Okruženje za simuliranje postrojenja zasnovano na Unity 3D razvojnom okviru

Hodak, David

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Sveučilišni studij

OKRUŽENJE ZA SIMULIRANJE POSTROJENJA ZASNOVANO NA UNITY 3D RAZVOJNOM OKVIRU

Diplomski rad

David Hodak

Osijek, 2024

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA **OSIJEK**

Obrazac D1: Obrazac za ocjenu diplomskog rada na sveučilišnom diplomskom studiju

Ocjena diplomskog rada na sveučilišnom diplomskom studiju

Ime i prezime pristupnika:	David Hodak
Studij, smjer:	Sveučilišni diplomski studij Računarstvo
Mat. br. pristupnika, god.	D-1122R, 13.10.2020.
JMBAG:	0165071284
Mentor:	izv. prof. dr. sc. Damir Filko
Sumentor:	
Sumentor iz tvrtke:	
Predsjednik Povjerenstva:	doc. dr. sc. Petra Pejić
Član Povjerenstva 1:	izv. prof. dr. sc. Damir Filko
Član Povjerenstva 2:	izv. prof. dr. sc. Emmanuel Karlo Nyarko
Naslov diplomskog rada:	Okruženje za simuliranje postrojenja zasnovano na Unity 3D razvojnom okviru
Znanstvena grana diplomskog rada:	**Programsko inženjerstvo (zn. polje računarstvo)**
Zadatak diplomskog rada:	Razviti program koji će omogućiti simuliranje tipičnih industrijskih postrojenja (npr. pokretnih traka, dizala, manipulatora i sl.) koristeći Unity 3D okvir i C# programski jezik. Program treba imati mogućnost učitavanja i simuliranja rada proizvoljnog tipičnog industrijskog postrojenja te provođenje sustava upravljanja takvim postrojenjem. Također, u okviru ovog diplomskog rada potrebno je realizirati manju bazu prototipnih dijelova postrojenja (pokretna traka, senzori, tipkala, prekidači, žarulje, 1D manipulatori i sl.) čije slaganje će omogućiti realizaciju proizvoljnog broja tipičnih
Datum ocjene pismenog dijela diplomskog rada od strane mentora:	05.09.2024.
Ocjena pismenog dijela diplomskog rada od strane mentora:	Izvrstan (5)
Datum obrane diplomskog rada:	17.09.2024.
Ocjena usmenog dijela diplomskog rada (obrane):	Izvrstan (5)
Ukupna ocjena diplomskog rada:	Izvrstan (5)
Datum potvrde mentora o predaji konačne verzije diplomskog rada čime je pristupnik završio sveučilišni diplomski studij:	19.09.2024.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA **OSIJEK**

IZJAVA O IZVORNOSTI RADA

Osijek, 19.09.2024.

Ime i prezime Pristupnika:	David Hodak
Studij:	Sveučilišni diplomski studij Računarstvo
Mat. br. Pristupnika, godina upisa:	D-1122R, 13.10.2020.
Turnitin podudaranje [%]:	6

Ovom izjavom izjavljujem da je rad pod nazivom**: Okruženje za simuliranje postrojenja zasnovano na Unity 3D razvojnom okviru**

izrađen pod vodstvom mentora izv. prof. dr. sc. Damir Filko

i sumentora

moj vlastiti rad i prema mom najboljem znanju ne sadrži prethodno objavljene ili neobjavljene pisane materijale drugih osoba, osim onih koji su izričito priznati navođenjem literature i drugih izvora informacija.
Izjavljujem da je intelektualni sadržaj navedenog rada proizvod mog vlastitog rada, osim u onom dijelu za koji mi je bila potrebna pomoć mentora, sumentora i drugih osoba, a što je izričito navedeno u radu.

Potpis pristupnika:

SADRŽAJ

1. UVOD

Virtualna okruženja za simuliranje stvarnih tvorničkih procesa u svrhu testiranja postala su jedan od glavnih alata industrije. Napredak softvera i hardvera omogućuje jednostavniji i brži razvoj što realističnijih okruženja, s naglaskom na simulaciju fizike stvarnog svijeta. Štedi se vrijeme i novac koje bi bilo utrošeno na materijal, ljudske resurse, skladištenje, rješavanje otpada i izgradnju, te se ljude osigurava od potencijalno opasnih testnih poligona. Virtualne simulacije kao i stvarna industrijska postrojenja imaju mogućnost upravljanja kroz PLC programe. Dolaze sa svojim verzijama kontrole, povezuju se s nekim od PLC simulatora ili se PLC programi integriraju u simulaciju.

Upravo je u ovom diplomskom radu kreiran jedan takav testni poligon koji predstavlja malo tvorničko postrojenje te ga se može upravljati kroz jednostavnu web aplikaciju. Ono se sastoji od tipičnih industrijskih elemenata (pokretne trake, strojevi, manipulatori itd.) čije slaganje omogućuje realizaciju proizvoljnog broja postrojenja. Cijeli sustav izrađen je u Unity programskom okruženju.

Korištena je najnovija verziji Unitya softvera, a napredak projekta se spremao na alat za upravljanje izvornim kodom Plastic SCM, koji prikladno, osim programskog koda, sprema i ostale potrebne segmente – slike, videa, 3D modele itd. Za kreiranje i prilagodbu 3D modela koji služe kao elementi postrojenja koristio se Blender, uz pripomoć službene Unity trgovine gotovih 3D modela koji su ubrzali razvoj simulacije. Web aplikacija za kontrolu virtualnog postrojenja je kreirana u Angularu i .NET platformi.

Potrebno je kreirati sustav koji će omogućiti korisniku stvaranje proizvoljnog broja postrojenja uz mogućnost njihovog spremanja za buduće korištenje. Cilj je razviti modularni sustav koji će se moći nadograđivati, prikazati životni ciklus razvoja projekta okruženja za simulaciju te kroz primjer pokazati zašto je ova metoda alat budućnosti.

U nastavku rada ukratko je opisan PLC i način rada, a zatim korištene tehnologije s naglaskom na Unity razvojni okvir. Nakon toga detaljno se objašnjava izrada projekta, rezultati i problemi.

1.1. Zadatak diplomskog rada

Razviti program koji će omogućiti simuliranje tipičnih industrijskih postrojenja (npr. pokretnih traka, dizala, manipulatora i sl.) koristeći Unity 3D okvir i C# programski jezik. Program treba imati mogućnost učitavanja i simuliranja rada proizvoljnog tipičnog industrijskog postrojenja te

provođenje sustava upravljanja takvim postrojenjem. Također, u okviru ovog diplomskog rada potrebno je realizirati manju bazu prototipnih dijelova postrojenja (pokretna traka, senzori, tipkala, prekidači, žarulje, 1D manipulatori i sl.) čije slaganje će omogućiti realizaciju proizvoljnog broja tipičnih postrojenja.

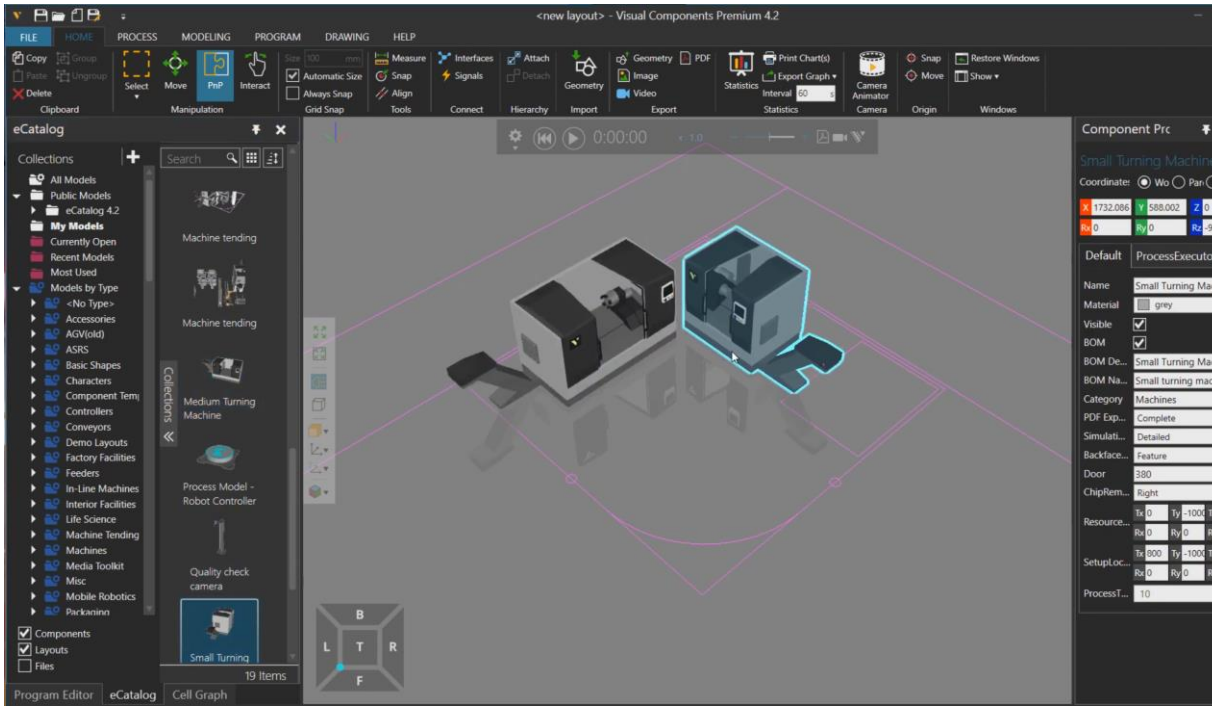
2. PREGLED PODRUČJA RADA

Postoji mnoštvo 3D simulacija koje se koriste u različitim industrijama i bilo bi preopširno navoditi ih, a kako je cilj rada simulacija tvorničkog postrojenja, fokus će biti na neke od njih.

Jedan od vjerojatno najpoznatijih primjera je Factory I/O. Razvila ju je Portugalska tvrtka Real Games sa sjedištem u Gondomaru. Osnovana 2006. godine, bavi se simulacijskim softverom za automatizaciju postrojenja. Factory I/O (Slika 2.1.) tvornička je simulacija za učenje tehnologija automatizacije. Dizajnirana je da bude jednostavna za korištenje te omogućuje brzu izgradnju virtualne tvornice koristeći uobičajene industrijske dijelove ponuđene na izbor. Ponuđena je i nekolicina već gotovih scena inspiriranih tipičnim industrijskim postrojenjima. Najčešće se koristi kao PLC platforma za treniranje, s obzirom na to da je PLC najrašireniji logički kontroler u industrijskoj primjeni. No, moguće ga je koristiti i s mikrokontrolerima, SoftPLC, Modbus i mnogim drugim tehnologijama. Postoji probna verzija na 30 dana nakon čega se mora kupiti licenca. [1]

Sl. 2.1. Factory I/O [1]

Visual Components, tvrtka osnovana 1999, s glavnim sjedištem u Finskoj, nudi svoj softver Visual Components 4.4 (Slika 2.2.) za izgradnju fleksibilnih, profesionalnih i skalabilnih simulacija proizvodnje. Tako je moguće kreirati tvorničko postrojene, povezati ga s kontrolnim sustavom (PLC), koristiti osnovno ali i napredno robotsko programiranje, ostvariti interakciju sa svojim postrojenjem kroz virtualnu stvarnost, modelirati komponente, uvesti modele iz AutoCAD programa te mnoštvo drugih opcija. [2]



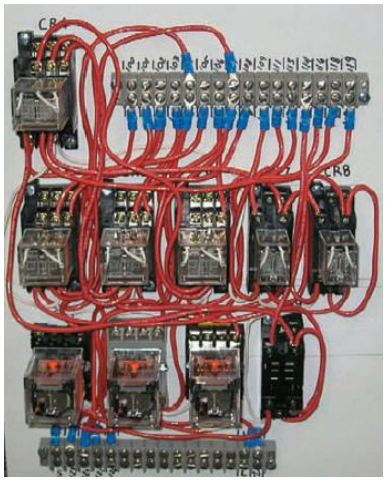
Sl. 2.2. Visual Components 4.4 [2]

Još jedan od primjera je i Fastsuite (Slika 2.3.). To je platforma za 3D simulaciju koja podržava cijeli životni ciklus proizvodnih sustava, od ranog koncepta dizajna i razvoja detalja procesa do virtualnog puštanja u rad te samog rada sustava. Kao i prošla dva primjera, bogat je mogućnostima, jednostavan za korištenje te prigodan za male tvrtke koje žele programirati robota ali i velike tvrtke koje izgrađuju cijelu digitalnu tvornicu. Razvila ga je tvrtka Cenit, sa sjedištem u Stuttgartu. [3]

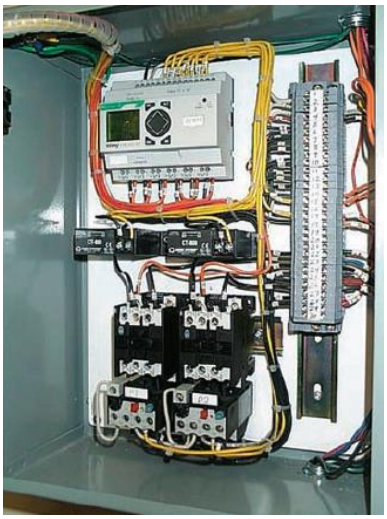
Sl. 2.3. Fastsuite [3]

3. PLC

Programabilni logički kontroler (eng. *programmable logic controller*) ili skraćenicom PLC prvi je put uveden krajem 60-ih godina. Prije uvođenja PLC tehnologije, za većinu industrijskih i procesnih kontrolnih sustava korišteni su elektromehanički releji. U tim sustavima, relejni logički uređaji povezivani su s ulazima i izlazima kontrolnih panela (Slika 3.1.). Releji su bučni, relativno veliki i skloni mehaničkim problemima. Kontrolni paneli koji ih sadrže su znatno veći od PLC sustava, a modifikacija takvog kontrolnog panela zahtijevala je ponovno ožičenje releja, što je skup i vremenski zahtjevan proces. Nasuprot tome, PLC sustav (Slika 3.2.) zahtijeva minimalno ožičenje, može biti modificiran i reprogramiran vrlo brzo.



Sl. 3.1. Relejni kontrolni panel [4]



Sl. 3.2. PLC kontrolni panel [4]

PLC je računalo dizajnirano za kontrolu postrojenja, odnosno za rad u industrijskom okruženju. Neke od prednosti korištenja prema [4] su :

- Pouzdanost – napisani programi se lako mogu koristiti i na drugim PLC uređajima. Budući da se sva logika nalazi u memoriji PLC-a, pogreške u ožičenju su svedene na minimum. Otpornost na razne mehaničke i elektromagnetske utjecaje, te općenito otporan na pogonske uvjete rada.
- Fleksibilnost – lakše je kreirati program nego provesti sva ožičenja, a posebno kada dođe potreba za promjenom. Proizvođači mogu lako unaprijediti svoje sustave tako da pošalju nove programe. Korisnici također mogu modificirati program, a postoje i sigurnosne značajke koje se mogu implementirati.
- Komunikacija – PLC može komunicirati s drugim kontrolerima ili računalnom opremom za nadzor, prikupljanje podataka, praćenje rada uređaja i procesnih parametara te preuzimanje i učitavanje programa.
- Brzina – industrijska postrojenja obrađuju tisuće podataka u sekundi te mnoštvo senzora reagira u vrlo kratkom vremenu pa je stoga i sposobnost PLC-a brzi odziv za primjenu u stvarnom vremenu.
- Dijagnostika – s pomoću funkcija za otklanjanje pogrešaka i dijagnostiku nudi mogućnost praćenja uređaja.

3.1. Dijelovi i način rada PLC-a

Osnovna hardverska struktura sastoji se od ulaznog dijela (digitalni, analogni ulazi), izlaznog dijela (digitalni, analogni izlazi), procesora, memorije, komunikacijskog sučelja i napajanja.

Ulazno/izlazna jedinica (U/I) (eng. *Input/Output* ili I/O) predstavlja sučelje procesora s vanjskim elementima (senzori, aktuatori). Dva su načina na koji je ugrađena u PLC, a to su fiksni i modularni. Fiksni je većinom za manje PLC-ove koji dolaze s ograničenim brojem priključaka te se ne mogu ukloniti, a procesor i U/I su u jednom dijelu. Prednost je niža cijena. Broj priključaka se može proširiti kupnjom dodatnih jedinica, no i tu postoje ograničenja u količini i vrsti. Modularni U/I omogućuje priključivanje odvojenih modula. Ovim se značajno utječe na fleksibilnost, moduli se slažu po želji.

Procesorska jedinica s memorijom predstavlja najbitniji dio PLC uređaja. Procesor obrađuje signale s ulaza u skladu s programom kojeg je napisao korisnik te dalje komunicira s ostalim modulima. Procesor dakle izvršava operativni sustav, upravlja memorijom, prati ulaze, izvršava

korisnički program i uključuje odgovarajuće izlaze. Procesor kontrolira sve aktivnosti. PLC program izvodi se kao ciklički proces sve dok je PLC u radnom načinu. Tipičan rad počinje tako da procesor očitava stanje ulaznih signala. Zatim se ta stanja prenose u memorijski registar procesora. Napisani program se zatim izvodi po zadanoj logici te se izlazni podaci spremaju u memoriju i prenose na izlaze. Nakon toga odvijaju se operacije dijagnostike i komunikacije.

Memorija je dio PLC-a gdje se sprema korisnički program i radni podaci. Kompleksnost programa određuje koliko će memorije biti potrebno. PLC izvršava rutinske provjere memorije kako bi se osiguralo da se program neće izvršiti ako je memorija oštećena. Memorija može biti izbrisiva i neizbrisiva. Izbrisiva memorija će izgubiti podatke ako se izgubi napajanje dok neizbrisiva zadržava memoriju u tom slučaju.

Modul za napajanje neosjetljiv je na smetnje koje dolaze iz električne mreže kao i na kraće ispade mrežnog napona. Standardni ulazi napajanja PLC uređaja su: 120/230 VAC i 24 VDC.

Komunikacijsko sučelje koristi se za uspostavu veze s drugim uređajima. Takve veze su obično uspostavljene s računalima, operatorskim stanicama, sustavima upravljanja procesa i drugim PLC-ovima. Postoje ugrađeni serijski i ethernet portovi ili moduli za ugradnju, a komunikacija se može ostvariti i preko mreže. Na ovaj način se spajaju i uređaji za programiranje. Najpopularnija metoda je korištenje osobnog računala u kombinaciji sa softverom proizvođača PLC-a. Ovo pruža najviše mogućnosti. Postoje i ručni uređaji koji su jednostavni za korištenje, imaju skromnu tipkovnicu za unos instrukcija i mali LCD ekran te se koriste za kraće programe ili kada treba nešto promijeniti direktno u pogonu.

3.2. Organizacija memorije

U ovom poglavlju bit će opisana organizacija memorije SLC500 kontrolera. SLC500 je serija PLC-ova koju je razvila tvrtka Allen-Bradley, današnja robna marka pod tvrtkom Rockwell Automation.

PLC dijeli raspoloživu memoriju u različite sekcije. Iz toga možemo izvući dvije šire kategorije: podatkovne datoteke (eng. *data files*) i programske datoteke (eng. *program files*). Programske datoteke su dio procesorske memorije gdje je smješten kod napisan u nekoj od tehnika programiranja za PLC. Uključuju:

- Datoteka sistemskih funkcija – datoteka koja je uvijek uključena i sadrži informacije o sustavu.
- Rezervirana datoteka – datoteka koju procesor koristi i nije dostupna korisniku.
- Glavna programska datoteka – uvijek uključena datoteka u kojoj se nalaze instrukcije o radu kontrolera koje je napisao korisnik.
- Datoteke potprograma – kreirane od korisnika te ih ima više i aktiviraju se ovisno o instrukcijama koje se nalaze u glavnoj programskoj datoteci.

Podatkovne datoteke su dio memorije gdje se spremaju statusi ulaza i izlaza, status procesora, statuse bitova i numeričke podatke. Ove informacije su dostupne pri programiranju. Organizirane su po tipu podatka koji sadrže i uključuju:

- *Output* datoteka – sadrži stanja izlaza kontrolera.
- *Input* datoteka – sadrži ulazna stanja kontrolera.
- *Status* datoteka –sprema informacije o operacijama kontrolera i korisna je za rješavanje problema.
- *Bit* datoteka – koristi se za spremanje vrijednosti veličine bita.
- *Timer* datoteka – pohranjuje vrijednosti mjerača vremena.
- *Counter* datoteka – pohranjuje vrijednosti brojača.
- *Control* datoteka – sprema veličinu, poziciju pokazivača i status bitova za specifične instrukcije kao što su pomični registri.
- *Integer* datoteka - koristi se za pohranu cijele brojčane vrijednosti do 16 bita.
- *Float* datoteka – koristi se za pohranu realnih brojeva do 32 bita.

Da bi se označilo koji PLC ulaz i izlaz su povezani na koji ulazni i izlazni uređaj potrebno je poznavati memorijske registre. Adresiranje ovisi od modela do modela ali svi koriste sličan ili modificiran sustav adresiranja. Adrese sadrže broj utora modula na koji se spajaju ulazni ili izlazni uređaji. Adrese su formatirane tako da sadrže tip datoteke, broj datoteke, broj utora, riječ i bit. Tip datoteke određuje radi li se o ulazu (I) ili izlazu (O). Riječ i utor koriste se za identifikaciju određenog U/I modula. Na slici 3.3. možemo vidjeti jedan takav primjer formatiranja adresa.

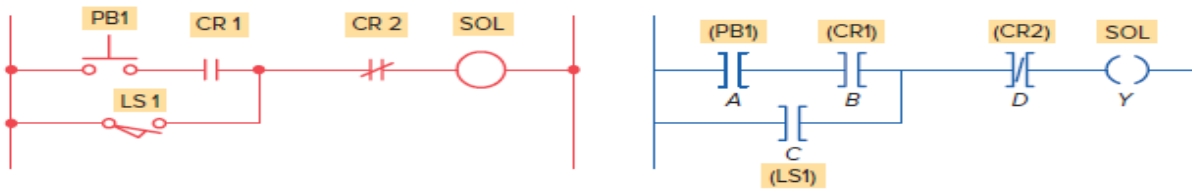
Sl. 3.3. Primjer formata adresiranja.

3.3. Osnove programiranja PLC uređaja

Pisanje programa za PLC najčešće se izvodi na računalu. PLC proizvođači često imaju svoj softver koji dolazi u kombinaciji s programskim uređivačem, kompilatorom te komunikacijskim softverom. Kod se piše u uređivaču koda te se on nakon provjere sintakse šalje na RAM memoriju PLC-a. Komunikacije između PLC-a i računala je najčešće serijska, a može biti aktivna i tijekom izvođenja programa. Programski jezici koji se koriste u programiranju PLC-a su ljestvičasti dijagrami (eng. *ladder diagram*), instrukcijske liste (eng. *instruction list*) i funkcijsko blokovski dijagrami (eng. function block diagram). Neki proizvođači nude i mogućnost programiranja u tekstualnim programskim jezicima kao što su PASCAL, BASIC ili C.

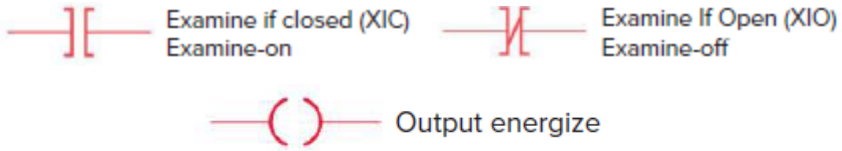
3.3.1. Ljestvičasti dijagrami

Najčešće upotrebljavan je ljestvičasti dijagram. Razlog tome je što je dizajniran da sliči relejnoj logici (Slika 3.4.) pa su se tako ljudi lakše obučavali pri prijelazu s relejne logike. Kasnije se zadržao jer se pokazao lako shvatljivim i onima koji se nisu bavili relejnim upravljanjima.



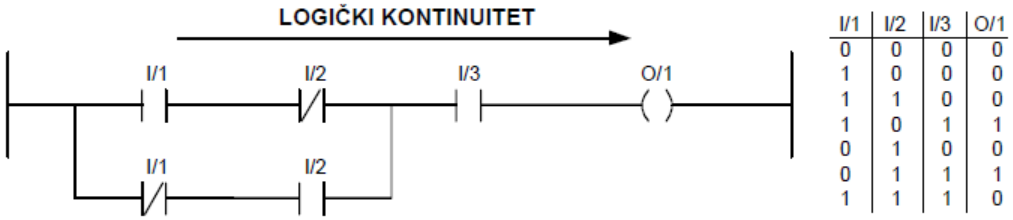
Sl. 3.4. Primjer relejnog kruga (lijevo) i ekvivalentni ljestvičasti dijagram (desno) [4]

Tri osnovna simbola (Slika 3.5.) koji se koriste u prebacivanju relejne upravljačke logike u kontaktnu simboličku logiku su XIC (eng. *Examine If Closed*), XIO (eng. *Examine If Open*) i OTE (eng. Output Energize). Ovo su zapravo instrukcije koje su predstavljene jednim bitom u PLC memoriji. Iako su XIO i XIC predstavljeni simbolima koji nalikuju otvorenom i zatvorenom relejnom kontaktnu , oni ne radi tako već djeluju kao naredbe koje ispituju vrijednosti (0 ili 1) bita podataka za određivanje logičkih uvjeta. Naredba XIC se koristi kada se želi odrediti je li adresirani bit u logičkom '1', odnosno provjera je li zatvoreno. Naredba XIO je suprotno, adresirani bit je u logičkom '0', odnosno provjera je li otvoreno. Naredba OTE koristi se da postavi stanje (0 ili 1) izlaza kada stanje kruga dođe u '1' odnosno '0'.



Sl. 3.5. Simboli XIC, XIO i OTE [4]

Program stalno kontrolira stanje fizičkih ulaza i prema tome upravlja izlazom. Može se sastojati od jednog ili više logičkih krugova, te se izvodi od prvog do zadnjeg logičkog kruga. Osnovne logičke operacije I i ILI se kombiniraju pa nam to daje bezbroj mogućnosti kod samog programiranja. Na slici 3.6. vidimo primjer jedno ljestvičastog dijagrama gdje će izlaz biti u logičkom stanju 1 kada su ulazi I1 i I2 u logičkim stanjima 1 i 0 odnosno 0 i 1 te ulaz I3 u logičkom stanju 1. Ovo je zapravo kombinacija paralelnog i serijski povezanog dijagrama.



I/1	I/2	I/3	O/1
0	0	0	0
1	0	0	0
1	1	0	0
1	0	1	1
0	1	0	0
0	1	1	1
1	1	1	0

Sl. 3.6. Primjer ljestvičastog dijagrama

4. KORIŠTENE TEHNOLOGIJE I ALATI

4.1. Unity Engine

Softver nastao 2005. pod tvrtkom Unity Technologies. Koristi se za izradu video igara, simulacija, virtualne i proširene stvarnosti u 3D i 2D prostoru. Ima mogućnost izrade u više platformi (Android, MacOS, Windows, Linux, Playstation, Web, itd.). Kreiran je u C++ programskom jeziku, ali koristi C# programski jezik za razvoj igara. Veliku popularnost stekao je među *indie*[1] developerima zbog toga što je u vrijeme kada je kreiran bio besplatan i najbolji na tržištu. S godinama je izrastao u stabilan i fleksibilan alat s bogatom dokumentacijom. Osim industrije igara, koristi se i u filmskoj, inženjerskoj, automobilskoj te arhitektonskoj industriji.

Grafičke sposobnosti Unitya su impresivne. Podržava DirectX, Metal, OpenGL i Vulkan grafičke API-je. Korisniku su na raspolaganju tri seta operacija renderiranja (eng. *Render Pipelines*). S pomoću njih se sadržaj scene prikazuje na ekran kroz mnoštvo obrade podataka i niza operacija. Unity nudi BRP, URP i HDRP. BRP (eng. Built-in Render Pipeline) je opće namjene s ograničavajućim svojstvima prilagođavanja. URP (eng. *Universal Render Pipeline*) i HDRP (eng. *High Definition Render Pipeline*) su bogatije svojstva, skalabilnije i boljih performansi gdje je HDRP za one igre koje žele grafički zadiviti svoje korisnike. Developer također može kreirati svoj *render pipeline* kako on vidi da se grafičke operacije trebaju odvijati.

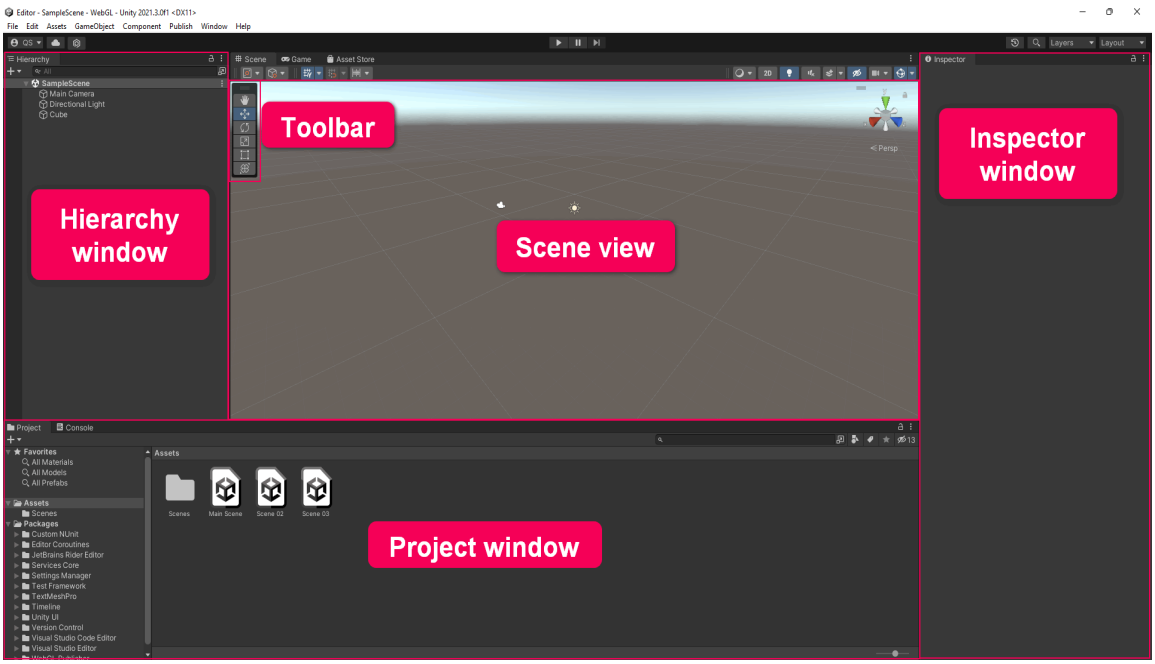
Sav proces kreiranja 3D ili 2D okruženja događa se kroz grafičko sučelje. Ovdje je ponuđeno mnoštvo komponenata i operacija za manipulaciju 3D ili 2D objekata. Za svu programsku logiku većinski se koristi skripte koje su u biti C# datoteke. Unity podržava vizualno programiranje iako ono nije dovoljno razvijeno da bi se oslonilo samo na njega.

U ovom poglavlju kratki je opis kako funkcioniraju osnovne značajke rada kroz Unity Editor, što su *GameObject* i *Prefab* komponente te na koji način programirati skripte koristeći Unity biblioteke, kako bi se lakše razumjeli procesi izrade ovog projekta.

[1] Indie – skraćenica za neovisnost (eng. *Independent*). U slučaju razvoja igara radi se o osobi ili manjoj grupi ljudi gdje je razvijanje neovisno o veliki produkcijskim tvrtkama te se troškovi snose samostalno.

4.1.1. Unity Editor

Unity Editor je grafičko korisničko sučelje koje se sastoji od mnoštva prozora (eng. *window*). Prozor je dio korisničkog sučelja koji prikazuje podatke, slike, modele, scenu te je omogućena manipulacija istim. Prozori se mogu rasporediti po želji, povećati, smanjiti, izvaditi iz rasporeda sučelja u jedinstveni prozor te grupirati pod jedan prozor. Ne postoji ograničenje koliko prozora će biti vidljivo u jednom trenutku, na korisniku je da organizira raspored. Po potrebi koristeći biblioteku koju Unity pruža moguće je kreirati i vlastite prozore te konfigurirati na koji način se



Sl. 4.1. Prikaz Unity Editora [6]

podaci u njima prikazuju. Na slici 4.1. možemo vidjeti da je korisničko sučelje Unity Editora podijeljeno u pet najznačajnijih dijelova. To su prikaz scene (eng. *scene view*), hijerarhijski prozor (eng. *hierarchy window*), prozor projekta (eng. *project window*), alatna traka (eng. *toolbar*) i informacijski prozor (eng. *inspector window*).

Prikaz scene je interaktivni prozor u kojem se stvara svijet manipulacijom objekata. Prikaz igre (eng. *game view*) najčešće se nalazi pored prikaza scene. Kroz ovaj prozor se testira igra bez potrebe za izgradnjom (eng. *build*) iste.

Alatna traka u prozoru scene sadrži navigacijske opcije. Uz pomoć njih se odabrani objekti mogu pomicati, rotirati i skalirati.

Hijerarhijski prozor sadrži sve objekte dodane u scenu. Ovdje se oni mogu organizirati. Ako dodamo objekt u scenu on će se dodati i u hijerarhijski prozor, a ako ga obrišemo nestaje iz hijerarhije.

Projektni prozor sadrži sve resurse u igri koje se nalaze u projektu. Funkcionira kao preglednik datoteka, datoteke se mogu smjestiti i organizirati po mapama. Ako određeni resurs dozvoljava, može ga se direktno povući iz projektnog prozora u scenu.

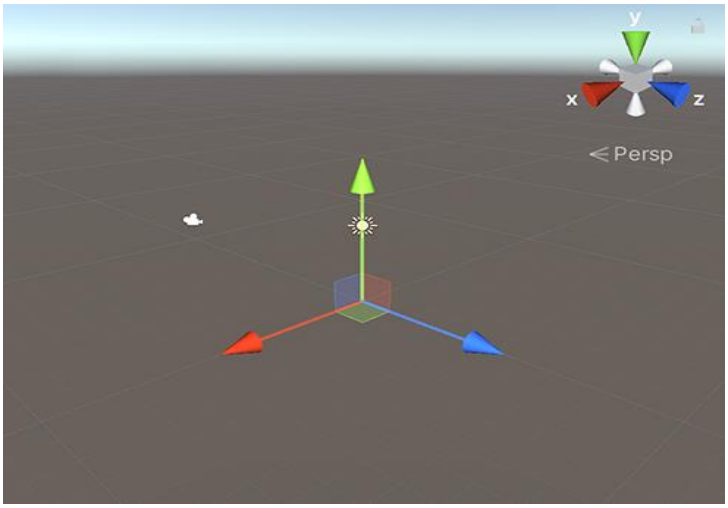
Informacijski prozor pokazuje detaljne informacije o objektima u igri. Ako se odabere neki objekt u sceni ovdje će se prikazati njegova svojstva i ponašanja. Primjerice, pozicija u 3D ili 2D svijetu, koje skripte sadrži na sebi, itd.

4.1.2. GameObject

Objekt koji čini temelj izgradnje igre. Svaki objekt u igri je zapravo tipa *GameObject*, bili to likovi u igri, zgrade, voda, oblaci ili pak kamere, svijetla i specijalni efekti. Sam po sebi je neupotrebljiv, ono što mu daje ponašanje i svojstva je komponenta ili više njih. Listu svih komponenti možemo vidjeti u informacijskom prozoru kada odaberemo *GameObject* u sceni. Komponente su zapravo skripte napisane u C# programskom jeziku. Može se dodijeliti više komponenti. Svaki *GameObject* sadrži samo jednu komponentu tipa *Transform*. Ova komponenta je određuje njegovu lokaciju, rotaciju i veličinu (Slika 4.2.). U 2D prostoru manipulacija se ostvaruje po x i y osi, a u 3D prostoru po x, y i z osi. Na slici 4.3. može s vidjeti kako su te osi predstavljene crvenom, zelenom i plavom bojom. [7]

Postoji i mogućnost grupiranja u odnos roditelj-dijete. Objekti koji se nalaze pod roditeljskim objektom bit će kontrolirani kroz roditeljsku *Transform* komponentu. Na primjer, ako pomaknemo roditeljski objekt pomičemo i sve pripadajuće objekte. Ovo ne znači da ne možemo manipulirati pripadajućim objektima, već je njihov prostor sad povezan uz lokalni prostor roditeljskog objekta. Važno je spomenuti da svaki objekt ima glavnu točku (eng. *pivot*) koja je zapravo referenta točka za pomicanje, rotiranje i skaliranje objekta.

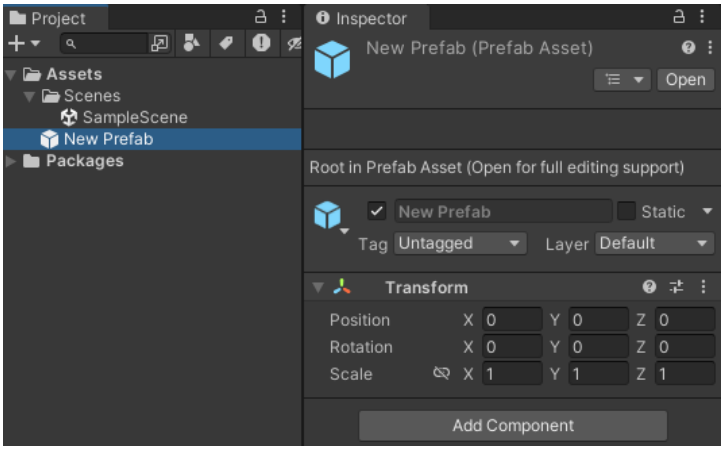
Sl. 4.2. *Transform* komponenta



Sl. 4.3. Prikaz osi po bojama *Transform* komponente

4.1.3. Prefab

GameObject sa svim svojim komponentama, vrijednostima svojstava i podređenim *GameObjectima* možemo pohraniti kao ponovno iskoristiv resurs za igru (eng. *game asset*). Ovo nam omogućuje *Prefab* sustav gdje se od *GameObjecta* stvara predložak (*prefab*), koji će poslužiti za stvaranje novih instanci u sceni. Kada se želi ponovno upotrijebiti GameObject konfiguriran na određeni način na više mjesta u sceni ili u više sceni tada je dobra ideja pretvoriti ga u *prefab*. Sve izmjene koje se naprave na *prefab* resursu se odražavaju na njegove instance, omogućujući jednostavne promjene na cijelom projektu bez pojedinačnih promjena. Ovo ne znači da sve instance *prefaba* moraju biti identične. Postavke pojedinih instanci se mogu razlikovati od ostalih, a moguće je kreirati i *prefab* varijante iz baznog *prefaba*. *Prefab* kreiramo tako da objekt iz scene prevučemo u projektni prozor ili kroz *Assets* meni na opciju *prefab*. Izgled prefaba u projektnom prozoru prikazan je na slici 4.4.



Sl. 4.4. Prikaz osi po bojama *Transform* komponente

4.1.4. Programiranje

Za programiranje se, kao što je već navedeno, koristi C# programski jezik. Unity podržava Visual Studio, Visual Studio Code i JetBrains Rider uređivače koda, odnosno integrirana razvojna okruženja (eng. *Integrated Developer Environment* – IDE). Nova skripta kreira se kroz meni *Assets*. Skripta će se kreirati u onoj mapi koju smo odabrali u projektnom prozoru. C# skripte također možemo kreirati i kroz navedene uređivače koda. Na slici 4.5. možemo vidjeti kako početno izgleda svaka skripta nakon kreiranja. Postoji i mogućnost prilagodbe, moguće je napraviti vlastiti predložak početnog izgleda svake C# skripte.

```
public class NewBehaviourScript : MonoBehaviour
{
    // Start is called before the first frame update
    Unity Message | 0 references
    void Start()
    {

    }

    // Update is called once per frame
    Unity Message | 0 references
    void Update()
    {

    }
}
```

Sl. 4.5. Prikaz početnog izgleda C# skripte

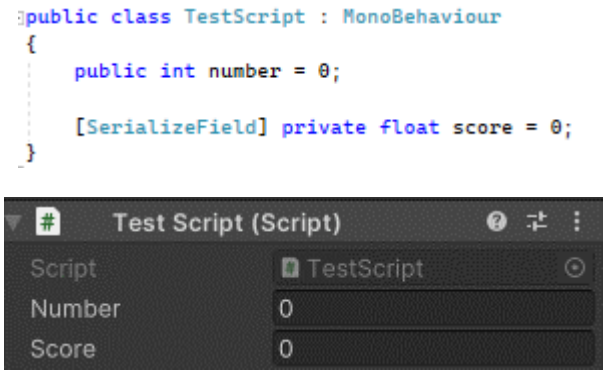
Korisniku je na raspolaganju mnoštvo klasa, ali neke od najčešće korištenih su [8]:

- ***GameObject*** – predstavlja tip objekta koji postoji u sceni
- ***MonoBehaviour*** – bazna klasa koju svaka Unity skripta nasljeđuje prema zadanim postavkama.
- ***Object*** – bazna klasa za sve objekte koji se mogu referencirati u editoru.
- ***Transform*** – omogućuje upravljane pozicijom, rotacijom i veličinom objekta u sceni
- ***Vectors*** – klasa za manipulaciju 2D, 3D točaka te linija i smjerova.
- ***Quaternion*** – klasa koja predstavlja apsolutnu ili relativnu rotaciju te nudi metode za njeno upravljanje.
- ***ScriptableObject*** – spremnik podataka za spremanje veće količine podataka.
- ***Time*** – klasa koja omogućuje mjerenje i kontroliranje vremena u igri te upravljane framerateom u igri.
- ***Mathf*** – klasa koja sadrži kolekciju čestih matematičkih funkcija koje se koriste u razvoju igara.

- ***Random*** - omogućuje lakše kreiranje nasumičnih vrijednosti različitih tipova.
- ***Debug*** – nudi niz metoda za pomoć pri otkrivanju pogrešaka.
- ***Gizmos*** – omogućuje iscrtavanje linije i oblika koji su vidljivi u prikazu scene i igre.
- ***Rigidbody*** - klasa koja omogućuje kontrolu pozicije objekta kroz simulaciju fizike.

Kao što je već navedeno *MonoBehaviour* je bazna klasa svake skripte. Ovdje je riječ o skriptama koje su kreirane kroz projektni prozor. Ako želimo čistu C# klasu uklonimo *MonoBehaviour* ili kreiramo skriptu kroz uređivač koda. *MonoBehaviour* nasljeđuje *Component* klasu. Ona je bazna klasa koja je potrebna za svaku skriptu koju želimo dodijeliti objektu u sceni. Unity koristi sustav baziran na komponentama (eng. *component based*). Svaki objekt u sceni može se sastojati od mnoštva komponenti, a to su skripte s klasama koje nasljeđuju *MonoBehaviour* [9].

Deklariramo li varijable s modifikatorom pristupa *public* u klasama koje nasljeđuju *MonoBehaviour* klasu, njihove vrijednosti bit će vidljive u informacijskom prozoru te ih možemo mijenjati. Osim *public* modifikatora pristupa možemo postaviti i *private* ali onda treba dodati i atribut *SerializedField* ispred modifikatora. Primjer navedenog možemo vidjeti na slici 4.6. Ova funkcionalnost omogućuje jednostavno testiranje različitih vrijednosti kroz grafičko sučelje bez potrebe za promjenama u kodu.



Sl. 4.6. Vidljivosti varijabli u *inspector* prikazu

MonoBehaviour klasa pruža funkcije za životni ciklus komponente. One najvažnije su *Awake*, *Start*, *Update*, *FixedUpdate*, *LateUpdate*, *OnDisable*, *OnEnable* i *OnDestroy*. Koriste se kao *event* funkcije, odnosno Unity će ih pozvati povremeno kao odgovor na događaje tijekom igranja, a developeru su dostupne za pisanje programske logike. Poziv se događa po imenu funkcije. Nakon što je funkcija završila s izvođenjem kontrola se vraća Unityu. *Start* i *Awake* se pozivaju samo jednom u cijelom životnom ciklusu skripte. Razlika je što se *Awake* uvijek poziva prvi, prilikom

inicijalizacije objekta u sceni. Osim toga, *Awake* funkcija se poziva na svim objektima u sceni prije nego se pozove bilo koja Start metoda. *Update* se poziva svaki *frame*[2]. Ovdje se često implementira najviše logike za igru. Za fiziku u igri i proračune koristi se *FixedUpdate* metoda. Zadana vrijednost poziva je 0.02 sekunde odnosno 50 poziva po sekundi. Može se staviti proizvoljna vrijednost poziva u postavkama. *LateUpdate* se također zove svaki *frame* ali tek nakon što su sve *Update* metode pozvane. Metoda *OnDisable* poziva se kada se objekt u igri onesposobi. Također poziva se i kada je objekt uništen, pa ovo može biti dobro mjesto za očistiti memoriju. *OnEnable* se poziva kada objekt više nije onesposobljen, odnosno vraćen je u aktivno stanje. Ako se objekt u igri uništi, scena se promijeni ili jednostavno igra ugasi, poziva se *OnDestroy* [10].

Da bismo razvili logiku u igri i učinili ju igrivom potrebno je stvoriti objekte u sceni kojima se onda dodjeljuju komponente. Te komponente mogu biti dio Unity *frameworka* kao tipa *RigidBody* ili vlastite C# skripte koje sadrže programsku logiku. Komponente možemo referencirati kroz *inspector* prozor. Ovo služi kao *dependency injection*. No možemo ih referencirati i kroz kod s pomoću metoda *GetComponent, GetComponents* itd. Treba paziti na arhitekturu koda jer dodavanjem sve više funkcionalnosti održavanje postaje teže te se stvaraju *bugovi*. Ovdje u pomoć priskače objektno orijentirani način programiranja, programski obrasci i SOLID načela. Programiranje igre je dug i mukotrpan proces no rezultat bi uvijek trebao biti pružiti igračima ugodno, zanimljivo i zabavno iskustvo.

[2] Frame – jedna cjelovita slika u nizu koji tvori video.

4.2. Blender

Blender (Slika 4.7.) je besplatan softver otvorenog koda za 3D modeliranje. Podržava cjelokupni 3D sustav obrade – modeliranje, namještanje, animacije, simulacije, renderiranje, sastavljanje i praćenje pokreta. Mogu se i uređivati video zapisi. Za korisničko grafičko sučelje koristi OpenGL, te ga se može proizvoljno urediti kroz Python skripte. Visoko kvalitetna 3D arhitektura, čime omogućava brzi i efikasni tijek stvaranja. Višeplatformni je pa jednako dobro radi na Linux, Windows i MacOS operacijskim sustavima. Njegov izvršni program zauzima malo memorije, što ga čini lako portabilnim. S obzirom na to da ima GNU licencu, nije samo besplatan, već se može koristiti u bilo koju svrhu te modificirati, distribuirati, poboljšati pa čak i objaviti vlastitu verziju. Ima veliku zajednicu developera te ga koriste mnogobrojni studiji za modeliranje, reklame, filmove i hobi projekte. [11]

Sl. 4.7. Blender

4.3. Gimp

Softver otvorenog koda koji dolazi pod akronimom GIMP – *GNU Image Manipulation Program* (Slika 4.8.). Uređivač je slika dostupan za Linux, MacOS i Windows operacijske sustave. Pruža sofisticirane alate za obradu slike – retuširanje, restauracija, kompozicije itd. Također, može se prilagoditi s dodacima koji nisu dio izvornog koda, a korisnik ih može sam razviti koristeći neke od programskim jezika (C, C++, Perl, Python itd) ili preuzeti već gotove kreirane od zajednice.

Sl. 4.8. GIMP

4.4. C# programski jezik

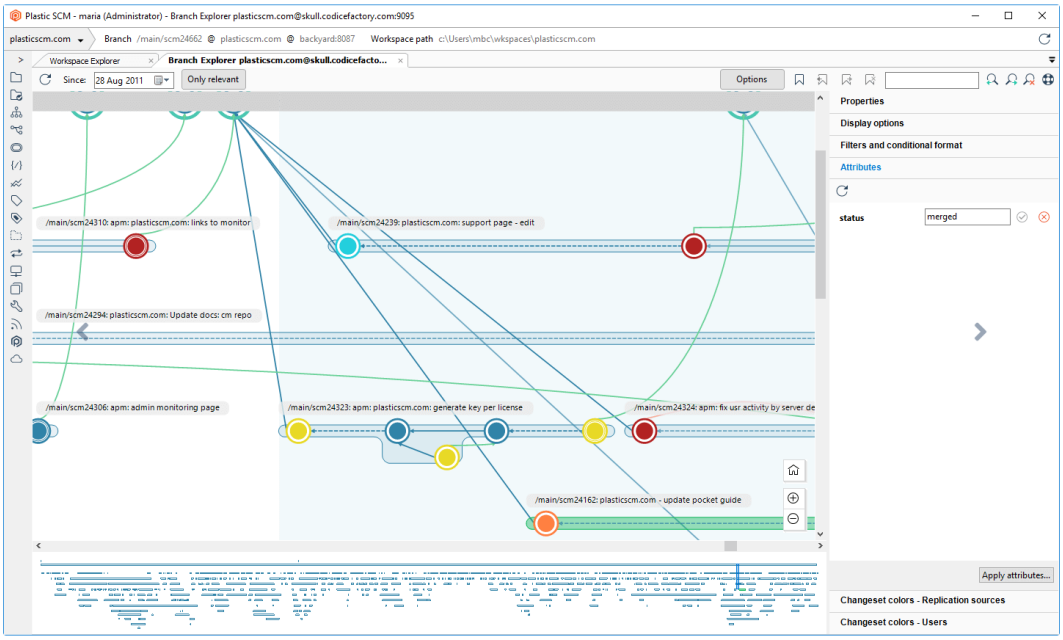
Nastao 2000. godine pod vodstvom Microsoftovog programskog inženjera Andresa Hejlsberga koji je ujedno i glavni arhitekt C# programskog jezika. Sličnost s Java programskim jezikom proizlazi iz toga da su oba jezika članovi obitelji C programskih jezika, a to potvrđuje i sintaktična sličnost s C++ programskim jezikom. U trenutku pisanja zadnja stabilna verzija je C# 12.0 objavljena u studenom 2023. godine.

Primarno je objektno-orijentiran što uključuje klase, objekte, konstruktore, destruktore, funkcijsko preopterećivanje, sučelja, nasljeđivanje klasa, polimorfizam itd. Naglašena je sigurnost tipova podataka gdje ako je neki podatak deklariran kao broj, ne može ostvariti interakciju s drugim podatkom koji je tekstualnog tipa jer će kompajler se izbaciti grešku. Sadrži mnoštvo korisnih značajki i veoma je fleksibilan jezik. Primjerice, iako se pokazivači iznimno rijetko koriste, može se spustiti na tu razinu. Zanimljivi su i atributi koje je moguće postaviti iznad metode, klase, strukture itd. te im na taj dodijeliti određene informacije. Primjerice, ako iznad metode dodamo atribut „[Obsolete]“ ostalim developerima će biti jasno da je ta metoda zastarjela. Bitno je napomenuti da C# ima sakupljanje smeća (eng. *garbage collection*), odnosno automatsko upravljanje memorijom gdje se memorija oslobađa u slučaju kad se objekti više ne koriste.

Programi napisani C# jezikom mogu se koristiti samo uz .NET. To je platforma koja se sastoji od .NET runtime-a i mnoštva biblioteka i klasa. Napisani kod .NET projekta se kompajlira u dll izvršnu datoteku (eng. *assembly*) koja sadrži kod CIL (eng. *Common Intermediate Language*) te se pokreće kroz komponentu .NET runtime-a CLR (eng. *Common Language Runtime*). Programi kreirani uz .NET mogu se osim na Windowsu koristiti na MacOS i Linux operacijskim sustavima. [12]

4.5. Plastic

Plastic SCM (Slika 4.9.) potpuni je sustav za upravljanje izvornim kodom. Kupljen je od Unity Technologies te je dio Unity DevOpsa. Dizajniran je da nudi brzinu, fleksibilnost i jasnu vizualizaciju prilikom odvajanja u grane (eng. *branch*) i spajanja grana. Pod riječju potpuni, misli se da Plastic nudi sve potrebne alate za upravljanje verzijama projekta – cloud server, web sučelje, uspoređivanje koda, različiti alati za spajanje grana, GUI alati za Linux, macOS i Windows te upravljanje kroz komandnu liniju. Mogućnost rada s tisućama grana i velikom količinom podataka. Nije baziran na Gitu iako je kompatibilan ali ne dijeli kodnu bazu, kreiran je zasebno sa zasebnim grafičkim sučeljem. Nudi distribuirani i centralizirani način rada. Upravljanje velikim datotekama i projektima što ga čini popularnim u industriji razvoja igara. Rješavanje kompleksnih konflikata prilikom spajanja grana uz alate koji pomažu odrediti razliku u kodu. Cloud serveri na raznim lokacijama diljem svijet te mogućnost njihova odabira prilikom kreiranja repozitorija. Sigurnost pristupa podacima bazirana je na *Fine-grained accessu,* odnosno potrebno je zadovoljiti više uvjeta kako bi se dobila kontrola nad podacima. Izrazito je brz, čak i kod velikih projekata. Kod problema, ima aktivnu podršku i često se unaprjeđuje kako bi zadovoljio želje korisnika i riješio neispravnosti. Jednostavno korištenje za Unity projekte.



Sl. 4.9. Plastic SCM Windows GUI

4.6. Angular i Typescript

Angular je razvojna platforma za web aplikacije. Održava ju Google. Njegova prva verzija. AngularJS, objavljena je 2010. godine i bila je napisana u JavaScriptu. Angular 2 objavljen je 2016. godine te je ova verzija napisana u TypeScriptu. U trenutku pisanja je na verziji 18.

TypeScript je proširenje JavaScript jezika te donosi brojne prednosti. Jedna od ključnih prednosti je statička provjera tipova podataka, koja omogućuje rano otkrivanje pogrešaka u kodu. Proširuje mogućnosti objektno orijentiranih značajki.

Arhitektura Angulara temelji se na modularnom pristupu. Jedna web aplikacija bit će podijeljena u mnoštvo modula koji se mogu ponovno iskoristiti i lako održavati. Komponenta je osnovna jedinica koja se sastoji od programske logike (TypeScript skripta), podataka i UI dijela (HTML i CSS skripte). Nudi robustan sustav upravljanja formama, stanjima aplikacije, navigacijom te mnoštvo drugih značajki.

Ima veliku zajednicu korisnika, pogotovo u poslovnom svijetu. Kontinuirano se poboljšava te prati trendove. [13]

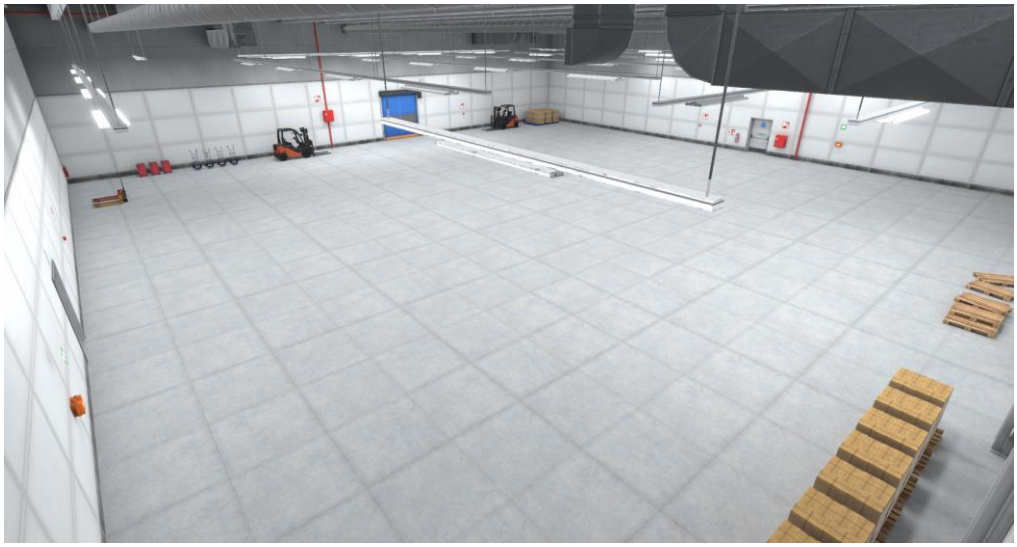
5. SIMULACIJA TVORNIČKOG POSTROJENJA

Projekt možemo podijeliti u četiri osnovne cjeline:

- Sustav gradnje objekata
- Funkcionalnosti tvorničkih elemenata
- Izrada korisničkog grafičkog sučelja
- Implementacija komunikacije s kontrolnom aplikacijom

Simulacija se sastoji od četiri scene. Prva scena koja se učitava prilikom pokretanja simulacije je *Startup*. Ovdje se nalazi *Bootstrap* komponenta koja inicijalizira korisničko sučelje i aktivira početni prikaz učitavanja nakon kojeg slijedi *Menu* scena. Također postoji i *LoadManager* komponenta koja sadrži metode za učitavanje i upravljanje scenama. Važno je napomenuti da ove dvije komponente moraju biti aktivne kroz cijeli životni ciklus aplikacije zbog njihovih metoda koje se koriste u sustavima drugih scena. Zato u *Awake* metodama imaju poziv *DontDestroyOnLoad* metode. Ova metoda osigurava da je komponenta dostupna u memoriji neovisno o promjeni scene s obzirom na to da promjena scene briše sve komponente koje se nalaze u njoj. Kako je *LoadManager* klasa za upravljanje scenama, potrebna nam je samo jedna njena instanca te mogućnost lakog referenciranja u drugim klasa. Iz toga razloga koristi *Singleton* obrazac. Scena *Background* paralelno se učitava sa scenom *Menu* (Slika 5.1.), sastoji se od jednostavnog tvorničkog postrojenja s pokretnim trakama, a služi da poboljša vizualni dojam ulaska u simulaciju. *Menu* scena ima gumb za startanje glavne scene i gumb za izlazak iz simulacije. Glavna scena *Factory* sadrži svu potrebnu programsku logiku, modele za kreiranje tvorničkog postrojenja i okruženje (Slika 5.2.) u kojem se postavljaju tvornički elementi.

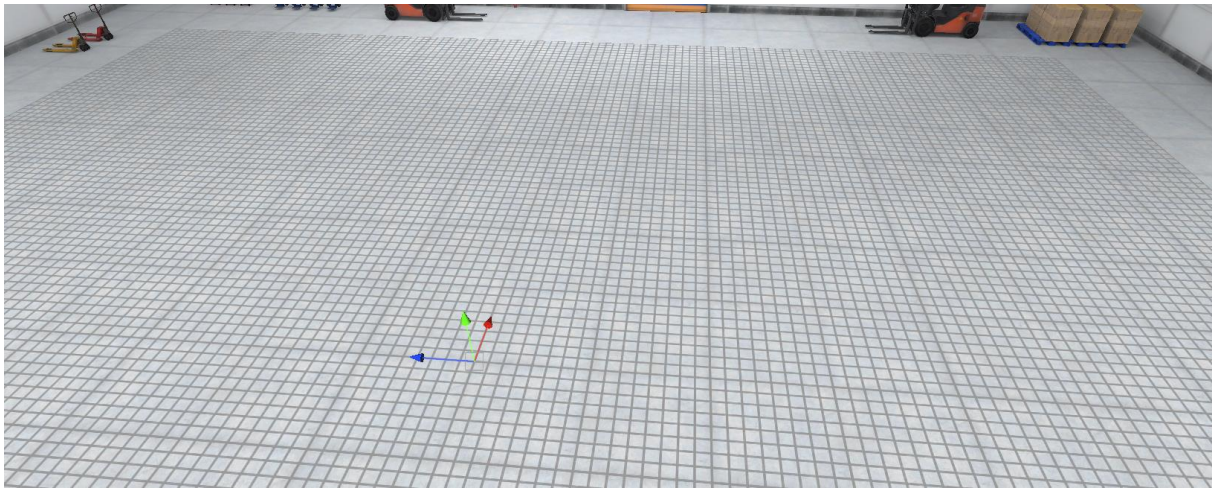
Sl. 5.1. Izgled scene *Menu* i pozadina u *Background* sceni



Sl. 5.2. Izgled tvorničkog okruženja

5.1. Sustav gradnje objekata

Za pozicioniranje elemenata tvorničkog postrojenja koristi se mreža prikazana na slici 5.3. Mreža je *GameObject* koji se sastoji od Unity komponente *Grid* i podloge s materijalom čiji *shader* tvori oblik mreže. *Grid* komponenta pomaže pri pozicioniranju objekata, odnosno transformira položaje ćelija mreže u odgovarajuće lokalne koordinate *GameObject-a*. Transform komponenta zatim pretvara te lokalne koordinate u globalne koordinate. Veličina ćelija postavljena je u skladu s veličinom modela tvorničkih elemenata kako bi se oni pravilno mogli postaviti jedan do drugoga. Klasa *GridManager* je osnovna komponenta mrežnog sustavom. Ona referencira *Grid* i *Renderer* komponente kako bi upravljala vizualnim prikazom i prostornom konfiguracijom mreže.



Sl. 5.3. Mreža za postavljanje tvorničkih elemenata.

Programski kod na slici 5.4 prikazuje metode ove klase:

- *Start* – inicijalizira konfiguraciju mreže pri pokretanju, postavljajući veličinu ćelija i prilagođavajući materijal mreže na temelju zadane skale.
- *ToggleGrid* – aktivira ili deaktivira objekt na kojem se nalazi *Renderer* komponenta, učinkovito prikazujući ili skrivajući mrežu u sceni.
- *GetWorldPosition* – vraća rezultat *CellToWorld* metode Grid komponente. Kada joj predamo koordinate ćelije vratit će nam koordinate globalne pozicije. Primjerice, za koordinate ćelije (5, 0, 4) će vratiti globalne koordinate (34.32, 2.13, 5).
- *GetCellPosition* – slično kao prethodna metoda samo što se ovdje predaje globalna pozicija, a vraća pozicija ćelije.
- *RaiseGrid* – podiže mrežu za određeni iznos, s ograničenjem kako bi se spriječilo da mreža izađe iz dozvoljenih okvira tvorničkog okruženja.
- *LowerGrid* – spušta mrežu za određeni iznos, s ograničenjem u dozvoljenim okvirima.
- *ResetGridHeight* – resetira mrežu na početni položaj, omogućujući brzo vraćanje na početni iznos konfiguracije.
- *SetGridHeight* – direktno postavlja položaj mreže na određenu visinu.

```
private void Start()
{
    _grid.cellSize = _gridCellSize;

    _gridPlane = _gridRenderer.gameObject;
    _defaultPosition = _gridPlane.transform.position;

    _gridRenderer.material.SetVector(_cellSizeParameter, new Vector2(1 /
_gridCellSize.x, 1 / _gridCellSize.z));
    _gridRenderer.material.SetVector(_defaultScaleParameter, new
Vector2(_defaultScale.x, _defaultScale.y));
}

public void ToggleGrid(bool value)
{
    _gridRenderer.gameObject.SetActive(value);
}

public Vector3 GetWorldPosition(Vector3Int cellPosition)
{
    return _grid.CellToWorld(cellPosition);
}

public Vector3Int GetCellPosition(Vector3 worldPosition)
{
    return _grid.WorldToCell(worldPosition);
}

public void RaiseGrid(float amount)
{
    float raiseAmount = _gridPlane.transform.position.y + amount;

    if(raiseAmount > 5)
        return;

    _gridPlane.transform.position = new Vector3(0, raiseAmount, 0);
}

public void LowerGrid(float amount)
{
    float lowerAmount = _gridPlane.transform.position.y - amount;

    if(lowerAmount < 0)
        return;

    _gridPlane.transform.position = new Vector3(0, lowerAmount, 0);
}

public void ResetGridHeight()
{
    _gridPlane.transform.position = _defaultPosition;
}

public void SetGridHeight(Vector3 position)
{
    _gridPlane.transform.position = position;
}
```

Sl. 5.4. Programski kod *GridManager* klase

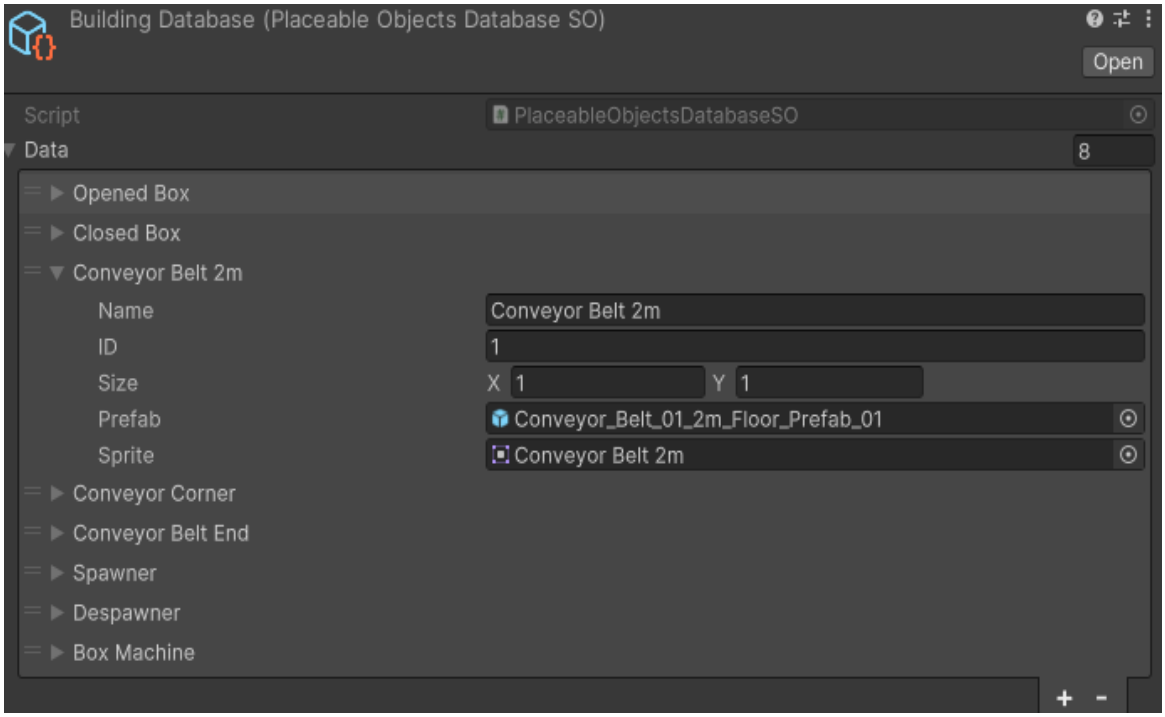
Objekti koji se postavljaju na mrežu moraju imati *Rigidbody*, *PlaceableObject* i *Collider* komponente. Kompleksniji modeli sadrže više *Collider* komponenti kako bi kolizija s drugim objektima pravilno funkcionirala. Da bi se objekt mogao postaviti na mrežu potrebno ga je učiniti *prefabom* i dodati u bazu objekata za gradnju. Klasa *PlaceableObjectDatabaseSO* (Slika 5.5.) nasljeđuje *ScriptableObject* komponentu te je osmišljena kao baza podataka za objekte koji se mogu postavljati na mrežu. Ova klasa omogućuje pohranu, upravljanje i validaciju podataka o objektima. Lista *Data* sprema podatke tipa *PlaceableData* te joj je dodijeljen atribut *SerializedField* što ju čini vidljivom u editoru. S obzirom na to da *PlaceableData* klasa ima ovaj atribut na svim varijablama, podaci se mogu uređivati unutar Unity grafičkog sučelja (Slika 5.6.). Varijable su sljedeće:

- *Name* – naziv objekta, koji se koristi za identifikaciju i referenciranje unutar simulacije.
- *ID* – jedinstveni identifikator objekta, koji omogućuje razlikovanje svakog objekta u bazi podataka.
- *Size* – dimenzije objekta izražene kao Vector2Int tip podatka, definiraju prostornu veličinu objekta unutar simulacije.
- *Prefab* – *prefab* objekta, predstavlja stvarni model koji će biti kreiran unutar simulacije.
- *Sprite* – slika koji se koristi za prikaz objekta u korisničkom sučelju ili drugim vizualnim elementima simulacije

Metodom *GetItemWithID* omogućuje se dohvaćanje objekta iz liste na temelju njegovog jedinstvenog identifikatora. *OnValidate* metoda je jedna od Unity *event* metoda. Poziva se svaki put kada se dogodi neka promjena u editoru. U ovom slučaju kada dodamo novi objekt u listu, a ne promijenimo mu identifikator, metoda će nas upozoriti na to.

```
[CreateAssetMenu]
public class PlaceableObjectsDatabaseSO : ScriptableObject
{
    [field: SerializeField] public List<PlaceableData> Data { get; private set; }

    public PlaceableData GetItemWithID(int id)
    {
        return Data.FirstOrDefault(x => x.ID == id);
    }

    private void OnValidate()
    {
        HashSet<int> IDs = new();
        foreach (var item in Data)
        {
            if (IDs.Contains(item.ID))
                Debug.LogError($"Dupliate ID found {item.ID} for {item.Name} in
PlaceableObjectsData");
            IDs.Add(item.ID);
        }
    }
}

[Serializable]
public class PlaceableData
{
    [field: SerializeField] public string Name { get; private set; }
    [field: SerializeField] public int ID { get; private set; }
    [field: SerializeField] public Vector2Int Size { get; private set; } =
Vector2Int.one;
    [field: SerializeField] public GameObject Prefab { get; private set; }
    [field: SerializeField] public Sprite Sprite { get; private set; }

}
```

Sl. 5.5. *ScriptableObject* kao baza objekata



Sl. 5.6. Dodavanje u *ScriptableObject* bazu

Glavna komponenta sustava gradnje je *BuildingSystem*. Kako sav kod ne bi bio smješten u jednu klasu, čime se otežava održavanje i preglednost, pomoćna logika sustava gradnje nalazi se u klasama *PreviewManager*, *PlacementManager*, *ObjectPlacer* i već spomenutoj klasi *GridManager*. Nakon što se u prozoru s tvorničkim elementima odabere jedan, poziva se *StartPlacement* metoda (Slika 5.7.). Predan joj je identifikacijski broj odabranog tvorničkog

```
public void StartPlacement(int id)
{
    if(_placementManager.IsPlacementActive)
        _placementManager.StopPlacement();

    _placeableData = _itemsDatabase.GetItemWithID(id);

    if (_placeableData == null)
    {
        Debug.LogError($"No item with id { id }");
        return;
    }

    _placementManager.IsPlacementActive = true;
    _gridManager.ToggleGrid(true);

    GameObject go = Instantiate(_placeableData.Prefab);
    PlaceableObject placeableObject = go.GetComponentInChildren<PlaceableObject>();

    _previewManager.SetPlacementPreviewObject(placeableObject);
    _placementManager.UpdatePlaceableObject(placeableObject);

    placeableObject.IgnorePhysics();

    _placementManager.ConnectInputToBuildingState();

    BuildingStarted?.Invoke(this, EventArgs.Empty);
}
```

Sl. 5.7. Metoda *StartPlacement*

elementa s pomoću kojeg se pronađe odgovarajući *prefab* iz baze objekata za postavljanje. Metoda prvotno provjerava je li gradnja već u tijeku. Ako je, poziva se *StopPlacement* metoda koja zaustavi gradnju, obriše odabrani element, sakrije mrežu i očisti memoriju. Ako nije, označava se da je gradnja aktivna postavljajući svojstvo *IsPlacementActive* na *true*, prikazuje se *GameObject* mreže i kreira se *GameObject* iz odabranog *prefaba* tvorničkog elementa. Na kreiranom *GameObject*-u dohvaća se komponenta *PlaceableObject* metodom *GetComponentInChildren*, koja se zatim preda metodama *SetPlacemenetPreviewObject* i *UpdatePlaceableObject* radi ažuriranja novo odabranog objekta za postavljanje. Na objektu tipa *PlaceableObject* poziva se *IgnorePhysics* metoda. Ona postavlja svojstvo *Rigidbody* komponente *isKinematic* na *true* kako bi se *GameObject* izuzeo iz djelovanja fizike. Razlog tome je što prilikom pomicanja odabranog tvorničkog elementa na željeno mjesto, ne trebamo djelovanje fizike jer bi to uzrokovalo sudare i utjecalo na pozicije ostalih već postavljenih tvorničkih elemenata. Kolizije i dalje postoje ali su drugog tipa, zato *IgnorePhysics* postavlja *isTrigger* svojstvo na *true* na sve *Collider* komponente

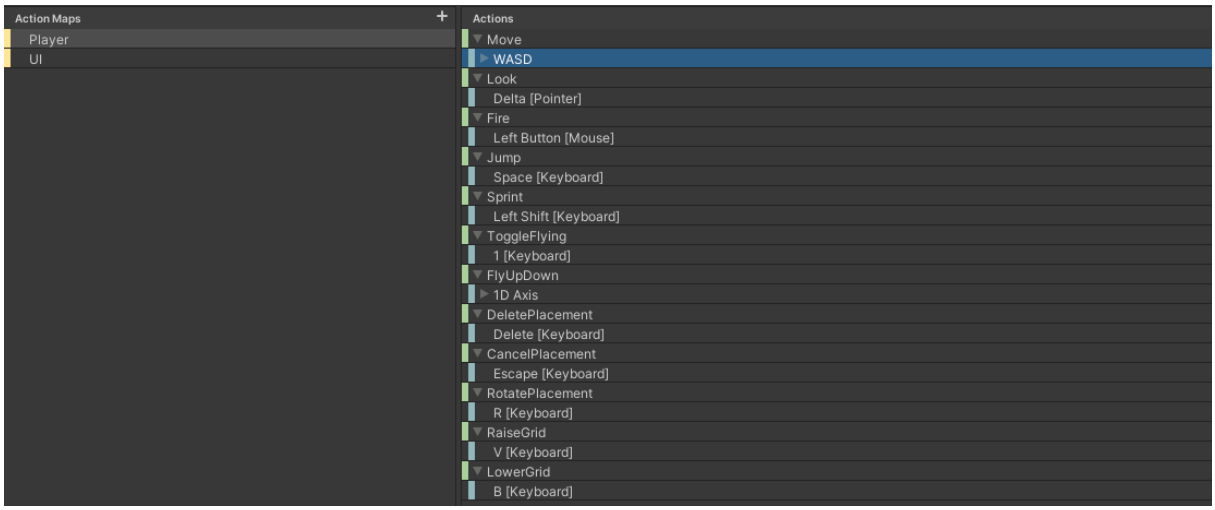
GameObject-a. Poanta takve kolizije je da i dalje očita dodire s drugim objektima ali bez utjecaja fizike. Nakon ove metode poziva se *ConnectInputToBuildingState* metoda. U njoj se povezuju događaji (eng. *events*) odziva kontrola s tipkovnice na odgovarajuće metode. Na kraju *StartPlacement* metode podiže se događaj *BuildingStarted* koji omogućuje ostalim klasama da znaju kada je započela gradnja i provedu potrebnu logiku pretplatom na njega.

Unity nudi kompleksan sustav ulaznih kontrola i mogućnost konfiguracije istih (Slika 5.9.). Pritiskom tipke na tipkovnici određena ulazna akcija podiže događaj *performed* koji će aktivirati pretplaćenu metodu. Pretplate na ove događaje nalaze se u metodi *ConnectInputToBuildingState* (Slika 5.8.). Akcijska mapa *Player* osim konfiguriranih akcija za kretanje ima i akcije za upravljanje gradnjom. To su:

- *DeletePlacement* – pritiskom na tipku *Delete* aktivira se *OnPlacementDeleted* metoda koja odabrani tvornički element obriše. Ovo vrijedi kada je on prethodno već postavljen na mrežu.
- *CancelPlacement* – pritiskom na tipku *Escape* aktivira se *OnPlacementCanceled* metoda koja otkazuje gradnju te vraća tvornički element na zadnju poziciju ako je bio postavljen na mrežu ili ga ukloni ako je bio odabran iz izbornika.
- *RotatePlacement* – pritiskom na tipku R aktivira se *OnRotation* metoda koja će rotirati objekt.
- *RaiseGrid* – pritiskom na tipku V podiže se *GameObject* mreže.
- *LowerGrid* – pritiskom na tipku B spušta se *GameObject* mreže.

```
public void ConnectInputToBuildingState()
{
    _inputActions.Player.DeletePlacement.performed += OnPlacementDeleted;
    _inputActions.Player.CancelPlacement.performed += OnPlacementCancelled;
    _inputActions.Player.RotatePlacement.performed += OnRotation;
    _inputActions.Player.RaiseGrid.performed += OnRaiseGrid;
    _inputActions.Player.LowerGrid.performed += OnLowerGrid;
}
```

Sl. 5.8. Programski kod metode *ConnectInputToBuildingState*



Sl. 5.9. Sustav konfiguriranja kontrola

U *Update* metodi pozivaju se metode *SelectionChanged* i *PlacementSelected* (Slika 5.10). *SelectionChanged* je početak programske logike za premještanje tvorničkog elementa po mreži (Slika 5.11.). Ako je gradnja neaktivna ili se pokazivač miša nalazi na korisničkom sučelju, izlazi se iz metode kako bi se spriječilo opterećivanje programa daljnjim provjerama s obzirom na to da se ovaj kod pokreće svaki *frame*.

```
private void Update()
{
    SelectionChanged();
    PlacementSelected();
}
```

Sl. 5.10. *Update* metoda klase *BuildingSystem*

```
private void SelectionChanged()
{
    if (!_placementManager.IsPlacementActive || SelectionManger.IsPointerOverUI())
        return;


_placementManager.HandleSelectionChanged(_inputManager.GetSelectedMapPosition());
}
```

Sl. 5.11. *SelectionChanged* metoda klase *BuildingSystem*

HandleSelectionChanged metodi *PlacementManager* klase predaje se rezultat metode *GetSelectedMapPosition* (Slika 5.12.). Ova metoda je dio klase *SelectionManager*. Prvotno se dohvaća trenutna pozicija miša na ekranu koja se sprema u varijablu *mousePos*. Varijabla je tipa *Vector3* te se njena z-koordinata postavlja na *nearClipingPlane* vrijednost. Ova vrijednost je najbliža udaljenost (ravnina) od kamere na kojoj se počinju prikazivati objekt u sceni. Ovo

osigurava da zraka počinje od te udaljenosti. Sljedeće se generira zraka iz točke na ekranu gdje se nalazi pokazivač miša. Njome se detektiraju objekti u 3D prostoru. Metoda *Raycast* zatim provjerava koliziju s objektima u sceni do udaljenosti od 999 jedinica. Parametar *hit* pohranjuje informacije o tome što je zraka pogodila. Varijabla *placementLayerMask* ograničava zraku na specifični sloj (eng. *Layer*) u sceni, a postavljena je na sloj *Placement*. Njime je označen *GameObject* mreže kako bi se zrakom dobile točke udara samo na njemu.

```
public Vector3 GetSelectedMapPosition()
{
    Vector3 mousePos = Input.mousePosition;
    mousePos.z = _sceneCamera.nearClipPlane;
    Ray ray = _sceneCamera.ScreenPointToRay(mousePos);
    RaycastHit hit;
    if(Physics.Raycast(ray, out hit, 999f, _placemenetLayerMask))
    {
        _lastPosition = hit.point;
    }

    return _lastPosition;
}
```

Sl. 5.12. *GetSelectedMapPosition* metoda klase *SelectionManager*

Nakon što je dobivena pozicija na mreži i predana metodi *HandleSelectionChanged* provode se dodatne provjere u metodi *ModifySelection* (Slika 5.13.) prije nego se *GameObject* pomakne na sljedeće polje. Prilikom svakog pomicanja pamti se pozicija te se u metodi *TryUpdatingPosition* uspoređuje s prethodnom kako se programski kod zadužen za premještanje ne bi izvodio konstantno ako se miš zadrži na istoj poziciji, a *GameObject* je već pozicioniran. Ako se radi o novoj poziciji odrađuje se njena validacija, odnosno provjera postoji li već neki *GameObject* u blizini s kojim bi se moglo doći u kontakt.

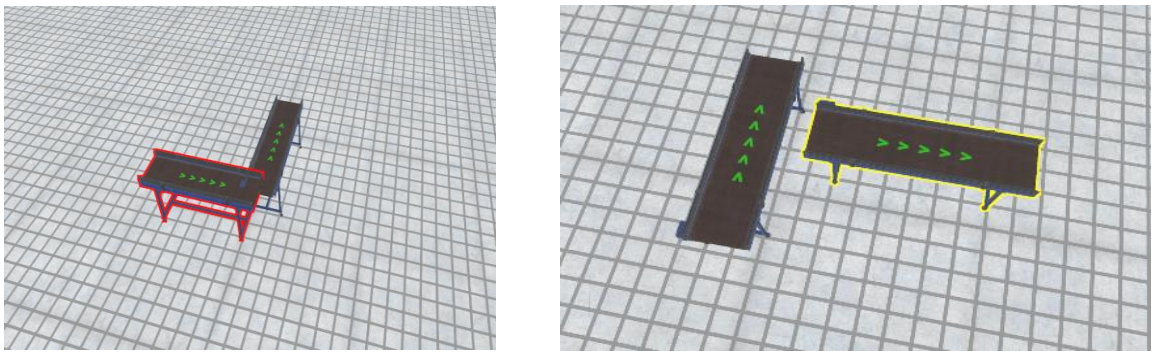
```
private bool ModifySelection(Vector3 mousePosition)
{
    Vector3Int tempPos = _gridManager.GetCellPosition(mousePosition);
    _selectionData.PlacementValidity = ValidatePlacement();

    if (_lastDetectedPosition.TryUpdatingPosition(tempPos))
    {
        _selectionData.SelectedWorldPosition =
_gridManager.GetWorldPosition(_lastDetectedPosition.GetPosition());
        _selectionData.PlacementValidity = ValidatePlacement();
        _selectionData.GridPosition = _gridManager.GridPosition;
        return true;
    }

    return false;
}
```

Sl. 5.13. Programski kod *ModifySelection metode*

Validacija je bazirana na sustavu kolizije, mreža služi samo za pozicioniranje i ograničenje gdje se može graditi. Validacija se obavlja u tri metode *PlaceableObject* komponente. To su *OnTriggerEnter*, *OnTriggerStay* i *OnTriggerExit* (Slika 5.14.). Sve tri metode postavljaju svojstvo *IsInCollision* na određeno stanje, kojim se označava da je objekt u sudaru. Kada se pomicanjem tvorničkog elementa po mreži dođe u doticaj s drugim tvorničkim elementima *OnTriggerEnter* i *OnTriggerStay* će postaviti *IsInCollision* na *true*. *OnTriggerEnter* se aktivira samo kod prvog kontakta, a *OnTriggerStay* se ponavlja sve dok postoji kontakt. Kada se kontakt prekine *OnTriggerExit* postavlja *IsInCollistion* na *false*. Validacija ne utječe na pomicanje tvorničkog elementa po mreži, on se može naći unutar drugih objekata, no postavljanje na zauzeto mjesto će biti zabranjeno. *GameObject* je označen žuto ako je mjesto na mreži slobodno, a crveno ako je zauzeto (Slika 5.15.).

```
private void OnTriggerEnter(Collider other)
{
    OnTrigger?.Invoke(this, other.gameObject);

    if (other.gameObject.GetComponentInParent<PlaceableObject>() == null)
        return;

    IsInCollision = true;
}

private void OnTriggerStay(Collider other)
{
    if(other.gameObject.GetComponentInParent<PlaceableObject>() == null)
        return;

    IsInCollision = true;
}

private void OnTriggerExit(Collider other)
{
    if(other.gameObject.GetComponentInParent<PlaceableObject>() == null)
        return;

    IsInCollision = false;
}
```

Sl. 5.14. Programski kod *OnTriggerEnter*, *OnTriggerExit* i *OnTriggerStay* metoda



Sl. 5.15. Prikaz validacije zauzetosti na mreži

Metoda *MovePreview* (Slika 5.16.) klase *PreviewManager* premješta *Transform* komponentu tvorničkog elementa na novu poziciju. Stanje premještanja je zapravo stanje pretpregleda (eng. *preview*) gdje se tvornički element samo pomiče po mreži bez postavljanja.

Izvođenje mehanizma za postavljanje objekta kreće od *PlacementSelected* metode (Slika 5.17.). Ova metoda prvo provjerava je li lijeva tipka miša pritisnuta i je li pokazivač iznad korisničkog sučelja. Ako bilo koji od ovih uvjeta nije ispunjen metoda završava izvršavanje. Ako je aktivno postavljanje objekata, poziva se metoda *HandleSelectionStarted* s pozicijom na mapi koju je odabrao igrač. U suprotnom, dohvaća se odabrani *GameObject* na mreži i pokušava se dobiti njegova komponenta *PlaceableObject* čija referenca se sprema u varijablu *placeableObject*. Ako je varijabla *placeableObject* različita od *null*, ažurira se stanje odabranog *GameObjecta* kroz metodu *UpdateSelectedPlaceableObject* te započinje premještanje objekta pozivom metode *StartMoving*.

```
public void MovePreview(Vector3 position, Quaternion rotation)
{
    _previewObject.transform.position = new Vector3(position.x, position.y,
position.z);
    _previewObject.transform.rotation = rotation;
}
```

Sl. 5.16. Programski *MovePreview* metode

```
private void PlacementSelected()
{
    if (!Mouse.current.leftButton.wasPressedThisFrame ||
EventSystem.current.IsPointerOverGameObject(PointerInputModule.kMouseLeftId))
        return;

    if (_placementManager.IsPlacementActive)
    {

_placementManager.HandleSelectionStarted(_inputManager.GetSelectedMapPosition());
    }
    else
    {
        var go = _inputManager.GetSelectedGameObject();
        var placeableObject = go.GetComponentInParent<PlaceableObject>();

        if (placeableObject != null)
        {
            _placementManager.UpdateSelectedPlaceableObject(placeableObject);
            StartMoving(placeableObject);
        }
    }
}
```

Sl. 5.17. Programski kod metode *PlacementSelected*

Pozivom *HandleSelectionStarted* metode (Slika 5.18.) resetira se posljednja detektirana pozicija, poziva metoda *ModifySelection* koja je prethodno objašnjena te ako ona vrati *true*, pokušava se postaviti objekt s pomoću metode *PlaceObject*.

Metoda *PlaceObject* (Slika 5.19.) pravi razliku radi li se o već postavljenom *GameObject*-u kojeg je korisnik odabrao za premještanje ili novo odabranom za postavljanje. U slučaju da je već postavljen, poziva metodu *PlaceObject*, a u suprotnom metodu *BuildObject*. Metode za gradnju i pozicioniranje, *BuildObject* i *PlaceObject*, nalaze se u komponenti *ObjectPlacer*. Obje metode pozivaju *SetPlaceableObject* metodu koja postavlja odabrani *GameObject* kao podređenu komponentu *ObjectPlacer GameObject*-a te svojstvo *IsKinematic* postavlja na *true* i isključuje *Collider* komponente. Zatim ažurira poziciju i rotaciju. Razlika je što *BuildObject* metoda sprema svaki izgrađeni objekt u listu i podiže događaj *ObjectBuilt* kako bi javila da je objekt izgrađen po prvi put.

```
public void HandleSelectionStarted(Vector3 selectedMapPosition)
{
    _lastDetectedPosition.Reset();
    if(ModifySelection(selectedMapPosition));
        PlaceObject(_placeableObject);
}
```

Sl. 5.18. Programski kod metode *HandleSelectionStarted*

```
public void PlaceObject(PlaceableObject placeableObject)
{
    if (!placeableObject.SelectionData.PlacementValidity)
        return;

    if (_selectedPlaceableObject != null)
    {
        _objectPlacer.PlaceObject(placeableObject);
    }
    else
    {
        _objectPlacer.BuildObject(placeableObject);

    }

    StopPlacement();
    ObjectPlaced?.Invoke(placeableObject);
}
```

Sl. 5.19. Programski kod metode *PlaceObject*

Premještanje objekta započinje pozivom metode *StartMoving* (Slika 5.20.). Slična je metodi *StartPlacement* ali ovdje se radi o *GameObject*-u koji je već postavljen na mrežu, a korisnik ga je odabrao. Metoda označuje da je postavljanje aktivno, isključuje fiziku na *GameObject*-u i ažurira trenutno postavljeni *GameObject* na objektima klasa *PlacementManger* i *PreviewManager*. Zatim uključuje prikaz mreže i postavlja njenu visinu. Konačno, povezuje unos korisnika s akcijama izgradnje metodom *ConnectIntputToBuildingState* i pokreće događaj *BuildingStarted*. Ostatak logike premještanja odvija se u *SelectionChanged* metodi koja je prethodno objašnjena.

```
private void StartMoving(PlaceableObject placeableObject)
{
    _placementManager.IsPlacementActive = true;

    placeableObject.IgnorePhysics();

    _previewManager.SetPlacementPreviewObject(placeableObject);
    _placementManager.UpdatePlaceableObject(placeableObject);

    _gridManager.ToggleGrid(true);
    _gridManager.SetGridHeight(placeableObject.SelectionData.GridPosition);

    _placementManager.ConnectInputToBuildingState();

    BuildingStarted?.Invoke(this, EventArgs.Empty);
}
```

Sl. 5.20. Programski kod metode *StartMoving*

5.2. Funkcionalnost tvorničkih elemenata

S obzirom na to da se radi o simulaciji, od tvorničkih elemenata očekuje se da simuliraju stvarne funkcionalnosti. U slučaju pokretnih traka, one moraju voditi objekt s jednog kraja trake na drugi određenom brzinom. Da bi se ovo postiglo, svaki *GameObject* koji predstavlja tvornički element ima komponente koja opisuje njegovu funkcionalnost.

Osnovna komponenta svakog tvorničkog elementa je *BaseFactoryElement* klasa (Slika 5.21.). Ona je apstraktna te pruža osnovne parametre i logiku. Izvedene klase mogu ju prilagoditi i proširiti prema specifičnim potrebama. Polja *Id* i *Name* određuju identifikacijski broj i ime tvorničkog elementa. Identifikacijski broj je samo poredak postavljanja, prvi tvornički element će imati *Id* jednak 1, a deseti 10. Ime se postavlja kroz *editor*. *Transform* komponenta omogućuje upravljanje pozicioniranjem, a *Rigidbody* komponenta fizikom. Dohvaćanje *PlaceableObject* komponente je zato što je tvornički element ipak objekt za postavljanje i neki parametri ove komponente mogu poslužiti izvedenim klasama. *HighlightEffect* komponenta služi za isticanje *GameObject*-a, a aktivira/deaktivira se metodom *SetHighlight* kojoj možemo predati željenu boju

isticanja. Virtualna metoda *SetFactoryBehaviour* pozvana je u trenutku pokretanja tvornice. Sprema početne pozicije na koje će se *GameObject* postaviti pozivom metode *ResetFactoryBehaviour* kada se tvornica ugasi. Vraćanje pozicije na izvornu bitno je u slučaju da se tvornički elementi pomaknu s mjesta na mreži gdje su početno stavljeni, primjerice kutije na pokretnoj traci. Izvedene klase mogu ove dvije metode prepisati i dodati proširenu logiku.

```
public abstract class BaseFactoryElement : MonoBehaviour
{
    public int Id { get; set; }
    [field: SerializeField] public string Name { get; set; }

    private Vector3 _startPosition;
    private Vector3 _childStartPosition;

    private Transform _childTransform;
    public Rigidbody Rigidbody { get; private set; }
    public PlaceableObject PlaceableObject { get; private set; }

    private HighlightEffect _highlightEffect;

    protected virtual void Awake()
    {
        _startPosition = transform.position;
        _childTransform = transform.GetChild(0).transform;

        Rigidbody = GetComponentInChildren<Rigidbody>();
        PlaceableObject = GetComponentInChildren<PlaceableObject>();
        _highlightEffect = GetComponentInChildren<HighlightEffect>();
    }

    public virtual void SetFactoryBehaviour()
    {
        _startPosition = transform.position;
        _childStartPosition = _childTransform.localPosition;
    }
    public virtual void ResetFactoryBehaviour()
    {
        transform.position = _startPosition;
        _childTransform.localPosition = _childStartPosition;
        _childTransform.localRotation = Quaternion.identity;
    }

    public void SetHighlight(bool highlight, Color colour = default)
    {
        _highlightEffect.outlineColor = colour;
        _highlightEffect.SetHighlighted(highlight);
    }
}
```

Sl. 5.21. Programski kod klase *BaseFactoryElement*

Određeni tvornički elementi imaju funkcionalnost stroja, odnosno simuliraju pokretne dijelove te se mogu upaliti i ugasiti. Bazna komponenta takvog elementa je *BaseMachine* (Slika 5.23.) apstraktna klasa. Ona implementira sučelje *IMachine* (Slika 5.22.) koje propisuje metode *StartMachine*, *StopMachine*, svojstvo *IsActive* i događaj *ActiveStateChanged*. *StartMachine* i *StopMachine* metode postavljaju *IsActive* svojstvo na određeno stanje, a *IsActive* svojstvo u *setteru* podiže događaj *ActiveStateChanged* kako bi se moglo reagirati na promjene stanja stroja. Obje su metode virtualne pa se u izvedenim klasama omogućuje proširivanje logike.

```
public interface IMachine
{
    bool IsActive { get; set; }

    event EventHandler<ActiveStateEventArgs> ActiveStateChanged;
    void StartMachine();
    void StopMachine();
}
```

Sl. 5.22. Programski kod sučelja *IMachine*

```
public abstract class BaseMachine : BaseFactoryElement, IMachine
{
    private bool _isActive;

    public bool IsActive
    {
        get => _isActive;
        set
        {
            _isActive = value;
            ActiveStateChanged?.Invoke(this, new ActiveStateEventArgs { Id = Id,
IsActive = value });
        }
    }

    public event EventHandler<ActiveStateEventArgs> ActiveStateChanged;

    public virtual void StartMachine()
    {
        IsActive = true;
    }

    public virtual void StopMachine()
    {
        IsActive = false;
    }
}
```

Sl. 5.23. Programski kod klase *BaseMachine*

Za upravljanje tvornicom zadužena je *FactoryManager* komponenta. U njenoj *Awake* metodi nalaze se pretplate na događaje *ObjectBuilt* i *ObjectRemoved* klase *BuildingSystem*. Na ovaj način točno se zna kada je tvornički element postavljen ili uklonjen s mreže. U slučaju podizanja *ObjectBuilt* događaja poziva se *OnObjectBuilt* metoda. Ona sprema tvornički element u listu *_factoryElements*, a ako se radi o elementu koji je stroj (sadrži *IMachine* sučelje) dodatno ga još spremi u listu *_machineElements*. *OnObjectRemoved* metoda reagira na podizanje događaja *ObjectRemoved* te čisti spomenute liste od tvorničkih elemenata koji su uklonjeni (Slika 5.24.).

```
private void OnObjectBuilt(object sender, PlaceableObjectEventArgs e)
{
    var factoryElement =
e.PlaceableObject.GetComponentInParent<BaseFactoryElement>();

    factoryElement.Id = _factoryElements.Count + 1;

    _factoryElements.Add(factoryElement);

    FactoryElementBuilt?.Invoke(this, factoryElement);

    if (factoryElement is IMachine)
    {
        _machineElements.Add(factoryElement);

        MachineBuilt?.Invoke(this, factoryElement);
    }
}

private void OnObjectRemoved(object sender, PlaceableObjectEventArgs e)
{
    var factoryElement =
e.PlaceableObject.GetComponentInParent<BaseFactoryElement>();

    _factoryElements.Remove(factoryElement);

    FactoryElementRemoved?.Invoke(this, factoryElement);

    if (factoryElement is IMachine)
    {
        _machineElements.Remove(factoryElement);

        MachineRemoved?.Invoke(this, factoryElement);
    }
}
```

Sl. 5.24. Programski kod klase *BaseMachine*

Pokretanje tvornice događa se preko *StartFactory* metode (Slika 5.25.). Ona postavlja svojstvo *IsFactoryRunning* na *true* čime se označuje početak rada tvornice. Petljom se prolazi kroz sve tvorničke elemente u listi te se poziva njihova *SetFactoryBehaviour* metoda. Provjerava se je li element stroj, odnosno sadrži li *IMachine* sučelje te se poziva *StartMachine* ako je uvjet zadovoljen. Za stopiranje tvornice poziva se *StopFactory* metoda (Slika 5.25.). Ona postavlja *IsFactoryRunning* na *false* i poziva *ResetFactoryBehaviour* na svakom elementu uz dodatni poziv *StopMachine* metode ako je element stroj. Na kraju obje metode nalazi se *EventBus* klasa s pozivom na Raise metodu. *EventBus* je statička klasa i funkcionalnošću je slična C# događaju, no klase koje žele pretplatu na ovakav događaj ne moraju znati za konkretnu implementaciju klase u kojoj se događaj nalazi i podiže. Trebaju znati samo za tip događaja na koji se žele pretplatili. U ovom slučaju su to događaji tipa *FactoryStarted* i *FactoryStopped*, odnosno događaji koji se podižu kada se tvornica pokrene i stopira. Ovo je posebno korisno za klase koje se nalaze u drugim

```
public void StartFactory()
{
    IsFactoryRunning = true;

    foreach (var factoryElement in _factoryElements)
    {
        factoryElement.SetFactoryBehaviour();

        if (factoryElement is IMachine)
        {
            var machine = factoryElement as IMachine;
            machine.StartMachine();
        }

    }

    EventBus<FactoryStarted>.Raise(new FactoryStarted());
}

public void StopFactory()
{
    IsFactoryRunning = false;

    foreach (var factoryElement in _factoryElements)
    {
        factoryElement.ResetFactoryBehaviour();

        if (factoryElement is IMachine)
        {
            var machine = factoryElement as IMachine;
            machine.StopMachine();
        }
    }

    EventBus<FactoryStopped>.Raise(new FactoryStopped());
}
```

Sl. 5.25. Programski kod metoda StartFactory i StopFactory

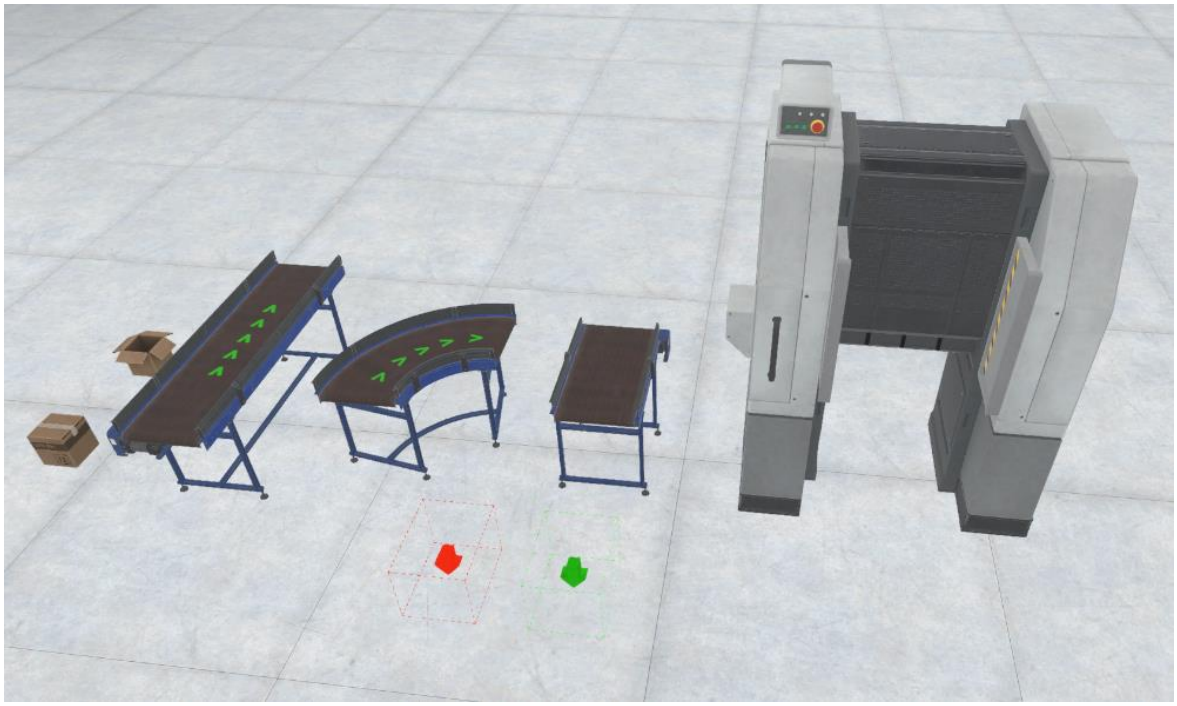
scenama ili čiji objekti su dinamički stvoreni te je dohvaćanje referenci drugih klasa otežano. Na slici 5.26. možemo vidjeti da potrebno kreirati objekt *EventBinding* s tipom *eventa* kojeg želimo slušati te mu predati metodu koja će se pozvati kod podizanja *eventa*. Za pretplatu predamo taj objekt *Register* metodi *EventBus* klase.

```
_factoryStartedEvent = new
EventBinding<FactoryStarted>(EventBinding_OnFactoryStarted);
_factoryStoppedEvent = new
EventBinding<FactoryStopped>(EventBinding_OnFactoryStopped);

EventBus<FactoryStarted>.Register(_factoryStartedEvent);
EventBus<FactoryStopped>.Register(_factoryStoppedEvent);
```

Sl. 5.26. Programski kod pretplate na event

Dostupni tvornički elementi za izgradnju proizvoljnog tvorničkog postrojenja su (Slika 5.27.):

- Pokretna traka
- Kutna pokretna traka
- Završna pokretna traka
- Otvorena kutija
- Zatvorena kutija
- Stroj za pakiranje kutija
- Uklanjač objekata
- Generator objekata



Sl. 5.27. Prikaz svih tvorničkih elemenata

5.2.1. Pokretna traka, kutna pokretna traka i kraj pokretne trake

Pokretne trake sadrže komponentu *Conveyor* koja nasljeđuje *BaseMachine* komponentu. U prepisanoj metodi *StartMachine* postavlja *isKinematic* i *IsRunning* svojstva na *true*. *IsRunning* je svojstvo klase *ConveyorBehaviour* (Slika 5.28.). Ova klasa je bazna za sve pokretne trake te je ključna za definiranje brzine, smjera i statusa, omogućujući dinamičko rukovanje objektima unutar sustava tvornice. Sadrži svojstva:

- *Speed* – određuje brzinu pokretne trake.
- *SpeedOffset* – odstupanje brzine koje se može koristiti za dodatne prilagodbe.
- *IsRunning* – *boolean* varijabla koja označava je li pokretna traka trenutno u pogonu.
- *DefaultSpeed* – zadana brzina koja s koristi kao referenca.

Klasa sadrži jedan događaj, *DirectionChanged*, koji se pokreće kada se promijeni smjer pokretne trake. Metoda *ChangeDirection* mijenja smjer kretanja pokretne trake. Promjena smjera postiže se množenjem trenutne brzine i offseta brzine s -1. Nakon što se smjer promijeni, pokreće se događaj *DirectionChanged* i prosljeđuje nova vrijednost brzine. Metoda *SetSpeed* postavlja novu brzinu pokretne trake. Ako je trenutna brzina negativna, nova brzina će također biti postavljena kao negativna, zadržavajući smjer kretanja. U suprotnom, brzina je pozitivna.

```
public class ConveyorBehaviour : MonoBehaviour
{
    [field: SerializeField] public float Speed { get; set; }
    [field: SerializeField] public float SpeedOffset { get; set; }
    [field: SerializeField] public bool IsRunning { get; set; }
    [field: SerializeField] public float DefaultSpeed { get; set; }

    public event Action<float> DirectionChanged;

    public void ChangeDirection()
    {
        Speed = Speed * (-1);
        SpeedOffset = SpeedOffset * (-1);
        DirectionChanged?.Invoke(Speed);
    }

    public void SetSpeed(float speed)
    {
        if (Speed < 0)
        {
            Speed = speed * -1;
        }
        else
        {
            Speed = speed;
        }
    }
}
```

Sl. 5.28. Programski klase *ConveyorBehaviour*

Na slici 5.29. nalazi se *GameObject* ravne pokretne trake. Za njen rad implementirana je klasa *BeltBehaviour* (Slika 5.30.) koja nasljeđuje *ConveyorBehaviour*. Dodaje funkcionalnosti za upravljanje vizualnim efektima i fizičkim kretanjem pokretne trake. U FixedUpdate metodi prvotno se provjerava *IsRunning* svojstvo. Ova provjera osigurava da se transportna traka pomiče samo kada je *IsRunning* svojstvo na *true*. Varijabla *newPosition* sprema rezultat izračuna nove pozicije koji se postiže množenjem *transform.forward*, *Time.fixedDeltaTime* i *Speed* varijabli. *Transform.forward* pruža vektor koji pokazuje u smjeru prema naprijed pokretne trake. *Time.fixedDeltaTime* je fiksni vremenski interval između ažuriranja fizike, osiguravajući kretanje bez obzira na *framerate*. *Speed* je brzina pokretne trake. Množenjem ovih vrijednosti dobivamo udaljenost koju traka treba prijeći u trenutnom koraku ažuriranja. Pozicija *Rigidbody* komponente postavlja se na poziciju koja nastaje oduzimanjem trenutne pozicije i prethodno izračunate pozicije *newPosition* varijable. Ovim se pozicija *Rigidbody* komponente postavlja malo unatrag, kako bi se pripremilo za pomicanje odmah unaprijed. U sljedećoj liniji koda *RigidBody* se postavlja na novu poziciju metodom *MovePosition* dodavanjem trenutne pozicije i vrijednosti *newPosition* varijable. *MovePosition* je metoda koja osigurava glatku interpolaciju i rukovanje sudarima, pružajući točniju simulaciju kretanja. Osim fizičkog kretanja *Update* metoda također mijenja teksturu kako bi se stvorio vizualni efekt pokretanja trake.

Sl. 5.29. Prikaz ravne pokretne trake

```
public class BeltBehavior : ConveyorBehaviour
{
    [SerializeField] private Renderer _renderer;

    private float texOffset;

    private Rigidbody rigidBody;
    private Material material;

    private void Awake()
    {
        rigidBody = GetComponent<Rigidbody>();
        material = _renderer.material;
    }

    private void FixedUpdate()
    {
        if(!IsRunning)
            return;

        Vector3 newPosition = transform.forward * Time.fixedDeltaTime * Speed;
        //rigidBody.position = (rigidBody.position - newPosition);
        rigidBody.MovePosition(rigidBody.position + newPosition);
    }

    private void Update()
    {
        if(!IsRunning)
            return;

        if (material)
        {
            texOffset += Speed * Time.deltaTime;
            texOffset = texOffset % 1;

            material.mainTextureOffset = new Vector2(0, texOffset);
        }
    }
}
```

Sl. 5.30. Programski kod klase *BeltBehaviour*

Kada se *Rigidbody* pokretne trake pomiče, stvara se efekt koji izgleda kao da se traka kreće. S obzirom na to da je Rigidbody svojstvo *isKinematic* postavljeno na *true*, on se ne pomiče na način na koji bi normalni fizički objekti reagirali na sile, ali još uvijek može utjecati na druge *Rigidbody* objekte s kojima dolazi u kontakt. Objekti na transportnoj traci, koji također imaju *Rigidbody* komponente, doživljavaju promjenu pozicije trake kao pomicanje prema naprijed. Razlog je što promjena pozicije uzrokuje trenje između trake i objekta. Zbog trenja objekti nastoje ostati u kontaktu s površinom trake i bivaju pomaknuti s njom. Ovaj efekt se stvara kroz sudare i kontaktne točke *Collider* komponenti.

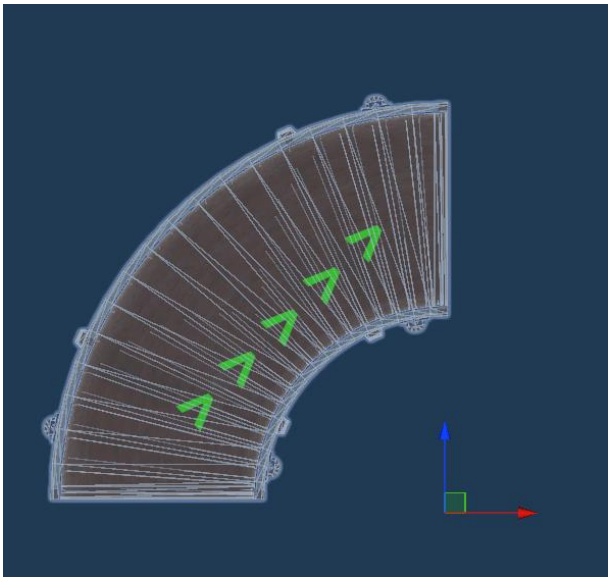
Na sličan način funkcionira i kutna pokretna traka (Slika 5.31.). Za njen rad implementirana je klasa *RadialConveyorBehaviour* koja također nasljeđuje *ConveyorBehaviour* klasu. U *FixedUpdate* metodi (Slika 5.32.), koja se u poziva fiksnim intervalima, provjerava se je li traka u radu. Ako jest, koristi se *AngleAxis* metoda strukture *Quaternion* za izračun rotacije objekta

Sl. 5.31. Prikaz kutne pokretne trake

oko definirane osi (*angleAxis*). Ova rotacija se temelji na vrijednostima *angle* varijable, *Speed* svojstva i vremenskog intervala *Time.fixedDeltaTime*. Zatim se izračunava nova rotacija s istim vrijednostima ali je u ovom slučaju *angle* varijabla negativna pa se ovdje zapravo poništava prva postavljena rotacija. Ovo se primjenjuje na *Rigidbody* metodom *MoveRotation* koja osigurava glatku tranziciju i preciznu rotaciju. Vrijednosti varijabli _angle i Speed su uzete proizvoljno metodom finog podešavanja kako bi kretanje po traci izgledalo što prirodnije. Princip je isti kao i kod ravne pokretne trake, Rigidbody svojstvo *IsKinematic* je postavljeno na *true* čime ostali objekti ne utječu na pokretnu traku ali ona svojim promjenama rotacije nazad i naprijed stvara trenje kojim gura objekte prema naprijed uzrokujući sudare na kontaktnim točkama *Collider* komponenti. Treba napomenuti i da je referentnu točka *GameObject-a* izvan njega, što je prikazano na slici 5.33. S obzirom na to da se rotacija izvršava oko referente točke, na ovaj način dobijemo putanju koja pravi krug koju onda i slijede objekti na traci.

```
void FixedUpdate ()
{
    if(!IsRunning)
        return;

    _rb.rotation *= Quaternion.AngleAxis ((_angle * Speed * Time.fixedDeltaTime),
_angleAxis);
    Quaternion newRot = _rb.rotation * Quaternion.AngleAxis ((-_angle * Speed *
Time.fixedDeltaTime), _angleAxis);
    _rb.MoveRotation (newRot);
}

private void Update()
{
    if(!IsRunning)
        return;

    if (!_material) return;

    _texOffset -= Speed * Time.deltaTime;
    _texOffset %= 1;
    _material.mainTextureOffset = new Vector2(0, _texOffset);
}
```

Sl. 5.32. Programski kod metoda *FixedUpdate* i *Update* klase *RadialConveyorBehaviour*



Sl. 5.33. Prikaz referentne točke kutne pokretne trake

Završna pokretna traka (Slika 5.34.) tvornički je element koji koristi istu logiku fizike kao i ravna pokretna traka te je dodatak na skup tvorničkih elemenata u slučaju da korisnik želi završiti liniju pokretnih traka s elementom koji vizualno predstavlja kraj.

Sl. 5.34. Prikaz završnog dijela pokretne trake

5.2.2. Stroj za pakiranje kutija

Tvornički element koji simulira pakiranje kutija (Slika 5.35.). S obzirom na to da je stroj, njegova komponenta *BoxMachine* nasljeđuje *BaseMachine* klasu. Na slici 5.36. prikazan je programski kod koji pokazuje da ova komponenta sadrži samo *Update* metodu. U njoj se prvo provjerava je li stroj aktivan. Ako jest, metodom *Physics* klase, *SphereCast*, stvara se zraka određene duljine iz referente točke *_sensorStartingPoint* koja na kraju ima sferu. Na ovaj način detektiraju se sudari s objektima koji dotaknu sferu. U ovom slučaju provjerava se je li detektirani objekt tipa *BoxElement*, odnosno kutija. Ako je, poziva se metoda *PackTheBox* u slučaju ako kutija nije zapakirana. Referentna točka je zapravo GameObject čiju Transform komponentu uzimamo za postavljanje početka zrake. Ona je postavljenja tako da se sfera nalazi točno iznad pokretnih traka koje će prolaziti kroz stroj.

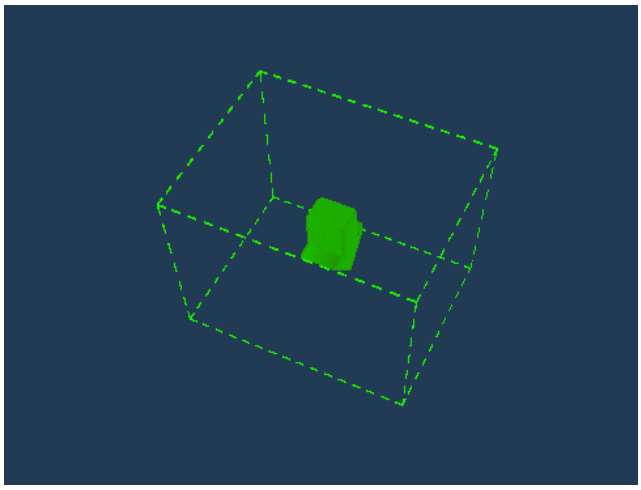
Sl. 5.35. Prikaz stroja za pakiranje kutija

```
public class BoxMachine : BaseMachine
{
    [SerializeField] private Transform _sensorStartingPoint;

    private void Update()
    {
        if(!IsActive) return;

        if(Physics.SphereCast(
            origin: _sensorStartingPoint.position,
            radius: 0.05f,
            direction: Vector3.down,
            hitInfo: out RaycastHit hit,
            maxDistance: 0.25f,
            drawDuration: 0.1f,
            hitColor: Color.green,
            noHitColor: Color.red
            ))
        {
            var box = hit.collider.GetComponentInParent<Box>();

            if(box != null)
            {
                if(box.IsPacked) return;

                box.PackTheBox();
            }
        }
    }
}
```

Sl. 5.36. Programski kod klase *BoxMachine*

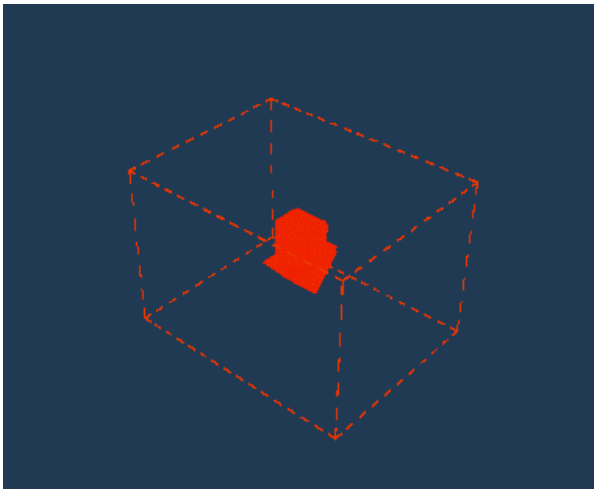
5.2.3. Generator i uklanjač objekata

Generator objekata (Slika 5.37.) je tvornički element zadužen za stvaranje objekata u određenom vremenskom intervalu. U ovom slučaju stvara nezapakirane kutije u intervalu od 3



Sl. 5.37. Prikaz generatora objekata

sekunde. S obzirom na to da se objekti stvaraju u njemu, kako ne bi došlo do naglih sudara i izbacivanja stvorenih objekta, njegov *Collider* je postavljen na *trigger* svojstvo. Uklanjač objekata (Slika 5.38.) je zadužen za uništavanje objekata koji dođu u doticaj s njime. Da bi objekt bio uništen potrebno je da njegova komponenta nasljeđuje sučelje *IDespawnTag*. Tako se zapravo zna je li objekt koji je došao u doticaj zapravo za uništenje. Proces uništenja objekta traje pola sekunde.



Sl. 5.38. Prikaz uklanjača objekata

5.2.4. Kutije

Postoje dvije vrste kutija, zapakirana i nezapakirana (Slika 5.39.). Obje vrste koriste komponentu *Box* koja nasljeđuje *BaseFactoryElement* klasu i *IDespawnTag* sučelje. Programski kod na slici 5.40. prikazuje da ova komponenta sadrži polja *boxOpenMesh* i *boxClosedMesh* koja su tipa Mesh. Ovo su reference na modele zapakirane i nezapakirane kutije. Kada se pozove *PackTheBox* metoda, postavlja se svojstvo *IsPacked* na *true* te se model kutije mijenja u onaj gdje je kutija zapakirana.

Sl. 5.39. Prikaz kutija

```
public class Box : BaseFactoryElement, IDespawnTag
{
    [SerializeField] private Mesh boxOpenMesh;
    [SerializeField] private Mesh boxClosedMesh;

    private MeshFilter _meshFilter;

    public bool IsPacked { get; set; }

    protected override void Awake()
    {
        base.Awake();

        _meshFilter = GetComponentInChildren<MeshFilter>();
    }

    public void PackTheBox()
    {
        IsPacked = true;
        _meshFilter.mesh = boxClosedMesh;
    }

    public override void SetFactoryBehaviour()
    {
        base.SetFactoryBehaviour();

        Rigidbody.isKinematic = false;
    }

    public override void ResetFactoryBehaviour()
    {
        base.ResetFactoryBehaviour();

        Rigidbody.isKinematic = true;
    }
}
```

Sl. 5.40. Programski kod klase *Box*

5.3. Korisničko sučelje

Korisniku je na raspolaganju interaktivno sučelje gdje može pokrenuti niz akcija. Sučelje se sastoji od panela i prozora. Panel je jednostavniji UI element koji može biti različitih veličina i oblika te može biti više od jednog prikazanog na ekranu. Prozor je UI element koji se prostire po cijelom ekranu i u svakom trenutku može biti samo jedan aktivan. Komponenta *UIFrame* omogućava efikasno upravljanje korisničkim sučeljem kroz dodavanje, uklanjanje i kontroliranje različitih UI elemenata. Kada se skripta *UIFrame* učita, ona može automatski inicijalizirati sve potrebne komponente ili čekati ručnu inicijalizaciju. Inicijalizacija se događa u *Awake* metodi, koja provjerava vrijednost *initializeOnAwake* varijable i poziva *Setup* metodu ako je ona postavljena na true (Slika 5.41.). Metoda *Setup* postavlja glavne komponente *Canvas*, *PanelUILayer*, *WindowUILayer* i *GraphicRaycaster*. Ako su komponente panela i prozora prisutni kao podređeni objekti glavnog objekta, oni se inicijaliziraju i pridružuju odgovarajuće događaje za blokiranje i deblokiranje ekrana. U slučaju da nisu prisutni, skripta ispisuje grešku.

```
public void Setup()
{
    _mainCanvas = GetComponent<Canvas>();

    if (_panelLayer == null)
    {
        _panelLayer = gameObject.GetComponentInChildren<PanelUILayer>(true);

        if (_panelLayer == null)
            Debug.LogError("[UI Frame] UI Frame lacks Panel Layer!");
        else
            _panelLayer.Initialize();
    }

    if (_windowLayer == null)
    {
        _windowLayer = gameObject.GetComponentInChildren<WindowUILayer>(true);

        if (_windowLayer == null)
            Debug.LogError("[UI Frame] UI Frame lacks Window Layer!");
        else
        {
            _windowLayer.Initialize();
            _windowLayer.RequestScreenBlock += OnRequestScreenBlock;
            _windowLayer.RequestScreenUnblock += OnRequestScreenUnblock;
        }
    }

    _graphicRaycaster = MainCanvas.GetComponent<GraphicRaycaster>();

    RegisterViews();
}
```

Sl. 5.41. Programski kod metoda *Awake* i *Setup* klase *UIFrame*

Registracija UI elemenata (panela i prozora) započinje metodom *RegisterViews* (Slika 5.42.). Ona prolazi kroz sve podređene objekte, pronalazi sve one koji implementiraju sučelje *IViewController* i registrira ih s pomoću metode *RegisterScreens*. Da bi *GameObject* bio panel, mora imati komponentu koja nasljeđuje *APanelController*, odnosno *IPanellController* sučelje koje je uvjetovano *IViewController* sučeljem. Ista stvar je i kod prozora, samo što je ovdje bazna komponenta *AWindowController*, a sučelje *IWindowController*.

```
private void RegisterViews()
{

    foreach (Transform t in _windowLayer.transform)
    {
            var view = t.GetComponent<IViewController>();

            if(view != null)
                RegisterScreen(t.name, view, t);
    }

    foreach (Transform t in _panelLayer.transform)
    {
        var view = t.GetComponent<IViewController>();

        if(view != null)
            RegisterScreen(t.name, view, t);
    }
}
```

Sl. 5.42. Programski kod metode *RegisterViews*

Skripta *UIFrame* omogućuje prikazivanje i sakrivanje panela na nekoliko načina. Metode *ShowPanel* i *HidePanel* koriste jedinstveni identifikator panela za prikazivanje ili sakrivanje odgovarajućeg UI elementa. Također, postoje generičke verzije ovih metoda koje koriste tip kontrolera za identifikaciju panela. Dodatno, metoda *ShowPanel* omogućuje prikazivanje panela s proslijeđenim svojstvima ili bez njih. Slične metode postoje i za prozore. Metode *OpenWindow* i *CloseWindow* koriste identifikator prozora za upravljanje njegovim prikazivanjem i sakrivanje, dok generičke verzije koriste tip kontrolera. Metoda *CloseCurrentWindow* zatvara trenutno aktivni prozor. Omogućena je i provjera jesu li paneli ili prozori otvoreni putem metoda *IsPanelOpen* i *IsWindowOpen*. Korisničko sučelje tvorničke simulacije sastoji se od glavnog prozora u igri, panela s tvorničkim elementima, serverskog panela i informacijskog panela.

5.3.1. Glavni prozor

Prozor koji je aktivan u glavnoj sceni (*Factory*). Na vrhu se nalazi alatna traka (Slika 5.43). Ona sadrži dugmad čije su slike i objašnjenja dani u tablici 5.1. Pri izlasku iz glavne scene ovaj prozor se zatvara.

Sl. 5.43. Alatna traka za upravljane simulacijom

	Gumb za izlazak iz scene *Factory*.
	Gumb za otvaranje i zatvaranje informacijskog panela.
	Gumb za otvaranje i zatvaranje server panela.
	Gumb za startanje i stopiranje tvornice.
	Gumb za promjenu načina kretanja u hodanje.
	Gumb za promjenu načina kretanja u letenje.
	Gumb za otvaranje i zatvaranje panela s tvorničkim elementima.

Tablica 5.1. Prikaz i objašnjenja gumbova alatne trake

5.3.2. Panel s tvorničkim elementima

Panel koji sadrži tvorničke elemente gdje je svaki tvornički element gumb predstavljen slikom i nazivom (Slika 5.44.). Odabirom jednog stvara se njegov *GameObject* u sceni. Prilikom pokretanja tvornice ovaj panel se zatvara te je njegovo otvaranje onemogućeno sve dok je tvornica u aktivnom statusu.

Sl. 5.44. Panel s tvorničkim elementima

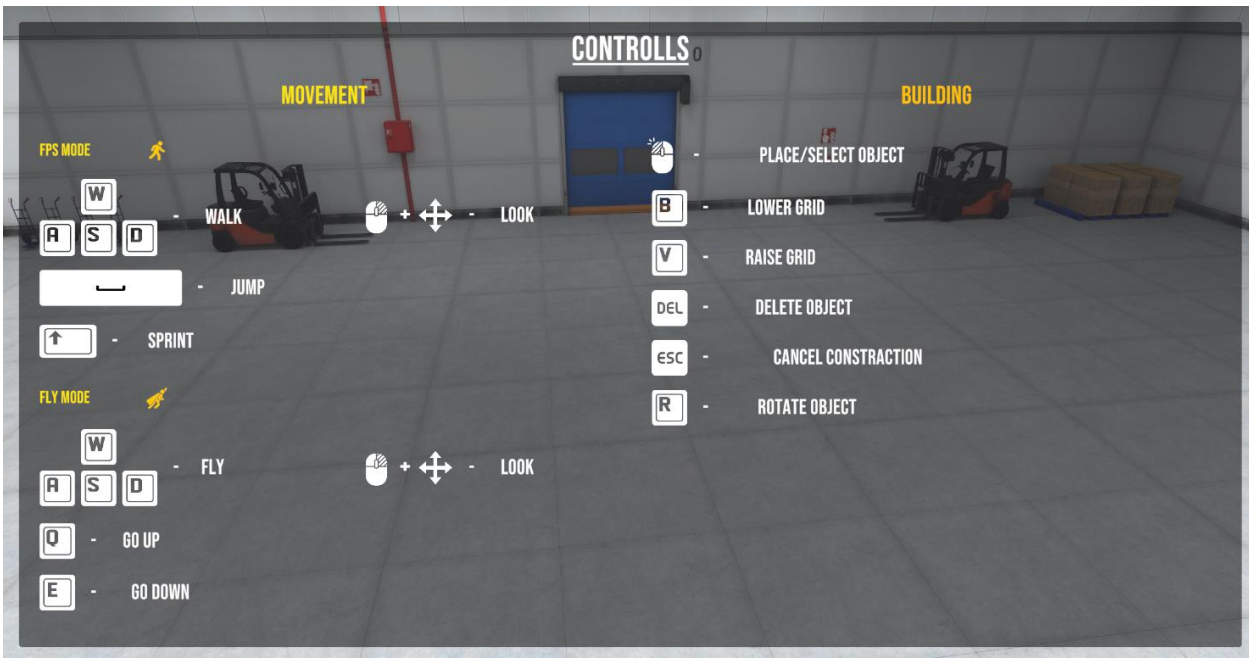
5.3.3. Serverski panel

Panel koji omogućuje konekciju s kontrolnom aplikacijom i prikazuje listu strojeva i njihov status. Sadrži gumb s tekstom *Connect* kojim se pokreće spajanje sa serverom. Ako je veza otvorena gumb mijenja tekst u *Disconnect*. Lista strojeva se mijenja ovisno o dodavanju ili uklanjanju tvorničkih elemenata. Prelaskom miša po elementu u listi isti se označava narančastom bojom te sukladno tome i njegov *GameObject* u sceni. Primjer ovoga možemo vidjeti na slici 5.45.

Sl. 5.45. Panel s tvorničkim elementima

5.3.4. Informacijski panel

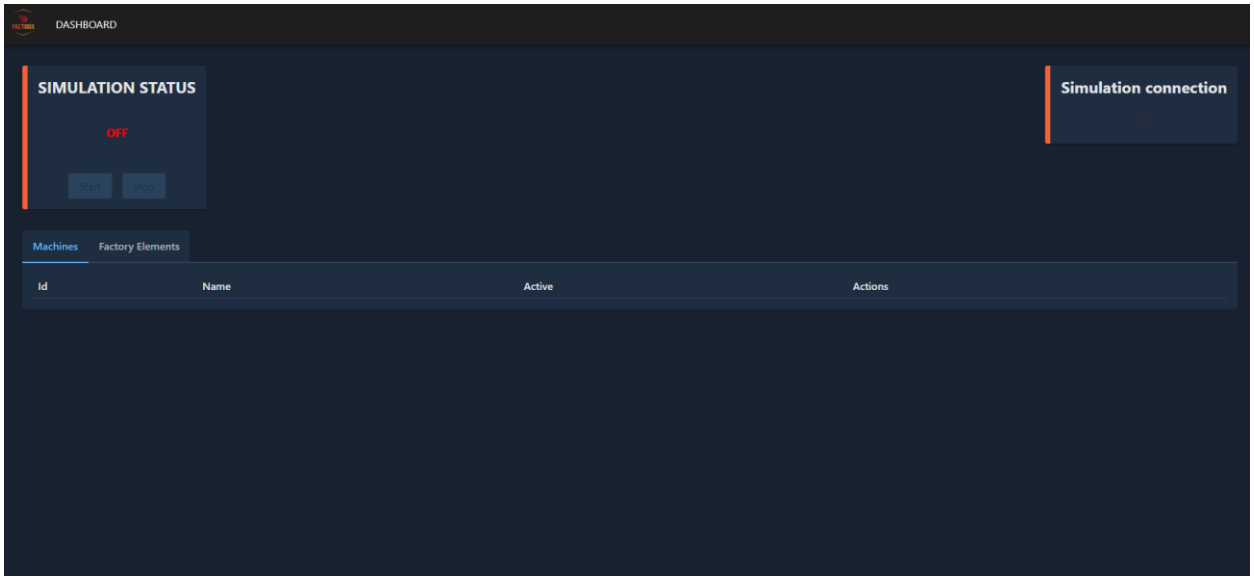
Panel s prikazom informacija kako kontrolirati kretanje i gradnju u simulatoru (Slika 5.46.). Može ga se uvijek prikazati.



Sl. 5.46. Panel s tvorničkim elementima

5.4. Kontrolna aplikacija

Aplikacija izrađena u Angularu omogućuje kontrolu i nadziranje simulacije u stvarnom vremenu. Na slici 5.47. prikaz je kontrolne ploče na kojoj se prikazuje status simulacije, konekcija sa simulatorom i liste tvorničkih elemenata. Lista *Machines* prikazuje postavljene tvorničke elemente kategorizirane kao strojevi, a lista *FactoryElements* prikazuje sve postavljene tvorničke elemente. Aktivna konekcija sa simulatorom bit će naznačena zelenom točkom u prozor *Simulator connection* te će se liste popuniti tvorničkim elementima ako ih ima postavljenih. Prozor *Simulation status* omogućuje pokretanje i zaustavljanje tvorničke simulacije. Također, ako je simulacija aktivna svaki stroj možemo zaustaviti i ponovno pokrenuti.



Sl. 5.47. Prikaz kontrolne aplikacije

Povezivanje s Unity aplikacijom odvija se preko ASP.NET Core aplikacije postavljene na Azure servis. Za komunikaciju u stvarnom vremenu korištena je SignalR biblioteka. Funkcioniranje web aplikacije u stvarnom vremenu je sposobnost servera da odmah prosljeđuje sadržaj povezanim klijentima čim postane dostupan, umjesto da server čeka klijenta da zatraži nove podatke. SignalR pruža jednostavan API za kreiranje serverskih poziva udaljenih procedura (eng. *remote procedure calls*) prema klijentima koji pozivaju funkcije u klijentskih aplikacijama. SignalR također uključuje API za upravljanje konekcijama i grupiranje konekcija. Koristi WebSocket transport gdje je dostupan i vraća se na starije transporte gdje je potrebno.

Na serverskoj strani klasa *FactoryHub* (Slika 5.48.), koja nasljeđuje klasu *Hub* iz biblioteke SignalR, koristi se za komunikaciju između klijenata. Statička varijabla tipa *HashSet* sadrži set svih Unity klijenata koji su povezani. Zbog jednostavnosti u komunikaciji s Unity klijentom metode slanja zahtjeva prema Unity aplikaciji uzimaju samo prvi član *unityConnections* seta čime se smatra da je uvijek samo jedna Unity aplikacija spojena. Metode koje se pozivaju s Angular strane su *SetFactoryActive, SetFactoryElementActive, GetFactoryInfo* i *IsUnityConnected*, dok Unity aplikacija poziva *SendFactoryInfo* metodu. Metoda *SendFactoryInfo* šalje informacije o tvorničkoj simulaciji svim klijentima osim pošiljaocu. Metoda *SetFactoryActive* postavit će status aktivacije tvornice, odnosno pokrenuti ju ili stopirati. Metoda *SetFactoryElementActive* omogućava postavljanje statusa aktivacije određenom tvorničkom elementu. Za dohvaćanje informacija o tvornici poziva se *GetFactoryInfo* metoda. *OnConnectedAsync* metoda se poziva kada se novi klijent poveže. Ako je dolazni zahtjev imao *header* vrijednost *ClientType* postavljen na *Unity* sprema se njegov konekcijski identifikator u *unityConnections* set i poziva metoda UnityConnected na Angular aplikaciji. Kada se klijent od spoji poziva se *OnDisconnectedAsync* gdje se uklanja njegov konekcijski identifikator i poziva metoda UnityDisconneted na Angular aplikaciji. Metodom *IsUnityConnected* provjerava se postoji li bilo koji povezani Unity klijent. Svi podaci koji dolaze ili se šalju su u JSON formatu.

```
public class FactoryHub: Hub
{
    private static HashSet<string> unityConnections = new HashSet<string>();

    public async Task SendFactoryInfo(FactoryInfo data)
    {
        await Clients.Others.SendAsync("ReceiveFactoryInfo", data);
    }

    public async Task SetFactoryActive(bool isActive)
    {
        var connectionId = unityConnections.ElementAt(0);
        await Clients.Client(connectionId).SendAsync("SetFactoryActive", isActive);
    }

    public async Task SetFactoryElementActive(FactoryElementStatus data)
    {
        var connectionId = unityConnections.ElementAt(0);
        await Clients.Client(connectionId).SendAsync("SetFactoryElementActive",
data);
    }

    public async Task<FactoryInfo> GetFactoryInfo()
    {
        try
        {
            var connectionId = unityConnections.ElementAt(0);
            var factoryInfo = await
Clients.Client(connectionId).InvokeAsync<FactoryInfo>("SendFactoryInfo", new
CancellationToken());
            return factoryInfo;
        }
        catch (Exception ex)
        {
            return new FactoryInfo();
        }
    }

    public async override Task OnConnectedAsync()
    {
        var clientType =
Context?.GetHttpContext()?.Request.Headers["ClientType"].ToString();

        if (clientType == "Unity")
        {
            unityConnections.Add(Context.ConnectionId);

            await Clients.Others.SendAsync("UnityConnected");
        }
    }

    public async override Task OnDisconnectedAsync(Exception? exception)
    {
        if (unityConnections.Contains(Context.ConnectionId))
        {
            unityConnections.Remove(Context.ConnectionId);

            await Clients.Others.SendAsync("UnityDisconnected");
        }
     }

    public bool IsUnityConnected()
    {
        return unityConnections.Count > 0;
    }
}
```

Sl. 5.48. Programski kod *FactoryHub* klase

Komunikacija sa serverom na Angular strani započinje u *FactorixService* klasi. Konstruktor (Slika 5.49.) inicijalizira SignalR konekciju pozivajući metodu *buildConnection* te postavlja metode koje će rukovati događajima *ReceiveFactoryInfo, UnityConnected* i *UnityDisconnected*.

```
 constructor(private messageService: MessageService) {
    this.buildConnection();

    this.hubConnection?.on("ReceiveFactoryInfo",
this.handleReceiveFactoryInfo.bind(this))
    this.hubConnection?.on('UnityConnected', async() => {
      console.log('Unity is connected!');
      this.unityConnectedSubject.next(true);
      const factoryInfo = await this.getFactoryInfo();
      if (factoryInfo) {
        this.factoryInfoSubject.next(factoryInfo);
        console.log('Factory Info', factoryInfo);
      }
    });

    this.hubConnection?.on('UnityDisconnected', () => {
      console.log('Unity is disconnected!');

      this.unityConnectedSubject.next(false);
      this.factoryInfoSubject.next(new IFactoryInfo());
    });
  }
```

Sl. 5.49. Programski kod konstruktora klase *FactorixService*

Metoda *buildConnection* (Slika 5.50.) kreira SignalR konekciju koristeći *HubConnectionBuilder* i poziva *startConnection* ako konekcija nije već uspostavljena.

```
public buildConnection(): void {
    if(!this.isConnected)
    {
      this.hubConnection = new HubConnectionBuilder()
        .withUrl(`${environment.apiUrl}factory`)
        .configureLogging(LogLevel.Information)
        .build();

       this.startConnection();
    }
}
```

Sl. 5.50. Programski kod metode *buildConnection*

Metoda *startConnection* (Slika 5.51.) pokušava uspostaviti konekciju sa serverom. Ako uspije, provjerava je li Unity povezan i preuzima informacije o tvornici. U slučaju greške, prikazuje poruku i pokušava ponovno da uspostavi konekciju nakon kratkog vremena.

```
async startConnection(): Promise<void> {
    this.hubConnection?.start()
      .then(async() => {
        this.isConnected = true;
        console.log("Connect to server with signal R");

        const unityConnected = this.hubConnection ? await
this.hubConnection.invoke<boolean>('IsUnityConnected') : false;
        console.log('Unity connected:', unityConnected);

        this.unityConnectedSubject.next(unityConnected);
        if(unityConnected){
          const factoryInfo = await this.getFactoryInfo();
          if (factoryInfo) {
            this.factoryInfoSubject.next(factoryInfo);
            console.log('Factory Info', factoryInfo);
          }
        }


      }, (error) => {
        console.log("Connection to server with signal R refused");
        this.messageService.add({severity: 'error', summary:  'Error', detail:
`${error}` });
      })
      .catch((error) => {
        console.log("Error while establishing signalR connection");
        this.messageService.add({severity: 'error', summary:  'Error', detail:
`${error}` });
        setTimeout(()=> this.startConnection(), 1000);
      })
  }
```

Sl. 5.51. Programski kod metode *startConnection*

Metode *startFactory* i *stopFactory* pozivaju serversku metodu *SetFactoryActive* s parametrom true ili false kojom se simulacija pokreće ili zaustavlja. Metoda *getFactoryInfo* dohvaća informacije o tvornici, sve tvorničke elemente i njihove statuse te status simulacije. Metoda *setFactoryElementActive* za predani ID tvorničkog elementa zaustavlja ili pokreće isti (Slika 5.52.).

```
public async startFactory(): Promise<void> {
    try {
      await this.hubConnection?.invoke("SetFactoryActive", true)
    } catch (error) {
      this.messageService.add({severity: 'error', summary:  'Error', detail:
`${error}` });
    }
  }

  public async stopFactory(): Promise<void> {
    try {
      await this.hubConnection?.invoke("SetFactoryActive", false)
    } catch (error) {
      this.messageService.add({severity: 'error', summary:  'Error', detail:
`${error}` });
    }
  }

  public async getFactoryInfo(): Promise<IFactoryInfo | undefined> {
    try {
      const factoryInfo =  await
this.hubConnection?.invoke<IFactoryInfo>("GetFactoryInfo");

      return factoryInfo;

    } catch (error) {
      this.messageService.add({ severity: 'error', summary: 'Error', detail:
`${error}` });
      return undefined;
    }
  }

  public async setFactoryElementActive(data: IFactoryElementStatus): Promise<void> {
    try {
      await this.hubConnection?.send("SetFactoryElementActive", data);
    } catch(error) {
      this.messageService.add({severity: 'error', summary:  'Error', detail:
`${error}` });
    }
  }
```

Sl. 5.52. Programski kod metoda *startFactory*, *stopFactory*, *getFactoryInfo* i *setFactoryElementActive*

Na Unity strani, *FactorixServerManager* komponenta je zadužena za konekciju i komunikaciju sa serverom preko SignalR-a te općenito kontroliranje ponašanja aplikacije tijekom aktivne konekcije. U *Awake* metodi postavlja se SignalR konekcija preko klase *HubConnectionBuilder*, registriraju se metode za različite događaje sa servera i lokalne događaje kao što su izgradnja i uklanjanje tvorničkih elemenata. Metoda *ConnectToServer* (Slika 5.53.) pokreće SignalR konekciju i podiže događaj *ServerConnected*. Suprotno tome metoda *DisconnectFromServer* (Slika 5.53.) prekida vezu sa serverom.

```
public async Task ConnectToServer()
{
    await _hubConnection.StartAsync();

    EventBus<ServerConnected>.Raise(new ServerConnected());
}
public async Task DisconnectFromServer()
{
    await _hubConnection.StopAsync();
}
```

Sl. 5.53. Programski kod metoda *ConnectToServer* i *DisconnectFromServer*

Metoda *SendFactoryInfo* (5.54.) služi za slanje informacija o tvornici serveru, Kreira DTO (eng. *data transfer object*) s trenutnim stanjem i šalje ga serveru.

```
public async Task SendFactoryInfo()
{
    var factoryModel = new FactoryResponseDto
    {
        IsFactoryOn = _factoryManager.IsFactoryRunning,
        FactoryActuators = _factoryManager.MachineElements.Select(factoryElement =>
        {
            var machine = factoryElement as IMachine;
            return new ActuatorDto { Id = factoryElement.Id, IsActive =
machine.IsActive, Name = factoryElement.Name };
        }).ToList(),
        FactoryElements = _factoryManager.FactoryElements.Select(factoryElement =>
new FactoryElementDto
        {
            Id = factoryElement.Id,
            Name = factoryElement.Name
        }).ToList()
    };

    await _hubConnection.SendAsync("SendFactoryInfo", factoryModel);
}
```

Sl. 5.54. Programski kod metode *SendFactoryInfo*

Metoda *HubConnection_OnSendFactoryInforAsync* (Slika 5.55.) vraća trenutne informacije o tvornici kada server zahtjeva. Kreira DTO s podacima. Kada server pošalje status aktivacije tvorničkog elementa poziva se *HubConnection_OnSetFactoryElementActive* (Slika 5.55.) metoda koja onda ažurira stanje tvorničkog elementa. Metoda *HubConnection_OnSetFactoryActive* (Slika 5.52.) poziva se kada server šalje status aktivacije tvornice. Pokreće ili zaustavlja tvornicu. Ove

dvije metode koriste *UnityMainThreadDispatcher* klasu čime se osigurava da se operacije koje uključuju Unity API izvode na glavnoj niti, s obzirom na to da SignalR povratni pozivi dolaze na pozadinske niti. Ovo je ograničenje samog Unitya, primjerice ažuriranje pozicije GameObject-a možemo učiniti samo na glavnoj niti.

```
private void HubConnection_OnSendFactoryInfoAsync()
{
    var factoryInfo = new FactoryResponseDto
    {
        IsFactoryOn = _factoryManager.IsFactoryRunning,
        FactoryActuators = _factoryManager.MachineElements.Select(factoryElement =>
        {
            var machine = factoryElement as IMachine;
            return new ActuatorDto { Id = factoryElement.Id, IsActive =
machine.IsActive, Name = factoryElement.Name };
        }).ToList(),
        FactoryElements = _factoryManager.FactoryElements.Select(factoryElement =>
new FactoryElementDto
        {
            Id = factoryElement.Id,
            Name = factoryElement.Name
        }).ToList()
    };

    return factoryInfo;
}

private void HubConnection_OnSetFactoryElementActive(FactoryElementStatus status)
{
    UnityMainThreadDispatcher.Instance().Enqueue(async () =>
    {
        var factoryElement = _factoryManager.MachineElements.Find(x => x.Id ==
status.Id);
        var machine = factoryElement as IMachine;

        if(status.IsActive)
            machine.StartMachine();
        else
            machine.StopMachine();

        await SendFactoryInfo();
    });
}

private void HubConnection_OnSetFactoryActive(bool isActive)
{
    UnityMainThreadDispatcher.Instance().Enqueue(() =>
    {
        if (isActive)
        {
            _factoryManager.StartFactory();
        }
        else
        {
            _factoryManager.StopFactory();
        }

    });
}
```

Sl. 5.55. Programski kod metoda za odgovor na serverske pozive

Metode *FactoryManager_FactoryElementRemoved* i *FactoryManager_FactoryElementBuilt* reagiraju na događaje kada se tvornički elementi uklone ili izgrade (Slika 5.56.). Ako je veza sa serverom aktivna, pozivaju *SendFactoryInfo* da pošalju ažurirane informacije o tvorničkom okruženju.

```
private async void FactoryElementRemoved(object sender, BaseFactoryElement e)
{
    if (_hubConnection.State == HubConnectionState.Connected)
    {
        await SendFactoryInfo();
    }
}

private async void FactoryElementBuilt(object sender, BaseFactoryElement e)
{
    if(_hubConnection.State == HubConnectionState.Connected)
    {
        await SendFactoryInfo();
    }
}
```

Sl. 5.56. Programski kod metoda *FactoryElementRemoved* i *FactoryElementBuilt*

6. ZAKLJUČAK

Korištenje tvorničkih simulacija uključuje korištenje sofisticiranih računalnih modela za prikaz proizvodnog pogona, ali u potpuno virtualnom prostoru. Proizvođači mogu vizualizirati različite aspekte proizvodnje, od svojih proizvodnih linija, njihove automatizacije i opreme do radnika u akciji. Na ovaj način štedi se vrijeme i novac jer se testni poligoni mogu izvesti simulacijom čime se smanjuje potencijalna šteta pri stvarnom testiranju. Razvojem 3D softvera proces izgradnje ovakvih simulacija je olakšan.

Simulacija tvorničkog okruženja kreirana u sklopu projekta ovog diplomskog rada, iako nije kompleksna kao suvremena rješenja, pruža uvid u mogućnosti koje donosi ovakav tip simulacije. Izrađena je u Unity softverskom alatu koji se pokazao dobrim rješenjem za ovakav tip aplikacije zbog svojih grafičkih sposobnosti, kompleksnog sustava fizike, intuitivnog programiranja i jednostavnog povezivanja komponenti u cjelinu.

Za izgradnju tvorničkog okruženja kreiran je sustav gradnje koji koristi mrežu za pozicioniranje i postavljanje tvorničkih elemenata. Simulacija nudi mogućnost kreiranja proizvoljnih linija pokretnih traka te transport kutija na njima. Moguće je i kreirati malo automatizirano postrojenje za pakiranje kutija s ponuđenim strojem za tu namjenu. Kontrolnom web aplikacijom napravljenom u Angularu moguće je pokrenuti i stopirati tvorničku simulaciju i pojedine tvorničke elemente.

Tijekom razvoja vodilo se računa o budućim unaprjeđenjima. Sustav gradnje omogućuje jednostavno dodavanje novih objekata te je neovisan o tipu, funkcionalnost tvorničkih elemenata je zaseban dio. Za unaprjeđenje funkcionalnosti tvorničke simulacije trebalo bi dodati nove tvorničke elemente. Primjerice različiti senzori, okretni stolovi, robotske ruke, različiti strojevi itd. Zatim omogućiti veću razinu prilagođavanja pojedinih tvorničkih elemenata i uvjeta izvođenja simulacije. Kontrolna aplikacija bi se mogla proširiti dodatnom statistikom, većom razinom upravljanja te implementacijom sustava automatizacije tvorničke simulacije korištenjem PLC-a.

LITERATURA

[1] Factory I/O About, RealGames, dostupno na: https://docs.factoryio.com/ (pristupljeno lipanj 2024.)

[2] Introducing Visual Components 4.4, Visual Components, dostupno na: https://www.visualcomponents.com/blog/introducing-visual-components-4-4/ (pristupljeno lipanj 2024.)

[3] Fastsuite Edition 2, Cenit Ag, dostupno na: https://www.cenit.com/en_EN/solutions-services/software-solutions/fastsuite-edition-2.html (pristupljeno lipanj 2024.)

[4] F. D. Petruzella, Programmable Logic Controllers, MyGraw-Hill Education, New York, 2017.

[5] M. Rabiee, Programmable Logic Controller Hardware and Programming, The Goodheart-Willcox Company, United States, 2018.

[6] Explore the Unity Editor, Unity Technologies, dostupno na: https://learn.unity.com/tutorial/explore-the-unity-editor-1# (pristupljeno lipanj 2024.)

[7] GameObjects, Unity Technologies, dostupno na: https://docs.unity3d.com/Manual/GameObjects.html (pristupljeno lipanj 2024.)

[8] Important Classes, Unity Technologies, dostupno na: https://docs.unity3d.com/Manual/ScriptingImportantClasses.html (pristupljeno lipanj 2024.)

[9] MonoBehaviour, Unity Technologies , dostupno na: https://docs.unity3d.com/Manual/class-MonoBehaviour.html (pristupljeno lipanj 2024.)

[10] Event Functions, Unity Technologies, dostupno na: https://docs.unity3d.com/Manual/EventFunctions.html (pristupljeno lipanj 2024.)

[11] Blender, Blender Foundation, dostupno na: https://www.blender.org/about/ (pristupljeno lipanj 2024.)

[12] A. Troelsen i Phil Japikse, Pro C# 10 with .NET 6, Apress Media, New York, 2022.

[13] What is Angular?, Google, dostupno na: https://angular.dev/overview (pristupljeno lipanj 2024.)

SAŽETAK

Simulacije tvorničkih postrojena omogućuju testiranje automatizacije tvorničkih pogona u virtualnom svijetu. Ove simulacije pružaju korisnicima priliku da identificiraju i otklone potencijalne probleme, optimiziraju procese te ispitaju različite scenarije bez potrebe za trošenjem resursa ili prekidanjem rada postrojenja. Tvornička simulacija u ovom projektu izrađena je u Unity razvojnom okruženju koji se koristi za izradu simulacija i razvoj 2D i 3D video igara. Opisana je izrada projekta i funkcioniranje simulacije. Korisniku je na raspolaganju baza tvorničkih elemenata koja se sastoji od pokretnih traka, kutija i stroja za pakiranje kutija. Kroz intuitivni sustav gradnje ovi elementi mogu se proizvoljno slagati u tvorničke pogone. Za upravljanje simulacijom, osim kroz korisničko sučelje simulacije, izrađena je i web aplikacija u Angularu.

Ključne riječi: simulacija, GameObject, PLC, Unity, komponenta

ABSTRACT

Factory simulations enable testing of production plant automation in a virtual environment. These simulations allow users to identify and resolve potential issues, optimize processes, and test different scenarios without spending resources or disrupting actual factory operations. The factory simulation in this project was created using the Unity game engine, which is used for creating simulations and developing 2D and 3D video games. The creation of the project and the functioning of the simulation are described. The user has access to a database of factory elements consisting of conveyor belts, boxes, and a box packaging machine. These elements can be freely arranged through an intuitive building system into factory setups. In addition to the simulation user interface, a web application built with Angular was also developed to manage the simulation.

Keywords: simulation, GameObject, PLC, Unity, component

ŽIVOTOPIS

David Hodak rođen 20. travnja 1997. godine u Puli, s prebivalištem u Rokovcima. Godine 2016. upisuje Fakultet elektrotehnike, računarstva i informacijskih tehnologija Osijek. Završava preddiplomski sveučilišni studij računarstva te upisuje diplomski sveučilišni studij računarstva, smjer informacijske i podatkovne znanosti. U periodu od listopada 2021. godine do listopada 2023. godine zaposlen kao programer videoigara u Orqa d.o.o. Nakon toga u listopadu 2023. godine zapošljava se kao programski inženjer u GDi GROUP LL d.o.o.

_____

Potpis autora