SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 6240

Simulacija obrnutog njihala

Dana Dodigović

Zagreb, lipanj 2019.

Za Rudolfa.

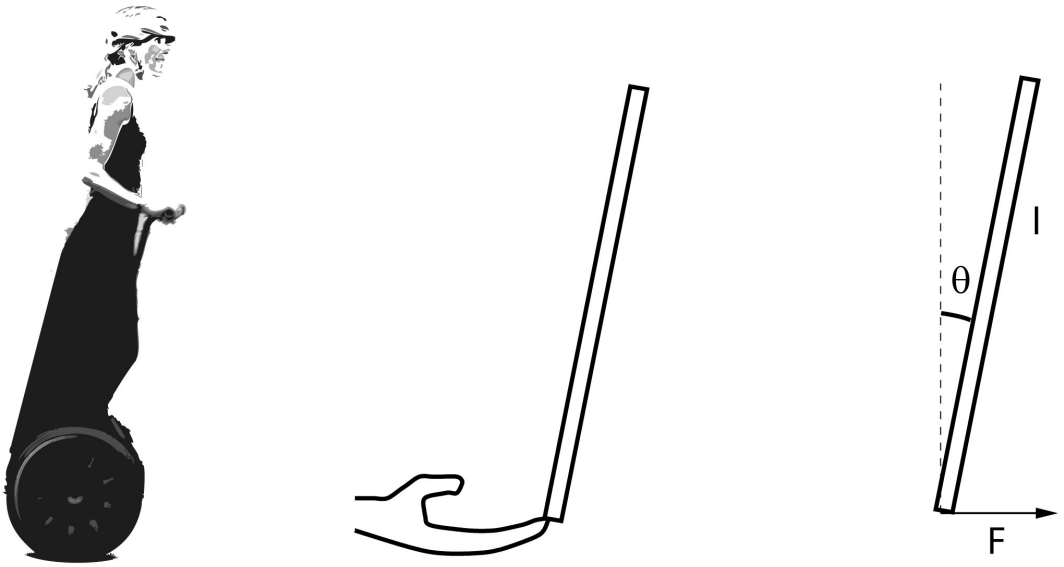
Sadržaj

1. Uvod

Tipičan primjer okrenutog njihala s kojim se većina ljudi susrela u svojem djetinjstvu (naravno, bez da su bili svijesni činjenice da je to često korišten fizikalni model koji opisuje vrlo zanimljiva ponašanja) je održavanje štapa na dlanu pomicanjem ruke u različitom smjeru i različitim intenzitetom s ciljem zadržavanja štapa u što mirnijem položaju što je moguće dulje vremena (slika 1.1).

Sustav okrenutog njihala ima mnogo upotreba u znanosti te je stabilizacija njihala klasičan problem u dinamici i teoriji upravljanja. Takav sustav dobro opisuje i kompleksnije sustave u području robotike i mehatronike [1] te je kao takav, osnovni i jedan od najčešćih načina za ispitivanje i prikaz upravljačkih strategija.

Cilj ovog rada je implementirati dvodimenzionalan grafički prikaz i kretanje okrenutog njihala pričvršćenog na kolica na koja se djeluje vanjskom silom fiksnog iznosa, ali varijabilnog smjera. Ideja je održati sustav što bliže ravnotežnoj točki, a u najgorem slučaju, barem osigurati da šipka njihala ne opadne na kolica što dulje vremena.



Slika 1.1: Primjeri okrenutog njihala.

2. Sustav okrenutog njihala

Okrenuto njihalo je popularan način demonstracije upravljanja korištenjem upravljačkog sklopa koji određuje vanjsku silu na temelju koje se može odrediti sljedeće stanje sustava u svakom vremenskom trenutku. Slika 2.1 prikazuje fizičku implementaciju ovakvog sustava.

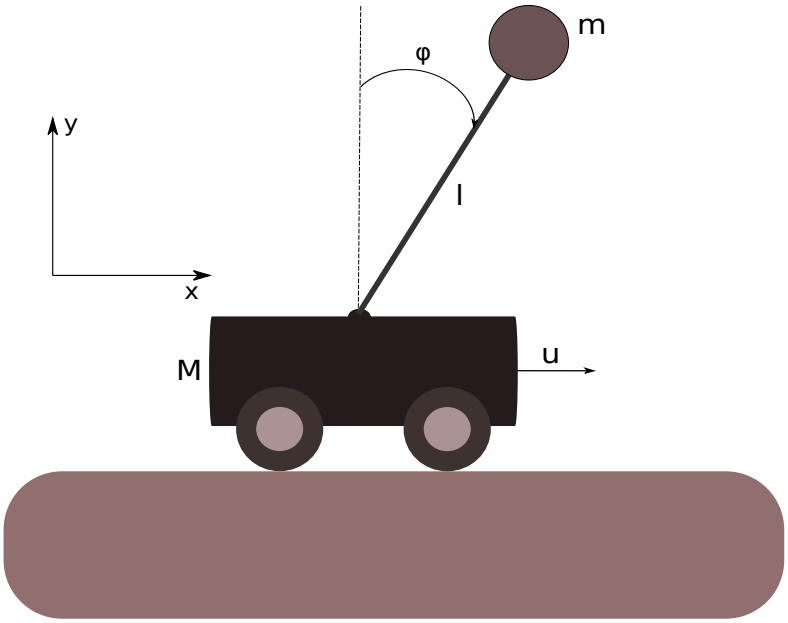
Slika 2.1: Primjer fizičkog okrenutog njihala.

Njihalo je nelinearan sustav čije se ponašanje može opisati s dvije diferencijalne jednadžbe drugog reda [2] (formula 2.1 i 2.2).

$$(M+m)\ddot{x} - mL\sin(\phi)\dot{\phi}^2 + mL\cos(\phi)\ddot{\phi} = u \tag{2.1}$$

$$m\ddot{x}\cos(\phi) + mL\ddot{\phi} = mg\sin(\phi) \tag{2.2}$$

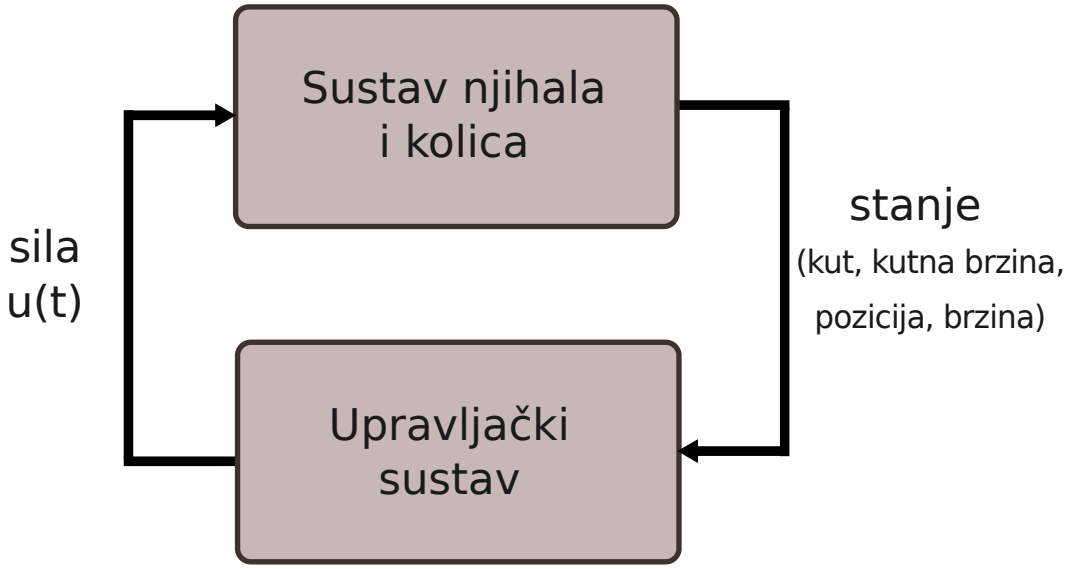
Slika 2.2 prikazuje sustav opisan u nastavku. Sustav se sastoji od pomičnih kolica mase M koja se mogu kretati samo duž x-osi. Na kolica je montirana šipka duljine L na čijem se vrhu nalazi kugla mase m. Šipka se nalazi na zglobu koji je fiksiran tako da je omogućeno kretanje samo u lijevu ili desnu stranu.



Slika 2.2: Sustav okrenutog njihala i kolica

Početno se sustav nalazi na poziciji za x_{cart} udaljen od ishodišta koordinatnog sustava, a šipka s y-osi zatvara kut ϕ. U svakom vremenskom trenutku na sustav djeluje horizontalna vanjska sila $u(t)$. Sve sile koje djeluju na tijelo su u svakom trenutku određene u samo jednoj ravnini (xy ravnina) - promatrani model je dvodimenzionalan. Kako bi sustav bio opisan jednadžbama gibanja, na dalje će vrijediti pretpostavka da su kolica i kugla aproksimirani točkastim masama, a da je masa šipke zanemariva.

Pod ovakvim pretpostavkama, ako se na trenutak zanemari djelovanje vanjske sile, pri pozitivnom početnom kutu (štap je nagnut prema desno), štap će početi rotirati sve više u desnu stranu, dok će smjer gibanja kolica biti suprotan.



Slika 2.3: Sustav okrenutog njihala i kolica

Slika 2.3 prikazuje petlju između upravljačkog sustava i njihala.

Navedeni sustav je moguće okarakterizirati na još jedan način. Stanje sustava je opisano s četiri podataka: $(x_{cart}, v_{cart}, \phi, \omega)$.

Pozicija x_{cart} predstavlja udaljenost kolica od ishodišta koordinatnog sustava po x-osi izraženu u metrima. Brzina v_{cart} označava brzinu kolica u m/s. Kut ϕ je kut u radijanima koji šipka zatvara s y-osi (za $\phi = 0$, šipka je u okomitom položaju), a kutna brzina ω u rad/s označava koliko se brzo njihalo rotira u odnosu na zglob kojim je pričvršćeno za kolica (pozitivan iznos kutne brzine znači povećanje kuta, dok negativan označava smanjenje).

Upravljački sustav u svakom vremenskom trenutku može primiti informacije o stanju sustava i na temelju toga generirati vanjsku silu kojom će djelovati na kolica. Na temelju dobivene sile i prethodnog stanja, sustav računa novo stanje. Ovakav postupak se ponavlja dokle god postoje promjene u sustavu pa se zbog toga i interakcija upravljačkog sustava i njihala predočava zatvorenom petljom.

3. Korištene tehnologije i alati

Program je implementiran u programskom jeziku C++ korištenjem programskog sučelja OpenGL trenutno aktualne inačice 4.6. Za sjenčanje na grafičkoj kartici korišten je jezik visoke razine apstrakcije GLSL (OpenGL Shading Language). GLSL je jezik sličan C-u koji pruža više izravne kontrole nad grafičkim cjevovodom bez potrebe za korištenjem asemblera.

Kako bi bilo omogućeno jednostavno korištenje modernih OpenGL funkcija koje su implementirane u grafičkim upravljačkim programima (engl. *driver*) u projekt je uključena i knjižnica samo s datotekama zaglavlja GLEW (OpenGL Extension Wrangler Library) bez koje bi bilo potrebno ručno provjeravati koja je verzija OpenGL-a podržana na pojedinom hardveru, koje su funkcije dostupne, te za svaku funkciju koju će biti korištena ručno inicijalizirati pokazivač. GLEW ovaj cijeli postupak pojednostavljuje na način da se uključi jedno zaglavlje te pozove jedna funkcija tijekom inicijalizacije prikazana u isječku koda 3.1.

Za jednostavnije stvaranje i manipulaciju prozorom, stvaranje konteksta, te rukovanje ulaznim uređajima (poput tipkovnice i miša) korištena je knjižnica GLFW 3.3. (Graphics Library Framework) koja za razliku od starije alternative GLUT omogućava veću kontrolu rada. GLFW ostavlja programeru da implementira vlastitu glavnu petlju (engl. *event loop*) što omogućava mnogo precizniji rad s vremenom i samim time i manja kašnjenja.

Velika prednost navedenih knjižnica je to što su višeplatformske knjižnice otvorenog koda.

Rad s vektorima olakšan je uporabom matematičke knjižnice GLM za C++ čija je sintaksa vrlo slična sintaksi GLSL-a. GLM je posebno prikladan za uporabu s OpenGL-om zato što koriste isti način pohranjivanja matrica u memoriju, odnosno pohranjuju matrice kao vektor stupce. To je pogodno iz dva razloga, glavni razlog su performanse, a drugi je praktičnost jer programer ne treba razmišljati je li unio matricu u formatu koji odgovara OpenGL-u kao što bi to bio slučaj kada bi matrice bile pohranjene u različitim formatima.

Sustav okrenutog njihala je opisan s dvije nelinearne diferencijalne jednadžbe, pa je u svrhu numeričkog rješavanja korištena knjižnica ODEINT [3].

```
if (glewInit() != GLEW_OK) {
   throw std::runtime_error("Failed to initialize GLEW.");
}
```

Isječak 3.1: Inicijalizacija modernog OpenGL-a.

3.1. Korištenje programa

U programu je prvo potrebno podesiti parametre sustava te željeni način upravljanja (tipkovnica ili gradijentni spust). U slučaju da se program želi pokrenuti bez dodatnih postavljanja, koriste se već ranije podešene postavke uz upravljanje tipkovnicom.

3.2. OpenGL

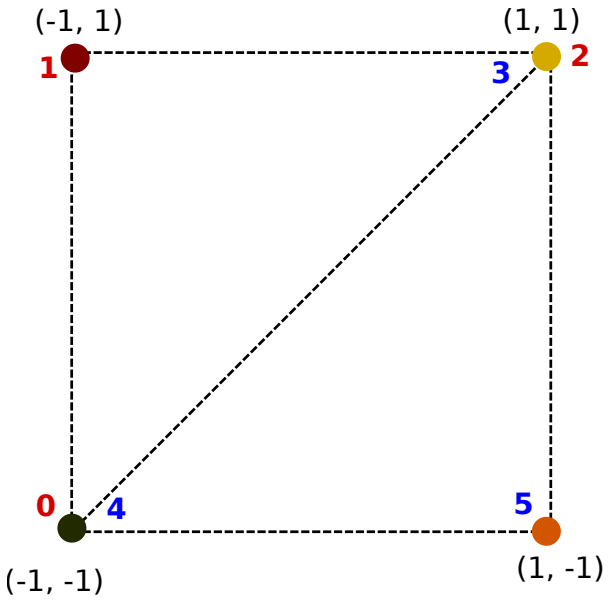
OpenGL (Open Graphics Library) je višeplatformska specifikacija koja je podržana u velikom broju programskih jezika, a služi za prikaz 2D i 3D scena [4]. Iako bi bilo moguće implementirati sve funkcionalnosti programski, danas su gotovo sve implementirane u hardveru kako bi se rasteretio procesor i kako bi se iskoristila paralelna moć grafičkih kartica. U nastavku ovog poglavlja će biti opisani neki osnovni koncepti s kojima je potrebno biti upoznat u svrhu izrade scena u modernom OpenGL-u inačice 3.3 na više.

3.2.1. Vertex Buffer Object (VBO)

`Vertex Buffer Object` je memorijski međuspremnik u kojem su spremljeni atributi poput koordinata točaka, vektora normala, boje i koordinata tekstura. Atributi su spremljeni u radnoj video memoriji i za razliku od OpenGL-a inačice 1.x koji iscrtava točke između poziva funkcija `glBegin()` i `glEnd()` neposredno nakon svakog slanja točke na grafičku karticu, `VBO` omogućava iscrtavanje cijelog spremnika nakon poziva odgovarajuće OpenGL funkcije za crtanje. Prednost `VBO`-a u odnosu na stariju alternativu je to što se podaci u memoriji čuvaju na način koji je optimalniji za grafičku karticu.

3.2.2. Index Buffer Object (IBO)

`Index Buffer Object` omogućava ponovno korištenje postojećih definicija vrhova. Primjerice, ako neki program treba iscrtati kvadrat koji se sastoji od dva trokuta, bez `IBO`-a je potrebno `VBO`-u proslijediti šest vrhova iako je dovoljno samo četiri kako bi kvadrat bio u potpunosti definiran. Indeks zapravo predstavlja redni broj pozicije vrha iz `VBO`-a koju je potrebno nacrtati. Indeksi i pozicije ovog primjera prikazani su na slici 3.1. Crvenom bojom su prikazani indeksi prvog trokuta, a plavom drugog gdje su pozicije označene točkama različitih boja.



Slika 3.1: Prikaz indeksa i koordinata točaka.

3.2.3. Vertex Array Object (VAO)

`Vertex Array Object` je poseban tip objekta koji enkapsulira sve podatke koji su povezani s procesiranjem vrhova. Umjesto da sadrži stvarne podatke, `VAO` sadrži reference na jedan ili više `VBO`-a i točno jedan `IBO` i raspored svakog vrha (u smislu redoslijeda pojavljivanja atributa, primjerice: u ovom `VBO`-u se nalazi tri broja s pomičnim zarezom koji definiraju poziciju i tri broja koja definiraju boju). Odnosno, `VAO` omogućava da se `VBO`-u jednom dodijeli njegov raspored atributa i zatim korisnik pri korištenju `VBO`-a više ne treba svaki put ponovno dodjeljivati taj isti raspored (isječak koda 3.2).

```
glVertexAttribPointer(i,
                      element.count,
                      element.type,
                      element.normalized ? GL_TRUE : GL_FALSE,
                      layout.get_stride(),
                      reinterpret_cast<const void*>(offset));
```

Isječak 3.2: Dodjela rasporeda atributa VAO-u.

3.2.4. Sjenčar vrhova

Sjenčar vrhova je program koji se izvršava na grafičkoj kartici za svaki vrh definiran `VBO`-om i pripadnim `IBO`-om. Njegova svrha je transformirati svaku 3D poziciju vrha u 2D prostor koji se prikazuje na ekranu i ovisno o načinu transformacije uzeti u obzir Z-spremnik. To se najčešće radi uz pomoć *model-pogled-projekcija* (`MVP`) matrice koja

je definirana preko *uniforma*. *Uniform* je globalna varijabla koju dijele svi vrhovi.

Sjenčar vrhova može manipulirati i bojom, koordinatama teksture i normalama i zbog toga pruža veliku kontrolu nad detaljima osvjetljenja, kretanja i boje u sceni.

```
#version 460 core

layout(location = 0) in vec4 position;
uniform mat4 MVP;

void main()
{
   gl_Position = MVP * position;
}
```

Isječak 3.3: Primjer jednostavnog sjenčara vrhova.

3.2.5. Sjenčar fragmenata

Sjenčar fragmenata je program koji se izvršava na grafičkoj kartici za svaki slikovni element u sceni. Kao ulazni podatak prima jedan slikovni element i promijenjeni element vraća na izlaz. Takav program računa boju i ostale atribute. Sjenčar fragmenata nije obavezan dio grafičkog cjevovoda i ako nije korišten boja elementa poprima nedefiniranu vrijednost.

```
#version 460 core

layout(location = 0) out vec3 out_color;
uniform vec3 color;

void main()
{
   out_color = color;
}
```

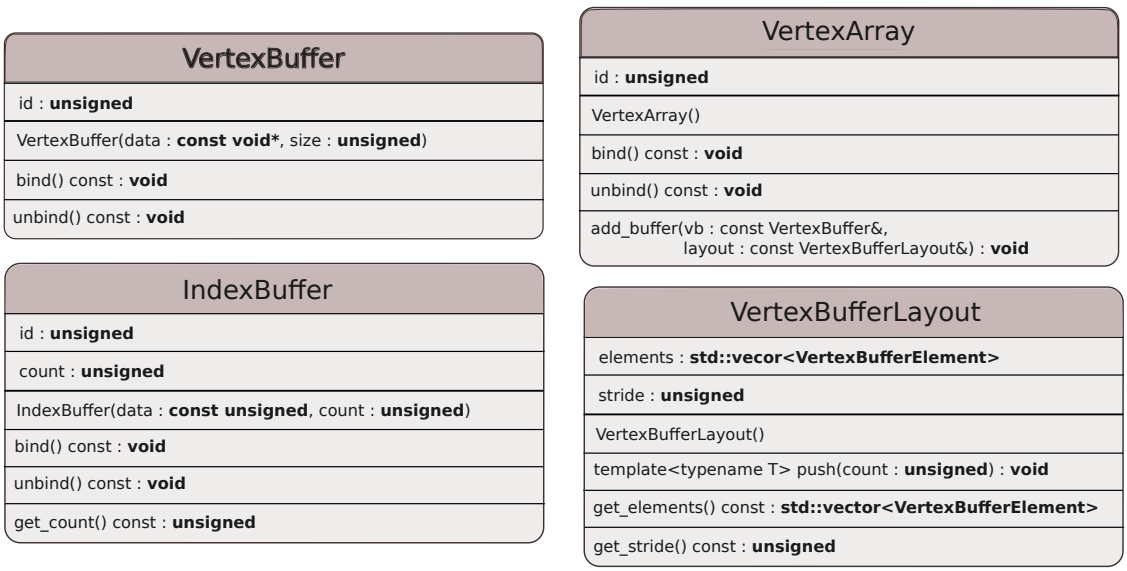
Isječak 3.4: Primjer jednostavnog sjenčara fragmenata.

4. Implementacija

U poglavlju 4.1 je opisana apstrakcija funkcija OpenGL-a. Cilj apstrakcije je napisati pregledan i proširiv kod koji će osigurati minimalno razmišljanje o objektima OpenGL-a jednom kada su apstrahirani u pripadajuće razrede. Nakon toga, u poglavlju 4.2 će biti opisani sustav okrenutog njihala iz implementacijske perspektive, dok će ostala poglavlja poduprti još nekoliko važnih funkcionalnosti vezanih uz rad i svojstva sustava i upravljačkog sklopa. Naglasak ovog dijela je upravo na grafičkom modeliranju navedenog sustava što će biti objašnjeno u poglavlju 4.4.

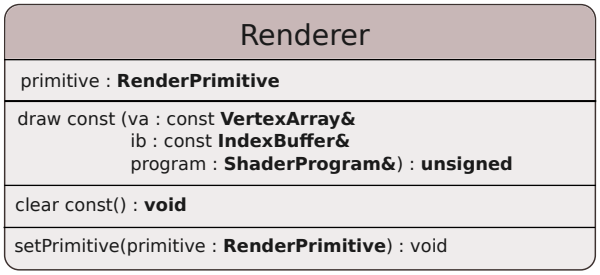
4.1. Apstrakcija OpenGL-a

Pri izradi bilo kakve grafičke aplikacije čija je scena iole složenija od jednog trokuta, kôd glavne funkcije postaje izrazito nepregledan i težak za razumijevanje. Također, pri prikazu jedne scene igrice s tisuću objekata, bilo bi potrebno za svaki pojedini objekt ponovno pisati sve inicijalizirajuće dijelove kôda što bi rezultiralo višestrukim i nepotrebnim ponavljanjima. Stoga se svaki objekt iz OpenGL-a u duhu objektno-orijentirane paradigme enkapsulira u zaseban razred. Cilj apstrakcije je od programera sakriti sve pozive funkcija iz knjižnica GLEW i GLFW kako bi se rad nastavio na višoj razini.



Slika 4.1: Apstrakcija objekata OpenGL-a.

Dijagram 4.1 prikazuje apstrakciju razreda opisanih u poglavlju 3.2 uz dodatak razreda `VertexBufferLayout` koji u sebi pamti tip objekata pohranjenih u `VBO`-u. Primjerice, ako svaki vrh iz `VBO`-a u sebi sadrži tri broja s pomičnim zarezom koja predstavljaju pozicije, tada će `VertexBufferLayout` pohraniti tri broja s pomičnim zarezom. Svaki objekt u sebi sadrži i metode `bind()` i `unbind()` zato što se OpenGL ponaša kao stroj stanja i tek kada mu se kaže da koristi te objekte (pozivom pripadne metode), oni postaju aktivni.



Slika 4.2: Apstrakcija objekta zaduženog za crtanje.

Na dijagramu 4.2 je prikazana apstrakcija objekta koji služi za crtanje svih objekata u sceni. Pri svakom pozivu taj objekt mora primiti reference na `VertexArray`, `IndexBuffer` i `ShaderProgram` kako bi znao što treba iscrtavati, kojim redoslijedom i koji sjenčar treba primjeniti na podatke. `RenderPrimitive` predstavlja enumeraciju koja u sebi sadrži OpenGL primitive za iscrtavanje.

Osim ovih razreda, apstahirani su također i prozor (`Window`) te program za sjenčanje (`ShaderProgram`). Prozor je zadužen za zamjenu međuspremnika, provjeru je li još uvijek otvoren, postavljanje povratnih funkcija za ulazne uređaje i prihvat događaja.

Program za sjenčanje postavlja i prevodi sjenčare zapisane u datotekama, te ima zadaću postavljati uniforme.

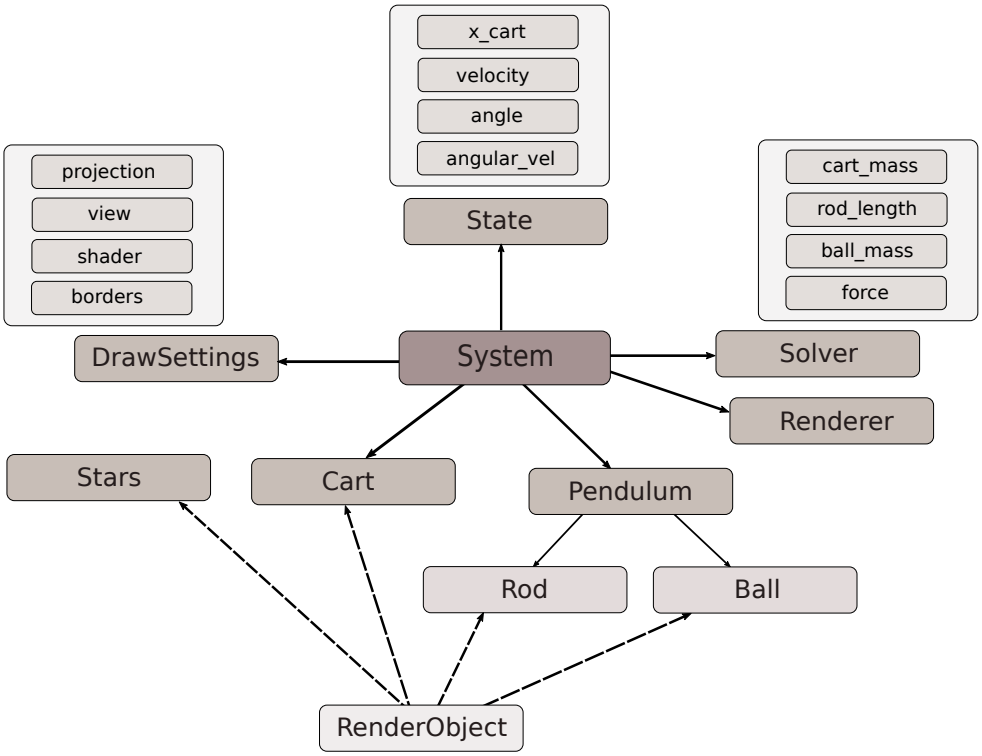
4.2. Opis sustava okrenutog njihala i kolica

Sustav okrenutog njihala i kolica oblikovan je kroz nekoliko razreda koji će biti objašnjeni u nastavku ovog poglavlja, a prikazani su na dijagramu 4.3.

Prvi od razreda je sam sustav (`System`). Sustav je zadužen za crtanje svih elemenata scene, pa tako u sebi sadrži objekt za iscrtavanje (`Renderer`). Sustav iscrtava kolica (`Cart`), njihalo (`Pendulum`), te pozadinske zvijezde (`Stars`) koje stvaraju vizualan učinak kretanja. Svi elementi koji se iscrtavaju nasljeđuju razred `RenderObject` (na dijagramu 4.3 prikazano iscrtkanom linijom).

Druga zadaća sustava je da na temelju trenutnog stanja koje je određeno četvorkom *(pozicija kolica, brzina kolica, kut njihala, kutna brzina njihala)* izračuna novo stanje

sustava pa tako sadrži i objekt za rješavanje diferencijalnih jednadžbi (`Solver`). Ako se nova pozicija kolica nalazi blizu ruba prozora, kamera se pomiče za određeni broj slikovnih elemenata. Unutar sustava je također definiran maksimalan kut između njihala i y-osi. Ako je postignut takav kut, grafički prikaz se gasi uz poruku o završetku prikaza u naredbenom retku.



Slika 4.3: Sustav okrenutog njihala s pripadnim razredima.

4.3. Učitavanje parametara sustava

Parametri sustava učitani su uporabom knjižnice za čitanje datoteka ekstenzije ”.ini”. INIreader (reff) je jednostavna knjižnica za parsiranje napisana u C-u koja sadrži samo jednu datoteku zaglavlja.

Korišteni parametri sustava navedeni su u tablici 4.1.

Tablica 4.1: Parametri zadani u .ini datoteci

m	Masa kugle (kg).
l	Duljina štapa (m).
ϕ	Kut između štapa i y-osi (rad).
M	Masa kolica (kg).
$u(t)$	Sila na kolica (N).
control	Način upravljanja (tipkovnica / grad. spust).

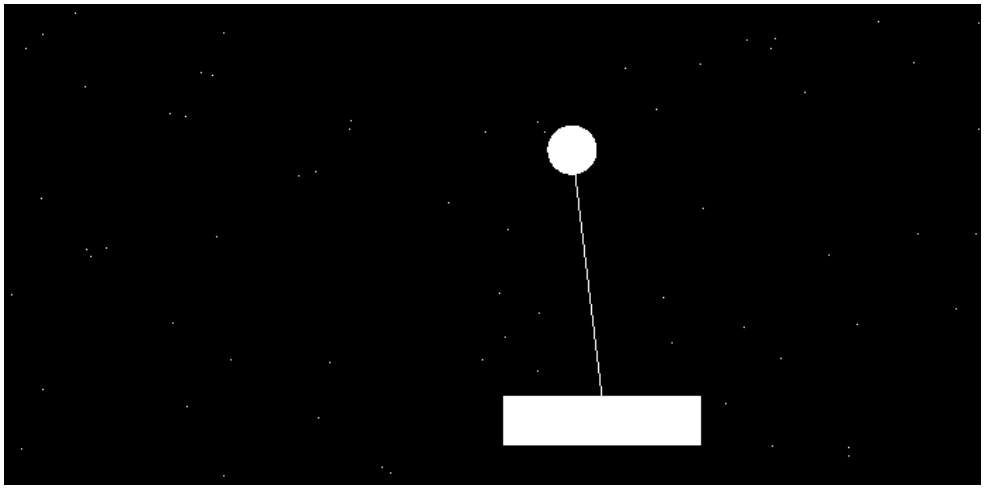
4.4. Grafički prikaz okrenutog njihala

Njihalo se sastoji od komponenata navedenih u poglavlju 2. U sklopu ovog sustava, za komponente nije korišten gotovi model, nego su prikazane nizom OpenGL primitiva. Svaka komponenta scene nasljeđuje `RenderObject` koji u sebi sadrži funkciju za crtanje čija je implementacija napisana niže u tekstu pod isječkom koda 4.1. Svaki objekt za iscrtavanje treba imati definiran `VAO`, `IBO` i `shader` kako bi se mogao izvršiti OpenGL poziv funkcije `glDrawElements(gl_primitive, ib.count(), GL_UNSIGNED_INT, nullptr)`.

```
void RenderObject::draw(Renderer& renderer)
{
   program.bind();
   program.set_uniform_mat4f("MVP", proj * view * model);
   renderer.set_primitive(render_primitive);
   renderer.draw(va, *ib, program);
}
```

Isječak 4.1: Kod za crtanje svih objekata u sceni.

Slika 4.4 prikazuje okrenuto njihalo tijekom upravljanja. Njihalo se pokretanjem programa nalazi pod proizvoljnim kutom te se prema tome treba prilagoditi sila koja će djelovati na kolica kako ono nebi palo na kolica.



Slika 4.4: Grafički prikaz sustava okrenutog njihala

Kolica i kugla su opisani nizom trokuta, a štap je prikazan jednostavnom linijom. Isječak koda 4.2 služi za izračun i pohranjivanje koordinata kugle i napisan je u nastavku. Izračunate koordinate se iscrtavaju OpenGL primitivom `GL_TRIANGLE_FAN` kako bi se spajanjem svaka dva vrha trokuta spojila s centrom kružnice čime se zapravo s nizom trokuta crta kružnica.

```
std::vector<glm::vec2> pos;
pos.reserve(n_segments + 1);
pos.push_back(center);

for (int i = 0; i < n_segments; i++) {
   float alpha = 2.0f * 3.1415926f
            * static_cast<float>(i)
            / n_segments;
   float x = center.x + radius * std::cos(alpha);
   float y = center.y + radius * std::sin(alpha);
   pos.emplace_back(x, y);
}
```

Isječak 4.2: Primjer zadavanja koordinata za kuglu.

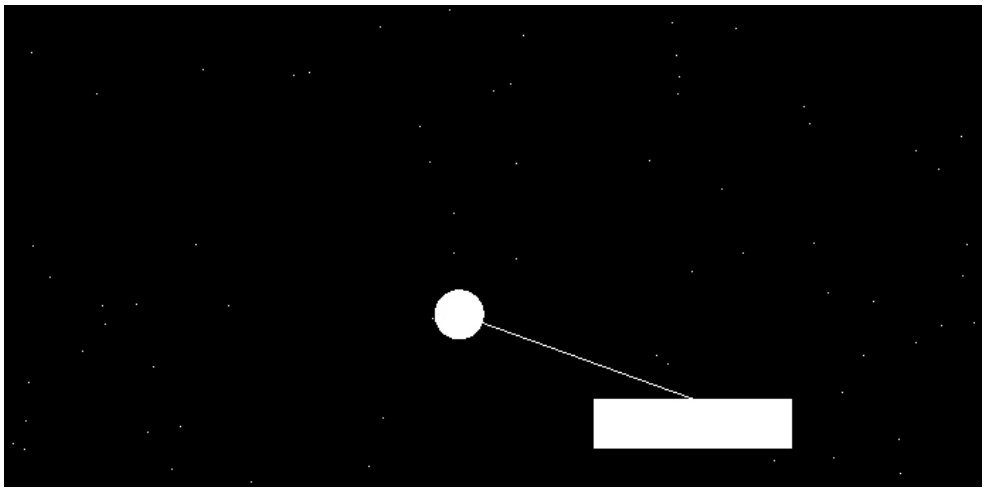
Svi elementi scene za iscrtavanje dijele matricu projekcije i pogleda i program za sjenčanje pa se unutar sustava koji sadrži te elemente definira struktura podataka koja čuva te informacije. Ovakva struktura je definirana u isječku koda 4.3 i ona sadrži strukturu `ViewBorders` koja definira trenutno vidno polje kamere (lijevi i desni rub) koje se ažurira prilikom prelaska kolica izvan tog polja kako bi se prema tim granicama mogla pomaknuti i kamera. Kamera je na početku programa smještena u ishodište koordinatnog sustava (sredina ekrana). Funkcija `glm::ortho(...)` definira parametre za paralelnu projekciju čime se koordinate x-osi ostavljaju u rasponu $[-1, 1]$, a koordinate y-osi skaliraju na raspon $[-0.2, 0.8]$.

```
struct DrawSettings {
   glm::mat4 proj = glm::ortho(-1.0f, 1.0f,
                        -0.2f, 0.8f,
                        -1.0f, 1.0f);
   glm::mat4 view = glm::mat4(1.0f);
   ShaderProgram program = ShaderProgram(
      "res/shaders/VertexShader.glsl",
      "res/shaders/FragmentShader.glsl"
   );
   ViewBorders borders;
};
```

Isječak 4.3: Struktura podataka koja sadrži podatke koje dijele svi elementi u sceni.

Pomicanje kamere prema lijevo i desno omogućuje prikaz kolica izvan početnog raspona koordinata i prikazano je u dodatku B.1.

Pri promjeni stanja sustava, komponente se translatiraju i rotiraju za određeni iznos u odnosu na početnu poziciju (slika 4.5 prikazuje veliki pomak kuta u odnosu na okomicu) zadanu parametrima opisanim u poglavlju 4.3 što je prikazano isječkom koda 4.4. Potrebno je primijetiti da OpenGL koristi konvenciju množenja matrice s točkom pa se zbog toga redoslijed operacija piše obrnuto od redoslijeda izvršavanja.



Slika 4.5: Prikaz njihala prije pada.

```
/* Calculate new model matrix based on new cart
 * position and pendulum angle.
*/
model = glm::translate(glm::mat4(1.0f),
                  glm::vec3(x_cart, 0.0f, 0.0f));
model = glm::rotate(model,
                angle,
                glm::vec3(0.0f, 0.0f, 1.0f));
model = glm::scale(model,
               glm::vec3(1.0f, 0.5f, 0.0f));
}
```

Isječak 4.4: Postavljanje model matrice štapa.

4.5. Upravljanje sustavom

Svrha upravljačkog sustava je generirati iznos i smjer vanjske sile na kolica u svakom vremenskom trenutku kako bi sustav mogao na temelju te sile prema jednadžbama 2.1 i 2.2 izračunati novo stanje sustava. Upravljanje sustavom je moguće ostvariti na više načina ovisno o cilju koji se želi postići. Za stabilizaciju sustava je učinkovito korištenje

neizrazitog upravljanja koje u okviru ovog rada nije implementirano. U okviru ovog rada implementirano je i opisano dvije vrsta upravljanja. Prvo je upravljanje tipkovnicom, a drugo je upravljanje algoritmom gradijentnog spusta što će detaljnije biti opisano u poglavlju 4.5.2. Koja vrsta upravljanja će biti korištena također je potrebno odrediti u inicijalizacijskim parametrima.

Zatvorena petlja opisana u poglavlju 2 i prikazana na slici 2.3 u kojoj komuniciraju sustav i upravljački sustav, neovisno o načinu implementacije upravljanja najbolje se može dočarati kodom glavne petlje 4.5 unutar funkcije `main()`.

```
while (w.open()) { // Loop until window is open.
   system.clear(); // Clear the screen.

   float force = gd->calculate_force();
   system.calculate_new_state(force);
   system.draw();

   w.swap_buffers(); // Show new frame.
   w.poll_events(); // Poll and process events.
}
```

Isječak 4.5: Glavna petlja.

4.5.1. Upravljanje tipkovnicom

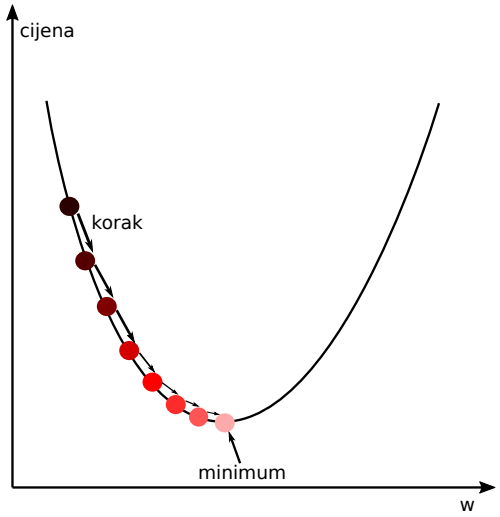
Pomicanje kolica tipkovnicom ostvareno je korištenjem povratne funkcije `keyboard_callback` koja vraća kôd pritisnute tipke. Korisnik pritiskom na lijevu ili desnu strelicu primjenjuje na kolica silu konstantnog iznosa. Ovakvo ponašanje implemenitrano je kao u isječku koda 4.6.

```
float KeyboardControl::calculate_force()
{
   switch (direction) {
      case Direction::none: return 0.0;
      case Direction::right: return FORCE_VALUE;
      case Direction::left: return -FORCE_VALUE;
      default: throw std::logic_error("Unknown direction.");
   }
}
```

Isječak 4.6: Određivanje sile prilikom upravljanja tipkovnicom.

4.5.2. Upravljanje gradijentnim spustom

Gradijentni spust je optimizacijski algoritam koji se koristi za minimizaciju funkcije tako da se iterativno korakom određene veličine kreće u smjeru negativnog gradijenta. U slučaju pozitivnog gradijenta, algoritam se vraća u prethodno stanje smanjujući veličinu koraka i ponavljajući navedeni postupak sve do ispunjenja određenih uvjeta.



Slika 4.6: Princip izračuna gradijentnog spusta.

U svrhu upravljanja, gradijentni spust se koristi za određivanje iznosa i smjera sile kojim je potrebno djelovati na kolica kako njihalo ne bi palo na kolica. Kôd kojim je to ostvaren prikazan je u dodatku B.2 Rezultat koji se postiže gradijentnim spustom je gibanje njihala oko centra. No, postupkom koji je implementiran u radu nije moguće postići stabilizaciju.

Algoritam početno kreće od sile čiji je iznos jednak nuli, te na temelju te sile računa sljedeće stanje sustava. U sljedećem koraku se iznos te sile povećava za određeni iznos. Smjer sile ovisi o tome je li trenutni kut između njihala i kolica pozitivan ili negativan. Ako je pozitivan, generira se sila pozitivnog smjera jer je cilj djelovanjem te sile podići njihalo prema okomici ($\phi = 0$), analogno je razmišljanje i za negativan kut.

U svakom koraku se provjerava je li generirana sila uzrokovala da njihalo prijeđe s pozitivnog u negativan kut ili obrnuto. U slučaju da se to dogodilo, znači da se uz ovaj iznos sile prelazi okomica što nije željeno ponašanje jer je cilj postaviti njihalo u vertikalan položaj. Zbog toga se iznos sile vraća na prethodni koji nije uzrokovao prelazak na drugu stranu.

Sada se sila povećava za manji iznos nego u prethodnom koraku jer je sila blizu traženog iznosa i potrebno je finije podesiti njen iznos. Ovaj postupak staje dok je postignuta dovoljno fina preciznost sile ili kada je postignut maksimalan iznos sile koji je moguće generirati (unaprijed određeni iznos ovisan o parametrima sustava).

Ovaj postupak uspješno održava njihalo da ne pada, ali njegov problem je to što ne

uzima u obzir kutnu brzinu koja se postiže pri djelovanju vanjske sile na kolica. Zbog velike kutne brzine, njihalo počinje jako ubrzavati i ne uspjeva se zaustaviti u željenom položaju nego završava na drugoj strani. Da bi ovaj postupak uspješno stabilizirao sustav bilo bi potrebno odrediti iznos sile koji bi se postepeno smanjivao što bi njihalo bilo bliže vertikalnom položaju.

4.6. Određivanje sljedećeg stanja

Sljedeće stanje sustava moguće je u potpunosti opisati dvjema diferencijalnim jednadžbama drugog reda navedenim u poglavlju 2. Kako bi bilo moguće numerički riješiti ovaj sustav, za računalo je prikladnije zapisati ga u matričnom obliku navedenom u formuli 4.1 (reff). Uvede li se vektor stanja $\vec{z} = z_1 \quad z_2 \quad z_3 \quad z_4$, i primjeti li se da vrijedi $z_2 = \dot{z_1}$ i $z_4 = \dot{z_3}$ dobiva se traženi upravo traženi oblik zapisa sustava koji uz poznato početno stanje omogućava rješavanje ovog sustava

$$\frac{d}{dt}\vec{z} = \frac{d}{dt}\begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \end{bmatrix} = \begin{bmatrix} z_2 \\ \frac{u\cos(z_1)-(M+m)g\sin(z_1)+mL\sin(z_1)\cos(z_1)z_2^2}{mL\cos^2(z_1)-(M+m)L} \\ z_4 \\ \frac{u+mL\sin(z_1)z_2^2-mg\sin(z_1)\cos(z_1)}{M+m-m\cos^2(z_1)} \end{bmatrix} \tag{4.1}$$

```
std::array<float, 4> new_state =
   solver->calculate_new_state({s.angle,
                          s.angular_vel,
                          s.x_cart,
                          s.x_cart_vel},
                         current_force);
s.angle = new_state[0];
s.angular_vel = new_state[1];
s.x_cart = new_state[2];
s.x_cart_vel = new_state[3];
```

Isječak 4.7: Postavljanje novog stanja.

4.6.1. Rješavanje sustava jednadžbi

Sustav diferencijalnih jednadžbi riješen je metodom Dormand-Prince. Metoda Dormand-Prince je eksplicitna metoda za rješavanje diferencijalnih jednadžbi i dio je porodice Runge-Kutta metodi. Za izračun koristi šest funkcija evaluacije kako bi izračunala rješenja četvrtog i petog reda. Za razliku od metode Runge-Kutta četvrtog reda, ova metoda omogućava adaptivno podešavanje koraka integracije što je čini preciznijom i računski efikasnijom. To je čini popularnom metodom za rješavanje u velikom dijelu gotovih alata za ovu namjenu.

U svrhu rješavanja jednadžbi zapisanih u matričnom obliku prema formuli 4.1 korištena je C++ knjižnica ODEINT [3]. To je knjižnica samo s datotekama zaglavlja koja služi za numeričko rješavanje diferencijalnih jednadžbi. Razvijena je uporabom metaprogramiranja pomoću predložaka (engl. *template metaporgramming*) što omogućuje visoku fleksibilnost bez lošeg učinka na performanse (citat s ode).

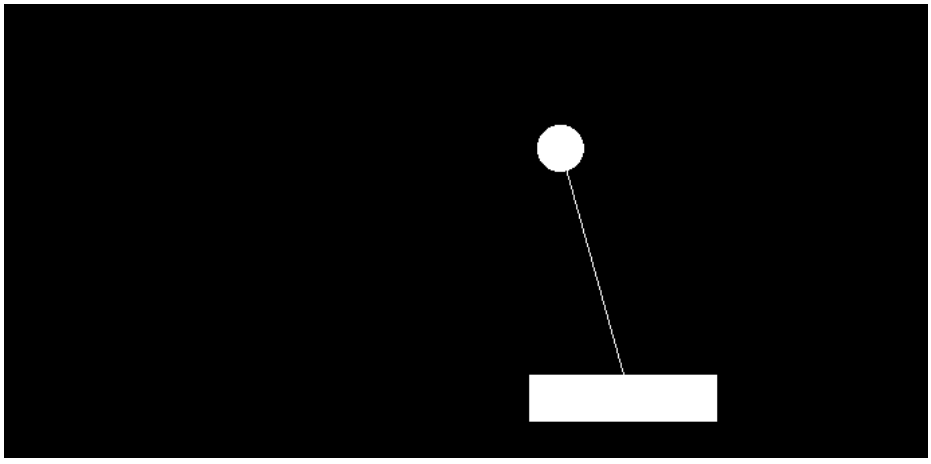
```
// Integration changes current state object.
boost::numeric::odeint::integrate(*this,
                                  current_state, // Start state.
                                  0.0, // Start time.
                                  0.002, // End time.
                                  0.002); // dt.
```

Isječak 4.8: Poziv ODEINT funkcije za numeričko integriranje.

U isječku koda 4.8 prikazan je poziv ODEINT [3] funkcije. Funkcija prima objekt `Solver` koji ima nadjačani operator poziva (engl. *call operator*) kako bi se prilagodilo ponašanje na način na koji to zahtijeva ODEINT integrator (isječak koda B.3).

4.7. Prikaz pozadine

Slika 4.8 prikazuje sustav prije dodavanja pozadinskih zvijezda. Pozadinske zvijezde predstavljene su slikovnim elementima bijele boje na slučajnim mjestima na ekranu. U slučaju kretanja uz rub prozora bez zvijezda je dobiven dojam statičnosti, odnosno, njihalo neovisno o prijeđenom putu djeluje kao da cijelo vrijeme miruje.



Slika 4.7: Prikaz sustava bez pozadinskih zvijezda

Kako bi se postigao vizualan učinak kretanja implementiran je razred `Stars` koji crta nasumično generirane zvijezde. Zvijezde se generiraju preko određenog broja zaslona, nakon čega se ponovno počinju ponavljati iste. U ovom slučaju uzeto je

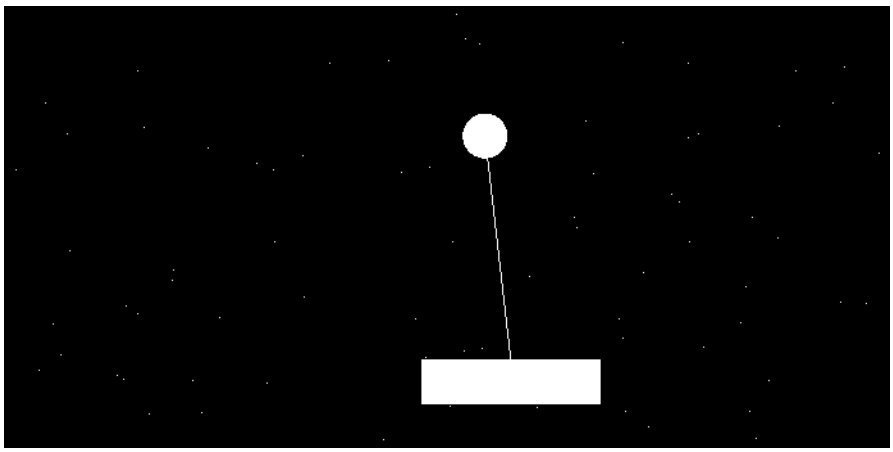
da su zvijezde generirane preko šest veličina zaslona što je i više nego dovoljno da bi se postigao učinak nasumičnosti kroz cijelo kretanje. Ovo je također vrlo učinkovito što se tiče performansi jer se ne trebaju aktivno stvarati nove zvijezde kretanjem kroz scenu. Način na koji se generiraju koordinate zvijezda napisan je u dodatku B.4.

Kôd 4.9 prikazuje način crtanja zvijezda lijevo i desno od ishodišta koordinatnog sustava u slučaju pomaka kolica izvan trenutno prikazanog ekrana.

```
void Stars::draw(Renderer& renderer)
{
   // Check whether new screen should be shown.
   if (borders.right >= start_right + n_screens) {
      start_left += n_screens;
      start_right += n_screens;
   }
   else if (borders.left <= start_left) {
      start_left -= n_screens;
      start_right -= n_screens;
   }

   // Initalize model matrix and draw stars.
   model = glm::translate(glm::mat4(1.0f),
                  glm::vec3(start_left, 0.0f, 0.0f));
   RenderObject::draw(renderer);
   model = glm::translate(glm::mat4(1.0f),
                  glm::vec3(start_right, 0.0f, 0.0f));
   RenderObject::draw(renderer);
}
```

Isječak 4.9: Funkcija za iscrtavanje pozadine.



Slika 4.8: Prikaz sustava s pozadinskim zvijezdama

4.8. Moguća proširenja

Jedna od mogućih nadogradnji ovog sustava bi bilo razvijanje neizrazitog sustava upravljanja [2]. Neizrazito upravljanje definira niz jezičnih pravila koja dolaze od eksperta primjenom kojih se određuje ponašanje sustava. U slučaju okrenutog njihala, pravila bi mogla biti definirana na temelju kuta i kutne brzine. Primjerice, ako je kut velik i kutna brzina velika, djeluj silom nekog iznosa ili ako je kut mali, a kutna brzina velika, djeluj silom nekog drugog, manjeg iznosa. Ovakva nadogradnja bi omogućila stabilizaciju njihala u točki ravnoteže.

Osim ovoga, sustav bi bilo moguće prikazati u tri dimenzije gdje bi se omogućilo kretanje kolica po sve tri osi. U tom slučaju, za prikaz bi se koristili gotovi modeli.

5. Zaključak

U sklopu ovog rada implementiran je dvodimenzionalni grafički prikaz okrenutog njihala u programskom sučelju OpenGL. U sklopu implementacije je napravljeno kretanje njihala opisano nelinearnim jednadžbama korištenjem programske knjižnice ODEINT. Osim kretanja ostvarenog preko tipkovnice, implementirano je i kretanje sustava gradijentnim spustom. To također daje zadovoljavajuće rezultate iako bi bilo potrebno dodatno nadograditi sustav u svrhu stabilizacije.

Okrenuto njihalo je općenito vrlo važan koncept pri opisivanju kretanja tako da je ovdje istražen mali dio koji bi se mogao još proširivati kroz implementaciju različitih vrsta upravljanja što ostavlja potencijalne mogućnosti nadogradnje na ostalim radovima tijekom studija. Naglasak ovog rada je bio upravo na programskom dijelu, odosno, modeliranju ovakvog sustava uz grafički prikaz kako bi se dobila simulacija s fizikom koja dobro opisuje stvarno ponašanje modela.

Literatura

[1] Olfa Boubaker, **The inverted pendulum: A fundamental benchmark in control theory and robotics**, 2012.

[2] Marko Čupić, Bojana Dalbelo Bašić i Marin Golub, **Neizrazito, evolucijsko i neuroračunrastvo.** 2013.

[3] **ODEINT**, url: `http://headmyshoulder.github.io/odeint-v2/`.

[4] Marko Čupić i Željka Mihajlović, **Interaktivna računalna grafika kroz primjere u OpenGL-u**, 2018.

Simulacija obrnutog njihala

Sažetak

Sustav okrenutog njihala je jedan od čestih sustava koji se koriste u svrhu prikaza sustava upravljanja. Takav sustav je moguće proširiti iz dvodimenzionalnog prostora u trodimenzionalni uz povećanja složenosti poput okreljivosti zgloba i slično. Kako bi se mogli razumjeti složeniji koncepti u sklopu ovog rada je opisan osnovni model i prikazana simulacija okrenutog njihala na kolicima u modernom OpenGL-u uz implementaciju fizike iz dviju diferencijalnih jednadžbi drugog reda. Jednadžbe su riješene numeričkom metodom integracije Dormand-Prince. Implementirano je kretanje tipkovnicom i algoritmom gradijentnog spusta. Naglasak rada je, uz opis sustava, i programsko ostvarenje u jeziku C++.

Ključne riječi: okrenuto njihalo, simulacija, OpenGL, C++, ODEINT, numerička integracija, Dormand-Prince metoda, upravljanje, gradijentni spust

Inverted pendulum simulation

Abstract

Inverted pendulum system is a common system used for control demonstration. Such system is extendable from 2D space to 3D space with complexity enhancements such as adding rotary joint. With intention to understand more complex inverted pendulum systems, this thesis describes basic model with graphic simulation of inverted pendulum on cart. Graphics is implemented using modern OpenGL. Physics is described by two second-order differential equations and solved using Dormand-Prince method for numerical integration. Two types of movement are implemented: one is basic movement using arrow keys and other is based on gradient descent algorithm. Proposed inverted pendulum model is realized using C++ programming language.

Keywords: inverted pendulum, simulation, OpenGL, C++, ODEINT, numerical integration, Dormand-Prince method, control, gradient descent algorithm

Dodatak A
Razvojno okruženje

U sklopu implementacije ovog rada korišteno je prijenosno računalo s procesorom Intel Core i5-5200 i grafičkom karticom NVIDIA GeForce 940M na operacijskom sustavu Arch Linux 5.1.8.

Korišten je GCC prevodioc inačice 8.3.0 uz zastavice za povezivanje `-lGL -lGLEW -lglfw` koje omogućavaju korištenje OpenGL-a. Program se uspješno prevodi bez upozorenja sa zastavicama `-Wall -Wextra -pedantic-errors`. Korišteni standard C++ jezika je C++17 omogućen sa zastavicom `std=c++17`. Za optimizacije se koriste zastavice `-O2 -march=native -mtune=native -DNDEBUG` kojim se osim standardnih optimizacija omogućavaju optimizacije specifične za procesor na kojem se programski kod prevodi.

Za implementaciju ovakve simulacije performanse računala nisu presudan faktor i program se prevodi u svega nekoliko sekundi. Performanse nisu problem zato što je korišten sjenčar za rad na grafičkoj kartici, a njegova zadaća se svodi na mali broj izračuna jer su objekti vrlo jednostavni (sastoje se od nekoliko vrhova). Za izračun fizike također nije potrebno puno procesorskog vremena.

Rad je napisan u uređivaču teksta Vim uz potporu za semantiku jezika C++ pomoću nadogradnje YouCompleteMe.

Dodatak B
Dodatni tekstovi programa

```
// Camera movement due to cart being close to the
// edge of the screen.
constexpr float padding = 0.1f;
float cart_reach_right = s.x_cart + 0.2f + padding;
float cart_reach_left = s.x_cart - 0.2f - padding;
// Move camera to the right.
if (cart_reach_right > ds.borders.right) {
   ds.view = glm::translate(ds.view,
            glm::vec3(-(cart_reach_right - ds.borders.right),
                   0.0f,
                   0.0f));
   ds.borders.left += cart_reach_right - ds.borders.right;
   ds.borders.right = cart_reach_right;
// Move camera to the left.
} else if (cart_reach_left < ds.borders.left) {
   ds.view = glm::translate(ds.view,
            glm::vec3(-(cart_reach_left - ds.borders.left),
                   0.0f,
                   0.0f));
   ds.borders.right += cart_reach_left - ds.borders.left;
   ds.borders.left = cart_reach_left;
}
```

Isječak B.1: Pomicanje kamere.

```
float GradientDescent::calculate_force() {
   System::State s = system.get_state();
   Solver solver = system.get_solver();

   float step = 10;
   float force = 0;
```

```
        // For positive angle, force direction is positive.
        int sign = s.angle > 0 ? 1 : -1;

        while (step >= 0.05 && std::abs(force) < 100) {
           std::array<float, 4> new_state =
               solver.calculate_new_state(
                  {s.angle, s.angular_vel, s.x_cart, s.x_cart_vel},
                   force);
           float& angle = new_state[0];

           // Check if angle sign changed.
           if ((sign == -1 && angle > 0)
               || (sign == 1 && angle < 0)) {
              force -= sign * step;
              // Decrease step size.
              step /= 5.0f;
           }
           force += sign * step;
        }
     }
     }
```

Isječak B.2: Gradijenti spust

```
void Solver::operator()(const std::array<float, 4> &s,
                  std::array<float, 4> &dxdt,
                  const double /* t */)
{
   constexpr float g = 9.80665f;
   dxdt = {
      s[1],
      static_cast<float>(
      (force * std::cos(s[0]) - (cart_mass + ball_mass)
      * g * sin(s[0]) + ball_mass * rod_length * std::sin(s[0])
      * std::cos(s[0]) * std::pow(s[1], 2)) /
         (ball_mass * rod_length * std::pow(std::cos(s[0]), 2)
         - (cart_mass + ball_mass) * rod_length)),
      s[3],
      static_cast<float>(
      (force + ball_mass * rod_length * std::sin(s[0])
```

```
      * std::pow(s[1], 2) - ball_mass * g * std::sin(s[0])
      * std::cos(s[0])) / (cart_mass + ball_mass - ball_mass
      * std::pow(std::cos(s[0]), 2)))
   };
}
```

Isječak B.3: Operator poziva.

```
// Create random vertex data - make stars random.
std::random_device dev;
std::mt19937 rng(dev());
std::uniform_real_distribution<float> dist_w( 0.0f, n_screens);
std::uniform_real_distribution<float> dist_h(-0.2f, 0.8f);
std::vector<glm::vec2> pos;
pos.reserve(n_stars);
for (size_t i = 0; i < n_stars; i++) {
   pos.emplace_back(dist_w(rng), dist_h(rng));
}
```

Isječak B.4: Nasumično generiranje zvijezda.