevitably consists from a multitude of relatively simple objects with limited tasks to perform. This is needed to gain the flexibility from the countless possibilities of combinations between these objects. Of course,

these objects have to get stored somewhere. Therefore the library provides several collections which do that. These collections are themselves relatively simple objects and have to get stored - in other collections.

For example a TrackSystem doesn't contain tracks, but so called TrackCollections that allow to handle several tracks as a group, e.g. if they are moved together.

To have all these objects work together, pointers or references are needed. Everything would be fine if only not the necessity would occure to save and to restore these connections. Would C++ allow to allocate an object at a specific address, we simply would store the pointer values and all would be right in the jungle. One obvious solution would be to write an own memory manager that provides own references and would be able to do just that. But this would also mean a runtime penalty on resolving these custom references, which I try to avoid at least by design.

So also it seems odd to me, every object provides an ID that gets assigned by the collection it is attached to. Of course these collections should be able to merge and thereby reassigning unique ids.

After having invested half a year of labour in this, my estimate is that with another year of work from my side and some help from the other project participants, it would be possible to write a separate railroad simulator based on this new codebase and sell it over the shop.

After a phase of extension and improvement were Eisenbahn Now and Eisenbahn.exe will be parallel projects, development can switch over to the new codebase completely.

Regarding the features this first Eisenbahn Now would have, I recommend to choose things for that the new technology can outperform the old easily. Namely these would be:

The dynamics that might get utilized by a cargo and transportation feature much better working than that which we have with Eisenbahn.exe today.

Another thing would be modules, making it possible to build

much bigger layouts than today and drive on them. The single modules would be created by EEP, using it as an editor.

Yet another thing might be a visual display of the logic layout using the Jack N' Plug System.

The real goal of this is not only to realize the above features, or maybe another dedicated feature set that's wanted; but it is to create a codebase that is flexible enough to create almost every kind of railroad simulation in a simpler way, more productive and with much less bugs.

Other than with more common technical problems like 3D rendering or physics engines, one can not ask anybody how to do railroad simulation. There is nobody who at the same time knows and is willing to explain, except maybe - us.

Hinterzarten, December 13th 2014 Marc Horstmann

References:

[Eccles] John C. Eccles: Das Gehirn des Menschen, Piper 1990

[Frenet] http://en.wikipedia.org/wiki/Frenet%E2%80%93Serret_formulas

[Gamma] Gamma, Helm, Johnson, Vlissides: Design Patterns, Addison Wesley 1995.

[Horstmann1]https://82.165.38.60/repos/eisenbahnnow/trunk/Docs/ModulesWithEEP.pdf 2014

[Horstmann2]https://82.165.38.60/repos/eisenbahnnow/trunk/Docs/AssetSelector.pdf and AssetSelector_Implementation.pdf, both 2014

[maxithing] http://www.das-eep-depot.de/index.php/Thread/903-Online-Workshop-Erstellen-von-Schaltungen/?pageNo=4

[ODE] http://ode.org/

[Peterson] http://www.saccade.com/writing/graphics/RE-PARAM.PDF

[PhysX] https://developer.nvidia.com/physx-sdk

[railML] http://www.railml.org/

[Strugatzki] Arkadi and Boris Strugatzki: Picknick am Wegesrand. 1972