TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Jorma Rebane 175744IDDR

3D KRAANA VIRTUAALSE FÜÜSIKALISE MUDELI LOOMINE UNREAL ENGINE KESKKONNAS

Diplomitöö

Juhendajad: Eduard Petlenkov
Aleksei Tepljakov
PhD

Tallinn 2019

Autorideklaratsioon

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

Autor: Jorma Rebane

20.05.2019

Annotatsioon

Antud diplomitöö käsitleb sildkraana virtuaalse füüsikalise mudeli loomist C++ keeles Unreal Engine 4 keskkonnas kasutades 3DKraana füüsikalise mudeli kirjeldust.

Eesmärk on luua alus eraldiseisva virtuaalse õpikeskkonna jaoks, läbi mille saaks realiseerida sardsüsteemide juhtimise praktikumi. Töö käigus valminud sildkraana moodulit on võimalik kergesti lisada mistahes Unreal Engine 4 projekti.

Töö koosneb kinemaatika valemitest, illustratsioonidest, koodinäidetest ja tekstist, mis kirjeldavad füüsikalise mudeli ja Unreal Engine 4 liidese loomist.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 35 leheküljel, 5 peatükki, 52 joonist, 3 tabelit.

Abstract

Creating a virtual 3D Crane physics model in Unreal Engine environment

This diploma thesis describes the creation of a digital twin of the physical laboratory model of a 3DCrane as a C++ library for Unreal Engine 4.

The goal is to create necessary groundwork for a future virtual study environment, which focuses on controlling embedded systems. The completed plugin can be easily added to any new or existing Unreal Engine 4 project.

This thesis consists of kinematics formulae, illustrations, code samples and text, which describes the creation of the physics model and Unreal Engine 4 plugin.

The thesis is in Estonian and contains 35 pages of text, 5 chapters, 52 figures, 3 tables.

Lühendite ja mõistete sõnastik

ATI	Tallinna Tehnikaülikooli Arvutitehnika Instituut
A-Lab	ATI Alpha Control Lab
UE4	Unreal Engine 4, versioon 4.22
Unreal Build Tool	UE4 programmikomponentide kompileerimissüsteem
MATLAB	*Mathworks*'i arendusplatvorm teadlastele ja inseneridele
Simulink	MATLAB'i graafiline simulatsiooni keskkond
C++	Programmeerimiskeel C++, versioon C++17
Komponent	Programmi taaskasutatav ja minimaalselt sõltuv alamosa
Actor	UE4 kasutajale nähtav komponent stseenis
Actor Component	UE4 *Actorile* lisatav nähtav või mittenähtav lisakomponent
Scene Component	UE4 komponentide üldine baasklass
Blueprint	UE4 *Actori* omadusi kirjeldav šabloon
3DKraana	"INTECO 3DCrane", miniatuurne sildkraana
Sardsüsteem	Manussüsteem, ingl *embedded system*
Plugin	Lisandprogramm, mis laiendab olemasoleva programmi funktsionaalsust

Sisukord

Jooniste loetelu

Tabelite loetelu

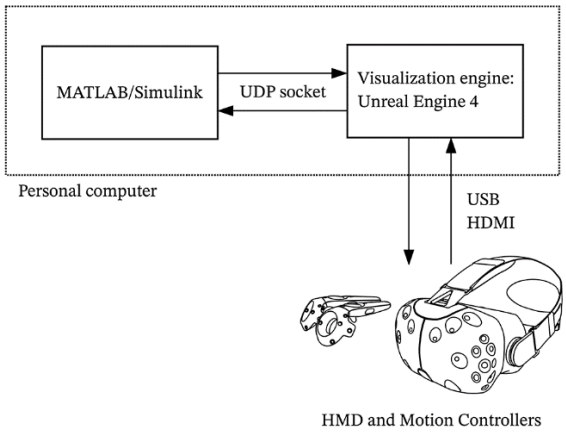
Sissejuhatus

Digitaalselt juhitavad sardsüsteemid on oluline osa tänapäeva elust ning on raske leida valdkonda kus need ei mängiks olulist rolli protsesside optimaalses juhtimises, alustades diiselmootori kontrollerist kuni isesõitvate autode või maanduvate rakettideni [1].

Sardsüsteemide edu tuleneb süsteemikontrolli valdkonna lähenemisest uue sardsüsteemi arendamiseks, millest esimene samm on sihtprobleemile mõjuvate füüsikaliste nähtuste analüüs ja sellest tulenevalt probleemi mudeli loomine. Teiseks sammuks on kontrollsüsteemide arendamine, eesmärgiga optimeerida süsteemi käitumist. Kolmandaks sammuks on süsteemi simulatsioon ja tulemuste valideerimine, enne süsteemi täieliku tootmisse viimist [1].

Käesolev töö keskendub just kolmandale sammule – 3DKraana füüsikalise mudeli simulatsioonile, eesmärgiga võimaldada stabiliseerivate juhtalgoritmide loomine virtuaalkeskkonnas, sardsüsteemide aine raames. Antud diplomitöö käigus loon alusraamistiku eraldiseisvatele virtuaalreaalsuse simulatsioonidele ATI A-Lab'is.

Eelnevate lõputööde käigus on välja arendatud mitmeid simulatsioonimudeleid MATLAB-Simulink keskkonnas, mis kirjeldavad 3DKraana dünaamikat [2] [3]. Kuid sõltuvus MATLAB keskkonnast piirab oluliselt simulatsioonide esitamiskiirust ja süsteemis saab olla ainult üks simulatsioon (Joonis 1).



Joonis 1. Eelnevate virtuaalreaalsuse simulatsioonide sõltuvus MATLAB'ist.

Antud diplomitöö eesmärk on luua eraldiseisev simulatsioonikeskkond, et ületada MATLAB-Simulink piirangud. Teostatamiseks on valitud üldotstarbeline programmeerimiskeel C++, mis sobitub otseselt erinevate virtuaalreaalsuse keskkondadega nagu Unreal Engine 4 ja teeb uute simulatsioonimudelite lisamise võrdlemisi lihtsaks.

Esimeses peatükis tutvustatakse sildkraanat, tema komponente ning miniatuurset sildkraana mudelit INTECO 3DCrane.

Teises peatükis kirjeldatakse sildkraana füüsikalised seosed ning esitatakse kaks mudelit: lihtsustatud ja täielik mudel.

Kolmandas peatükis realiseeritakse füüsikaline komponent C++ keeles ja kirjeldatakse mõlema mudeli integreerimise protsessi.

Neljandas peatükis kirjeldatakse Unreal Engine 4 C++ mooduli loomist koos kraana simulatsioonikomponendiga ja seadistatakse taaskasutatav kraana *Blueprint*.

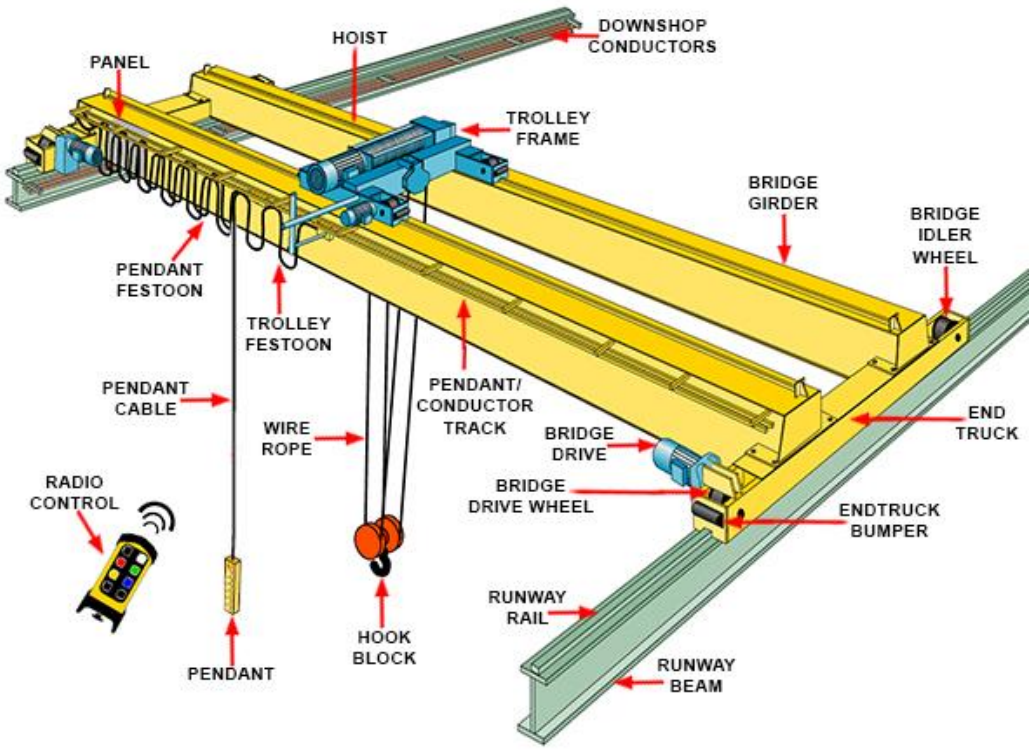
Viiendas peatükis võrreldakse täieliku mudeli simulatsiooni reaalse 3DKraana käitumisega ning analüüsitakse simulatsiooni täpsust. Samuti tehakse vastav järeldus, kas loodud simulatsioon on piisav juhtalgoritmide väljatöötamiseks.

1 Sildkraana tutvustus

Järgnevas tutvustuses vaadatakse üle, mida kujutab endast sildkraana, ning mis on selle peamised rakendused tööstusvaldkonnas. Samuti vaatame üle A-Lab'is kasutatavat sildkraana laboratoorset mudelit INTECO 3DCrane ning selle spetsifikatsiooni.

1.1 Sildkraanad

Sildkraana *(ingl overhead crane)* on kindlat tüüpi tööstuslik kraana, mida iseloomustab kahe rööpa vaheline sild. Sellist kraanatüüpi kasutatakse erinevates tööstusharudes ja sobib paigutuseks tehaste või ladude siseruumides, kus seadmete suurus on piiratud. Käesoleva töö käigus on kraana jaotatud kolmeks põhikomponendiks: sild, vanker ja vints.



Joonis 2. Tööstusliku sildkraana illustratsioon koos ingliskeelsete terminitega [4].

Sild on sildkraana kõige nähtavam osa ning selle liikumine toimub rööbastee peal edasi-tagasi. See komponent on tihti väga massiivne, sest peab toetama kogu süsteemi raskust. Rööbastee on kinnitatud kandvale konstruktsioonile.

Vanker on silla peal küljelt küljele liikuv komponent, mille ajam on tavaliselt vankri enda peal. Vints on kinnitatud vankri külge ja võimaldab trossiga koorma üles-alla liikumist. Sildkraana tõstevõime on harilikult kuni 50 tonni [5]. Sildkraanade puhul esineb ka mitmeid erinevaid konfiguratsioone, kuid antud töös käsitletakse kõige klassikalisemat sildkraanat (Joonis 1).

1.2 INTECO 3DCrane

Antud töös simuleeritakse miniatuurset sildkraanat 3DCrane (edaspidi 3DKraana), mille tootjaks on INTECO. Tegemist on laboritingimusteks sobiva mudeliga, mille raami mõõtmed on 1m x 1m x 1m. 3DKraana juhtimine toimub läbi MATLAB-Simulink keskkonna üle RT-CON liidese [6]. Kraanat saab juhtida nii käsitsi kui ka juhtprogrammiga.

Joonis 3. INTECO 3DKraana laboratoorne mudel [6].

3DKraana on ümbritsetud metallist konstruktsiooniga, mille külge on kinnitatud silla rööpad ja ajamid (Joonis 3). Konstruktsioon ise on kinnitatud põranda külge, et takistada kraana kõikumist.

Selleks, et 3DKraana oleks virtuaalses keskkonnas korrektselt simuleeritud, on vajalik välja tuua elementaarsed parameetrid, mis kirjeldavad kraana dünaamikat. 3DKraana maksimaalne kiirus ja kiirendus on selgitatud eelnevalt A. Aksjonovi magistritöös [3]. Liikumisala ja kraana mõõtmed on mõõdetud A-Lab'is kohapeal [3, lk 12].

Tabel 1. 3DKraana põhiparameetrid.

Kraana parameeter	**Mõõdetud suurus**
Maksimaalne kiirus	0.3 m/s
Maksimaalne kiirendus	0.6 m/s^2
Liikumisala mööda rööpaid	0.72 m
Liikumisala mööda silda	0.88 m
Trossi liikumispikkus	0.62 m
Silla mass	2.2 kg
Vankri mass	1.155 kg
Koormuse mass	1 kg

Andmetest tulenevalt saab märkida, et liikumisala rööbastel on märgatavalt väiksem kui kraana raami mõõtmed võiksid lubada, see on tingitud 3DKraana mootorite konstruktsioonist. Trossi pikkus on laboratoorses mudelis piiratud, et vältida põrkumist kraana komponentidega.

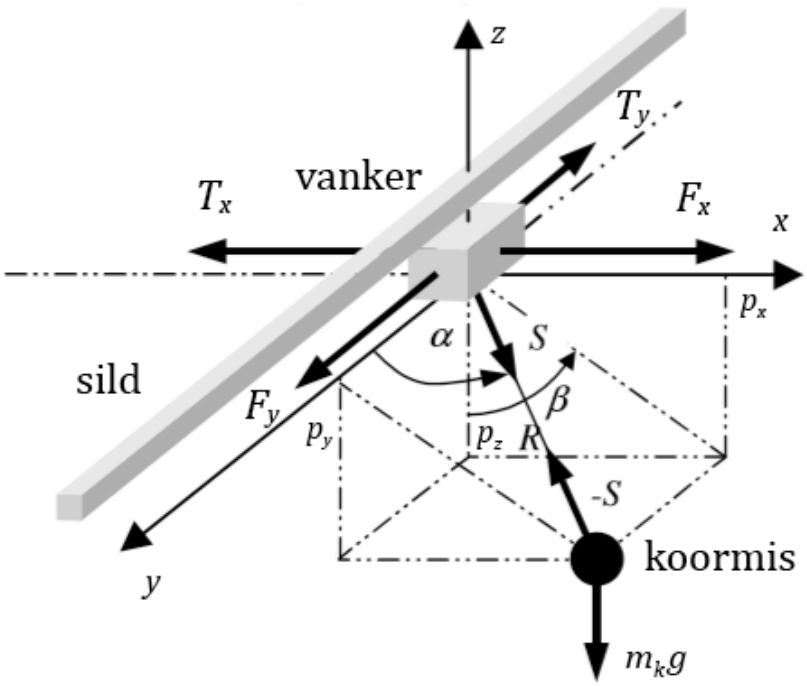
Lähtuvalt maksimaalsest kiirusest ja rööbaste liikumise alast, saab järeldada, et 3DKraana sild liigub ühest otsast teise minimaalselt ~2.5 sekundi jooksul. Seega on tegemist võrdlemisi kiiresti liikuva kraanaga, mis on ideaalne koormise võnkumise näitamiseks ning stabiliseerivate juhtalgoritmide väljatöötamiseks.

2 Sildkraana füüsikaline mudel

3DKraana käsiraamatus on kirjeldatud sildkraana üldine füüsikaline mudel, mille abil on võimalik taasluua kraana simulatsioon [7]. Simulatsiooni loomiseks on vaja luua matemaatiline olekumudel, mida integreeritakse üle aja. Selleks on igal ajahetkel vaja leida kraana komponentide hetkkiirendused.

Järgnevalt kirjeldatud mudel on modifitseeritud ja lihtsustatud eesti keeles mõistmiseks.

2.1 Füüsikalise mudeli põhiseosed



Joonis 4. 3DKraana füüsikalise mudeli skeem [7].

Kraana mudelis on esitatud 5 põhikomponenti:

$X - vankri\ X\ koordinaat\ /\ silla\ asukoht\ rööbastel$

$Y - vankri\ Y\ koordinaat\ /\ vankri\ asukoht\ sillal$

$R - vintsi\ trossi\ pikkus$

$\alpha - y\ telje\ ja\ trossi\ vaheline\ nurk$

$\beta - trossi\ xz\ projektsiooni\ ja\ negatiivse\ z\ telje\ vaheline\ nurk$

Põhiliste sümbolite ja seoste definitsioonid koos füüsikaliste ühikutega:

$F_X - silla\ veojõud\ (N)$
$F_Y - vankri\ veojõud\ (N)$
$F_R - trossi\ vintsi\ veojõud\ (N)$
$T_X - sillale\ mõjuv\ hõõrdejõud\ (N)$
$T_Y - vankrile\ mõjuv\ hõõrdejõud\ (N)$
$T_R - trossile\ mõjuv\ hõõrdejõud\ (N)$
$m_s - silla\ mass\ (kg)$
$m_v - vankri\ mass\ (kg)$
$m_k - koormise\ mass\ (kg)$
$p_x\ , p_y\ , p_z\ - koormise\ asukoht\ (m)$
$S - vankrile\ mõjuv\ reaktsioon\ (N)$

$\mu 1 = \frac{m_k}{m_v}$ – reaktsioonijõu ülekandetegur koormise ja vankri vahel

$\mu 2 = \frac{m_k}{m_v + m_s}$ – reaktsioonijõu ülekandetegur koormise ja silla vahel

$$F_{NX} = F_X - T_X\ , \qquad F_{NY} = F_Y - T_Y\ , \qquad F_{NR} = F_R - T_R\ .$$

Kus F_{NX}, F_{NY}, F_{NR} on X, Y ja R komponentidele mõjuvad summaarsed jõud. Avaldis $F_X - T_X$ kujutab endast staatilise ja kineetilise hõõrdejõu mõjumist vastassuunaliselt veojõule. Newtoni notatsioonis esitatakse hetkkiirendus ja hetkkiirus järgnevalt:

$$\overrightarrow{a_X} = \ddot{X} = \frac{d^2X}{dt^2}\ , \qquad \overrightarrow{v_X} = \dot{X} = \frac{dX}{dt}\ .$$

Et lihtsustada olekumudeli kirjeldust, märgitakse põhikomponentide jõududest tulenevad hetkkiirendused ja nende komponentide hetkkiirused järgnevalt:

$a_X\ , a_Y\ , a_R\ , a_\alpha\ , a_\beta\ , a_{px}\ , a_{py}\ , a_{pz} - hetkkiirendused$
$v_X\ , v_Y\ , v_R\ , v_\alpha\ , v_\beta\ , v_{px}\ , v_{py}\ , v_{pz} - hetkkiirused$

Kraana matemaatilise mudeli kirjeldamiseks on valitud sfääriline koordinaadisüsteem, kuna pendli liikumise kirjeldamiseks on Cartesiuse süsteem ebamugav [7, lk 69]. Sellest tulenevalt on defineeritud koormise asukoht $p_x,\ p_y,\ p_y$:

$$p_x = X + R\ sin\alpha\ sin\beta \tag{3.1}$$
$$p_y = Y + R\ cos\alpha \tag{3.2}$$
$$p_z = -R\ sin\alpha\ cos\beta \tag{3.3}$$

Reaktsioonivektor $\overrightarrow{S}$ koosneb jõukomponentidest S_x, S_y, S_z (ühik N). Siinkohal tuleb mainida, et skalaarne reaktsioonijõud S pole sama mis reaktsioonivektor $\overrightarrow{S}$.

$$S_x = S\ sin\alpha\ sin\beta \tag{3.4}$$

$$S_y = S\ cos\alpha \tag{3.5}$$

$$S_z = -S\ sin\alpha\ cos\beta \tag{3.6}$$

Kraana üldine dünaamika on kirjeldatav Newtoni teise seaduse [8] avaldisest $a = \frac{F}{m}$.

$$a_{px} = -\frac{S_x}{m_k} \tag{3.7}$$

$$a_{py} = -\frac{S_y}{m_k} \tag{3.8}$$

$$a_{pz} = -\frac{S_z}{m_k} - g \tag{3.9}$$

$$a_X = \frac{F_{NX} + S_x}{m_v + m_s} \tag{3.10}$$

$$a_Y = \frac{F_{NY} + S_y}{m_v} \tag{3.11}$$

Dünaamika on täielikult kirjeldatud läbi komponentidele mõjuvate hetkkiirenduste. See võimaldab ka süsteemi modelleerimist ilma sfääriliste koordinaatideta, juhul kui igal ajahetkel on leitud reaktsioonivektor $\overrightarrow{S}$.

Selleks et mudelit edaspidi lihtsustada, tähistatakse X, Y ja R komponentidele mõjuvad summaarsed veokiirendused vastavalt a_{NX}, a_{NY}, a_{NR}:

$$a_{NX} = \frac{F_{NX}}{m_v + m_s} \tag{3.12}$$

$$a_{NY} = \frac{F_{NY}}{m_v} \tag{3.13}$$

$$a_{NR} = \frac{F_{NR}}{m_k} \tag{3.14}$$

2.2 Lihtsustatud mudel

Lihtsustatud mudelis saame teha eelduse, et koormise liikumine on võrdlemisi väike [7, lk 70]. See võimaldab mittelineaarse pendli liikumist kirjeldada hoopis lineaarselt. Defineerime nurgad $\Delta\alpha, \Delta\beta$:

$$cos\alpha = \cos(90° + \Delta\alpha) \cong -\Delta\alpha \quad (3.15)$$

$$sin\alpha = \sin(90° + \Delta\alpha) \cong 1 \quad (3.16)$$

$$cos\beta \cong 1 \quad (3.17)$$

$$sin\beta \cong \Delta\beta \quad (3.18)$$

Sellest tulenevalt saame ümber defineerida reaktsioonivektori $\overrightarrow{S}$ järgnevalt asendades (3.15) – (3.18) definitsioonidesse (3.4) – (3.6) [7, lk 70]:

$$S_x = S\Delta\beta \quad (3.19)$$

$$S_y = -S\Delta\alpha \quad (3.20)$$

$$S_z = -S \quad (3.21)$$

Asendades (3.19) – (3.21) definitsioonidesse (3.7) – (3.11) ja kasutades leitud hetkkiirendusi a_{NX}, a_{NY}, a_{NR}, saame lihtsustatud dünaamika:

$$a_{px} = -a_{NR}\Delta\beta \quad (3.22)$$

$$a_{py} = a_{NR}\Delta\alpha \quad (3.23)$$

$$a_{pz} = a_{NR} - g \quad (3.24)$$

$$a_X = a_{NX} + a_{NR}\mu_2\Delta\beta \quad (3.25)$$

$$a_Y = a_{NY} - a_{NR}\mu_1\Delta\alpha \quad (3.26)$$

Kuna $\Delta\alpha, \Delta\beta$ on väga väikesed, siis on a_{px}, a_{py}, a_{pz} samuti väga väikesed ja domineerivad a_X, a_Y. Lihtsustustest (3.15) – (3.18) saame koormise asukoha [7, lk 70]:

$$p_x = X + R\Delta\beta \quad (3.27)$$

$$p_y = Y + R\Delta\alpha \quad (3.28)$$

$$p_z = -R \quad (3.29)$$

Leitud asukohtadest saab võtta teist järku tuletised, $\ddot{p}_x, \ddot{p}_y, \ddot{p}_z$ mis on füüsikalises tähenduses nende hetkkiirendused a_{px}, a_{py}, a_{pz} [7, lk-d 70-71]:

$$a_{px} = a_X + a_R\Delta\beta + 2v_R v_{\Delta\beta} + Ra_{\Delta\beta} \tag{3.30}$$

$$a_{py} = a_Y - a_R\Delta\alpha - v_R v_{\Delta\alpha} - Ra_{\Delta\alpha} \tag{3.31}$$

$$a_{pz} = -a_R \tag{3.32}$$

Viimase sammuna on vaja tuletada $a_R, a_{\Delta\alpha}, a_{\Delta\beta}$ [7, lk 71]:

$$a_X = a_{NX} + a_{NR}\mu_2\Delta\beta \tag{3.33}$$

$$a_Y = a_{NY} - a_{NR}\mu_1\Delta\alpha \tag{3.34}$$

$$a_{\Delta\alpha} = (a_Y - g\Delta\alpha - 2v_{\Delta\alpha}v_R)/R \tag{3.35}$$

$$a_{\Delta\beta} = -(a_X + g\Delta\beta + 2v_{\Delta\beta}v_R)/R \tag{3.36}$$

$$a_R = -a_{NR} + g \tag{3.37}$$

Olles saanud $a_X, a_Y, a_R, a_{\Delta\alpha}, a_{\Delta\beta}$ avaldised, on kõik olemas kraana lihtsustatud mudeli integreerimiseks üle aja.

2.3 Täielik mudel

Täielikus mudelis käsitletakse trossi koormise liikumist pendli võrranditega, mis on mittelineaarsed [9]. Et kirjeldada mudelit täielikult, asendatakse võrrandid (3.4) – (3.6) dünaamika võrranditesse (3.7) – (3.11) [7, lk 73], rakendades eelnevalt defineeritud lihtsustusi (3.12) – (3.14):

$$a_{px} = -a_{NR}\, sin\alpha\, sin\beta \tag{3.38}$$

$$a_{py} = -a_{NR}\, cos\alpha \tag{3.39}$$

$$a_{pz} = a_{NR}\, sin\alpha\, cos\beta - g \tag{3.40}$$

$$a_X = a_{NX} + a_{NR}\,\mu_2\, sin\alpha\, sin\beta \tag{3.41}$$

$$a_Y = a_{NY} + a_{NR}\,\mu_1\, cos\alpha \tag{3.42}$$

Koormise asukoha leidmise võrranditest (3.1) – (3.2) leitakse uuesti teist järku tuletised $\ddot{p}_x, \ddot{p}_y, \ddot{p}_z$, mis on ühtlasi nende hetkkiirendused a_{px}, a_{py}, a_{pz} [7, lk 74]:

$$a_{px} = a_X + \left(a_R - Rv_\alpha^2 - Rv_\beta^2\right)sin\alpha sin\beta + 2Rv_\alpha v_\beta cos\alpha cos\beta + (2v_R v_\alpha + Ra_\alpha)cos\alpha sin\beta + \left(2v_R v_\beta + Ra_\beta\right)sin\alpha cos\beta \quad (3.43)$$

$$a_{py} = a_Y + (a_R - Rv_\alpha^2)cos\alpha - (2v_R v_\alpha + Ra_\alpha)sin\alpha \quad (3.44)$$

$$a_{pz} = \left(-a_R + Rv_\alpha^2 + Rv_\beta^2\right)sin\alpha cos\beta + 2Rv_\alpha v_\beta cos\alpha sin\beta - (2v_R v_\alpha + Ra_\alpha)cos\alpha cos\beta + \left(2v_R v_\beta + Ra_\beta\right)sin\alpha sin\beta \quad (3.45)$$

Viimase sammuna on vaja tuletada a_R, a_α, a_β [7, lk 74], mis annab meile terve dünaamika kirjelduse:

$$V_\alpha = R\, v_\beta^2\, cos\alpha\, sin\alpha\ - 2v_R v_\alpha + g\, cos\alpha\, cos\beta \quad (3.46)$$

$$V_\beta = 2v_\beta(Rv_\alpha cos\alpha + v_R sin\alpha) + gsin\beta \quad (3.47)$$

$$V_R = Rv_\beta^2 \sin^2\alpha + gsin\alpha cos\beta + Rv_\alpha^2 \quad (3.48)$$

$$a_X = a_{NX} + a_{NR}\, \mu_2\, sin\alpha\, sin\beta \quad (3.49)$$

$$a_Y = a_{NY} + a_{NR}\, \mu_1\, cos\alpha \quad (3.50)$$

$$a_\alpha = (a_{NY} sin\alpha - a_{NX} cos\alpha sin\beta + (\mu_1 - \mu_2 \sin^2\beta)a_{NR} cos\alpha sin\alpha + V_\alpha)/R \quad (3.51)$$

$$a_\beta = -(a_{NX} cos\beta + a_{NR}\mu_2 sin\alpha cos\beta sin\beta + V_\beta)/(R\sin\alpha) \quad (3.52)$$

$$a_R = -a_{NY} cos\alpha - a_{NX} sin\alpha sin\beta - a_{NR}(1 + \mu_1 \cos^2\alpha + \mu_2 \sin^2\alpha \sin^2\beta) + V_R \quad (3.51)$$

Olles saanud $a_X, a_Y, a_R, a_\alpha, a_\beta$ avaldised, on kõik olemas kraana täieliku mudeli integreerimiseks üle aja.

3 Füüsikalise mudeli realiseerimine

Käesoleva töö üks põhietappe on eraldiseisva kraana mudeli loomine C++ keeles, mida oleks võimalik rakendada mistahes platvormil ning vajadusel lisada ka uusi kraana mudeleid [10]. Füüsikalise mudeli kogu lähtekood on teksti kujul saadaval (Lisa 2). Samuti on kogu lähtekood saadaval digitaalses vormis (Lisa 1, Crane3dPlugin).

3.1 Kinemaatika seoste loomine

Et vältida füüsikaliste suuruste ebakorrektset segunemist, on esimeseks sammuks kinemaatika seoste loomine. Eesmärk on luua tugevad andmetüübid ***Force***, ***Mass***, ***Accel*** vastavalt jõu, massi ja kiirenduse ühikute jaoks.

Kinemaatika seosed $F = ma$ ning $a = \frac{F}{m}$ avalduvad andmetüüpide seostena $Force = Mass * Accel$ ning $Accel = Force/Mass$.

Et füüsikaliste ühikute aritmeetilisi seoseid mitte korduvalt defineerida, tuleb luua uus C++ *template* andmetüüp. Tegemist on andmetüübi malliga, mida on võimalik parameetriga spetsialiseerida.

```
template<class T> struct Unit
{
    double Value = 0.0;
    static constexpr Unit Zero() { return { 0.0 }; }
    Unit operator+(Unit b) const { return { Value + b.Value }; }
    Unit operator-(Unit b) const { return { Value - b.Value }; }
    ...
};
struct _Force {};
struct _Mass  {};
struct _Accel {};
using Force = Unit<_Force>;
using Mass  = Unit<_Mass>;
using Accel = Unit<_Accel>;
```

Joonis 5. Füüsikalise ühiku Unit<T> definitsioon.

Andmetüübi **`Unit<T>`** (Joonis 5) spetsialiseerimine teostatakse läbi tühjade andmetüüpide. Üldiselt pole spetsialiseeritav andmetüüp T oluline, kuna definitsioonid peidetakse varjunimede ***Force***, ***Mass***, ***Accel*** taha.

Peale ühikusüsteemi ja varjunimede loomist on võimalik luua vajalikud lisaseosed (Joonis 6). Nendest seostest piisab, et teostada enamus kraana matemaatilises mudelis kirjeldatud kiirenduste leidmine.

```
template<class T>
inline Unit<T> abs(Unit<T> x) { return { std::abs(x.Value) }; }

template<class T>
inline Unit<T> operator*(double x, Unit<T> u) { return { x * u.Value }; }

// F = ma
inline Force operator*(Mass m, Accel a) { return { m.Value * a.Value }; }

// a = F/m
inline Accel operator/(Force F, Mass m) { return { F.Value / m.Value }; }
```

Joonis 6. Füüsikalise ühikusüsteemi seoste defineerimine C++ keeles.

3.2 Füüsikaline komponent

Kraana füüsikalises mudelis on defineeritud 5 põhilist komponenti X, Y, R, α, β, millel on sarnased füüsikalised omadused $m, x, \vec{v}, \vec{a}, \mu_{st}, \mu_{ki}$.

Lisaks on komponentidel kindlad miinimum- ja maksimumpiirid. Piirid on olulised, et kraana komponendid ei sõidaks rööbastelt välja ning et veojõudu ei saaks rakendada rööbaste piiridel. Viimaks hoitakse komponendis ka sellele mõjuvad jõud ja summaarne veokiirendus, mis on kasulik komponendi parameetrite kuvamiseks reaalajas (Tabel 2).

Tabel 2. Füüsikalise komponendi põhiomadused, piirid ja mõjuvad jõud

Põhiomadused	Piirid	Mõjuvad jõud
$m - mass$ $x - asukoht$ $\vec{v} - hetkkiirus$ $\vec{a} - hetkkiirendus$ $\mu_{st} - staat.hõõrd.koef.$ $\mu_{ki} - kineet.hõõrd.koef.$	$x_{MIN} - min\ piir$ $x_{MAX} - max\ piir$ $v_{MAX} - max\ kiirus$ $a_{MAX} - max\ kiirendus$	$F_x - veojõud$ $F_{st} - staatiline\ hõõrdej.$ $F_{ki} - kineetiline\ hõõrdej.$ $F_N - summaarne\ veojõud$ $a_N - summ.veokiirendus$

Antud tabelile vastavalt on loodud kinemaatika komponent C++ keeles, kasutades eelnevalt defineeritud füüsikaliste ühikute andmetüüpe (Joonis 7). Kõikidele füüsikalise komponendi omadustele on antud vaikeväärtused. Kiiruse ja kiirenduse piirajad on vaikimisi välja lülitatud väärtusega 0, sest veojõudusid ja hõõrdejõukoefitsiente kasutatakse ainult X, Y, R, komponentide puhul.

```
struct Component
{
    // base properties:
    Mass Mass = 1_kg;  // mass
    double Pos = 0.0;  // current position within limits
    double Vel = 0.0;  // instantaneous velocity
    Accel Acc = 0_ms2; // instantaneous acceleration

    // limits:
    double LimitMin = 0.0;
    double LimitMax = 0.0;
    double VelMax = 0.0; // velocity limit, 0 = disabled
    double AccMax = 0.0; // acceleration limit, 0 = disabled

    // driving forces:
    Force Applied;   // applied / driving force
    Force SFriction; // static friction
    Force KFriction; // kinematic friction
    Force Fnet;      // net force
    Accel NetAcc;    // net driving acceleration

    double CoeffStaticColoumb  = 5.0;   // resistance to starting movement
    double CoeffKineticViscous = 100.0; // resistance as velocity increases

    ...
};
```

Joonis 7. Kinemaatika komponent C++ keeles.

Kasutades sellist standardiseeritud komponenti, on võimalik ühiselt rakendada liikumispiire, arvutada summaarset veojõudu ning integreerida komponendi asukoha väärtust üle aja. Ühtlasi vähendab see sarnase loogika duplitseerimist ning võimaldab uute jõumudelite lisamist ja konkreetset probleemi kergemini mõista.

3.3 Hõõrdejõu rakendamine

Kraana matemaatilises mudelis on esitatud summaarsed veojõud F_{NX}, F_{NY}, F_{NR}:

$$F_{NX} = F_X - T_X\,, \qquad F_{NY} = F_Y - T_Y\,, \qquad F_{NR} = F_R - T_R$$

Kus T_X, T_Y, T_R on veojõududele F_X, F_Y, F_R mõjuv vastassuunaline hõõrdejõud. Hõõrdejõu modelleerimiseks on 3DKraana füüsikalises mudelis kasutatud Coloumb'i ja Viskoosse hõõrdejõu kombineeritud mudelit [11, lk 3], mis sobib paremini vankri liikumise kirjeldamiseks. Hõõrdejõu arvutamiseks kasutatakse järgnevat valemit [11, lk 3]:

$$T_x = f_c sign(v_x) + f_v v_x$$

Kus v_x on komponendi hetkkiirus, f_c on Coloumb'i hõõrdejõu konstant, f_v on viskoosse hõõrdejõu konstant ja $sign$ on funktsioon, mis määrab jõu suuna vastavalt argumendile. Tuleb mainida, et $f_c \leq F_X$ vastavalt Coloumb'i hõõrdejõu seadusele [11, lk 3]. 3DKraana mudeli kirjelduses on mõõtmiste tulemustena saadud hõõrdejõu konstandid (Tabel 3):

Tabel 3. 3DKraana hõõrdejõu konstandid [6, lk 41].

Komponent	f_c	f_v
Sild	5	100
Vanker	7.5	82
Tross	10	75

Summaarjõud F_{NET} arvutatakse osana füüsikalisest komponendist (Joonis 8):

```
void Component::ApplyForce(Force applied)
{
    applied = ClampForceByPosLimits(applied); // no Fapp at edges
    KFriction = Force{CoeffKineticViscous} * Vel;
    SFriction = Force{CoeffStaticColoumb} * sign(Vel);

    // Coloumb static friction: never greater than net force
    if (SFriction != 0.0 && abs(SFriction) > abs(Fnet))
        SFriction = sign(SFriction) * abs(Fnet);

    Applied = applied;
    Fnet = applied - FrictionDir * (SFriction + KFriction);
    Fnet = dampen(Fnet); // removes sigma sized force flip-flopping
    NetAcc = Fnet / Mass;
}
```

Joonis 8. Summaarjõu Fnet arvutamine.

Jõudude arvutamisel on arvestatud ka komponendi liikumispiiridega. Kui komponent on saavutanud kindla piiri, siis peab rakendatud jõud olema $F_X = 0$. Ilma piireteta laguneb simuleeritud mudel koost ning sild ja vanker sõidavad rööbastelt välja.

Piiramiseks kasutatakse järgmist füüsikalise komponendi funktsiooni (Joonis 9), mis arvestab komponendi asukohaga ja hõõrdejõu suunaga, mis on kaabli puhul teistpidi. Samuti kasutatakse piiride kontrollimiseks konstanti 0.01, sest asukoha integreerimisel esineb $O(\Delta t^2)$ suurusjärgus ebatäpsusi.

```
Force Component::ClampForceByPosLimits(Force force) const
{
    Force F = FrictionDir*force;
    if (F > 0.0 && Pos > (LimitMax-0.01)) return Force::Zero();
    if (F < 0.0 && Pos < (LimitMin+0.01)) return Force::Zero();
    return force;
}
```

Joonis 9. Summaarjõu Fnet piiramine.

Jõu arvutamisele on lisatud ka sumbumisfunktsioon $dampen(x)$, mis summutab ülimalt väikesed jõud, et vältida simulatsiooni ebastabiilsust (Joonis 10).

```
inline Force dampen(Force force)
{
    return { std::abs(force.Value) < 0.001 ? 0.0 : force.Value };
}
```

Joonis 10. Sumbumisfunktsioon *dampen*.

Viimase sammuna on arvutatud jõukomponendile mõjuv summaarne hetkkiirendus a_{NET}, mis avaldub Newtoni teisest seadusest (Joonis 11) [8]. Ühtlasi on tegu ka komponendile mõjuva jõu peamise väljundiga, mis on matemaatilises kirjelduses a_{NX}, a_{NY}, a_{NR}.

```
NetAcc = Fnet / Mass;
```

Joonis 11. Summaarse hetkkiirenduse NetAcc arvutus lõppväljundina.

Komponentides arvutatud hetkkiirendusi saab kasutada kraana olekumudeli integreerimiseks üle aja. Ühtlasi sobivad hetkkiirendused sisendiks mistahes simulatsioonimudelile, avades laiema võimaluse uute mudelite väljatöötamiseks.

3.4 Tulemuse integreerimine

Kui eelmiste sammudega on kätte saadud peamiste füüsikaliste komponentide X, Y, R kiirendused a_{NX}, a_{NY}, a_{NR}, siis nende rakendamiseks on vaja kiirendust integreerida üle aja.

Antud töös on valitud *Velocity Verlet* integreerimisemeetod, mis on variatsioon üldisest *Störmer-Verlet* integreerimise meetodist. Tegemist on väga kiire lineaarse lähendusega, mis on täpsem kui *Euleri* meetod, ei sõltu kiirusest ning töötab ajanihkega Δt [12].

Velocity Verlet matemaatiline kirjeldus on järgnev [12, lk Velocity Verlet]:

$$x_{i+1} = x_i + v_i \Delta t + \frac{a_i \Delta t^2}{2}$$
$$v_{i+1} = v_i + \frac{(a_i + a_{i+1})\Delta t}{2}$$

Velocity Verlet integreerimise meetod sobib reaalajas töötavate süsteemide rakenduses, kuid miinuseks on kumulatiivne viga, esitatud kujul $error\big(x(t_0 + T)\big) = O(\Delta t^2)$ [12, lk Error terms]. Sellest väljendub, et vea suurus on ruutsõltuvuses ajasammuga – mida väiksem ajasamm, seda väiksem on viga ja seda täpsem on simulatsioon.

Velocity Verlet meetodi lisapiiranguna peab Δt olema muutumatu. Sellest tulenevalt saame valida piisava täpsusega konstantse ajasammu $\Delta t = \frac{1s}{10000}$, mis annab 10000 integreerimist sekundis.

Lisaks on simulatsiooni ülesehitusest tulenevalt ka komponendi asukoha x piirid x_{MIN}, x_{MAX}, mida tuleb käsitleda erijuhuna. Kui uus asukoht on väljaspool simulatsiooni piire, siis piirame integreerimise käigus x väärtust ja arvutame v_{i+1} hoopis keskmise kiirusena:

$$v_{i+1} = \frac{(x_{i+1} - x_i)}{\Delta t}$$

Nendest valemitest tulenevalt on kirja pandud 4 iseseisvat funktsiooni kiirenduse ja kiiruse integreerimiseks ning piiride kontrollimiseks (Joonis 12). Lisaks peab piiride arvutamisel arvestama integreerimise ebatäpsusega, kasutades konstanti ± 0.01, sest muidu mõjuks komponentidele hetkkiirus, kui nad on piiridest kaugusel $O(\Delta t^2)$.

```
inline double integrate_verlet_pos(double x, double v, Accel a, double dt)
{
    return x + v*dt + a.Value*dt*dt*0.5;
}

inline double integrate_verlet_vel(double v, Accel a0, Accel a1, double dt)
{
    return v + (a0.Value + a1.Value)*0.5*dt;
}

inline double average_velocity(double x1, double x2, double dt)
{
    return (x2 - x1) / dt;
}

inline bool inside_limits(double x, double min, double max)
{
    return (min+0.01) < x && x < (max-0.01);
}
```

Joonis 12. *Velocity Verlet* meetodiga asukoha ja kiiruse leidmine.

Antud funktsioonidest koosneb füüsikalise komponendi **Component::Update** meetod, mis võtab sisendina uue hetkkiirenduse ning integreerib komponendi uue asukoha ja hetkkiiruse (Joonis 13). Näha on ka piiride rakendamine kasutades **clamp** funktsiooni ning keskmise kiiruse arvutamine piiride lähedal.

```
void Component::Update(Accel newAcc, double dt)
{
    double newPos = integrate_verlet_pos(Pos, Vel, Acc, dt);
    double newVel;
    if (inside_limits(newPos, LimitMin, LimitMax))
    {
        newVel = integrate_verlet_vel(Vel, Acc, newAcc, dt);
    }
    else // pos must be clamped, use Vavg
    {
        newPos = clamp(newPos, LimitMin, LimitMax);
        newVel = average_velocity(Pos, newPos, dt);
    }

    if (VelMax > 0.0 && std::abs(newVel) > VelMax) {
        newVel = sign(newVel) * VelMax;
    }
    Pos = newPos;
    Vel = newVel;
    Acc = newAcc;
}
```

Joonis 13. Füüsikalise komponendi integreerimisfunktsioon.

3.5 Simulatsioonimudel

Simulatsiooni üldises mudelis pannakse kokku kõik kraana komponendid ja tuuakse esile mudeli parameetrid. Määratakse komponentide massid m_s, m_v, m_k, piirid $X_{MIN}, X_{MAX}, Y_{MIN}, Y_{MAX}, R_{MIN}, R_{MAX}$ ja R algväärtus. Samuti määratakse komponentide maksimaalne kiirus v_{MAX} ja kiirendus a_{MAX} (Joonis 14).

```
class Model
{
public:
    /** NOTE: These are the customization parameters of the model */
    Mass Mpayload = 1.000_kg; // Mc mass of the payload
    Mass Mcart    = 1.155_kg; // Mw mass of the cart
    Mass Mrail    = 2.200_kg; // Ms mass of the moving rail
    Accel g = 9.81_ms2; // gravity constant, 9.81m/s^2
    // describes distance of the rail from the center of the frame
    Component Rail { 0.0, -0.30, +0.30 };
    // describes distance of the cart from the center of the rail;
    Component Cart { 0.0, -0.35, +0.35 };
    // describes the length of the lift-line
    Component Line { 0.5, +0.16, +0.90 };
    // describes α angle between y axis (cart left-right) and the lift-line
    Component Alfa { 0.0, -0.05, +0.05 };
    // describes β angle between negative direction on the z axis and
    // the projection of the lift-line onto the xz plane
    Component Beta { 0.0, -0.05, +0.05 };
    // Maximum crane component velocity for Rail, Cart, Line
    double VelocityMax = 0.3; // m/s
    // Maximum crane component acceleration
    double AccelMax = 0.6; // m/s^2
```

Joonis 14. Simulatsioonimudeli komponentide seadistus.

Antud töö eesmärk on võimaldada ka uute simulatsiooniviiside lisamist, milleks on kasutatud objektorienteeritud lähenemist abstraktse baasklassiga *IModelImplementation* (Joonis 14), mis hoiab endas ka viiteid füüsikalistele komponentidele.

```
class IModelImplementation
{
protected:
    Model& Model;
    Component &Rail, &Cart, &Line, &Alfa, &Beta;
    Accel& g;
public:
    IModelImplementation(crane3d::Model& model);
    virtual string Name() const = 0;
    virtual void Update(double dt, Force Frail, Force Fcart, Force Fwind) = 0;
    virtual CraneState GetState() const = 0;
};
```

Joonis 15. Simulatsioonimudelite abstraktne baasklass.

Model klassis on võimalik jooksvalt muuta rakendatavat simulatsiooni teostust. Samuti on võimalik simulatsioon igal hetkel viia tagasi algseisu. Selles töös käsitleme siiski ainult lihtsustatud ja täieliku mudeli teostusi (Joonis 16).

```
// The most basic crane model with minimum pendulum movement
class BasicLinearModel : public IModelImplementation
{
public:
    using IModelImplementation::IModelImplementation;
    string Name() const override { return "Linear"; }
    void Update(double dt, Force Frail, Force Fcart, Force Fwind) override;
    CraneState GetState() const override;
};

// Non-linear fully dynamic model with all 3 forces
class NonLinearComplete : public NonLinearModel
{
public:
    using NonLinearModel::NonLinearModel;
    string Name() const override { return "NonLinearComplete"; }
    void Update(double dt, Force Frail, Force Fcart, Force Fwind) override;
};
```

Joonis 16. Töös käsitletavate lihtsustatud ja täieliku mudeli teostusklassid.

Mudeli integreerimiseks on kaks meetodit **Update** ja **UpdateFixed**. Update integreerib otseselt antud ajasammuga mudelit ühe sammu võrra edasi.

UpdateFixed võimaldab mudeli fikseeritud ajasammuga uuendamist kasutades sisendina varieeruvat ajamuutu, kus suurem ajamuut jagatakse fikseeritud ajasammuga. Jagamisel üle jäänud murdosa salvestatakse järgmiseks integreerimiseks (Joonis 17).

```
ModelState Model::UpdateFixed(double fixedTime, double elapsedTime,
                              Force Frail, Force Fcart, Force Fwind)
{
    // we run the simulation with a constant time step
    SimulationTimeSink += elapsedTime;
    int iterations = static_cast<int>(SimulationTimeSink / fixedTime);
    SimulationTimeSink -= iterations * fixedTime;

    for (int i = 0; i < iterations; ++i)
    {
        Update(fixedTime, Frail, Fcart, Fwind);
    }
    return GetState();
}
```

Joonis 17. Varieeruva ajasammu muutmine fikseeritud ajasammuks.

Update funktsioonis uuendatakse kraana massi põhiseosed ning integreeritakse valitud simulatsioonimudel fikseeritud ajasammuga (Joonis 18).

```
void Model::Update(double fixedTime, Force Frail, Force Fcart, Force Fwind)
{
    Rail.Mass = Mrail+Mcart;
    Cart.Mass = Mcart;
    Line.Mass = Mpayload;
    CurrentModel->Update(fixedTime, Frail, Fcart, Fwind);
}
```

Joonis 18. Fikseeritud ajasammuga Update.

Pärast mudeli integreerimist on võimalik kätte saada kraana mudeli olek, mis koosneb X, Y, R, p_x, p_y, p_z komponentidest (Joonis 19). Iga simulatsioonimudel defineerib oleku väljundi arvutamise iseseisvalt, tulenevalt kõikide võimalike mudelite erinevustest.

```
CraneState BasicLinearModel::GetState() const
{
    CraneState s { Rail.Pos, Cart.Pos, Line.Pos };
    s.PayloadX = s.RailOffset + s.LiftLine * Beta.Pos;
    s.PayloadY = s.CartOffset - s.LiftLine * Alfa.Pos;
    s.PayloadZ = -s.LiftLine;
    return s;
}

CraneState NonLinearModel::GetState() const
{
    CraneState s { Rail.Pos, Cart.Pos, Line.Pos };
    double A = Alfa.Pos, B = Beta.Pos;
    s.PayloadX = s.RailOffset + s.LiftLine * sin(A) * sin(B);
    s.PayloadY = s.CartOffset + s.LiftLine * cos(A);
    s.PayloadZ = -s.LiftLine * sin(A) * cos(B);
    return s;
}
```

Joonis 19. Kraana mudeli oleku kätte saamine peale integreerimist.

Kraana mudeli rakendamine triviaalses programmis võib välja näha järgnevalt (Joonis 20), kus fikseeritud ajasamm on 0.001s, varieeruv ajamuut on 0.1s ja sisenditena rakendatakse veojõud Frail, Fcart, Fwind.

```
Force Frail = 0_N;  // force driving the rail
Force Fcart = 25_N; // force driving the cart
Force Fwind = 0_N;  // force winding the cable

Model model { "NonLinearComplete" };
ModelState state = model.UpdateFixed(0.001, 0.1, Frail, Fcart, Fwind);
```

Joonis 20. Kraana mudeli kasutamine triviaalses programmis.

3.6 Mudelite realiseerimine

Järgnevalt on välja toodud lihtsustatud lineaarse mudeli ja täieliku mittelineaarse mudeli realiseerimine kasutades eelnevalt üles seatud füüsikalisi komponente ja seoseid.

3.6.1 Lihtsustatud mudel

Esimese sammuna lihtsustatud mudelis rakendatakse silla veojõud F_X, vankri veojõud F_Y ja trossi vintsi kerimise veojõud F_R. Selle käigus rakendatakse vastavalt hõõrdejõud ning piiratakse summaarset jõudu vastavalt piiretele (Joonis 21). Tulemuseks on kiirendused a_{NX}, a_{NY}, a_{NR}, mis on mudelis nimedega Rail.NetAcc, Cart.NetAcc, Line.NetAcc.

```
void BasicLinearModel::Update(double dt, Force Frail, Force Fcart, Force Fwind)
{
    Rail.ApplyForce(Frail);
    Cart.ApplyForce(Fcart);
    Line.ApplyForce(Fwind);
    // cable driven tension/friction coefficients:
    double μ1 = Model.Mpayload / Cart.Mass; // μ1 = Mc / Mw
    double μ2 = Model.Mpayload / Rail.Mass; // μ2 = Mc / (Mw + Ms)
    double R = Line.Pos;
```

Joonis 21. Lineaarse mudeli summarjõudude arvutus.

Summaarsetest veokiirendustest saab arvutada komponentidele mõjuvad hetkkiirendused $a_X, a_Y, a_R, a_{\Delta\alpha}, a_{\Delta\beta}$, mis sõltuvad eelmises sammus integreeritud hetkkiirustest $v_R, v_{\Delta\alpha}, v_{\Delta\beta}$ ja integreeritud väärtustest $R, \Delta\alpha, \Delta\beta$ (Joonis 22). Kiirenduse valemid on tuletatud 3. peatükis, valemitest (3.33) – (3.37):

```
Accel aX = Rail.NetAcc + Line.NetAcc*μ2*Beta.Pos;
Accel aY = Cart.NetAcc - Line.NetAcc*μ1*Alfa.Pos;
Accel aA =  (Cart.Acc - g*Alfa.Pos - 2*Alfa.Vel*Line.Vel) / R;
Accel aB = -(Rail.Acc + g*Beta.Pos + 2*Beta.Vel*Line.Vel) / R;
Accel aR = g - Line.NetAcc;
```

Joonis 22. Lihtsustatud mudeli hetkkiirenduste arvutamine.

Viimase sammuna integreeritakse põhilised füüsikalised komponendid $X, Y, R, \Delta\alpha, \Delta\beta$. Trossi pikkus hoitakse konstantsena juhul kui $F_R = 0$ (Joonis 23). Sellega on saavutatud lihtsustatud olekumudel, mis realiseerub läbi pideva integreerimise.

```
Rail.Update(aX, dt);
Cart.Update(aY, dt);
Alfa.Update(aA, dt);
Beta.Update(aB, dt);
Line.Update(aR, dt);
if (Fwind == 0.0) Line.Pos = R;
```

Joonis 23. Lihtsustatud mudeli integreerimine.

3.6.2 Täielik mudel

Sarnaselt lihtsustatud mudelile, rakendatakse täielikus mittelineaarses mudelis esmalt silla veojõud F_X, vankri veojõud F_Y ja trossi vintsi kerimise veojõud F_R. Selle käigus rakendatakse vastavalt hõõrdejõud ning piiratakse summaarset jõudu vastavalt piiretele. Tulemuseks on kiirendused a_{NX}, a_{NY}, a_{NR}, mis on mudelis nimedega Rail.NetAcc, Cart.NetAcc, Line.NetAcc.

Täielik mudel on olemuselt keerulisem kui lihtsustatud mudel, seega on tarvis enne luua abimuutujad (Joonis 24), mis vastavad 3. peatüki valemitele (3.46) – (3.48).

```
void NonLinearComplete::Update(double dt, Force Frail, Force Fcart, Force Fwind)
{
    Rail.ApplyForce(Frail);
    Cart.ApplyForce(Fcart);
    Line.ApplyForce(Fwind);

    double sA = sin(Alfa.Pos), cA = cos(Alfa.Pos);
    double sB = sin(Beta.Pos), cB = cos(Beta.Pos);
    double R = Line.Pos;
    double G = g.Value;
    double vB2 = Beta.Vel*Beta.Vel;
    double VA = R*vB2*cA*sA - 2*Line.Vel*Alfa.Vel + G*cA*cB;
    double VB = 2*Beta.Vel*(R*Alfa.Vel*cA + Line.Vel*sA) + G*sB;
    double VR = R*vB2*sA*sA + G*sA*cB + R*Alfa.Vel*Alfa.Vel;
    // cable driven tension/friction coefficients:
    double μ1 = Model.Mpayload / Cart.Mass; // μ1 = Mc / Mw
    double μ2 = Model.Mpayload / Rail.Mass; // μ2 = Mc / (Mw + Ms)
```

Joonis 24. Mittelineaarse mudeli seoste defineerimine.

Summaarsetest veokiirendustest saame arvutada komponentidele mõjuvad hetkkiirendused $a_X, a_Y, a_R, a_\alpha, a_\beta$ (Joonis 25). Kiirenduse valemid on tuletatud 3. peatükis, valemitest (3.49) – (3.53):

```
    Accel aX = Rail.NetAcc + Line.NetAcc*μ2*sA*sB;
    Accel aY = Cart.NetAcc + Line.NetAcc*μ1*cA;
    Accel aA = (Cart.NetAcc*sA - Rail.NetAcc*cA*sB +
                (μ1 - μ2*sB*sB)*Line.NetAcc*cA*sA + VA) / R;
    Accel aB = -(Rail.NetAcc*cB + Line.NetAcc*μ2*sA*cB*sB + VB) / (R*sA);
    Accel aR = - Cart.NetAcc*cA - Rail.NetAcc*sA*sB
                - Line.NetAcc*(1 + μ1*cA*cA + μ2*sA*sA*sB*sB) + VR;
```

Joonis 25. Mittelineaarse mudeli hetkkiirenduste arvutamine.

Viimase sammuna integreerime põhilised füüsikalised komponendid X, Y, R, α, β. Samamoodi kui lihtsustatud mudelis, hoitakse trossi pikkus konstantsena juhul kui $F_R = 0$ (Joonis 26).

```
Rail.Update(aX, dt);
Cart.Update(aY, dt);
Alfa.Update(aA, dt);
Beta.Update(aB, dt);
Line.Update(aR, dt);
if (Fwind == 0.0) Line.Pos = R;
```

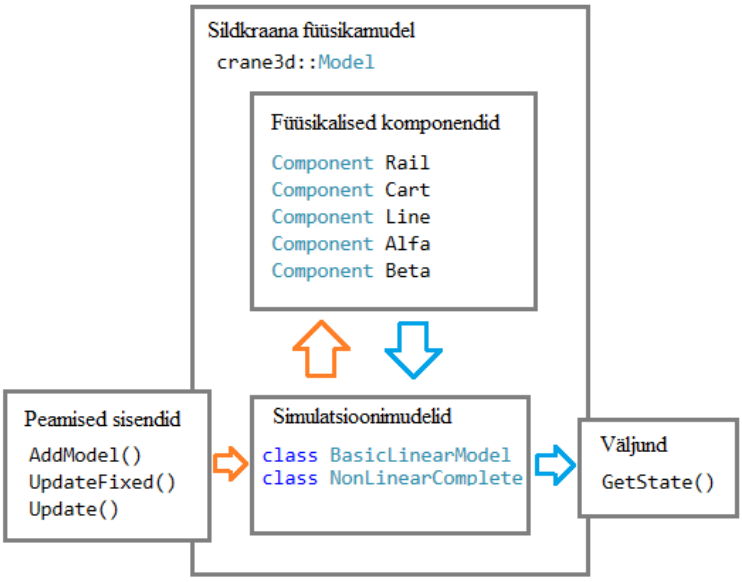
Joonis 26. Mittelineaarse mudeli komponentide integreerimine.

Sellega on saavutatud kraana täieliku mittelineaarse olekumudeli integreerimine üle aja. Mittelineaarse mudeli puhul on korrektselt modelleeritud trossi külge kinnitatud koormise liikumine kui kolmemõõtmeline pendel.

3.7 Mudeli kokkuvõte

Antud peatükis valminud eraldiseisev kraana füüsikamudel võimaldab kraana simuleerimist mistahes platvormil ja uute simulatsioonimudelite lisamist, mis täidab ühtlasi diplomitöö esimese nõude.

Mudeli arhitektuuri lihtsustatud ülevaate saame allolevast diagrammist, milles on näha peamised sisendid, simulatsioonimudelite suhtlus füüsikaliste komponentidega ning väljundina kraana hetkeoleku saamine (Joonis 27).



Joonis 27. Sildkraana füüsikamudeli arhitektuuri lihtsustatud diagramm.

4 Unreal Engine 4 mooduli loomine

Unreal Engine 4 toetab plugin komponente, mis võivad sisaldada C++ koodi ja sisufaile *(“Content”)*. Selleks, et saada kraana füüsikaline mudel tööle Unreal Engine 4 sees, on töö käigus loodud eraldi komponendikiht, mis hoiab endas kraana füüsikamudelit ja tõlgendab UE4 sisendid ja parameetrid sobivalt kraana mudelile. Järgnevalt on kirjeldatud vastava komponendikihi loomisprotsess.

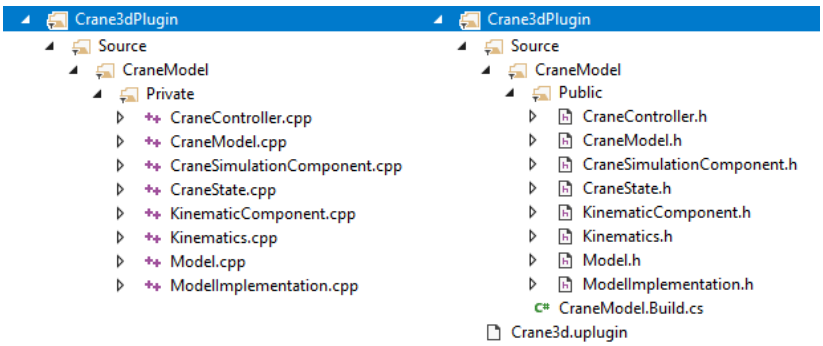
4.1 Uue mooduli loomine

Uue plugin-i ülesseadmine on olemuselt lihtne, suuresti *Unreal Build System* abiga [13]. Esmalt tuleb luua plugin-i kirjeldusega fail, mille nimi on $Crane3d.uplugin$ (Joonis 28):

```
{
        "VersionName": "1.0",
        "FriendlyName": "Crane3d",
        "Description": "3DCrane Physics Simulation Plugin",
        "CreatedBy": "Jorma Rebane",
        "CreatedByURL": "https://github.com/RedFox20/Crane3dPlugin",
        "CanContainContent": true,
        "Modules": [ {
                        "Name": "CraneModel",
                        "Type": "Runtime",
                        "LoadingPhase": "Default"
        } ]
}
```

Joonis 28. UE4 plugin-i kirjeldusfail.

Järgmisena tuleb luua *Epic Games* poolt ettenähtud kaustastruktuur [13], kuhu paigutatakse Crane3d.uplugin failis kirjeldatud **CraneModel** moodul (Joonis 29):



Joonis 29. Crane3dPlugin mooduli kaustastruktuur.

Antud CraneModel moodulisse paigutame eelnevalt loodud kraana mudeli failid. Source/CraneModel/Private/ kausta paigutame .cpp laiendiga C++ moodulid ja Source/CraneModel/Public/ kausta lähevad .h laiendiga C++ päised (Joonis 29).

Lisaks on tarvis kirjeldada mooduli kompileerimiseks kasutatavad sätted, mille jaoks on fail ***Source/CraneModel/CraneModel.Build.cs***. Antud failis on ka seadistatud, millistest Unreal Engine 4 moodulitest sõltub CraneModel.

Samuti on *Unreal Build Tooli* seadistus selline, et plugin projekt ehitataks iteratiivsete alamkomponentidega (Joonis 30), mis on antud väikese projekti puhul oluliselt kiirem [13].

```
public class CraneModel : ModuleRules
{
    public CraneModel(ReadOnlyTargetRules Target) : base(Target)
    {
        PCHUsage = ModuleRules.PCHUsageMode.UseExplicitOrSharedPCHs;
        bFasterWithoutUnity = true;
        bEnforceIWYU = true;
        bPrecompile = false;
        PublicDependencyModuleNames.AddRange(
            new [] { "Core", "CoreUObject", "Engine" });
    }
}
```

Joonis 30. UE4 plugin-i kompileerimisseadistus.

Sellega on kirjeldatud minimaalne UE4 koodimoodul, millele on võimalik lisada juurde taaskasutatavaid komponente, sisufaile ja *Blueprinte*.

4.2 Kraana komponendi plugin

Selleks, et lisada 3. peatükis loodud kraana füüsikaline mudel Unreal Engine 4 mängumootoris nähtava **Actor**-i külge, on tarvis luua uus alamklass **UActorComponent** järgi. Selle alamklassi nimeks saab **UCraneSimulationComponent** (Joonis 31) ja see paigutatakse vastavatesse failidesse CraneSimulationComponent.h/.cpp, mis on nähtav eelnevalt esitatud kaustastruktuurist (Joonis 29).

```
UCLASS( ClassGroup=(Custom), meta=(BlueprintSpawnableComponent) )
class CRANEMODEL_API UCraneSimulationComponent : public UActorComponent
{
    GENERATED_BODY()
public:
    UCraneSimulationComponent();
    void TickComponent(float DeltaTime, ELevelTick TickType,
                       FActorComponentTickFunction* ThisTickFunction) override;
};
```

Joonis 31. Uue *Actor* komponendi loomine.

UE4 simulatsiooni komponendile on lisatud viit kraana mudelile nimega **Model** (Joonis 32). Samuti on kirjeldatud *Blueprintides* nähtavad simulatsiooni parameetriväljad ja simulatsiooni väljundiks olevad muutujad. Antud **USceneComponent** tüüpi parameetrid on kraana visuaalseteks elementideks.

```
std::unique_ptr<crane3d::Model> Model;

// This should be an invisible node which marks the 0,0,0 of the crane system
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Crane Components")
USceneComponent* CenterComponent = nullptr;

// crane rail (X-axis in the model)
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Crane Components")
USceneComponent* RailComponent = nullptr;

// crane cart (Y-axis in the model)
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Crane Components")
USceneComponent* CartComponent = nullptr;

// payload at PayloadPosition
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Crane Components")
USceneComponent* PayloadComponent = nullptr;

// OUTPUT: X-offset of the rail and Y-offset of the cart (cm)
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "Crane Outputs")
FVector CartPosition;

// OUTPUT: Payload 3D position, offset to CenterComponent
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "Crane Outputs")
FVector PayloadPosition;
```

Joonis 32. UE4 simulatsiooni komponendi parameetrid.

Et kraana kontrollimine oleks läbi *Blueprintide* lihtsam, on lisatud komponentide initsialiseerimise funktsioon, kraana füüsikalistele komponentidele mõjuvate jõudude funktsioonid ning simulatsioonimudeli vahetamise funktsiooni (Joonis 33).

```
UFUNCTION(BlueprintCallable, Category = "Crane Initializer")
void SetCraneComponents(USceneComponent* center, USceneComponent* rail,
                        USceneComponent* cart,   USceneComponent* payload);

// Adds force to the rail across the X axis
UFUNCTION(BlueprintCallable, Category = "Crane Force Inputs")
void AddRailX(float axisValue, float multiplier);

// Adds force to the cart across the Y axis
UFUNCTION(BlueprintCallable, Category = "Crane Force Inputs")
void AddCartY(float axisValue, float multiplier);

// Adds force to the winding mechanism across the Z axis
UFUNCTION(BlueprintCalslable, Category = "Crane Force Inputs")
void AddWindingZ(float axisValue, float multiplier);

// Switches to the next simulation model
UFUNCTION(BlueprintCallable, Category = "Crane Misc. Inputs")
void NextModelType();
```

Joonis 33. Kraana *Actor* komponendi *Blueprint* funktsioonid.

Kraana füüsikalisest mudelist tulnud olekust uuendatakse väljundiks olevad olekumuutujad **CartPosition** ja **PayloadPosition**. Lisaks on vajalik ühikute teisendamine sentimeetritest meetritesse, sest Unreal Engine 4 koordinaadisüsteem on vaikimisi esitatud sentimeetrites (Joonis 34).

```
void UCraneSimulationComponent::UpdateVisibleFields(const crane3d::ModelState& state)
{
    // UE4 is in centimeters, crane model is in meters
    CartPosition.X = float(state.RailOffset * 100);
    CartPosition.Y = float(state.CartOffset * 100);
    PayloadPosition = FVector(state.PayloadX, state.PayloadY,state.PayloadZ)*100;
```

Joonis 34. UE4 simulatsiooni komponendi väljundite kirjutamine.

Unreal Engine 4 mootori taktsammuks on polümorfne funktsioon **TickComponent**, mis annab parameetrina ühe kaadri arvutamiseks möödunud aja DeltaTime. Kui mäng jookseb ~60 kaadrit sekundis, siis on DeltaTime väärtus ~1/60.

Kraana integreerimine on eelnevas peatükis üles seatud vastavalt sellele teadmisele, mis võimaldab simulatsiooni ühendada ilma suurema pingutuseta kasutades **Model::UpdateFixed** meetodit (Joonis 35).

```
void UCraneSimulationComponent::TickComponent(float DeltaTime,
    ELevelTick TickType, FActorComponentTickFunction* ThisTickFunction)
{
    Super::TickComponent(DeltaTime, TickType, ThisTickFunction);

    UpdateModelParameters();

    using crane3d::Force;
    crane3d::ModelState state = Model->UpdateFixed(1.0/10'000.0, DeltaTime,
                Force{ForceRail}, Force{ForceCart}, Force{ForceWinding});

    UpdateVisibleFields(state);
    UpdateVisibleComponents();

    // reset input forces for this frame
    ForceRail = ForceCart = ForceWinding = 0.0;
}
```

Joonis 35. UE4 simulatsiooni komponendi sidumine kraana füüsikamudeliga.

Kraana visuaalsete komponentide uuendamine toimub funktsioonis **UpdateVisibleComponents**, milles antaks silla, käru ja koormise relatiivsed asukohad. Kraana koordinaadisüsteemi keskpunktiks UE4 stseenis on **CenterComponent**.

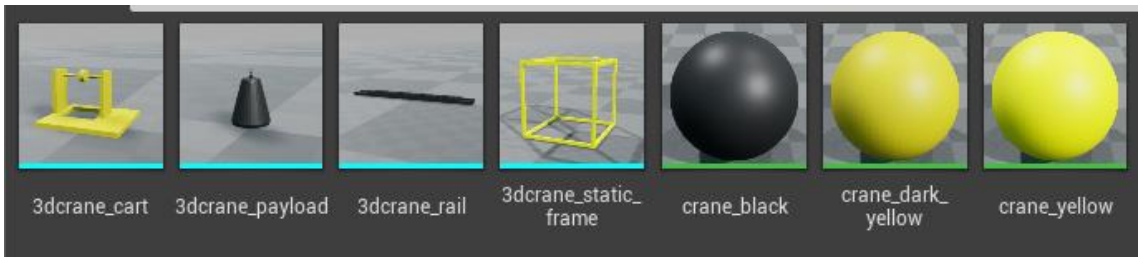
Lisaks asukohtadele pööratakse koormis otsaga käru poole, mis on tingitud sellest, et koormis on trossiga kinnitatud vankri külge. Viimane samm lisab musta värvi kaabli joone, kasutades UE4 silumisfunktsiooni **DrawDebugLine** (Joonis 36).

```
void UCraneSimulationComponent::UpdateVisibleComponents()
{
    if (!CenterComponent || !PayloadComponent || !CartComponent || !RailComponent)
        return;

    FVector railPos = RailComponent->RelativeLocation;
    FVector cartPos = RailComponent->RelativeLocation;
    railPos.X = CartPosition.X;
    cartPos.X = CartPosition.X;
    cartPos.Y = CartPosition.Y;
    RailComponent->SetRelativeLocation(railPos);
    CartComponent->SetRelativeLocation(cartPos);
    PayloadComponent->SetRelativeLocation(PayloadPosition);

    // rotate the payload towards cart
    FVector dir = CartComponent->GetComponentLocation()
                - PayloadComponent->GetComponentLocation();
    dir.Normalize();
    FRotator rot = dir.Rotation(); rot.Pitch -= 90;
    PayloadComponent->RelativeRotation = rot;

    // cable line
    DrawDebugLine(GetWorld(), CartComponent->GetComponentLocation(),
        PayloadComponent->GetComponentLocation(),
        FColor(15, 15, 15), false, -1, 0, 1);
}
```

Joonis 36. UE4 visuaalsete komponentide asukohtade määramine.

4.3 Kraana *Blueprinti* loomine

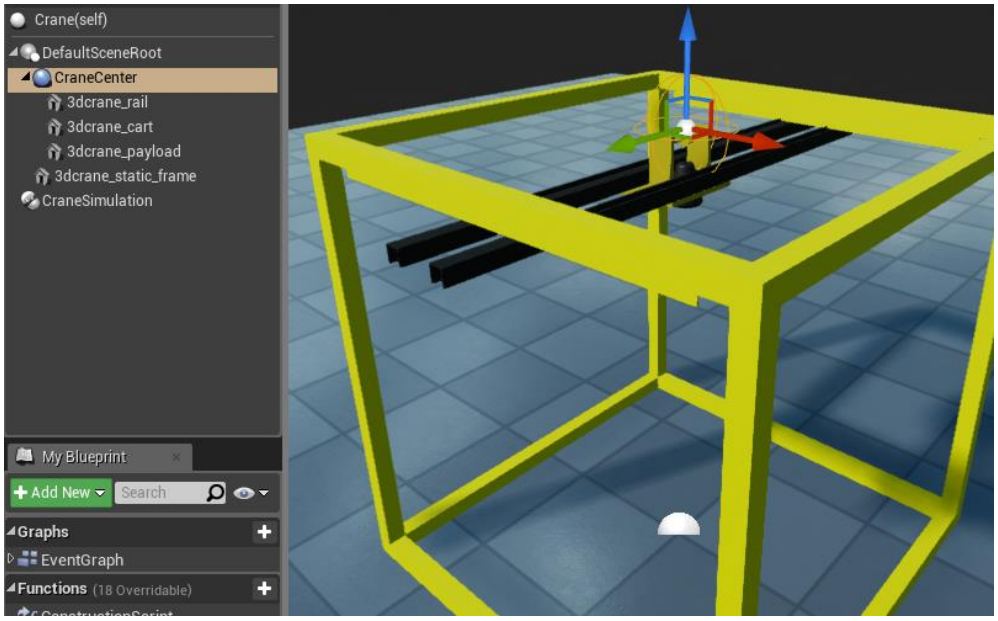
Valminud **UCraneSimulationComponent** kasutamiseks luuakse visuaalne *Actor* mille nimeks saab Crane. UE4 projekti on imporditud D. Freibergi [2] poolt modelleeritud 3DKraana 3D mudel, mis koosneb osadest 3dcrane_cart, 3dcrane_payload, 3dcrane_rail ja 3dcrane_static_frame (Joonis 37). Importimise käigus on UE4 loonud ka vastavad pinnamaterjalid crane_black, crane_dark_yellow ja crane_yellow (Joonis 37).



Joonis 37. 3DKraana 3D mudelid ja materjalid.

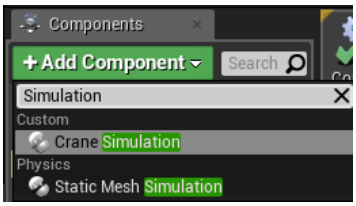
Blueprint Crane puhul on loodud vastav hierarhia (Joonis 38), kus CraneCenter defineerib füüsikalise mudeli koordinaadisüsteemi keskpaiga. Liikuvad visuaalsed komponendid 3dcrane_rail, 3dcrane_cart, 3dcrane_payload on lisatud CraneCenter alamobjektidena, mis loob vajaliku relatiivse koordinaadisüsteemi.

Kraana raam on lisatud eraldi kraana komponendina 3dcrane_static_frame. Kogu kraana paigutuse keskpunkt asub raami all keskel (Joonis 38).



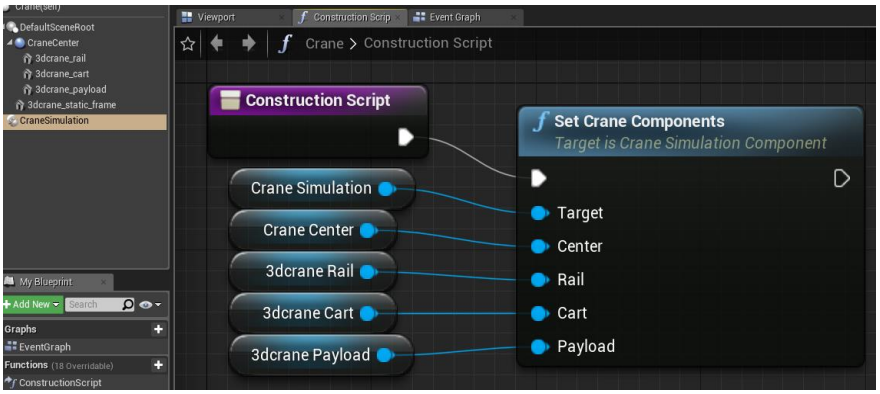
Joonis 38. Crane *Blueprinti* visuaalne hierarhia.

Blueprint Crane kujutab endast *Actorit* millele on võimalik lisada mistahes **ActorComponent**. Selleks, et lisada C++ keeles kirjutatud plugin **UCraneSimulationComponent**, on vajalik avada menüüst „Add Component“ ja leida eelnevalt loodud „Crane Simulation” (Joonis 39).



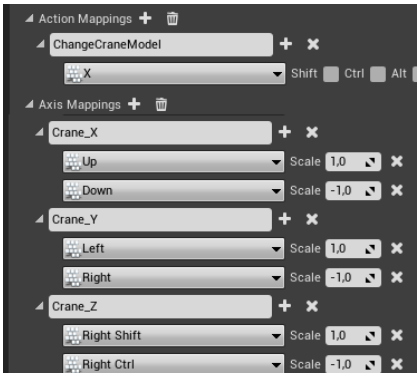
Joonis 39. Uue *Actor* komponendi isendi tekitamine.

Järgmise sammuna initsialiseeritakse kraana kohustuslikud parameetrid, ilma milleta simulatsiooni pole võimalik näha. Selleks on kasutatav eelnevalt C++ komponendis loodud *Blueprintile* nähtav funktsioon **SetCraneComponents** (Joonis 40).



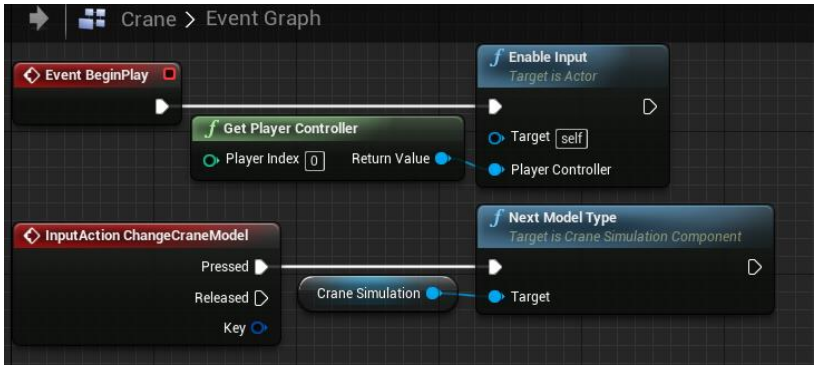
Joonis 40. Kraana komponendi initsialiseerimine.

Selleks, et kraana simulatsioon saaks vajalikud juhtsisendid, on vajalik luua sisendite definitsioonid, mis on esitatud “Settings -> Project Settings -> Input” menüüs. Sisendite nimedeks on ChangeCraneModel, Crane_X, Crane_Y, Crane_Z (Joonis 41).



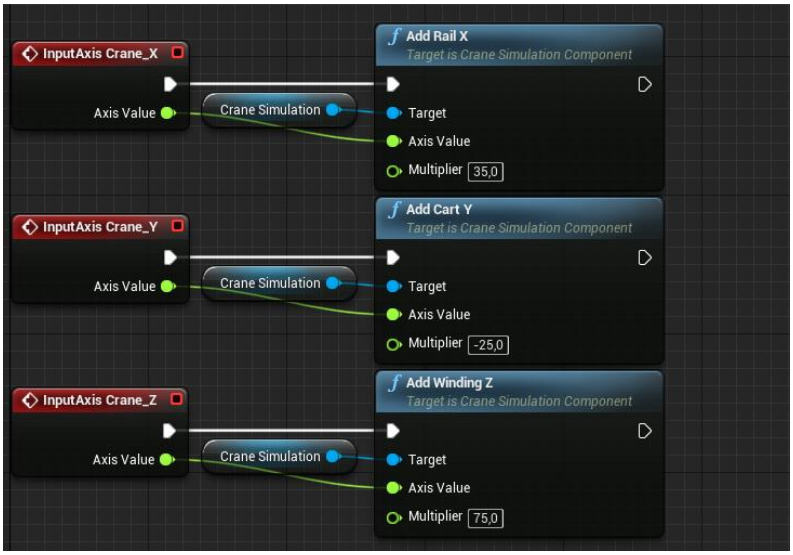
Joonis 41. Kraana sisendite defineerimine.

Loodud sisendsündmuseid kasutatakse Crane *Blueprintis* vastavalt kraana simulatsiooni komponendile parameetrite saatmiseks. Kraana loomisel lülitatakse sisse sisendsündmuste saatmine Crane *Actorile* kasutades **EnableInput** funktsooni. Kraana mudeli muutmise sündmuse puhul kutsutakse C++ komponendis defineeritud funktsioon **NextModelType** (Joonis 42).



Joonis 42. Sisendi sisselülitamine ja mudeli vahetamine.

Kasutuses on eelnevalt C++ defineeritud funktsioonid veojõudude lisamiseks (Joonis 43).



Joonis 43. Kraana simulatsiooni veojõudude lisamine.

Sellega on loodud kõik vajalik funktsionaalsus Crane *Actoris*. Kraana simulatsiooni kasutamiseks on vaja projektile lisada Crane3dPlugin ning vedada Crane *Actor* vastavasse UE4 stseeni.

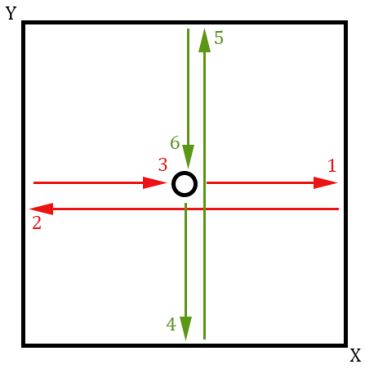
5 Simulatsiooni kontrollimine

Veendumaks, et loodud simulatsioon on piisav stabiliseerivate juhtalgoritmide loomiseks, on vajalik lihtsustatud ja täieliku mudeli käitumise võrdlemine. Käesolevas peatükis on analüüsitud mudeli komponentide ja 3DKraana laboratoorse mudeli käitumine graafikutel ning Unreal Engine 4 keskkonnas.

Antud simulatsioonimudelite puhul on tegemist ilma juhtalgoritmita süsteemidega, seega on ka 3DKraana laboratoorsel mudelil juhtalgoritmid välja lülitatud. Simulatsiooni sisenditeks on lineaarsed jõud $F_{sild}, F_{vanker}, F_{vints}$, mis põhjustavad kõrvalmõjuna nähtavat koormise võnkumist.

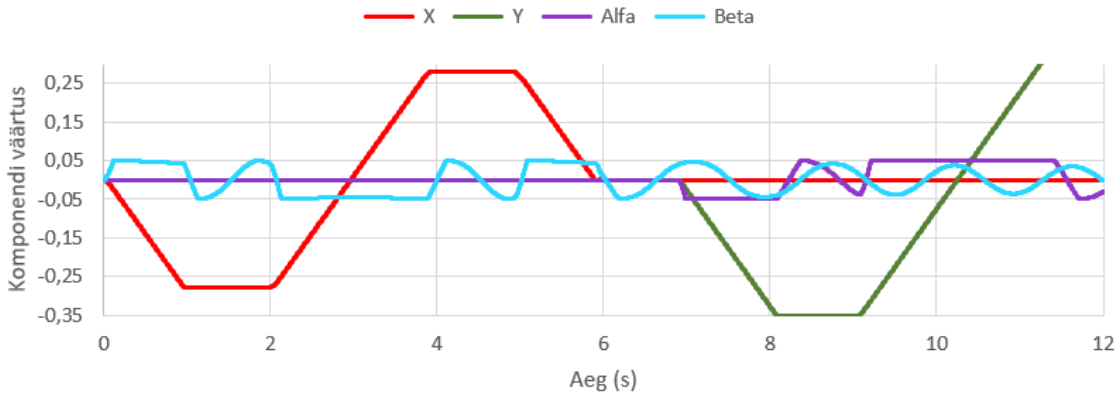
5.1 Lihtsustatud ja Täieliku mudeli võrdlus

Mudelite võrdluseks on loodud lihtne juhtimisalgoritm, mis 12 sekundi vältel joonistab kraana silla ja vankri liikumisega risti kuju (Joonis 44). Antud liikumismuster on piisavalt lihtne, et võimaldada komponentide analüüsimist graafikutelt. Komponentide sisenditena on kasutatud veojõudusid $F_{sild} = 30N,\ F_{vanker} = 30N,\ F_{vints} = 34N$.



Joonis 44. Mudelite võrdluseks loodud juhtalgoritm (X-punane, Y-roheline).

Simulatsiooni väljundid on esitatud graafikuga, millel on näidatud X, Y komponentide muutuste mõju nurkadele α, β (Joonis 45). Antud graafikutel on esitatud ainult 4 komponenti, et teha graafikute esitamine lihtsamaks. Samuti on need 4 komponenti sobivad võrdlemiseks 3DKraana laboratoorse mudeliga.

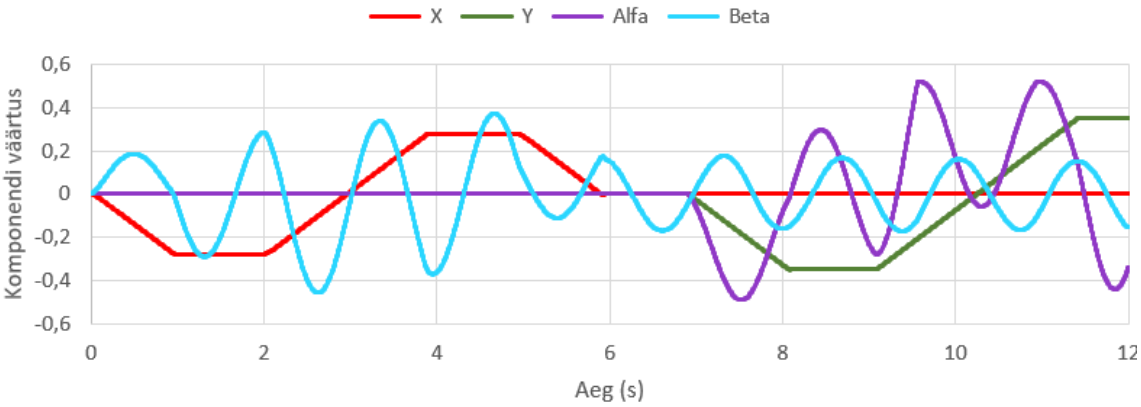


Joonis 45. Lihtsustatud mudeli Rist-juhtalgoritmi graafik.

Vaadates lihtsustatud mudeli graafikul X, β komponentide vahelist suhet, on esmalt näha lihtsustatud mudeli kehv võnkumismudel. Samuti tuleb ilmsiks lihtsustatud trossi nurga muutumises kerge viide. Kui sild jääb seisma, siis koormis jätkab teljel võnkumist. Mõlema komponendi liikumise puhul on jälgitav sarnane muster (Joonis 45).

Hoolikal vaatlemisel on graafikult võimalik täheldada ka kerge β võnkumise sumbumine. Kui on kiirendus X teljel, siis β väärtus muutub vastassuunas kuni komponendi piirideni. See nähtus kujutab endast 3DKraana vintsi piiret, mis takistab liiga suure võnkumise tekkimist [6].

Analüüsides täieliku mudeli käitumist, on näha väga suurt erinevust lihtsustatud mudeliga (Joonis 46).



Joonis 46. Täieliku mudeli Rist-juhtalgoritmi graafik.

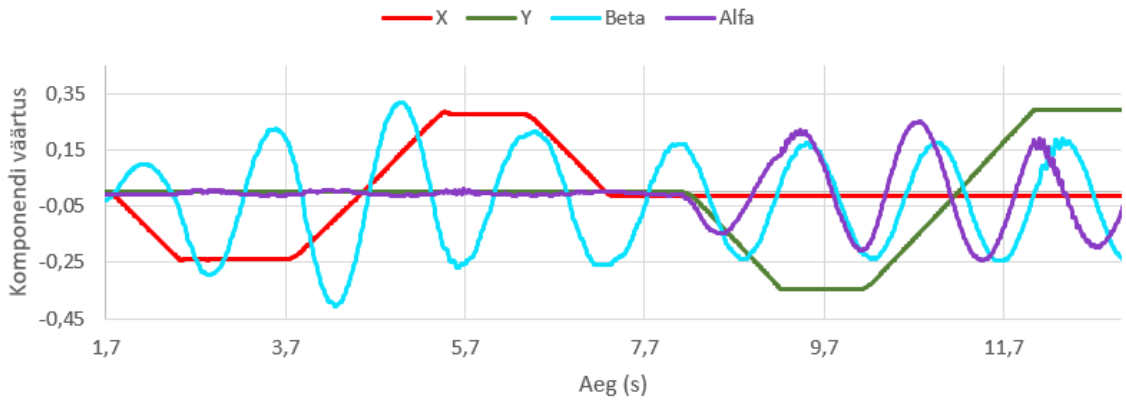
Täielikus mudelis on β võnkumine palju dünaamilisem ning ei järgi silla X väärtuse liikumist lineaarselt, kuigi siiski on seos nähtav. Avaldub ka sarnane võnkumismuster pärast silla liikumise seisma jäämist. Jälgitav on kerge β võnkumise sumbumine X teljel, mis on tingitud vankri ja koormise vahel mõjuvast reaktsioonijõust.

Kraana α komponendi muutumine on silmnähtavalt mittelineaarne ning saab lisamõjutusi teistest komponentidest (Joonis 46).

Lineaarse ja täieliku mudeli graafikute võrdlemisest saab teha järelduse, et lihtsustatud mudel käitub tõesti lineaarselt, mis ilmneb asjaolust, et komponendid ei mõjuta üksteist kaudselt ning on käitumiselt ettearvatavad.

Täieliku mudeli graafikust on nähtav, et komponendid mõjutavad üksteist mittelineaarselt, mida võib täheldada α, β nurkadele teistes telgedes rakenduvatest jõududest.

Järgnevalt on esile toodud 3DKraana laboratoorse mudeliga jäädvustatud andmete graafik (Joonis 47). Andmete salvestamisel on α, β mõõdetud teises skaalas kui simulatsioonis, kuid graafikult on siiski näha palju sarnasusi täieliku mudeliga.



Joonis 47. 3DKraana laboratoorse mudeli käsitsi juhitud Risti graafik.

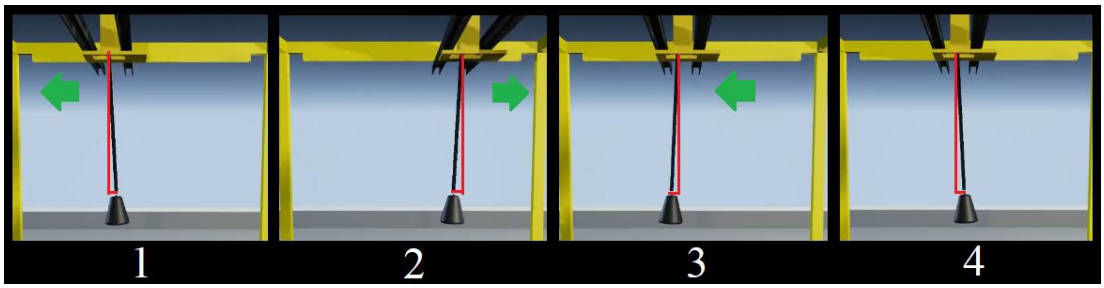
Täielik mudel on palju sarnasem reaalsusele, kuid siiski esineb võnkumiste suurustes erinevusi (Joonis 47, 8. sekund). Erinevused võivad olla tingitud ideaalse pendli mittetäielikust vastavusest reaalse pendliga, veojõudude suurustest kui ka juhtalgoritmide erinevustest.

Kokkuvõtteks võib järeldada, et täielik simulatsioon on adekvaatsem koormise pendlitaolise liikumise kirjeldamiseks, sest pendli liikumine on tuntud mittelineaarne probleem kinemaatikas [9].

5.2 Kraana visuaalne analüüs

Visuaalse võrdlemise jaoks on lindistatud mõlema simulatsioonimudeli käitumine UE4 keskkonnas ja 3DKraana laboratoorse mudeli käitumine A-Lab'is (Lisa 1, 3DKraana laboratoorne mudel). Antud lindistusest on tehtud kaadrite seeria erinevate etappide analüüsimiseks. Komponentide sisenditena on kasutatud $30N$ suuruseid veojõudusid, mis annab parema pildi võnkumise jälgimiseks.

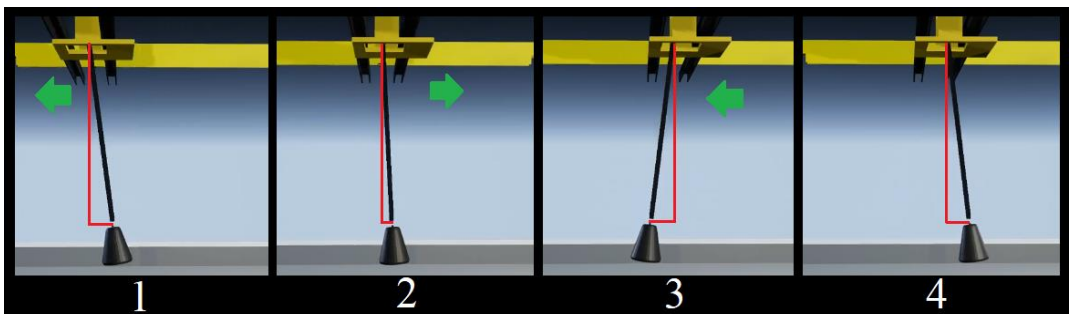
Seerias on jälgitud 3 põhilist etappi: kiirendamine vasakule (Joonis 48, 1), kiirendamine paremale (Joonis 48, 2), käitumine pärast tsentrisse naasmist (Joonis 48, 3 – 4).



Joonis 48. Lihtsustatud mudeli liikumisseeria UE4 keskkonnas.

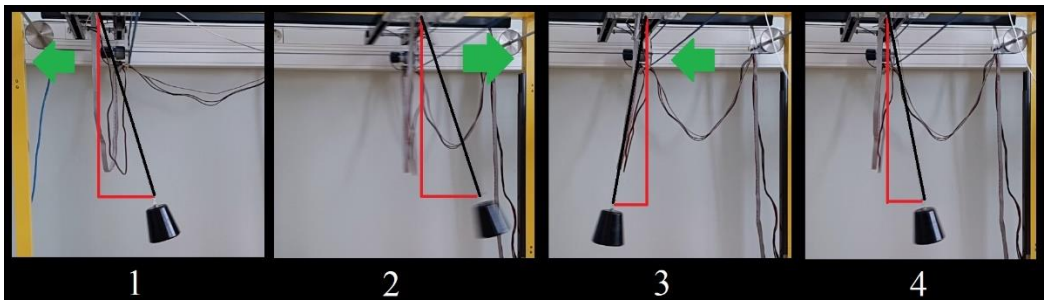
Lihtsustatud mudeli käitumisest on näha, et koormise asukoht on suures osas sõltuvuses vankri liikumise suunast ning võnkumine on piiratud vastavalt füüsikalises mudelis tehtud lihtsustustest. Sellest tulenevalt pole lihtsustatud mudeliga võimalik luua stabiliseerivaid juhtalgoritme [7].

Täieliku mudeli puhul on visuaalselt märgata palju tõetruumat pendli liikumist. Samuti on pendli liikumine proportsionaalselt suurem, pidurdades ka vankri liikumist. Teises etapis on näha et koormis võngub vankri liikumisest ettepoole (Joonis 49, 2). Viimastes sammudes on nähtav ka koormise võnkumise sumbumine (Joonis 49, 3 – 4).



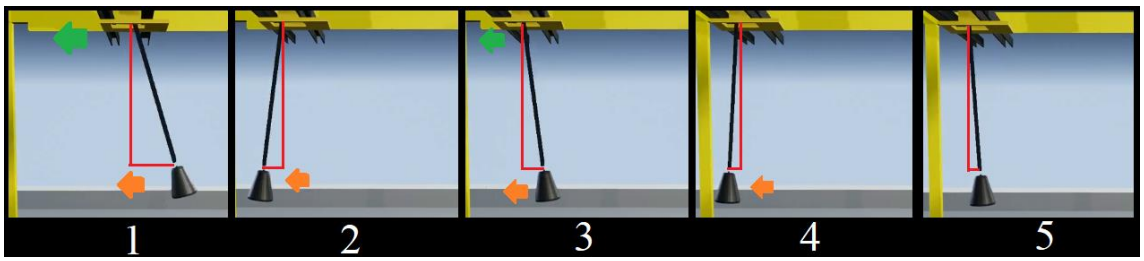
Joonis 49. Täieliku mudeli liikumisseeria UE4 keskkonnas.

Laboratoorse mudeli puhul on koormise võnkumine väga sarnane täielikule mudelile. Peamiseks erinevuseks on võnkumise suurus, mis on tingitud 3DKraana suurematest veojõududest, kuid üldises pildis on võnkumise muster väga sarnane (Joonis 50).



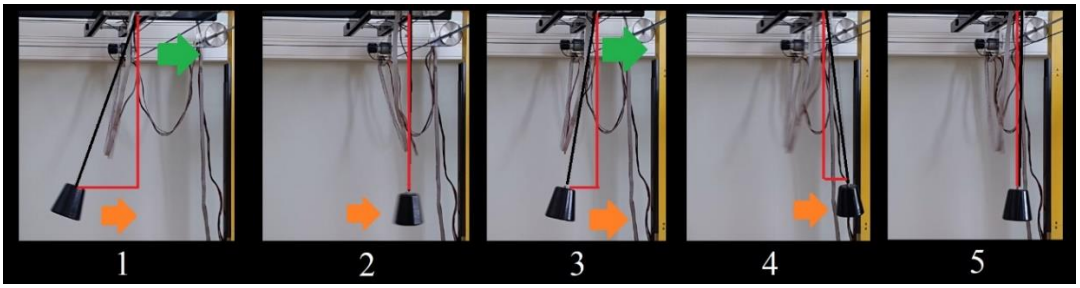
Joonis 50. 3DKraana laboratoorse mudeli liikumisseeria.

Veendumaks, et täielik mudel sobib juhtalgoritmide väljatöötamiseks, on viimase võrdlusena simuleeritud koormise võnkumise nullimine (Joonis 51).



Joonis 51. Täieliku mudeli võnkumise nullimine.

Rakendades pendli võnkumise hetkel õiges suunas veojõudu (Joonis 51, 1 – 4), sumbub kiiresti koormise võnkumine ja pendel naaseb tasakaaluseisu lähedale (Joonis 51, 5). See ühtib ka laboratoorse mudelis jälgitava käitumisega (Joonis 52).



Joonis 52. 3DKraana laboratoorse mudeli võnkumise nullimine.

Sellest tulenevalt on täielik mudel piisavalt võrdväärne 3DKraana laboratoorse mudeliga ja sobib stabiliseerivate juhtalgoritmide väljatöötamiseks virtuaalreaalsuse keskkonnas, mis saavutab antud diplomitöö eesmärgi.

Kokkuvõte

Antud diplomitöö käigus valmis ATI A-Lab'i jaoks eraldiseisev 3DKraana simulatsiooniteek "**Crane3dPlugin**" C++ keeles, mis on ühilduv Unreal Engine 4 arhitektuuriga ning võimaldab kraanat simuleerida virtuaalreaalsuses.

Töö käigus lihtsustati olemasoleva sildkraana füüsikalist mudelit, avaldades kõik valemid sõltumaks komponentide hetkkiirendustest. Samuti analüüsiti füüsikalises mudelis kasutatud hõõrdejõu mudelit, mis 3DKraana *User's Manualis* [6] oli poolikult kirjeldatud. Jõu rakendamise ja komponentide väärtuste piiramiseks sai välja töötatud uus lahendus (3.3, Hõõrdejõu rakendamine), mis MATLAB-Simulink versioonis puudus [2].

Lisaks tehti füüsikalise mudeli analüüsi käigus järeldus, et tervet füüsikalist mudelit oleks võimalik simuleerida vektoritega, kui on leitud reaktsioonivektor $\overrightarrow{S}$ [ptk 2.1]. Antud lähenemine lihtsustaks üldist lahendust oluliselt, kuid ei garanteeriks sama täpsust.

Simulatsiooniteegi arhitektuur on loodud uute simulatsioonimudelite lisamise võimaldamiseks, pakkudes vajalikud tööriistad füüsikaliseks integreerimises üle aja ning kontrolljõudude rakendamiseks koos hõõrdejõududega.

Ühtlasi on antud diplomitöö alusnäiteks tulevastele Unreal Engine 4-ga seonduvatele töödele A-Lab'is, et teha kergemaks virtuaalreaalsusega seotud lahenduste loomine.

Viimases peatükis analüüsiti simulatsiooni graafikuid ning leiti, et lihtsustatud mudeli puhul on tegemist lineaarse lähendusega, mis ei vasta reaalsusele. Täieliku mudeli analüüsi käigus leiti, et koormise võnkumine on mittelineaarne ning suures osas sarnane 3DKraana laboratoorse mudeli käitumisega.

Mudelite visuaalsel uurimisel UE4 keskkonnas oli selgelt näha, et täielik mudel on piisav stabiliseerivate juhtalgoritmide loomiseks, kuna toimub jõudude sumbumine. Sellega on saavutatud diplomitöö mõlemad peamised eesmärgid.

Kasutatud kirjandus

[1] F. Lamnabhi-Lagarrigue, “Systems & Control for the future of humanity, research agenda: Current and future roles, impact and grand challenges“, *Elsevier*, nr April, 2017.

[2] D. Freiberg, “*3DKraana modelleerimine ja simuleerimine virtuaalmaailmas*“, *BSc*, Tallinna Tehnikaülikool, Infotehnoloogia teaduskond, 2016.

[3] A. Aksjonov, “*3D Kraana juhtimissüsteem*“, *Magistritöö,* Tallinna Tehnikaülikool, Elektrotehnika instituut, 2015.

[4] Munck Cranes, [Võrgumaterjal]. munck-cranes.no.

[5] A. Lukašin, “Tööstuslikud elektriseadmed ja -paigaldised“, 2008-2013. [Võrgumaterjal].http://opiobjektid.tptlive.ee/Paigaldised/kraanad.html.

[6] INTECO, “3DCrane User's Manual“, 2016. [Võrgumaterjal]. https://a-lab.ee/man/3DCrane-user-manual.pdf.

[7] INTECO, “Mathematical model of the 3DCrane, lk 68“, 2016. [Võrgumaterjal]. https://a-lab.ee/man/3DCrane-user-manual.pdf.

[8] I. Newton, “Philosophiæ Naturalis Principia Mathematica“, 1687.

[9] D. Simpson, “The Nonlinear Pendulum“, 31 Detsember 2010. [Võrgumaterjal]. http://www.pgccphy.net/ref/nonlin-pendulum.pdf.

[10] J. Rebane, “Crane3dPlugin“, May 2019. [Võrgumaterjal]. https://github.com/RedFox20/Crane3dPlugin.

[11] Y. F. Liu, J. Li, Z. M. Zhang, X. H. Hu ja W. J. Zhang, “Experimental comparison of five friction models on the“, 2015. [Võrgumaterjal]. https://www.mech-sci.net/6/15/2015/ms-6-15-2015.pdf.

[12] “Verlet integration“, 2018. [Võrgumaterjal]. https://en.wikipedia.org/wiki/Verlet_integration.

[13] Epic Games, “An Introduction to UE4 Plugins“, 2018. [Võrgumaterjal]. https://wiki.unrealengine.com/An_Introduction_to_UE4_Plugins.

[14] EKI, “Eesti Keele Instituudi Keelenõu“, [Võrgumaterjal]. http://kn.eki.ee.

[15] Q. C. Nguyen, H. Q. Le ja K.-S. Hong, “Improving Control Performance of a Container Crane Using Adaptive Friction“, *International Conference on Control, Automation and Systems*, nr 14, 2014.

Lisa 1 – Digitaalsed Allikad

Projekti **Crane3dPlugin** lähtekood: https://github.com/RedFox20/Crane3dPlugin

Projekti **CraneVR** lähtekood: https://github.com/RedFox20/CraneVR

INTECO **3DCrane User's Manual**: https://a-lab.ee/man/3DCrane-user-manual.pdf

Lihtsustatud mudel UE4 keskkonnas reaalajas: https://youtu.be/iCY0CYne-IY

Täielik mudel UE4 keskkonnas reaalajas: https://youtu.be/8925NJwqJWE

INTECO **3DKraana laboratoorne mudel** A-Lab'is: https://youtu.be/zASyncrXt3M

Lisa 2 – Kraana simulatsioonimudeli lähtekood

Kinematics.h

```
// Copyright (c) 2019 - Jorma Rebane Crane3D
// Distributed under MIT License
#pragma once
#include <cmath>

namespace crane3d
{
    ////////////////////////////////////////////////////////////////////
    // Basic physics relations

    template<class T> struct Unit
    {
        double Value = 0.0;
        static constexpr Unit Zero() { return { 0.0 }; }
        Unit operator+(Unit b) const { return { Value + b.Value }; }
        Unit operator-(Unit b) const { return { Value - b.Value }; }
        bool operator>(Unit b) const { return Value > b.Value; }
        bool operator<(Unit b) const { return Value < b.Value; }
        Unit operator+(double x) const { return { Value + x }; }
        Unit operator-(double x) const { return { Value - x }; }
        Unit operator*(double x) const { return { Value * x }; }
        Unit operator/(double x) const { return { Value / x }; }
        bool operator>(double b) const { return Value > b; }
        bool operator<(double b) const { return Value < b; }
        bool operator==(double b) const { return Value == b; }
        bool operator!=(double b) const { return Value != b; }
        double operator/(Unit b) const { return Value / b.Value; }
        Unit operator-() const { return { -Value }; }
    };
    struct _Force {};
    struct _Mass  {};
    struct _Accel {};
    using Force = Unit<_Force>;
    using Mass  = Unit<_Mass>;
    using Accel = Unit<_Accel>;

    inline double sign(double x) { return x > 0.001 ? 1.0 : (x < -0.001 ? -1.0 : 0.0); }
    template<class T> inline double sign(Unit<T> x) { return sign(x.Value); }
    using std::abs;
    template<class T> inline Unit<T> abs(Unit<T> x) { return { std::abs(x.Value) }; }
    template<class T> inline Unit<T> operator*(double x, Unit<T> u) { return { x*u.Value }; }
    template<class T> inline bool operator>(double x, Unit<T> u) { return x > u.Value; }

    // F = ma
    inline Force operator*(Mass m, Accel a) { return { m.Value * a.Value }; }
    inline Force operator*(Accel a, Mass m) { return { m.Value * a.Value }; }
    // a = F/m
    inline Accel operator/(Force F, Mass m) { return { F.Value / m.Value }; }

    constexpr Force operator""_N (long double newtons) { return { double(newtons) }; }
    constexpr Mass  operator""_kg(long double kilos)   { return { double(kilos)   }; }
    constexpr Accel operator""_ms2(long double accel)  { return { double(accel)   }; }

    constexpr Force operator""_N (unsigned long long newtons) { return { double(newtons) }; }
    constexpr Mass  operator""_kg(unsigned long long kilos)   { return { double(kilos)   }; }
    constexpr Accel operator""_ms2(unsigned long long accel)  { return { double(accel)   }; }

    ////////////////////////////////////////////////////////////////////

    // compute new velocity using Euler's method:
    // v' = v + a*dt
    inline double integrate_euler_velocity(double v0, Accel a, double dt)
    {
        return v0 + a.Value*dt;
    }
```

```
    // compute new position using Euler's method:
    // x' = x + v'*dt
    inline double integrate_euler_pos(double x, double v1, double dt)
    {
        return x + v1*dt;
    }

    // integrate position using Velocity Verlet method:
    // x' = x + v*dt + (a*dt^2)/2
    inline double integrate_verlet_pos(double x, double v, Accel a, double dt)
    {
        return x + v*dt + a.Value*dt*dt*0.5;
    }

    // integrate velocity using Velocity Verlet method:
    // v' = v + (a0+a1)*0.5*dt
    inline double integrate_verlet_vel(double v, Accel a0, Accel a1, double dt)
    {
        return v + (a0.Value + a1.Value)*0.5*dt;
    }

    // calculate average velocity from position change
    // v_avg = dx / dt
    inline double average_velocity(double x1, double x2, double dt)
    {
        return (x2 - x1) / dt;
    }

    ////////////////////////////////////////////////////////////////////////

    inline double clamp(double x, double min, double max)
    {
        if (x <= min) return min;
        if (x >= max) return max;
        return x;
    }

    // dampens values that are very close to 0.0
    inline double dampen(double x)
    {
        return std::abs(x) < 0.001 ? 0.0 : x;
    }

    inline Force dampen(Force force)
    {
        return { std::abs(force.Value) < 0.001 ? 0.0 : force.Value };
    }

    inline bool inside_limits(double x, double min, double max)
    {
        return (min+0.01) < x && x < (max-0.01);
    }

    ////////////////////////////////////////////////////////////////////////
    // Math helper constants
    constexpr double _PI = 3.14159265358979323846;
    constexpr double _180degs = _PI;
    constexpr double _90degs = (_180degs / 2);
    constexpr double _60degs = _180degs/3.0;
    constexpr double _45degs = _180degs/4.0;
    constexpr double _30degs = _180degs/6.0;

    ////////////////////////////////////////////////////////////////////////
}
```

KinematicComponent.h

```
// Copyright (c) 2019 - Jorma Rebane Crane3D
// Distributed under MIT License
#pragma once
#include "Kinematics.h"

namespace crane3d
{
    ////////////////////////////////////////////////////////////////////////

    /**
     * Single force component with its own Position, Velocity, Acceleration and Net Force
     */
    struct Component
    {
        // base properties:
        Mass Mass = 1_kg;  // mass
        double Pos = 0.0;  // current position within limits
        double Vel = 0.0;  // instantaneous velocity
        Accel Acc = 0_ms2; // instantaneous acceleration

        // limits:
        double LimitMin = 0.0;
        double LimitMax = 0.0;
        double VelMax = 0.0; // velocity limit, 0 = disabled
        double AccMax = 0.0; // acceleration limit, 0 = disabled

        // driving forces:
        Force Applied;   // applied / driving force
        Force SFriction; // static friction
        Force KFriction; // kinematic friction
        Force Fnet;      // net driving force
        Accel NetAcc;    // net driving acceleration

        double FrictionDir = 1.0;

        // Coloumb-Viscous friction model
        // https://www.hindawi.com/journals/mpe/2013/946526/
        double CoeffStaticColoumb  = 5.0;   // resistance to starting movement
        double CoeffKineticViscous = 100.0; // resistance as velocity increases

        Component() = default;
        Component(double pos, double limitMin, double limitMax)
            : Pos{pos}, LimitMin{limitMin}, LimitMax{limitMax} {}

        void SetLimits(double min, double max) { LimitMin = min; LimitMax = max; }

        // Resets all dynamic variables: Pos, Vel, Acc, Fnet, NetAcc
        void Reset();

        // Update pos and vel using "Velocity Verlet" integration
        void Update(Accel new_acc, double dt);

        // Apply driving forces and friction forces
        void ApplyForce(Force applied);

        // Prevent applying force when against frame
        Force ClampForceByPosLimits(Force force) const;
    };

    ////////////////////////////////////////////////////////////////////////
}
```

KinematicComponent.cpp

```
// Copyright (c) 2019 - Jorma Rebane Crane3D
// Distributed under MIT License
#include "KinematicComponent.h"
#include <cmath>

namespace crane3d
{
    ////////////////////////////////////////////////////////////////////

    void Component::Reset()
    {
        Pos = Vel = 0.0;
        Acc = NetAcc = Accel::Zero;
        Applied = SFriction = KFriction = Fnet = Force::Zero;
    }

    void Component::Update(Accel newAcc, double dt)
    {
        double newPos = integrate_verlet_pos(Pos, Vel, Acc, dt);
        double newVel;
        if (inside_limits(newPos, LimitMin, LimitMax))
        {
            newVel = integrate_verlet_vel(Vel, Acc, newAcc, dt);
        }
        else // pos must be clamped, use Vavg
        {
            newPos = clamp(newPos, LimitMin, LimitMax);
            newVel = average_velocity(Pos, newPos, dt);
        }

        if (VelMax > 0.0 && std::abs(newVel) > VelMax) {
            newVel = sign(newVel) * VelMax;
        }
        Pos = newPos;
        Vel = newVel;
        Acc = newAcc;
    }

    // Simplified Coloumb-Viscous friction model:
    // F = f_c*sign(v) + f_v*v
    void Component::ApplyForce(Force applied)
    {
        applied = ClampForceByPosLimits(applied); // no Fapp at edges
        KFriction = Force{CoeffKineticViscous} * Vel;
        SFriction = Force{CoeffStaticColoumb} * sign(Vel);

        // Coloumb static friction: never greater than net force
        if (SFriction != 0.0 && abs(SFriction) > abs(Fnet))
            SFriction = sign(SFriction) * abs(Fnet);

        Applied = applied;
        Fnet = applied - FrictionDir * (SFriction + KFriction);
        Fnet = dampen(Fnet); // removes sigma sized force flip-flopping
        NetAcc = Fnet / Mass;
    }

    Force Component::ClampForceByPosLimits(Force force) const
    {
        Force F = FrictionDir*force;
        if (F > 0.0 && Pos > (LimitMax-0.01)) return Force::Zero;
        if (F < 0.0 && Pos < (LimitMin+0.01)) return Force::Zero;
        return force;
    }

    ////////////////////////////////////////////////////////////////////
}
```

Model.h

```
// Copyright (c) 2019 - Jorma Rebane Crane3D
// Distributed under MIT License
#pragma once
#include "KinematicComponent.h"
#include "ModelImplementation.h"
#include <vector>  // std::vector
#include <fstream> // std::ofstream
#include <memory>  // std::shared_ptr
#include <unordered_map> // std::unordered_map

namespace crane3d
{
    using std::vector;
    using std::shared_ptr;
    ////////////////////////////////////////////////////////////////////////

    // Coordinate system of the Crane model
    // X: outermost movement of the rail, considered as forward
    // Y: left-right movement of the cart
    // Z: up-down movement of the payload
    class Model
    {
    public:
        /** NOTE: These are the customization parameters of the model */
        Mass Mpayload = 1.000_kg; // Mc mass of the payload
        Mass Mcart    = 1.155_kg; // Mw mass of the cart
        Mass Mrail    = 2.200_kg; // Ms mass of the moving rail
        Accel g = 9.81_ms2; // gravity constant, 9.81m/s^2

        // Rail component
        //   describes distance of the rail from the center of the frame
        Component Rail { 0.0, -0.28, +0.28 };

        // Cart component
        //   describes distance of the cart from the center of the rail;
        Component Cart { 0.0, -0.39, +0.39 };

        // Line component
        //   describes the length of the lift-line
        Component Line { 0.5, +0.18, +0.70 };

        // Alfa component
        //   describes α angle between y axis (cart left-right) and the lift-line
        Component Alfa { 0.0, -0.05, +0.05 };

        // Beta component
        //   describes β angle between negative direction on the z axis and
        //   the projection of the lift-line onto the xz plane
        Component Beta { 0.0, -0.05, +0.05 };

        // Maximum Rail, Cart, Line component velocity
        double VelocityMax = 0.3; // m/s

        // Maximum Rail, Cart, Line component acceleration
        double AccelMax = 0.6; // m/s^2

    private:

        shared_ptr<IModelImplementation> CurrentModel;
        std::unordered_map<string, shared_ptr<IModelImplementation>> Models;

        double SimulationTimeSink = 0.0; // accumulator for running N iterations every update
        double TotalSimTime = 0.0; // total simulation time elapsed
        int64_t TotalUpdates = 0; // total number of discrete steps taken

        // for debugging:
        double DbgFixedTimeStep = 0.0;
        double DbgAvgIterations = 1.0;
        std::unique_ptr<std::ofstream> DbgCsv; // CSV output file stream
```

```
public:

    /**
     * Initialize model with a default model type. Throws if default model doesn't exist.
     */
    Model(const string& selectedModel = "Linear");

    Model(Model&&)      = delete; // NoMove
    Model(const Model&) = delete; // NoCopy
    Model& operator=(Model&&)      = delete; // NoMove
    Model& operator=(const Model&) = delete; // NoCopy

    /**
     * Resets all simulation components. Does not modify customization parameters.
     */
    void Reset();

    /**
     * Sets the simulation type and Resets the simulation if the type changed.
     * Throws if model with this name is not found.
     */
    void SetCurrentModelByName(const string& modelName);
    string GetCurrentModelName() const { return CurrentModel->Name(); }

    /** @return List of all supported model type names */
    vector<string> GetModelNames() const;

    /**
     * Register a new model implementation
     * @param setAsCurrent [false] If TRUE, sets the added model as CurrentModel
     */
    void AddModel(shared_ptr<IModelImplementation> model, bool setAsCurrent = false);

    /**
     * @return Current time state of the simulation
     */
    double GetSimulationTime() const { return TotalSimTime; }

    /**
     * Writes all simulation data points to a CSV file for later analysis
     * Columns exported:
     * t; X; Y; R; Alfa; Beta; pX; pY; pZ
     */
    void SetOutputCsv(const std::string& outCsvFile);

    /**
     * Updates the model using a fixed time step
     * @param fixedTime Size of the fixed time step. For example 0.01
     * @param elapsedTime Time since last update
     * @param Frail force driving the rail with cart (Fx)
     * @param Fcart force driving the cart along the rail (Fy)
     * @param Fwind force winding the lift-line (Fr)
     * @return New state of the crane model
     */
    CraneState UpdateFixed(double fixedTime, double elapsedTime,
                           Force Frail, Force Fcart, Force Fwind);

    /**
     * Updates the model using deltaTime as the time step.
     * This can be unstable if deltaTime varies.
     * @param deltaTimeStep Fixed time step
     * @param Frail force driving the rail with cart (Fx)
     * @param Fcart force driving the cart along the rail (Fy)
     * @param Fwind force winding the lift-line (Fr)
     * @return New state of the crane model
     */
    void Update(double deltaTimeStep, Force Frail, Force Fcart, Force Fwind);
```

```
        /**
         * @return Current state of the crane:
         *  distance of the rail, cart, length of lift-line and swing angles of the payload
         */
        CraneState GetState() const { return CurrentModel->GetState(); }

        /**
         * @return A full multi-line debug string with all dynamic variables shown
         */
        std::wstring GetStateDebugText() const;

    private:
        void AppendStateToCsv();
    };

    ////////////////////////////////////////////////////////////////////////
}
```

Model.cpp

```
// Copyright (c) 2019 - Jorma Rebane Crane3D
// Distributed under MIT License
#include "Model.h"
#include <cmath>
#include <cstdio>
#include <cstdarg>
#include <sstream> // std::stringstream
#include <iomanip> // std::setprecision
#include <algorithm>

namespace crane3d
{
    ////////////////////////////////////////////////////////////////////

    Model::Model(const string& selectedModel)
    {
        Rail.CoeffStaticColoumb = 5.0;
        Cart.CoeffStaticColoumb = 7.5;
        Line.CoeffStaticColoumb = 10.0;

        Rail.CoeffKineticViscous = 100.0;
        Cart.CoeffKineticViscous = 82.0;
        Line.CoeffKineticViscous = 75.0;

        Line.FrictionDir = -1.0;

        AddModel(std::make_shared<BasicLinearModel>(*this));
        AddModel(std::make_shared<NonLinearComplete>(*this));

        SetCurrentModelByName(selectedModel);
    }

    void Model::Reset()
    {
        SimulationTimeSink = 0.0;
        TotalSimTime = 0.0;
        TotalUpdates = 0;

        Rail.Reset();
        Cart.Reset();
        Line.Reset();
        Alfa.Reset();
        Beta.Reset();

        Line.Pos = 0.5;

        Rail.VelMax = VelocityMax;
        Cart.VelMax = VelocityMax;
        Line.VelMax = VelocityMax;

        Rail.AccMax = AccelMax;
        Cart.AccMax = AccelMax;
        Line.AccMax = AccelMax;

        if (CurrentModel->Name() == "Linear")
        {
            Alfa.SetLimits(-0.05, +0.05);
            Beta.SetLimits(-0.05, +0.05);
        }
        else
        {
            // Invert the alfa angle as required by non-linear models
            Alfa.Pos = _90degs;
            // the crane cannot physically swing more than X degrees due to cart edges
            Alfa.SetLimits(_60degs, _180degs - _60degs);
            Beta.SetLimits(-_30degs, _30degs);
        }
    }
```

```
void Model::SetCurrentModelByName(const string& modelName)
{
    auto model = Models.find(modelName);
    if (model == Models.end())
        throw std::runtime_error("Could not find model with id='"+modelName+"'");

    if (CurrentModel != model->second) {
        CurrentModel = model->second;
        Reset();
    }
}

vector<string> Model::GetModelNames() const
{
    vector<string> names;
    std::transform(Models.begin(), Models.end(), std::back_inserter(names),
                   [](const auto& kv){ return kv.first; });
    return names;
}

void Model::AddModel(shared_ptr<IModelImplementation> model, bool setAsCurrent)
{
    Models[model->Name()] = model;
    if (setAsCurrent) {
        SetCurrentModelByName(model->Name());
    }
}

void Model::SetOutputCsv(const std::string & outCsvFile)
{
    DbgCsv = std::make_unique<std::ofstream>(outCsvFile);
    if (!DbgCsv->is_open())
        throw std::runtime_error("Failed to create CSV file");
    std::locale mylocale("et_EE.UTF-8");
    DbgCsv->imbue(mylocale);
    *DbgCsv << "t; X; Y; R; Alfa; Beta; payX; payY; payZ\n";
    AppendStateToCsv();
}

void Model::AppendStateToCsv()
{
    CraneState s = GetState();
    *DbgCsv << TotalSimTime << "; " << Rail.Pos << "; "
            << Cart.Pos   << "; " << Line.Pos   << "; "
            << s.Alfa     << "; " << s.Beta     << "; "
            << s.PayloadX << "; " << s.PayloadY << "; " << s.PayloadZ << "\n";
}

static void format(std::wostream& out, const wchar_t* fmt, ...)
{
    const int max = 2048;
    wchar_t buf[max];
    va_list ap; va_start(ap, fmt);
    int len = _vsnwprintf_s(buf, max, fmt, ap);
    if (len < 0) { buf[max - 1] = L'\0'; len = (int)wcslen(buf); }
    out.write(buf, len);
}
```

```
    std::wstring Model::GetStateDebugText() const
    {
        CraneState s = GetState();
        std::wstringstream ss;
        format(ss, L"Model: %hs \n", CurrentModel->Name().c_str());
        format(ss, L" Mpayl %6.1fkg \n", Mpayload.Value);
        format(ss, L" payl %+6.2f, %+6.2f, %+6.2f \n",
                    s.PayloadX, s.PayloadY, s.PayloadZ);
        format(ss, L" pXYR %+6.2f, %+6.2f, %+6.2f \n",
                    s.RailOffset, s.CartOffset, s.LiftLine);
        format(ss, L" vXYR %+6.2f, %+6.2f, %+6.2f m/s \n",
                        Rail.Vel, Cart.Vel, Line.Vel);
        format(ss, L"  α  %+6.2f  vα %+6.2f rad/s  aα %+6.2f rad/s^2 \n",
                        Alfa.Pos, Alfa.Vel, Alfa.Acc.Value);
        format(ss, L"  β  %+6.2f  vβ %+6.2f rad/s  aβ %+6.2f rad/s^2 \n",
                        Beta.Pos, Beta.Vel, Beta.Acc.Value);
        auto formatComponent = [&](const char* which, const Component& c) {
            format(ss, L"  %hs a %+6.2f m/s², Fnet %+5.1f, Fapp %+5.1f, "
                       L"Fst %+5.1f, Fki %+5.1f \n",
                which, c.Acc.Value, c.Fnet.Value, c.Applied.Value,
                c.SFriction.Value, c.KFriction.Value);
        };
        formatComponent("Rail", Rail);
        formatComponent("Cart", Cart);
        formatComponent("Line", Line);
        format(ss, L"  iter# %5lld  dt %f  iters/update %.1f  iter/s %.0f\n",
            TotalUpdates, DbgFixedTimeStep, DbgAvgIterations, TotalUpdates/TotalSimTime);
        return ss.str();
    }

    ////////////////////////////////////////////////////////////////////

    CraneState Model::UpdateFixed(double fixedTime, double elapsedTime,
                                  Force Frail, Force Fcart, Force Fwind)
    {
        // we run the simulation with a constant time step
        SimulationTimeSink += elapsedTime;
        int iterations = static_cast<int>(SimulationTimeSink / fixedTime);
        SimulationTimeSink -= iterations * fixedTime;

        DbgFixedTimeStep = fixedTime;
        DbgAvgIterations = (DbgAvgIterations + iterations) * 0.5;

        for (int i = 0; i < iterations; ++i)
        {
            Update(fixedTime, Frail, Fcart, Fwind);
        }
        return GetState();
    }

    void Model::Update(double fixedTime, Force Frail, Force Fcart, Force Fwind)
    {
        ++TotalUpdates;
        TotalSimTime += fixedTime;

        Rail.Mass = Mrail+Mcart;
        Cart.Mass = Mcart;
        Line.Mass = Mpayload;
        CurrentModel->Update(fixedTime, Frail, Fcart, Fwind);

        if (DbgCsv) {
            AppendStateToCsv();
        }
    }

    ////////////////////////////////////////////////////////////////////
}
```

CraneState.h

```
// Copyright (c) 2019 - Jorma Rebane Crane3D
// Distributed under MIT License
#pragma once

namespace crane3d
{
    ////////////////////////////////////////////////////////////////////////

    /**
     * Output state of the model
     */
    struct CraneState
    {
        // X distance of the rail with the cart from
        // the center of the construction frame
        double RailOffset = 0.0;
        // Y distance of the cart from the center of the rail
        double CartOffset = 0.0;
        // R lift-line length
        double LiftLine = 0.0;

        // Payload 3D coordinates
        double PayloadX = 0.0;
        double PayloadY = 0.0;
        double PayloadZ = 0.0;

        void Print() const;
    };

    ////////////////////////////////////////////////////////////////////////
}
```

CraneState.cpp

```
// Copyright (c) 2019 - Jorma Rebane Crane3D
// Distributed under MIT License
#include "CraneState.h"
#include <cstdio>

namespace crane3d
{
    ////////////////////////////////////////////////////////////////////////

    void CraneState::Print() const
    {
        printf("Rail: %.2f Cart: %.2f Line: %.2f X: %.2f Y: %.2f Z: %.2f\n",
                RailOffset, CartOffset, LiftLine, PayloadX, PayloadY, PayloadZ);
    }

    ////////////////////////////////////////////////////////////////////////
}
```

ModelImplementation.h

```
// Copyright (c) 2019 - Jorma Rebane Crane3D
// Distributed under MIT License
#pragma once
#include "CraneState.h"
#include "KinematicComponent.h"
#include <string>  // std::wstring

namespace crane3d
{
    using std::string;
    ////////////////////////////////////////////////////////////////////////

    class Model;

    class IModelImplementation
    {
    protected:
        Model& Model;
        Component &Rail, &Cart, &Line, &Alfa, &Beta;
        Accel& g;
    public:
        IModelImplementation(crane3d::Model& model);

        /** @return Name of this model as an unique identifier */
        virtual string Name() const = 0;

        /**
         * Updates the model using deltaTime as the time step.
         * @param dt Fixed time step
         * @param Frail force driving the rail with cart (Fx)
         * @param Fcart force driving the cart along the rail (Fy)
         * @param Fwind force winding the lift-line (Fr)
         */
        virtual void Update(double dt, Force Frail, Force Fcart, Force Fwind) = 0;

        /** @return Current observable state of the Crane */
        virtual CraneState GetState() const = 0;
    };


    ////////////////////////////////////////////////////////////////////////


    // The most basic crane model with minimum pendulum movement
    class BasicLinearModel : public IModelImplementation
    {
    public:
        using IModelImplementation::IModelImplementation;
        string Name() const override { return "Linear"; }
        void Update(double dt, Force Frail, Force Fcart, Force Fwind) override;
        CraneState GetState() const override;
    };


    ////////////////////////////////////////////////////////////////////////


    // Base abstract class for non-linear models
    class NonLinearModel : public IModelImplementation
    {
    public:
        using IModelImplementation::IModelImplementation;
        CraneState GetState() const override;
    };
```

```
    // Non-linear fully dynamic model with all 3 forces
    class NonLinearComplete : public NonLinearModel
    {
    public:
        using NonLinearModel::NonLinearModel;
        string Name() const override { return "NonLinearComplete"; }
        void Update(double dt, Force Frail, Force Fcart, Force Fwind) override;
    };

    ////////////////////////////////////////////////////////////////////
}
```

ModelImplementation.cpp

```
// Copyright (c) 2019 - Jorma Rebane Crane3D
// Distributed under MIT License
#include "ModelImplementation.h"
#include "Model.h"

namespace crane3d
{
    ////////////////////////////////////////////////////////////////////

    IModelImplementation::IModelImplementation(crane3d::Model& model)
        : Model{model}, Rail{model.Rail}, Cart{model.Cart},
          Line{model.Line}, Alfa{model.Alfa}, Beta{model.Beta}, g{model.g}
    {
    }

    ////////////////////////////////////////////////////////////////////

    void BasicLinearModel::Update(double dt, Force Frail, Force Fcart, Force Fwind)
    {
        Rail.UpdateForce(Frail);
        Cart.UpdateForce(Fcart);
        Line.UpdateForce(Fwind);

        // cable driven tension/friction coefficients:
        double μ1 = Model.Mpayload / Cart.Mass; // μ1 = Mc / Mw
        double μ2 = Model.Mpayload / Rail.Mass; // μ2 = Mc / (Mw + Ms)
        double R = Line.Pos;

        Accel aX = Rail.NetAcc + Line.NetAcc*μ2*Beta.Pos;
        Accel aY = Cart.NetAcc - Line.NetAcc*μ1*Alfa.Pos;
        Accel aA =  (Cart.Acc - g*Alfa.Pos - 2*Alfa.Vel*Line.Vel) / R;
        Accel aB = -(Rail.Acc + g*Beta.Pos + 2*Beta.Vel*Line.Vel) / R;
        Accel aR = g - Line.NetAcc;

        Rail.Update(aX, dt);
        Cart.Update(aY, dt);
        Alfa.Update(aA, dt);
        Beta.Update(aB, dt);
        Line.Update(aR, dt);
        if (Fwind == 0.0)
            Line.Pos = R;
    }

    CraneState BasicLinearModel::GetState() const
    {
        CraneState s { Rail.Pos, Cart.Pos, Line.Pos };
        s.PayloadX = s.RailOffset + s.LiftLine * Beta.Pos;
        s.PayloadY = s.CartOffset - s.LiftLine * Alfa.Pos;
        s.PayloadZ = -s.LiftLine;
        return s;
    }

    ////////////////////////////////////////////////////////////////////
```

```
    ////////////////////////////////////////////////////////////////////

    void NonLinearComplete::Update(double dt, Force Frail, Force Fcart, Force Fwind)
    {
        Rail.UpdateForce(Frail);
        Cart.UpdateForce(Fcart);
        Line.UpdateForce(Fwind);

        double sA = sin(Alfa.Pos), cA = cos(Alfa.Pos);
        double sB = sin(Beta.Pos), cB = cos(Beta.Pos);
        double R = Line.Pos;
        double G = g.Value;
        double vB2 = Beta.Vel*Beta.Vel;
        double VA = R*vB2*cA*sA - 2*Line.Vel*Alfa.Vel + G*cA*cB;
        double VB = 2*Beta.Vel*(R*Alfa.Vel*cA + Line.Vel*sA) + G*sB;
        double VR = R*vB2*sA*sA + G*sA*cB + R*Alfa.Vel*Alfa.Vel;
        // cable driven tension/friction coefficients:
        double μ1 = Model.Mpayload / Cart.Mass; // μ1 = Mc / Mw
        double μ2 = Model.Mpayload / Rail.Mass; // μ2 = Mc / (Mw + Ms)

        Accel aX = Rail.NetAcc + Line.NetAcc*μ2*sA*sB;
        Accel aY = Cart.NetAcc + Line.NetAcc*μ1*cA;
        Accel aA = (Cart.NetAcc*sA - Rail.NetAcc*cA*sB +
                    (μ1 - μ2*sB*sB)*Line.NetAcc*cA*sA + VA) / R;
        Accel aB = -(Rail.NetAcc*cB + Line.NetAcc*μ2*sA*cB*sB + VB) / (R*sA);
        Accel aR = - Cart.NetAcc*cA - Rail.NetAcc*sA*sB
                    - Line.NetAcc*(1 + μ1*cA*cA + μ2*sA*sA*sB*sB) + VR;

        Rail.Update(aX, dt);
        Cart.Update(aY, dt);
        Alfa.Update(aA, dt);
        Beta.Update(aB, dt);
        Line.Update(aR, dt);
        if (Fwind == 0.0)
            Line.Pos = R;
    }

    CraneState NonLinearModel::GetState() const
    {
        CraneState s { Rail.Pos, Cart.Pos, Line.Pos };
        double A = Alfa.Pos, B = Beta.Pos;
        s.PayloadX = s.RailOffset + s.LiftLine * sin(A) * sin(B);
        s.PayloadY = s.CartOffset + s.LiftLine * cos(A);
        s.PayloadZ = -s.LiftLine * sin(A) * cos(B);
        return s;
    }

    ////////////////////////////////////////////////////////////////////
}
```

CraneSimulationComponent.h

```
// Copyright (c) 2019 - Jorma Rebane 3DCrane UE4
// Distributed under MIT License
#pragma once
#include "CoreMinimal.h"
#include "Components/ActorComponent.h"
#include "Components/SceneComponent.h"
#include "Model.h"
#include <memory>
#include "CraneSimulationComponent.generated.h"

/**
 * Type of crane simulation used by CraneSimulationComponent
 */
UENUM(BlueprintType)
enum class ECraneModelType : uint8
{
    // Non-linear fully dynamic model with all 3 forces
    NonLinearComplete,

    // The most basic crane model with minimum pendulum movement
    Linear,
};


/**
 * Adapts crane simulation and parameters as an actor component
 */
UCLASS( ClassGroup=(Custom), meta=(BlueprintSpawnableComponent) )
class CRANEMODEL_API UCraneSimulationComponent : public UActorComponent
{
    GENERATED_BODY()

    std::unique_ptr<crane3d::Model> Model; // need a stable pointer for UE4 editor.

public:

    // This should be an invisible node which marks the 0,0,0 of the crane system
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Crane Components")
    USceneComponent* CenterComponent = nullptr;

    // crane rail (X-axis in the model)
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Crane Components")
    USceneComponent* RailComponent = nullptr;

    // crane cart (Y-axis in the model)
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Crane Components")
    USceneComponent* CartComponent = nullptr;

    // payload at PayloadPosition
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Crane Components")
    USceneComponent* PayloadComponent = nullptr;

    // Input force driving the rail (reset every tick) - see AddRailX()
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Crane Force Inputs")
    float ForceRail = 0;

    // Input force driving the cart (reset every tick) - see AddCartY()
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Crane Force Inputs")
    float ForceCart = 0;

    // Input force driving the cable winding (reset every tick) - see AddWindingZ()
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Crane Force Inputs")
    float ForceWinding = 0;

    // Crane simulation type
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Crane Parameters")
    ECraneModelType ModelType = ECraneModelType::NonLinearComplete;
```

```
    // Mass of the rail (kg)
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Crane Parameters")
    float RailMass = 2.2f;

    // Mass of the cart (kg)
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Crane Parameters")
    float CartMass = 1.155f;

    // Mass of the payload (kg)
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Crane Parameters")
    float PayloadMass = 1.0f;

    // Gravity constant (m/s^2)
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Crane Parameters")
    float Gravity = 9.81f;

    // Min limit for the rail (cm)
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Crane Limits")
    float RailLimitMin = -28;

    // Max limit for the rail (cm)
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Crane Limits")
    float RailLimitMax = +28;

    // Min limit for the cart (cm)
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Crane Limits")
    float CartLimitMin = -39;

    // Max limit for the cart (cm)
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Crane Limits")
    float CartLimitMax = +39;

    // Min limit for the line (cm)
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Crane Limits")
    float LineLimitMin = 18;

    // Max limit for the line (cm)
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Crane Limits")
    float LineLimitMax = 70;

    // OUTPUT: X-offset of the rail and Y-offset of the cart (cm)
    UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "Crane Outputs")
    FVector CartPosition;

    // OUTPUT: Payload 3D position, offset to CenterComponent
    UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "Crane Outputs")
    FVector PayloadPosition;

    UCraneSimulationComponent();

    UFUNCTION(BlueprintCallable, Category = "Crane Initializer")
    void SetCraneComponents(USceneComponent* center,
                            USceneComponent* rail,
                            USceneComponent* cart,
                            USceneComponent* payload);

    // Adds force to the rail across the X axis
    UFUNCTION(BlueprintCallable, Category = "Crane Force Inputs")
    void AddRailX(float axisValue, float multiplier);

    // Adds force to the cart across the Y axis
    UFUNCTION(BlueprintCallable, Category = "Crane Force Inputs")
    void AddCartY(float axisValue, float multiplier);

    // Adds force to the winding mechanism across the Z axis
    UFUNCTION(BlueprintCallable, Category = "Crane Force Inputs")
    void AddWindingZ(float axisValue, float multiplier);
```

```
    // Switches to the next simulation model
    UFUNCTION(BlueprintCallable, Category = "Crane Misc. Inputs")
    void NextModelType();

protected:
    void BeginPlay() override;
    void UpdateVisibleFields(const crane3d::CraneState& state);
    void UpdateModelParameters();
    void UpdateVisibleComponents();

public:
    void TickComponent(float DeltaTime, ELevelTick TickType,
                       FActorComponentTickFunction* ThisTickFunction) override;
};
```

CraneSimulationComponent.cpp

```
// Copyright (c) 2019 - Jorma Rebane 3DCrane UE4
// Distributed under MIT License
#include "CraneSimulationComponent.h"
#include "EngineMinimal.h"
#include "DrawDebugHelpers.h"

UCraneSimulationComponent::UCraneSimulationComponent()
{
    PrimaryComponentTick.bCanEverTick = true;
}

void UCraneSimulationComponent::SetCraneComponents(USceneComponent* center,
                                                   USceneComponent* rail,
                                                   USceneComponent* cart,
                                                   USceneComponent* payload)
{
    CenterComponent = center;
    RailComponent = rail;
    CartComponent = cart;
    PayloadComponent = payload;
}

void UCraneSimulationComponent::AddRailX(float axisValue, float multiplier)
{
    ForceRail = axisValue * multiplier;
}
void UCraneSimulationComponent::AddCartY(float axisValue, float multiplier)
{
    ForceCart = axisValue * multiplier;
}
void UCraneSimulationComponent::AddWindingZ(float axisValue, float multiplier)
{
    ForceWinding = axisValue * multiplier;
}

static std::string to_string(ECraneModelType type)
{
    if (UEnum* uenum = FindObject<UEnum>(ANY_PACKAGE, TEXT("ECraneModelType"), true))
    {
        FString typeName = uenum->GetNameStringByValue((int64)type);
        return TCHAR_TO_ANSI(*typeName);
    }
    return {};
}

void UCraneSimulationComponent::BeginPlay()
{
    Super::BeginPlay();

    Model = std::make_unique<crane3d::Model>(to_string(ModelType));
    UpdateModelParameters();
    UpdateVisibleFields(Model->GetState());
}

void UCraneSimulationComponent::NextModelType()
{
    if (UEnum* uenum = FindObject<UEnum>(ANY_PACKAGE, TEXT("ECraneModelType"), true))
    {
        int next = (int)ModelType + 1;
        if (next > (int)ECraneModelType::Linear) next = 0;
        ModelType = (ECraneModelType)next;
    }
}

void UCraneSimulationComponent::UpdateVisibleFields(const crane3d::CraneState& state)
{
    // UE4 is in centimeters, crane model is in meters
    CartPosition.X = float(state.RailOffset * 100);
    CartPosition.Y = float(state.CartOffset * 100);
    PayloadPosition = FVector(state.PayloadX, state.PayloadY,state.PayloadZ) * 100;
```

```
    auto text = Model->GetStateDebugText();
    GEngine->AddOnScreenDebugMessage(1, 5.0f, FColor::Red, FString::Printf(L"Frail: %.2f N", ForceRail));
    GEngine->AddOnScreenDebugMessage(2, 5.0f, FColor::Red, FString::Printf(L"Fcart: %.2f N", ForceCart));
    GEngine->AddOnScreenDebugMessage(3, 5.0f, FColor::Red, FString::Printf(L"Fwndg: %.2f N", ForceWinding));
    GEngine->AddOnScreenDebugMessage(4, 5.0f, FColor::Red, FString{text.c_str()});
}

void UCraneSimulationComponent::UpdateModelParameters()
{
    // update model with parameters
    // we do this every frame to allow full dynamic tweaking of
    // the crane while the game is running
    Model->Mrail = crane3d::Mass{RailMass};
    Model->Mcart = crane3d::Mass{CartMass};
    Model->Mpayload = crane3d::Mass{PayloadMass};
    Model->g = crane3d::Accel{Gravity};

    Model->Rail.LimitMin = RailLimitMin / 100.0f;
    Model->Rail.LimitMax = RailLimitMax / 100.0f;
    Model->Cart.LimitMin = CartLimitMin / 100.0f;
    Model->Cart.LimitMax = CartLimitMax / 100.0f;
    Model->Line.LimitMin = LineLimitMin / 100.0f;
    Model->Line.LimitMax = LineLimitMax / 100.0f;

    Model->SetCurrentModelByName(to_string(ModelType));
}

void UCraneSimulationComponent::TickComponent(float DeltaTime,
    ELevelTick TickType, FActorComponentTickFunction* ThisTickFunction)
{
    Super::TickComponent(DeltaTime, TickType, ThisTickFunction);

    UpdateModelParameters();

    constexpr double ItersPerSecond = 1'000'000.0;
    constexpr double FixedTimeStep = 1.0 / ItersPerSecond;

    using crane3d::Force;
    crane3d::CraneState state = Model->UpdateFixed(FixedTimeStep, DeltaTime,
                Force{ForceRail}, Force{ForceCart}, Force{ForceWinding});

    UpdateVisibleFields(state);
    UpdateVisibleComponents();

    // reset input forces for this frame
    ForceRail = ForceCart = ForceWinding = 0.0;
}

void UCraneSimulationComponent::UpdateVisibleComponents()
{
    if (!CenterComponent || !PayloadComponent || !CartComponent || !RailComponent)
        return;

    FVector railPos = RailComponent->RelativeLocation;
    FVector cartPos = RailComponent->RelativeLocation;
    railPos.X = CartPosition.X;
    cartPos.X = CartPosition.X;
    cartPos.Y = CartPosition.Y;
    RailComponent->SetRelativeLocation(railPos);
    CartComponent->SetRelativeLocation(cartPos);
    PayloadComponent->SetRelativeLocation(PayloadPosition);

    // rotate the payload towards cart
    FVector dir = CartComponent->GetComponentLocation()
                - PayloadComponent->GetComponentLocation();
    dir.Normalize();
    FRotator rot = dir.Rotation();
    rot.Pitch -= 90;
    PayloadComponent->SetWorldRotation(rot);

    // cable line
    DrawDebugLine(GetWorld(), CartComponent->GetComponentLocation(),
        PayloadComponent->GetComponentLocation(),
        FColor(15, 15, 15), false, -1, 0, 1);
}
```

CraneController.h

```
// Copyright (c) 2019 - Jorma Rebane Crane3D
// Distributed under MIT License
#pragma once
#include "Model.h"
#include <vector>
#include <deque>

namespace crane3d
{
    ////////////////////////////////////////////////////////////////////

    struct WayPoint
    {
        double X = 0.0;
        double Y = 0.0;
        double R = 0.0;
        /**
         * Duration in seconds for following this waypoint
         * Duration <= 0.0: no duration, terminate waypoint when we reach it
         */
        double Duration = 0.0;
    };

    /**
     * Simple waypoint based controller of the crane
     */
    class CraneController
    {
        crane3d::Model* Model = nullptr;
        Force Frail, Fcart, Fwind;
        std::deque<WayPoint> WayPoints;

    public:

        CraneController(crane3d::Model* model);

        /** Max driving forces applied when following waypoints */
        void SetDrivingForces(Force FrailMax, Force FcartMax, Force FwindMax);

        /**
         * Adds a new waypoint to follow.
         * @note X,Y,R are clamped to Model limits to prevent unreachable coordinates
         * @param X Desired X position
         * @param Y Desired Y position
         * @param R Desired R position
         * @param duration [0.0] Number of seconds to follow+stay at target pos.
         *     If duration <= 0.0: terminate waypoint immediately when we reach it.
         */
        void AddWayPoint(double X, double Y, double R, double duration = 0.0)
        {
            AddWayPoint(WayPoint{ X, Y, R, duration });
        }

        void AddWayPoint(WayPoint p);

        /** Set waypoints for the crane to follow */
        void SetWayPoints(const std::vector<WayPoint>& wayPoints);

        /** Clears the current waypoint list */
        void ClearWayPoints();

        /**
         * Runs the simulation for the specified number of seconds while following waypoints.
         * @param fixedTimeStep Small delta time
         * @param runTimeSeconds Total run time of the controller
         */
        void Run(double fixedTimeStep, double runTimeSeconds);
    };
    ////////////////////////////////////////////////////////////////////
}
```

CraneController.cpp

```
// Copyright (c) 2019 - Jorma Rebane Crane3D
// Distributed under MIT License
#include "CraneController.h"
#include <cassert>

namespace crane3d
{
    ////////////////////////////////////////////////////////////////////

    CraneController::CraneController(crane3d::Model* model) : Model{model}
    {
        assert(Model != nullptr && "CraneController Model* cannot be null!");
    }
    void CraneController::SetDrivingForces(Force FrailMax, Force FcartMax, Force FwindMax)
    {
        Frail = FrailMax, Fcart = FcartMax, Fwind = FwindMax;
    }

    void CraneController::AddWayPoint(WayPoint p)
    {
       p.X = clamp(p.X, Model->Rail.LimitMin, Model->Rail.LimitMax);
       p.Y = clamp(p.Y, Model->Cart.LimitMin, Model->Cart.LimitMax);
       p.R = clamp(p.R, Model->Line.LimitMin, Model->Line.LimitMax);
       WayPoints.push_back(p);
    }

    void CraneController::SetWayPoints(const std::vector<WayPoint>& wayPoints)
    {
        ClearWayPoints();
        for (const WayPoint& p : wayPoints)
            AddWayPoint(p);
    }

    void CraneController::ClearWayPoints() { WayPoints.clear(); }

    void CraneController::Run(double fixedTimeStep, double runTimeSeconds)
    {
        double simTime = 0.0;
        for (; simTime < runTimeSeconds; simTime += fixedTimeStep)
        {
            WayPoint wp = WayPoints.empty() ? WayPoint{ 0.0, 0.0, 0.5 } : WayPoints.front();
            double dx = (wp.X - Model->Rail.Pos);
            double dy = (wp.Y - Model->Cart.Pos);
            double dr = (wp.R - Model->Line.Pos);

            if (!WayPoints.empty())
            {
                WayPoint& first = WayPoints.front();
                if (first.Duration > 0.0) // this is a timed waypoint
                {
                    first.Duration -= fixedTimeStep;
                    if (first.Duration < 0.0) {
                        WayPoints.pop_front();
                    }
                }
                else // this is an immediate waypoint, terminate if we arrive closeby
                {
                    if (abs(dx) < 0.001 && abs(dy) < 0.001 && abs(dr) < 0.001) {
                        WayPoints.pop_front();
                    }
                }
            }

            Force FdriveRail = sign(dx) * Frail;
            Force FdriveCart = sign(dy) * Fcart;
            Force FdriveLine = sign(dr) * Fwind;

            Model->Update(fixedTimeStep, FdriveRail, FdriveCart, FdriveLine);
        }
    }
    ////////////////////////////////////////////////////////////////////
}
```