CHALMERS

CampusQuest

Utvecklandet av ett interaktivt spel för nyanställda på Chalmers

Examensarbete inom högskoleprogrammet Datateknik

AMANDA THORÉN
REBECCA ALBO

INSTITUTIONEN FÖR DATA- OCH INFORMATIONSTEKNIK
CHALMERS TEKNISKA HÖGSKOLA
Göteborg 2025
www.chalmers.se

Examensarbete 2025

CampusQuest

Utvecklandet av ett interaktivt spel för nyanställda på Chalmers

AMANDA THORÉN
REBECCA ALBO

CHALMERS

Institutionen för Data- och informationsteknik
Chalmers Tekniska Högskola
Göteborg 2025

CampusQuest
Utvecklandet av ett interaktivt spel för nyanställda på Chalmers
AMANDA THORÈN
REBECCA ALBO

© AMANDA THORÉN & REBECCA ALBO, 2025.

Handledare: Sakib Sistek
Examinator: Nicholas Smallbone

Examensarbete 2025
Institutionen för Data- och informationsteknik
Chalmers Tekniska Högskola
SE-412 96 Göteborg
Telefon +46 31 772 1000

Omslagsbild: Campus Johanneberg med spelets baskaraktär vinkandes.

Skriven i LaTeX
Göteborg 2025

CampusQuest
Utvecklandet av ett interaktivt spel för nyanställda på Chalmers
AMANDA THORÉN
REBECCA ALBO
Institutionen för data och informationsteknik
Chalmers Tekniska Högskola

Sammanfattning

Chalmers har behov av introducera nyanställda och doktorander till universitetet. Informationen är idag ofta utspridd, texttung och kan vara svår att ta till sig. Vår lösning är att kombinera nytta med nöje genom ett interaktivt onboarding-spel i en redan befintlig app – Chalmers Onboarding. Spelet gör det möjligt att utforska campus virtuellt, interagera med karaktärer och ta del av viktig information stegvis, på ett engagerande sätt.

Målet är att skapa en spelupplevelse som är lärorik, lättnavigerad och visuellt tilltalande, där användaren själv styr takten i sin introduktion till arbetsplatsen. Lösningen ska inte bara öka förståelsen för Chalmers som organisation och plats, utan också göra introduktionen mer underhållande.

Nyckelord: spelifiering, utbildning, Godot, Flutter.

Förord

Detta examensarbete har genomförts under vårterminen 2025 som en del av kandidatprogrammet för datateknik, Chalmers tekniska högskola. Projektet har genomförts i samarbete med Chalmers, med målet att förbättra introduktionsprocessen för nyanställda genom ett spel.

Idén växte fram ur ett gemensamt intresse för spelutveckling och en vilja att använda teknik för att lösa verkliga problem. Vi ville skapa något som inte bara informerar utan också engagerar. Resultatet blev ett spel som guidar nya medarbetare genom campus och verksamheten på ett underhållande sätt.

Vi vill rikta ett stort tack till vår handledare Sakib Sistek på Chalmers för värdefull återkoppling och stöd under projektets gång.

Vi hoppas att vår lösning kan bidra till en smidigare och mer inspirerande start för nya medarbetare på Chalmers – och kanske i framtiden även på andra arbetsplatser.

Amanda Thorén, Rebecca Albo, Göteborg, Juni 2025

Innehåll

Figurer

1
Inledning

Detta kapitel introducerar bakgrunden till utvecklingen av spelet CampusQuest, syftet och mål med arbetet, samt arbetets avgränsningar.

1.1 Bakgrund

Chalmers har behov av att sammanfatta och förenkla nödvändig information till kandidater i rekryteringar och nyanställda. Det innebär bland annat att informera om Chalmers verksamhet på ett enkelt, lustfyllt, inspirerande och överskådligt sätt. Detta avser främst till befintlig och nyanställd personal, men också för att attrahera potentiella kandidater till Chalmers – i linje med Chalmers vision att attrahera excellens.

När man är nyanställd och ska börja på en ny arbetsplats finns det ofta mycket ny information att ta in, många dokument att läsa och dessa kan vara utspridda på olika ställen. Man kommer också till en ny fysisk plats man kanske aldrig varit på, där det kan vara svårt att hitta till en början och lokalisera sig bland de olika byggnaderna. Det kan vara tröttsamt att sätta sig ner och läsa igenom text efter text, eller att gå vilse bland alla byggnader i början. Man kanske också väljer att bara ta in den information som är absolut mest nödvändig och missar helhetsbilden av Chalmers. Genom att ta in ny information och lära sig genom att inte bara läsa utan också interagera, kan lärande och underhållning förenas. Detta kan motivera den anställde att lära sig mer om Chalmers och sin nya arbetsplats.

1.2 Syfte

För att inkludera underhållning i inlärningen om Chalmers och den nya arbetsplatsen, är vår lösning att skapa ett innovativt och interaktivt campus-spel i den redan befintliga appen Chalmers Onboarding. I detta spel kan nyanställda på Chalmers engagera sig på ett roligt och lärorikt sätt. Syftet är att låta nyanställda på ett unikt sätt utforska universitetets två campus, Johanneberg och Lindholmen, och få relevant information om platsen de befinner sig på. Den anställde ska själv få styra en karaktär och välja vart på campus denne vill ta sig och ta in delar av informationen i taget.

1.3 Mål

- Modulära funktioner: återanvändbara funktioner, både inom projektetet och andra projekt.
- Anpassningsbar dialog och interaktioner: universitetet ska när som helst enkelt kunna ändra dialoger och eventuellt interaktioner och visuella element.
- Implementera ett spel som är både lärorik och uppmuntrar fortsatt användning
- Möjliggör en integration av spelet i appen Chalmers Onboarding

1.4 Avgränsningar

Arbetet går ut på att från grunden skapa ett helt nytt spel. Med grund i detta ligger fokus på de mer formella delarna av Chalmers, exempelvis dess institutioner och relevanta byggnader. Spelet omfattar inte caféer, restauranger eller liknande. Vidare innehåller spelet ingen information om vad som finns inuti byggnaderna, exempelvis föreläsningssalar och datorsalar. På grund av begränsad tidsram ansågs det viktigare att i början av utvecklingen av spelet fokusera på utsidan av byggnaderna och det som finns utomhus.

2
Metod

Arbetet med spelet kan delas in i två delar: projektplanering och implementation. Dessa två delar beskrivs i mer detalj nedan.

2.1 Projektplanering

Det hela började som en enkel idé om ett spel som kan hjälpa nyanställda lära känna Chalmers, men det fanns inte många krav eller specifikationer. Detta ledde till att relativt mycket tid i början behövde läggas på att få ner en grundidé om vad spelet skulle innehålla och vilken ambitionsnivå som var rimlig. Inledningsvis skedde ett möte med beställarna från Chalmers, där de fick lyfta sina tankar kring vad spelet skulle leverera. Sedan skedde även ett möte med de studenter som jobbar på den app som spelet ska integreras i. Efter detta fanns en bättre bild av vad kraven för spelet skulle vara. När grundidén för spelet var på plats, behövdes det fattas beslut om vilken spelmotor som skulle användas och vilka andra verktyg som skulle kunna bli aktuella.

2.1.1 Val av verktyg

För att utveckla spel finns det flera alternativa spelmotorer att välja mellan, bland annat Unity, Unreal Engine och Godot. Eftersom ingen av oss tidigare hade arbetat med spelutveckling gjordes efterforskning. Unreal Engine är en av de mest använda spelmotorerna och även en av de mest kraftfulla. En nackdel med Unreal Engine är att det är en relativt hög inlärningskurva. Unity är en annan populär spelmotor som anses användarvänlig. Det är till viss del gratis att använda, men med ett begränsat utbud jämfört med om man betalar för det. Spelmotorn Godot är däremot helt gratis att använda, är effektivt och tillåter exportering till andra plattformar, inklusive mobil och dator [1]. Baserat på specifikationerna som tagits fram ansågs Godot vara en lämplig spelmotor att arbeta med. Som programmeringsspråk valdes GDScript, ett högnivåspråk som är objektorienterat. GDScript är utvecklat och anpassat för Godot. Språket påminner om (men är inte baserat på) Python [2]. Eftersom GDScript är utvecklat specifikt för Godot ansågs det lämpligast att använda detta som primärt programeringsspråk i spelutvecklingen.

När det gäller verktyg för 3D-modellering finns det också flera alternativ att välja mellan. Autodesk 3ds Max, ZBrush och Houdini är kraftfulla verktyg när det kommer till 3D-modellering, men det kostar att använda dem. Blender är ett alternativ till dessa då Blender är helt gratis att använda. Blender har också ett brett utbud av

verktyg för modellering och animering [3]. Av dessa anledningar valdes Blender som verktyg för att skapa karaktärer och annan rekvisita till spelet.

2.2 Implementation

När grunderna var på plats och nästa steg var att börja implementera spelet behövdes arbetet delas upp. Vi såg att arbetet skulle involvera både backend- och frontend-arbete. Vi valde att då inledningsvis dela upp arbetet så att en av oss fokuserade mer på backend och implementera spelet i den redan befintliga appen. Den andra fokuserade mer på frontend och på att skapa själva spelet i Godot. Det bestämdes att under arbetets gång ha kontinuerliga möten där vi uppdaterade varandra om vad som skett sedan senast och diskutera potentiella problem och idéer som dyker upp under arbetets gång.

Under arbetets gång användes GitHub för att organisera och samordna arbetet. Inledningsvis skedde arbetet i två olika grenar, för att inte störa huvudkoden. Efter ett tags arbete med koden övergick det dock till att endast den som arbetade frontend gjorde större förändringar i koden, så vi övergick till att arbeta direkt i huvudkoden. Med hjälp av Github kunde ett effektivt samarbete ske, då båda hela tiden hade tillgång till vad den andre implementerat och vi kunde hålla varandra uppdaterade.

I början av arbetet hade vi inte riktigt en uppfattning om vilken omfattning spelet skulle kunna ha inom den tidsram som var satt. Detta ledde till att de inledande specifikationerna var relativt grundläggande. Under implementeringen av spelet insåg vi dock att det fanns utrymme för att utveckla spelet vidare, bortom de grundläggande specifikationerna. Vi hade då möten där vi diskuterade de idéer vi hade för vidare utveckling och kom gemensamt fram till vad vi ville lägga till för att göra spelet mer omfattande. Allt som implementerades testades kontinuerligt under arbetets gång. När nya delar lades till, testades både dessa och de nya delarna för att säkerställa att allt fortfarande fungerade som önskat. När spelet var färdigt genomfördes grundliga genomgångar av spelet och dess funktioner för att säkerställa att allting fungerade som det skulle.

2.3 AI

Under arbetets gång har AI-verktyg använts som stöd. Framför allt har ChatGPT används för att hjälpa till vid felsökning och vid tolkning av felmeddelanden. Eftersom ingen av oss var bekanta med varken Blender eller Godot i början av arbetet, var det tidssparande att kunna få hjälp av AI för att tolka felmeddelanden. Vid användandet av ChatGPT gavs den bit kod som gav felmeddelandet samt själva felmeddelandet. ChatGPT förstod oftast vad problemet var och kunde förklara det. Tack vare detta kunde vi själva förstå problemet och snabbt ta oss vidare. Exempel på prompts som använts är:

- *I'm in blender in weight paint and want to select a bone. The video I'm watching is telling me to shift-click on the bone but nothing is happening*
- *[long error message] not sure what this means*
- *what does "cant operate on nodes the current scene inherits from" mean?*

3
Teori

I detta kapitel presenteras den teoretiska och tekniska bakgrund som behövs för att bättre kunna följa resten av rapporten.

3.1 Spelifiering

Något som får ökad uppmärksamhet inom media och forskning är spelifiering (gamification). Det kan definieras som användandet av speldesign och spelmekanismer inom andra kontexter än spelvärlden. I detta sammanhang kommer det att talas om kontexten utbildning. Tanken är att spelifiering ska bidra till ett ökat engagemang inom inlärning och motivation att lära sig. Det har på senare år gjorts mycket forskning på ämnet och Murillo-Zamorano m.fl. tar i sin artikel upp flera exempel. De nämner bland annat forskning har visat att spelifiering kan ha stora fördelar för elever, såsom bättre betyg, bättre inlärningsförmåga och bättre kritiskt tänkande. De lyfter också att spelifiering kan ha fördelar för lärare. Vidare nämner författarna att elever värdesätter spelifiering och att det i vissa kontexter kan ha större potential än de mer traditionella metoderna som används inom utbildning [4].

Ett specifikt exempel på spelifiering inom utbildning presenteras i en artikel av Strmečki, Bernik och Radošević. De undersökte huruvida det var mer effektivt att använda spelifiering i en onlinekurs, jämfört med en vanlig onlinekurs med bara text och illustrationer. Deras resultat var att det var en betydande skillnad i resultat, till fördel för de som tog onlinekursen som var spelifierad. Författarna till artikeln lyfter fram ett antal element som fördelaktigt kan inkorporeras när utbildning och spelifiering kombineras. Bland annat nämner de poängsystem, utmaningar, en topplista (leaderboard) och anpassning av karaktärer (customization). Poäng är något som indikerar framgång i spelet och motiverar spelaren att fortsätta spela. Utmaningar hjälper spelaren att ta sig fram i spelet och kan vara ett sätt att samla poäng. Topplistor är en plats där andra kan se en spelares poäng och kan användas för att motivera spelare som är tävlingsinriktade. Det finns även andra element som kan inkorporeras för den sociala spelaren, exempelvis samarbetsmoment [5].

Även om det finns mycket som talar för hur effektivt det är med spelifiering inom utbildning, finns det också forskning som visar på att det inte alltid är det. I en artikel av Jiyuan m.fl. lyfter författarna förvisso mycket positivt med spelifiering, men även att det inom högre utbildning har begränsad effektivitet. Författarna föreslår att det kan bero på att det är färre lärotimmar på universitet och att mer tid läggs på praktiska aspekter [6]. Den begränsade effektiviteten av spelifiering inom högre utbildning är också något som nämns i den ovan nämnda artikeln av

Murillo-Zamorano m.fl.

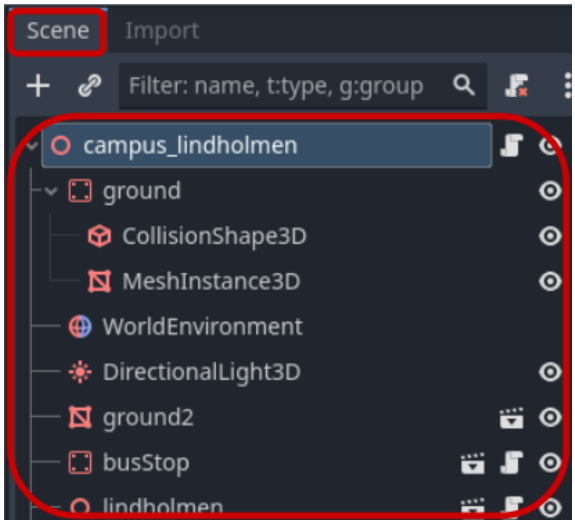
3.2 Teknisk bakgrund

I arbetet med spelet har tre huvudsakliga verktyg använts och kommer att refereras till nedan i rapporten. För att få en bättre förståelse för dessa, presenteras de nedan.

3.2.1 Godot

Godot är en spelmotor som används för att skapa spel, både i 2D och i 3D. Det är gratis att använda och använder sig av öppen källkod. I Godot finns ett stort utbud av verktyg som ger användaren möjlighet att skapa allt från enkla spel till mer avancerade spel. Dessa verktyg ger användaren möjlighet att direkt börja skapa spelet, utan att behöva "återuppfinna hjulet", som de beskriver det på sin webbplats. När spelet är färdigt finns möjligheter att exportera till många olika plattformar, såsom macOS, Windows och Android. Det som användare skapar i Godot ägs av användarna själva [7]. I Godot finns ett antal begrepp som är bra att ha en övergripande förståelse för. Dessa beskrivs nedan.

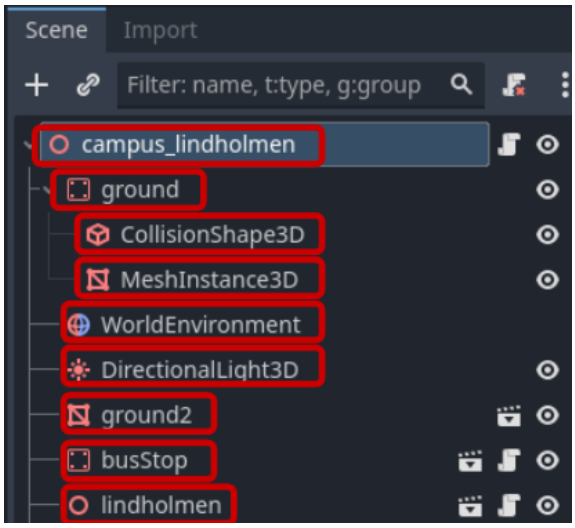
Scener
Spel som skapas i Godot bygger på scener som används en gång eller återanvänds flera gånger. En scen kan vara en karaktär, en meny eller en hel bana i ett spel. Scener kan ha andra scener i sig, då som noder, exempelvis om man vill ha en karaktär placerad i en specifik bana [8]. Figur 3.1 visar en scen bestående av ett antal noder.



Figur 3.1: Exempel på en scen

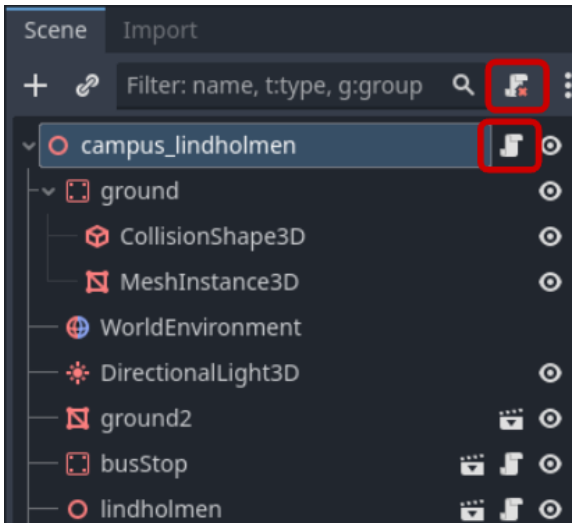
Noder
Varje scen består av en eller flera noder. De har en hierarkisk struktur, där varje scen-träd har en rotnod, till vilken resterande noder på något vis är kopplade. Noderna har ett förälder-barn-förhållande, där varje nod har en enda förälder. En

föräldernod behöver inte ha en barnnod. En scen kan bli en nod i en annan scen, såsom när en karaktär placeras i en bana [8]. I Figur 3.2 kan ett nodträd ses, där `campus_lindholmen` är rotnoden.



Figur 3.2: Exempel på noder i ett nodträd

Skript
Till varje nod kan ett skript läggas till. Ett skript är något som förlänger funktionaliteten på det objekt som skriptet tillhör [9]. I Figur 3.3 visas samma scen som i Figur 3.1. Rotnoden är den nod som är överst i listan och den har i sin tur ett antal barnnoder under sig. Till respektive nod kan ett skript läggas till för att ytterligare förlänga funktionaliteten hos noden. I Figur 3.4 visas ett exempel på hur ett skript kan se ut i kod. Det språk som används är GDScript.



Figur 3.3: Exempel på en nod med ett skript

```
func update_pant_color():
    var pant = $Armature/Skeleton3D/Pants1
    var material = pant.get_surface_override_material(0)
    if material == null:
        material = pant.get_active_material(0).duplicate()
        pant.set_surface_override_material(0, material)
    material.albedo_color = pant_colors[current_pant_index]
```

Figur 3.4: Exempel på kod i ett skript

3.2.2 Blender

Blender är ett gratis verktyg, baserat på öppen källkod, som bland annat kan användas till att 3D-modellera. Det finns ett stort utbud av skulpteringsverktyg som hjälper användaren att skapa allt från enkla modeller till stora avancerade modeller. Efter att modellen är skapad finns det även verktyg för att bland annat rigga modellerna och animera dem [10]. Det som användare skapar i Blender ägs helt och hållet av användarna själva [11].

3.2.3 Android Studio och Flutter

Appen *ChalmersOnboarding* är utvecklad med hjälp av användargränssnittsramverket Flutter, ett ramverk skapat av Google för plattformsoberoende apputveckling. Som integrerad utvecklingsmiljö (IDE) har Android Studio använts för att utveckla, testa och felsöka applikationen.

Efter installation av Android Studio genomfördes följande steg: För att kunna utveckla applikationen krävdes följande komponenter från Android Studios SDK Manager:

- Android SDK Platform (API 35.0.2)
- Android SDK Command-line Tools
- Android SDK Build-Tools
- Android SDK Platform-Tools
- Android Emulator

1. Installation av Flutter-plugin, vilket automatiskt installerade tillhörande Dart-plugin.
2. Godkännande av Android SDK-licenser via kommandotolken med kommandot:
   `flutter doctor --android-licenses`
3. Hämtning av alla nödvändiga paket och beroenden genom kommandot:
   `flutter pub get`

Denna konfiguration valdes baserat på rekommendationer från projektets tidigare utvecklare. Denna konfiguration garanterade Kompatibilitet med befintlig kodbas.
widgets
Flutter använder widgets, vilka är klasser som beskriver UI-element.

4
Genomförande

För att skapa ett spel krävs arbete på flera olika håll med olika sorters verktyg. Dessa ska sedan kombineras och integreras med varandra på ett sätt som skapar ett helhetligt spel som uppfyller förutsatta krav. I detta kapitel presenteras de olika typer av arbeten som utförts och hur de integrerats med varandra för att skapa ett fullständigt spel.

4.1 Karaktärer

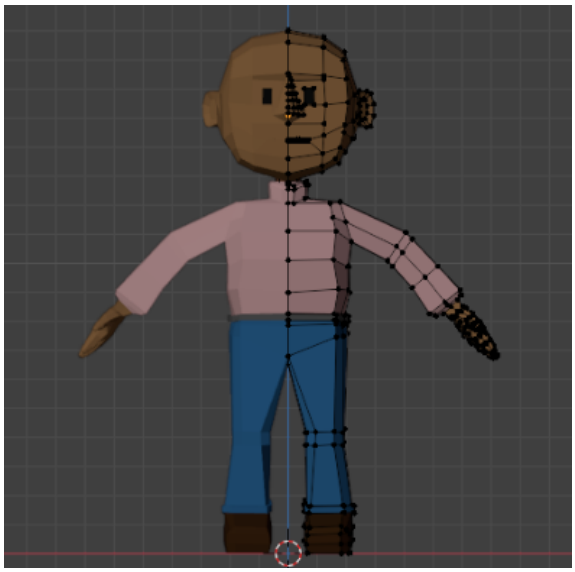
En huvuddel av spelet är karaktären som spelaren styr och alla de andra karaktärerna som spelaren kan interagera med under spelets gång. För att särskilja de olika karaktärerna i spelet kommer sido-karaktärerna att refereras till som NPCs (Non Playable Characters). Karaktären som spelaren styr kommer att refereras till som huvudkaraktären eller bara karaktären när det är tydligt vad som refereras. Användaren av spelet kommer att refereras till spelaren. Skapandet och integreringen av karaktärerna delades under arbetets gång in i två delar. Den första delen gick ut på att skapa karaktärerna i Blender och den andra delen gick ut på att importera och programmera dem i Godot. Respektive del presenteras nedan i detalj.

4.1.1 Skapandet i Blender

Som nämnt i tidigare kapitel är Blender ett effektivt verktyg att arbeta i när 3D-karaktärer ska skapas. Det finns möjligheter att skapa avancerade karaktärer som kan bli verklighetstrogna och har många detaljer. Eftersom arbetet med spelet var tidsbegränsat och relativt omfattande i andra aspekter, gjordes ett val att inte lägga ner alltför mycket tid på att modellera karaktärerna. Detta innebar att det inte fanns tid att lägga för mycket fokus på detaljer, utan göra karaktärerna mer lågupplösta med färre polygoner. Karaktärerna får då ett enklare och kantigare utseende.

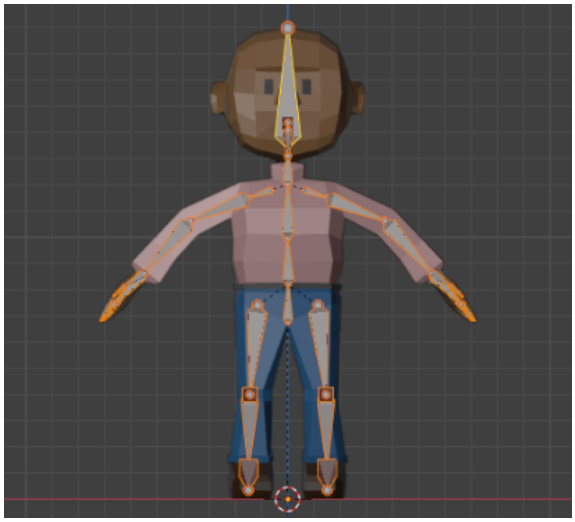
Arbetet med att skapa karaktärerna inleddes med en del experimenterande för att hitta den stil som skulle kunna anses passa spelet bäst. Eftersom spelet utspelar sig på olika campus tillhörande ett universitet ansågs det bäst att utforma karaktärerna så att de har människoformer. För att inte behöva lägga för mycket tid på att få proportioner helt rätt och verklighetstrogna utformades dock karaktärerna lite annorlunda jämfört med hur en riktig människa ser ut. Exempelvis är huvudet betydligt rundare och proportionerligt större än resten av kroppen, inspirerat av karaktärer som kan ses i exempelvis tecknade program. På så sätt blir karaktärerna unika för just detta spel och annorlunda proportioner blir mer ursäktbara.

Mest tid lades på den första karaktären som skapades och detta var för att få en bra bas som resterande karaktärer kan byggas på. Den första karaktären modellerades med enkla kläder som skulle vara lätta att sedan alterera och ändra färg på. Den skapades även utan hår, för att varje karaktär som skapas därefter skulle kunna få sin egen frisyr. Baskaraktären kan ses nedan i figur 4.1.



Figur 4.1: Baskaraktären

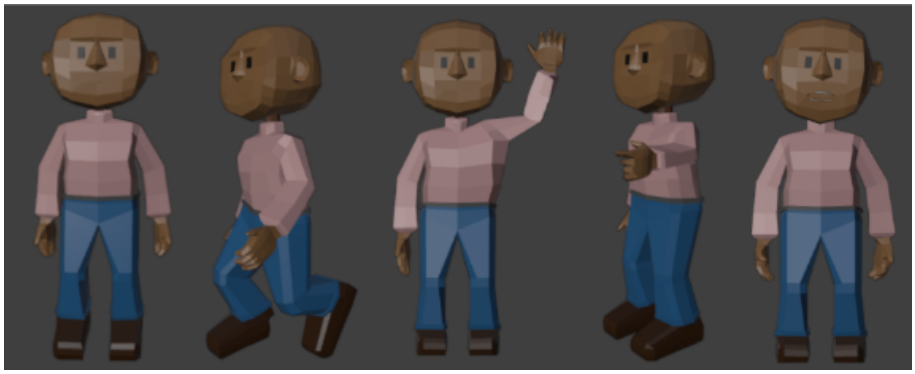
Efter att modelleringen av baskaraktärer var klar var nästa steg att lägga till en armatur, karaktärens skelett, och skapa några enkla animeringar som kan användas i spelet. Med hjälp av armaturen kan karaktären styras på olika sätt och animeras till att exempelvis prata, gå och vinka. Armaturen kan ses nedan i figur 4.2.



Figur 4.2: Baskaraktärens armatur

De animationer som skapades baserades på enkla rörelser som skulle kunna tänkas vara användbara i spelet. Dessa var följande:

- Idle: karaktären står i en vilande position och andas
- Walking: karaktären går
- Waving: karaktären vinkar
- Pointing: karaktären pekar framåt
- Talking: karaktärens pratar



Figur 4.3: Animationer

Eftersom det i spelet skulle finnas en NPC för alla de olika byggnaderna som kan hittas på Campus, behövdes cirka 25 karaktärer. Det hade tagit lång tid att skapa alla dessa karaktärer från grunden och ge varje karaktär sina egna armaturer och animationer. Istället användes baskaraktären, fullständig med modellering, armatur och animationer, för att skapa alla de NPCs som skulle ingå i spelet. Enkla förändringar gjordes på varje karaktär för att särskilja dem från varandra och ge dem ett eget utseende, men i grunden är de till stor del kopior av baskaraktären. Exempel på förändringar är att de fick olika tröjor, olika frisyrer, olika hudfärger och olika färger på kläderna. Ett urval av karaktärerna illustreras i figur 4.4 nedan.

Figur 4.4: Ett urval av NPCs i spelet

Det sista steget för karaktärerna i Blender var att exportera dem och göra dem till .glb filer, för att på så sätt vara kompatibla med Godot för att kunna importera dem

till spelet. Vid exporting följer hela karaktären med, inklusive armatur, animeringar och de paletter som används för att lägga till färger.

För att överföra 3D-modeller från Blender till Godot användes alltså filformatet `.glb` (GLTF Binary). Vid import i Godot konverteras `.glb`-filen till en `.import`-resurs och blir till en *inherited scene* med följande struktur:

- Mesh-objekt konverteras till `MeshInstance3D`
- Material från Blender kopplas till respektive mesh som `StandardMaterial3D`.

Detta gör att modellen kan användas direkt i spelet, inklusive korrekt positionering och material, utan ytterligare manuell konfiguration.

4.1.2 Implementering i Godot

Med karaktärerna exporterade från Blender, importerades de till Godot. Det första steget var att skapa en ny ärvd scen baserat på `.glb`-filen. Då skapas en `.tscn`-fil, varpå nya noder kan läggas till och karaktären kan vidareutvecklas i Godot. Inledningsvis är rotnoden en `Node3D`. Detta behövde ändras till `CharacterBody3D` på varje importerad karaktär, så att relevanta barnnoder skulle kunna läggas till. Efter detta steg skiljer sig hanteringen av huvudkaraktären och resterande NPC's och beskrivs var för sig nedan.

4.1.2.1 Huvudkaraktären

För att göra karaktären redo att användas i spelet behövdes ett antal noder läggas till. En sådan nod är `CollisionShape3D`, som används för att karaktären ska kollidera med andra ting i världen, såsom byggnader och andra karaktärer. Utan en sådan nod skulle karaktären kunna gå rakt igenom dem. Eftersom det är karaktären som spelaren kommer följa under spelets gång behövdes det också läggas till en `Camera3D` och ett par andra noder som hjälper till att kontrollera kameran. Det lades även till noder för att hjälpa karaktären känna av om den befinner sig inom ett område där handlingar kan äga rum, exempelvis starta en dialog med en NPC. Till sist lades en nod kallad `AnimationTree` till, som användes för att skapa smidiga övergångar mellan animationerna. Exempelvis undviks då en skarp övergång från när karaktären går från ett vilande läge till att börja gå.

Efter att noderna var på plats behövde huvudkaraktären sitt egna skript. Här kopplas exempelvis tangenterna som används till att styra karaktären. Vidare behövdes också kod som styr hur kameran följer karaktären och följer musens rörelse. I skriptet finns också funktionen `in_dialogue()` som låser karaktären under dialog med en NPC. På så vis kan spelaren inte lämna området under tiden en dialog förs. Efter avslutad dialog kallas en annan funktion (`end_dialogue()`) som låter karaktären röra sig igen.

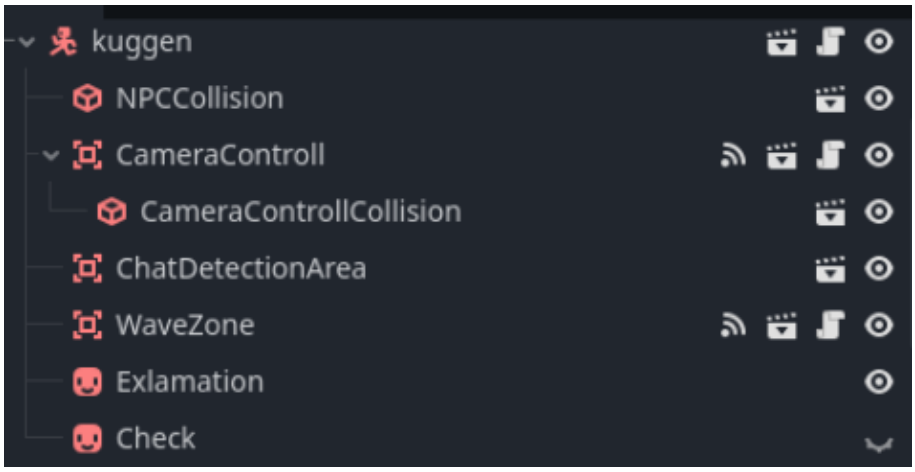
Med noderna på plats och ett färdigt skript var karaktären redo att användas och placerades ut i Campus-scenerna på den plats karaktären ska befinna sig vid spelets start.

4.1.2.2 NPC

Eftersom planen var att ha åtminstone en NPC för varje institution på Chalmers var det tydligt att många karaktärer skulle behöva importeras, där varje karaktär ska ha ett tillhörande skript och ett antal noder. För att undvika att behöva lägga till samma noder, om och om igen, och skriva ett likadant skript för varje karaktär, valdes ett mer modulärt tillvägagångssätt. Till att börja med skapades en mall för skriptet. På så vis behövde skriptet endast kodas en gång och sedan enkelt läggas till på varje ny karaktär. Den nya karaktären får då sitt eget skript, med den färdiga mallen som grund. Ska karaktären ha annan eller ytterligare funktionalitet i sitt skript, kan detta ändras och läggas till. I skript-mallen finns en funktion (`face_toward()`) som vänder NPCn mot huvudkaraktären när en dialog inleds och en funktion (`face_back()`) som vänder tillbaks NPCn efter avslutad dialog. Det finns också en funktion som hanterar animationen för NPCn, så att den exempelvis byter från viloläge till pratandes vid dialog eller börjar vinka när huvudkaraktären är i närheten

För att undvika att lägga till samma noder på alla NPC när de importeras skapades ett antal olika scener med de relevanta noderna. Till att börja med behövde varje NPC, liksom huvudkaraktären, en `CollisionShape3D`. När både huvudkaraktären och varje NPC har en sådan nod kan de inte gå rakt igenom varandra. Vidare behövdes också en scen som avgör om huvudkaraktären befinner sig inom ett område där en dialog kan inledas med en NPC. Denna scen består av en nod av typen `Area3D` och placeras framför varje NPC. Eftersom huvudkaraktären också har en sådan nod kan en funktion implementeras som avgör om dessa överlappar eller inte. Om de överlappar kan en dialog inledas med en NPC. Vid inledandet av en dialog behövdes ytterligare en scen som bland annat ser till att rätt kamera används under dialogen. Denna scen sätter också igång rätt dialog och kallar på relevanta funktioner. Den sista scen som delas av alla NPC's är en `Area3D` som används på liknande sätt som den för dialogen. Denna nod avgör om huvudkaraktären är inom ett visst område av en NPC. Om huvudkaraktären befinner sig inom området så kallas en funktion i skriptet som tillhör en specifik NPC och får den att börja vinka.

Genom ovan nämnda tillvägagångssätt är det relativt enkelt att lägga till en ny NPC. Det som behöver göras med varje specifik NPC är att lägga till en ny ärvd scen och ändra rotnoden till `CharacterBody3D`. Efter detta läggs skriptet enkelt till eftersom det är en färdig mall och NPCn kan läggas till i Campus-scenen. Där kan de tilldelas namn. I spelet är de döpta efter de institutioner de representerar. Sedan väljer man att instansiera en barn-scen (Instantiate Child Scene) och väljer de färdiga scenerna. Ett exempel på hur det ser ut i nod-trädet kan ses nedan i figur 4.5.



Figur 4.5: Exempel på NPC med noder

4.2 Spelplan

Spelplanen genererades med hjälp av Blender i kombination med tillägget Blosm, vilket möjliggör import av 3D-modeller från OpenStreetMap (OSM) – ett öppet, kollaborativt projekt som tillhandahåller geografisk data som vägar, byggnader och landskap, fritt tillgängligt för alla. Blosm genererar initialt ett antal mesh-strukturer baserade på denna data. De importerade byggnaderna, vägarna, vattenytorna, grönområdena och skogsområdena visualiseras i Figur 4.6.

Figur 4.6: Importerade mesh-objekt från Blosm: byggnader, vägar, vattenytor, grönområden och skogsområden.

4.2.1 Modifiering av den genererade geometrin

Den genererade OSM-modellen krävde modifieringar för att fungera i spelet. Dessa modifieringar presenteras nedan.

4.2.2 En uppdelning av byggnader

Byggnaderna färgades och modiferades på olika sätt enligt den grupp byggnaden tillhörde för att tydligt visa för spelaren vilka områden som är viktiga och därmed styra spelarens uppmärksamhet

- Campus byggnader
- Byggnader inom räckhåll till spelaren.
- Byggnader utom räckhåll för spelaren – dessa modifierades dessutom med en "Decimate"-modifierare i Blender, med en ratio på 0,3 för att minska antalet polygoner och därmed förbättra spelets prestanda.

4.2.3 Förhindring av Z-konflikt

Z-konflikt (Z-fighting) är ett vanligt grafiskt problem som uppstår när två eller flera ytor ligger mycket nära varandra att grafiksmotorn har svårt att avgöra vilken yta som ska visas framför den andra. Detta leder till att ytorna "flimrar" visuellt. För att undvika detta problem genomfördes följande höjdjusteringar i modellen:

- Alla grönområden fick tjockleken 0.1.
- Alla skogsområden fick tjockleken 0.2.
- Mindre viktiga vägar togs bort, medan de återstående modifierades för att ges tjocklek.

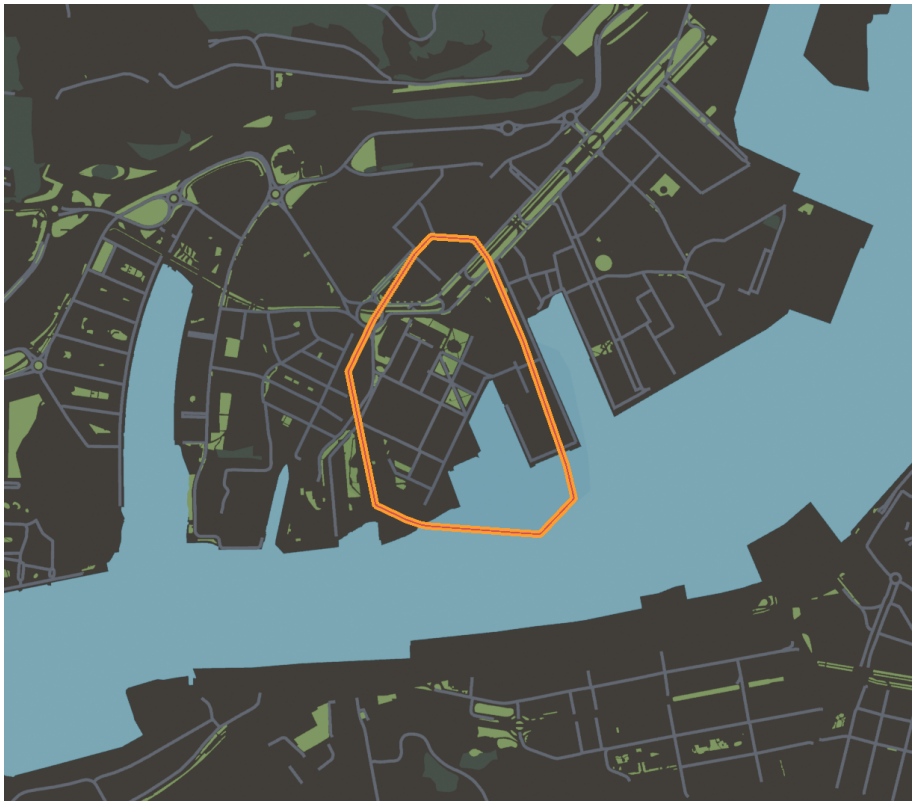
Vägarnas ursprungliga geometri bestod av tunna, plana polygoner utan volym vilket blev problematiskt. Vägarna gavs en tjocklek genom ett skript som skapar en volym mellan en nedre och en övre yta och fyllde ut sidorna däremellan.

4.2.4 Begränsning av spelområde

För att begränsa spelarens rörelseområde implementerades en osynlig barriär genom följande process:

1. **Insamling av punkter:** Alla hörnpunkter (vertex-koordinater) från de valda objekten samlas in.
2. **Beräkning av konvex hölje:** Punkterna sorteras och bearbetas med algoritmen `monotone chain` för att skapa en konvex polygon som bildar gränsen som omsluter en samling punkter, ungefär som ett tajt gummiband runt dem. Denna polygon, kallad konvex hölje, används för att beräkna den minsta ytan som innehåller alla punkter [12].
3. **Offset av polygon:** En offset-polygon skapas genom att förskjuta den konvexa polygonens kanter utåt. En inre polygon genereras också baserat på offset-polygonen för att definiera kantens tjocklek.
4. **Skapande av 3D-modell:** Med hjälp av de yttre och inre polygonerna byggs en tredimensionell struktur med botten- och toppytor samt väggar.

5. **Skapande av mesh i Blender:** Slutligen genereras ett nytt mesh-objekt i Blender med namnet `pass_border-colonly`.



Figur 4.7: Kanten runt campus.

4.2.5 Namngivningssystem

Ett strukturerat namngivningssystem har implementerats för att underlätta import av objekt från Blender till spelmotorn Godot. Systemet bygger på att använda specifika prefix och suffix i objektnamn för att automatiskt styra hur objekten hanteras vid import. Följande konventioner tillämpas:

- Suffixet `-colonly` används för att markera objekt som endast ska generera kollisionsgeometri i Godot. Godot stöder automatiskt vissa suffix vid import, vilket innebär att objekt med detta suffix får korrekt kollisionsdata utan ytterligare manuell konfiguration.
- `Standard objekt` som saknar specifierade prefix och suffix tolkas som fysiska objekt som ska ha kollisionsdata.
- Prefixet `pass_` används för att identifiera objekt som inte ska ha kollisionsdata. Till denna kategori hör objekt grönområden, skogsområden, vägar och objekt som ska synas men som spelaren bör kunna passera igenom så som bron mellan Kemi och Fysik byggnaden på Johanneberg campus.
- Prefixet `arrow_` används för att identifiera en pil som ska hjälpa spelaren att gå i rätt rikting. Den ska behandlas som `standard objekt` men ska också kopplas till en funktion som ger spelaren ett poäng och visar en dialog när spelaren går över pilen.

Många objekt på spelplanen ska ha kollison och vara synliga. För detta används ett skript, `buildingManager`, som kopplas till varje campus-scen och går igenom varje `MeshInstance3D` objekt.

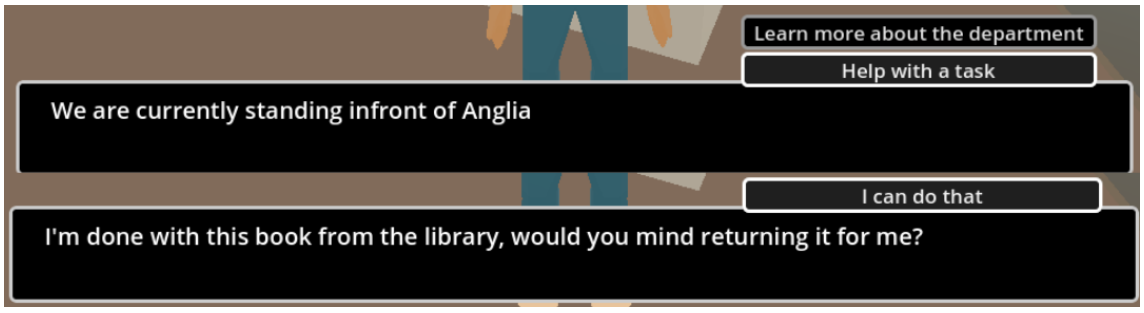
4.3 Övriga implementeringar

Utöver karaktärerna och spelplanen behövdes ett antal andra funktioner läggas till för att ge spelaren olika möjligheter under spelets gång. En sådan funktion var en möjlighet att transportera huvudkaraktären mellan de två olika campuserna, Lindholmen och Johanneberg. För att åstadkomma detta skapades en enkel busskur i Blender. Denna placerades ut på respektive campus där det i verkligheten går att ta en buss mellan Johanneberg och Lindholmen. Busskuren fick bland annat en `Area3D`-nod för att kunna avgöra när spelaren befinner sig inom räckhåll för busskuren. När spelaren befinner sig inom detta område dyker en text upp som säger `Press F to change campus`. Om spelaren trycker på F byts nuvarande campus-scen mot den andra och spelaren kan fortsätta utforska det andra campuset.

Utöver möjligheten att byta campus under spelets gång lades ytterligare funktioner till, vilka presenteras nedan.

4.3.1 Spelifiering

För att ge spelaren ytterligare motivation till att spela spelet, utöver spelets funktion att lära spelaren om Chalmers, lades ett poängsystem till. Detta innebär att spelaren kan samla poäng genom att utföra vissa uppgifter i spelet. Exempelvis får spelaren ett poäng då de väljer att lära sig mer om institutionen de besöker. Utöver detta finns det också ett antal NPCs i spelet som behöver hjälp med någon typ av uppgift. Ett exempel är den uppgift som syns i figur 4.8, där en NPC behöver hjälp med att lämna tillbaka en bok till biblioteket. Ett annat exempel är att en NPC ber spelaren lämna en kopp kaffe till en annan NPC och informerar om vilken byggnad den andra NPCn befinner sig utanför. Tanken med dessa uppgifter är dels, som tidigare nämnt, att ytterligare motivera spelaren till att spela spelet, men också att vidare lära spelaren lokalisera sig på Lindholmen och Johanneberg.



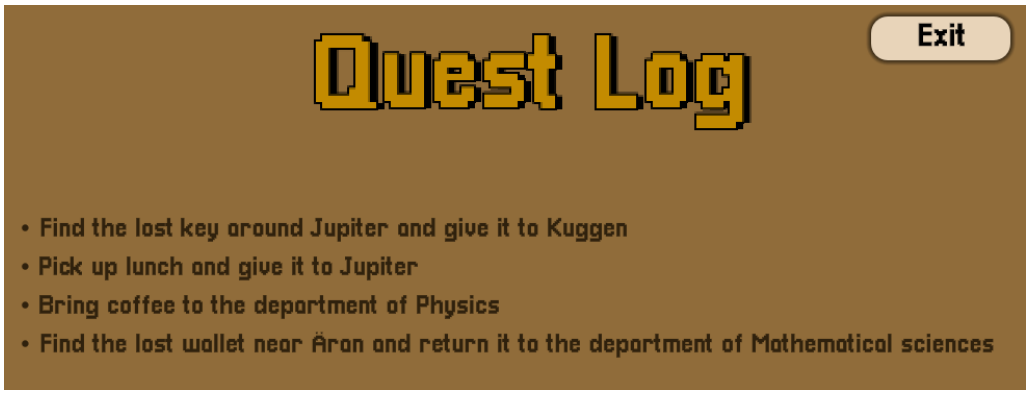
Figur 4.8: Exempel på en uppgift som spelaren kan utföra

Om spelaren väljer att hjälpa till med uppgiften i figur 4.8 läggs en bok till i spelarens förråd (inventory), som illustreras i figur 4.9. Spelaren får sedan gå till

ett av Chalmers bibliotek, där det finns en box för spelaren att lämna tillbaka boken. När boken är tillbakalämnad får spelaren ett poäng. På motsvarande sätt kan spelaren hitta en nyckel eller en plånbok på marken och plocka upp dem. De hamnar då i förrådet och spelaren behöver hitta den NPC som tappat nämnda föremål.

Figur 4.9: Spelarens förråd

Om spelaren väljer att samla på sig flera uppdrag innan de utförs, kan det vara svårt att komma ihåg vad uppdragen går ut på. För att hjälpa spelaren hålla reda på uppdragen finns det en uppdragslogg (quest log), som går att se i figur 4.10. Om spelaren inte har några uppdrag står det `No active quests`. När spelaren får uppdrag och de är aktiva läggs en beskrivning av uppdraget till i loggen. De försvinner då uppdraget är slutfört och när alla tillgängliga uppdrag är slutförda står det `You have finished all quests`.



Figur 4.10: Uppdragsloggen

För att hålla koll på all logik tillhörande uppdragen och poängen används tre skript. Poängen sparas i `gameState.gd` och presenteras i paus-menyn (paus-menyn presenteras nedan). Det är även från denna paus-meny som spelaren hittar sitt förråd. För att spelaren upprepade gånger ska kunna samla poäng genom att lära sig om samma institution om och om igen, sparas en lista i `gameState`. Denna lista håller koll på vilka NPCs spelaren har interagerat med. När spelaren interagerar med en NPC genomsöks listan, om det är första gången spelaren interagerar med NPCn får spelaren

ett poäng. När det gäller uppdragen samlas informationen i skriptet `MiniQuests.gd`. Här finns bland annat information om vad spelaren har i sitt förråd, vilka uppdrag som är aktiva och vilka som är avslutade. Skriptet innehåller även information om varje uppdrag, som har ett unikt id och en tillhörande beskrivning. Till sist används skriptet uppdragsloggens egna skript `quest_log.gd`, som med hjälp av `miniQuest` uppdaterar uppdragsloggen som spelaren kan se (figur 4.10).

4.3.2 Dialoger

För att göra dialoger till karaktärerna i Godot används ett plugin som heter `Dialogue Manager 2`. Detta plugin erbjuder ett färdigt exempel på hur det kan se ut med den textbubbla som dyker upp under en dialog och hur dialogerna kan kodas. Därefter går det att anpassa efter eget önskemål. I figur 4.8 kan textbubblan ses och ett exempel på hur en dialog kan skrivas i kod syns nedan. Det inleds med ett namn som identifierar vilket dialog som ska startas. I detta fall är varje enskild dialog döpt efter de namn som givits till varje enskild NPC. Den text som inte har några tecken framför sig är det som NPCn säger. När det är ett bindestreck framför erbjuds val för spelaren att välja. Vidare kan olika globala skript kopplas till dialogfilen. Tack vare detta kan skriptens olika funktioner kallas och dess variabler nås från dialogfilen. I exemlet nedan kallas `add_score` i skriptet `GameState`. Exemplet som syns nedan är ett av de enklare, eftersom denna NPC inte erbjuder något uppdrag. Om det istället erbjuds ett uppdrag står det `Help with a task` istället för `End conversation`.

Listing 4.1: Exempel på en kodad dialog

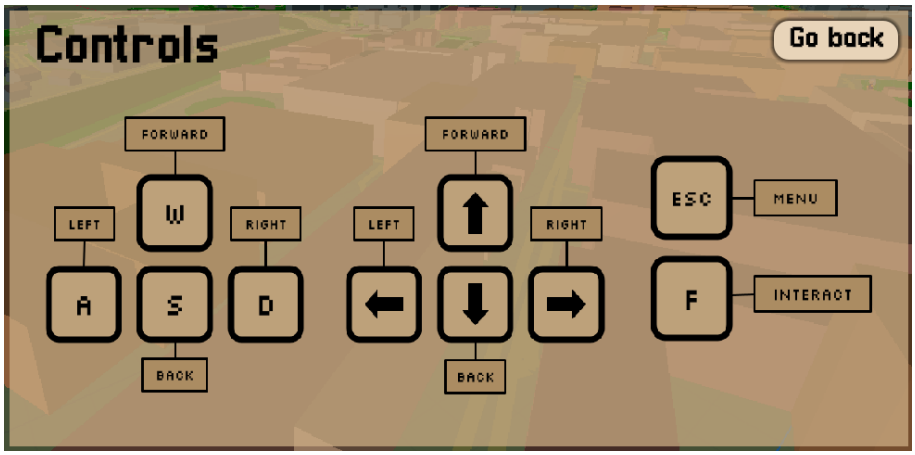
```
~ electrical
Oh hello there!

We are currently standing infront of the department of
Electrical Engineering
- Learn more about the department
        do show_extra_info(true, "electrical")
        if not talked_to_npcs.has("electrical")
                do add_score(1, "electrical")
        => END
- End conversation

do set_mouse_state(GameState.MouseState.GAMEPLAY)
=> END!
```

4.3.3 Menyer

Det första som dyker upp när spelaren startar spelet är en huvudmeny. I huvudmenyn står spelets namn, CampusQuest, och spelaren presenteras för fem olika olika knappar: `Start new game`, `Continue`, `Controls`, `Exit`, samt en symbol som representerar inställningar. Denna symbol kan ses uppe i vänster hörn i figur 4.11. Knapparna är programmerade i ett skript som är kopplat till rotnoden. Om spelaren trycker på `Exit` avslutas spelet. Om spelaren trycker på knappen `Controls` tas spe-

laren till en annan scen, vilken beskriver de olika tangenterna som används i spelet. Detta illustreras i figur 4.12. Från denna scen kan spelaren trycka en knapp, `Go Back`, som tar spelaren tillbaka till huvudmenyn. Har spelaren redan startat ett spel och sparat detta, kan spelaren återuppta detta spel genom att trycka på `Continue`. I detta fall tas spelaren direkt till en scen där spelaren får välja vilket campus denne vill ta sig till. När spelaren trycker på `Start new Game` påbörjas ett nytt spel. Om det finns ett tidigare sparat spel skrivs detta över när ett nytt spel påbörjas. Spelaren tas vidare till en annan scen, där spelaren kan välja hur huvudkaraktären ser ut. När spelaren trycker på symbolen som representerar inställningar kan spelaren välja att aktivera `Text-To-Speech` och ställa in volymen. Detta illustreras i figur 4.13.

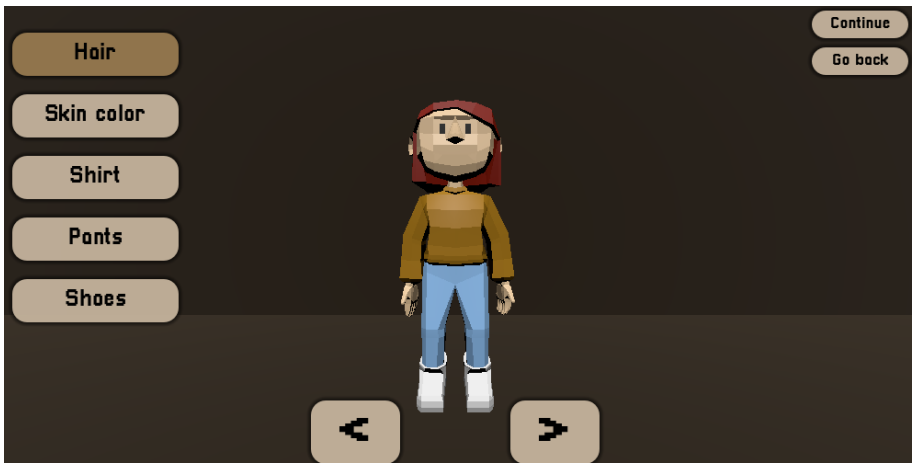
Figur 4.11: Spelets huvudmenyn



Figur 4.12: Spelets kontroller

Figur 4.13: Spelets inställningar

För att ge spelaren fler valmöjligheter i spelet, valdes det att lägga till en scen i spelet där spelaren själv kan välja hur huvudkaraktären ska se ut. Denna scen illustreras i figur 4.14. Scenen består av huvudkaraktären, samt ett antal valbara knappar: `Hair`, `Skin`, `Shirt`, `Pants`, `Shoes`, `Go Back` och `Continue`. För att få rätt funktionalitet behövdes tre olika skript: det skript som tillhör huvudkaraktären, ett som lyssnar på vilken knapp som trycks in och kallar på funktioner i huvudkaraktärens skript och ett globalt skript som registrerar de slutgiltiga valen. Om spelaren vill gå tillbaka till huvudmenyn väljs `Go Back`. När `Hair` är valt kan spelaren välja olika, förbestämda, hårfrisyrer. När en av de andra kategorierna (`Skin`, `Shirt`, `Pants`, `Shoes`) är valda kan spelaren välja färg i respektive kategori. När spelaren är nöjd med sina val och trycker på `Continue`, sparas det som har valts i varje kategori i form av en integer, som representerar ett index, i det globala skriptet. Indexet används sedan i huvudkaraktärens skript för att uppdatera utseendet på huvudkaraktärerna som är utplacerade på Johanneberg och Lindholmen.



Figur 4.14: Val och anpassning av karaktär

När spelaren är klar med att välja utseende på huvudkaraktären, tas spelaren till en scen där spelaren får möjlighet att börja utforska på Johanneberg eller på Lind-

holmen, vilket går att se i figur 4.15 När spelaren gjort sitt val byts scenen till det valda campusets scen.

Figur 4.15: Val av Campus

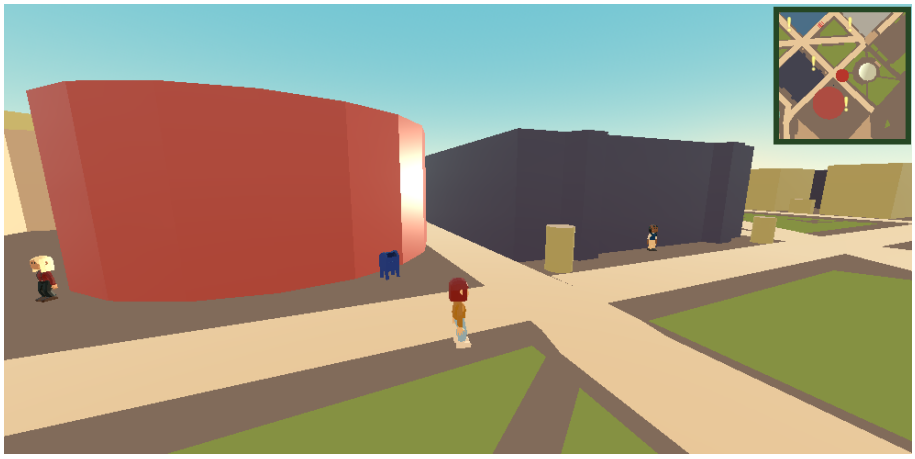
Innan spelet kommer igång går det att byta fram och tillbaka mellan ett antal olika meny-scener. Det går också att komma till samma scen från olika scener. Exempelvis går att komma till scenen där campus väljs från scenen där huvudkaraktärens utseende bestäms och från huvudmenyn om spelaren väljer att fortsätta ett sparat spel. Är spelaren i scenen där campus väljs och väljer att gå tillbaka behöver spelet veta vilken som är korrekt föregående scen. Detta hålls koll på genom ett enkelt globalt skript (`SceneManager.gd`) med en variabel `previous_path`. Innan byte av scen sparas nuvarande scens sökväg under `previous_path`. När spelaren i nästa scen väljer att gå tillbaka används denna variabel i SceneManager för att komma tillbaka till korrekt föregående scen.

Under spelets gång har spelaren möjlighet att pausa spelet och får då upp den paus-meny som illustreras i figur 4.16. Respektive campus-scen har en nod som är av typen `CanvasLayer`. Paus-menyn är en egen scen som lagts till som barnnod till `CanvasLayer`. Under spelets gång är denna nod osynlig och blir synlig först när spelaren väljer att pausa spelet. Här kan spelaren se sina nuvarande poäng, välja att fortsätta spela eller välja att avsluta spelet. Om spelaren väljer att återgå till spelet blir paus-menyn återigen osynlig.

Figur 4.16: Paus-meny

4.3.4 Kartor

Eftersom syftet med spelet bland annat är att hjälpa nyanställda lokalisera sig på Chalmers två campus, togs beslutet att lägga in två olika typer av kartor i spelet. En av kartorna är en form av minikarta (mini map) som ovanifrån visar var spelaren befinner sig och de byggnader som finns inom en viss radie. Minikartan illustreras i figur 4.17. Denna karta är hela tiden närvarande uppe i högra hörn under spelets gång. I scenen för minikartan har rotnoden ett enkelt skript med en referens till huvudkaraktären och en referens till den relevanta kameran. Genom skriptet följer kameran huvudkaraktären ovanifrån vart karaktären än går.



Figur 4.17: Mini-karta uppe i högra hörn

Den stora kartan visar också ovanifrån var spelaren befinner sig, men visar hela campuset som spelaren befinner sig på. Denna kartan illustreras i figur 4.18. Kartan täcker hela fönstret för att ge spelaren en bättre överblick av var denne befinner sig på campus. Här finns även möjlighet för spelaren att se vilka relevanta byggnader som finns på kartan. Detta gör spelaren genom att låta muspekaren vara över en av byggnaderna. Om byggnaden tillhör Chalmers dyker en text upp över denna byggnad som talar om vilket namn byggnaden har. I figur 4.18 befann sig muspilen över byggnaden Saga.

Figur 4.18: Stora kartan över Lindholmen

Båda kartorna har samma struktur. De består av rotnod, som sedan har en `SubViewportContainer`, som i sin tur har en `Subviewport` och den har i sin tur en `Camera3D`. Eftersom kamerorna behöver ha olika positioner för respektive campus är kartorna inte egna scener. Istället består de av noder direkt inlagda i respektive campus scen. Dessa noder ligger under samma föräldernod som paus-menyn, alltså `CanvasLayer`. Minikartan är synlig när spelaren inte befinner sig i en meny, och blir osynlig då paus-menyn aktiveras. Den stora kartan är tvärtom osynlig under spelets gång och dyker upp först när spelaren väljer att ta fram den från paus-menyn.

För att göra det tydligare på kartorna var NPCs befinner sig, fick varje NPC två noder av typen `Sprite3D`. Dessa har två olika bilder: ett gult utropstecken, som till en början är synligt, och en gul bock, som till en början är osynlig. Dessa kan ses i 4.18. När spelaren interagerar med en NPC och lär sig mer om institutionen byts utropstecknet ut mot bocken. Detta görs genom att en funktion (`change_mark()`) kallas i NPCns skript, som gör utropstecknet osynligt och bocken synlig.

4.3.5 Ljud och musik

För att hantera de två ljudtyperna ljudeffekter och bakgrundsmusik används ett globalt skript `SoundManager` som är tillgängligt för samtliga noder för att spela upp ljudeffekter och styra bakgrundsmusik.

Ljudeffekter utgörs av korta ljudklipp, exempelvis klickljud eller poängljud, som spelas vid specifika händelser. Dessa hanteras genom att skapa temporära ljudspelare (`AudioStreamPlayer`) som automatiskt frigörs när ljudet är färdigt, vilket möjliggör uppspelning av flera ljudeffekter utan att påverka huvudmusiken.

Bakgrundsmusiken spelas kontinuerligt och hanteras via en dedikerad ljudspelare med stöd för att byta låt, pausa och göra volymjusteringar med hjälp av tweening. Musiken är kopplad till olika campus och menyer i spelet, och dess volym kan justeras dynamiskt beroende på spelets tillstånd, till exempel när en dialog spelas.

4.3.5.1 Pausmeny och återupptagning av musik

Pausmenyn aktiveras från båda campusen, vilket ställer krav på att `SoundManager` sparar den låt som spelades innan pausen. Detta görs genom att spara referensen till den aktuella musiken när paus musiken aktiveras genom `play_pause_music` och sedan återuppta samma låt när spelet fortsätter genom `exit_pause_music`.

4.3.5.2 Menyer och nivåmusik

Musiken i huvudmenyn, karaktärmenyn samt campusen Lindholmen och Johanneberg, hanteras på ett standardiserat sätt. Varje musikspår är förladdat och kan spelas med eller utan fade-in-effekt. Fade-in implementeras med hjälp av tweening. Tweening används för att skapa mjuka övergångar i volym, exempelvis vid musikbyte eller när dialog pågår och musiken ska dämpas. För att undvika att en pågående tween skriver över volymvärdet och orsakar oväntat låg ljudnivå bör alltid befintliga tweens avslutas innan en ny startas. Detta hanteras genom att kontrollera om en fade-tween finns och är aktiv, och i så fall avbryta den innan en ny tween påbörjas.

4.3.5.3 Don't Repeat Yourself (DRY)-principen i ljudvolym

För att undvika upprepningar och hårdkodning används variabeln `final_volume` som bas för all volymhantering. Dialogvolymen (`dialogue_volume`) beräknas dynamiskt utifrån `final_volume` genom att multiplicera den med en konstant faktor, vilket gör det enkelt att ändra volymen i spelet. Detta underlättar även framtida justeringar och underhåll av ljudkoden.

Sammanfattningsvis möjliggör denna struktur att musik och ljudeffekter kan anpassas efter spelaren och programmerarens behov, samtidigt som koden är lätt att underhålla.

4.4 Integrering av Godot-spelet i Chalmers Onboarding appen

För integrationen av spelet Campus Quest, utvecklat i Godot, med Flutter-applikationen Chalmers Onboarding användes exportprofilen `Web (HTML5)`. Den exporterade HTML-filen döptes till `index.html`. Detta rekommenderas eftersom `index.html` vanligtvis är den fil som webbservrar automatiskt hämtar, vilket förenklar inbäddningen till Flutter-projektet [13]. Exporten genererade bland annat `index.html`, en .(`.js`) fil, en (`.wasm`) fil samt en (`.pck`) fil. Dessa filer placerades i Flutter-projektets mappstruktur under `web/assets/godot/`.

4.4.1 Spelfönstrets hjälparklass

För att möjliggöra återanvändning och flexibilitet skapades en Flutter-widget, `GodotEmbedder`, som använder `HtmlElementView` från `dart:ui` (via `dart:ui_web`) för att rendera en inbäddad webbsida i en Flutter websida. I `GodotEmbedder` används

en hjälparklass, `GodotBridge`, som ansvarar för att skapa ett `<iframe>`-element samt möjliggöra kommunikation mellan Flutter och Godot.

Användning av `GodotEmbedder` kräver endast en import i den Flutter-widget där spelet ska visas:

```
import 'widgets/godot_embedder.dart';
```

Spelet kan därefter enkelt bäddas in med hjälp av följande kod:

```
children: [
  const SizedBox(height: 50),
  AspectRatio(
    aspectRatio: 2 / 1,
    child: GodotEmbedder(
      alignment: Alignment.center,
    ),
  ),
],
```

Användningen av `AspectRatio` är viktig eftersom `HtmlElementView` i Flutter Web måste ha layoutbegränsningar för att kunna renderas korrekt. Genom att ange ett bildförhållande (i detta fall 2:1) säkerställs att spelet följer krav på att ha en layoutbegränsning, men också att layouten anpassas dynamiskt efter fönstrets storlek.

4.4.2 Integration i Chalmers Onboarding appen

Integrationen genomfördes genom att bygga vidare på befintlig kodstruktur. Precis som andra sidor i appen, så som quiz-sidan, adderades en ikon till menyn som tar spelaren till "CampusQuest"-sidan vilket illustreras i figur 4.19.

I profil-sidan visas också antal poäng från spelet campusQuest på samma sätt som quiz spelets poäng vilket visas i figur 4.20.

Admin, en speciell typ av användare på Chalmers Onboarding, har dessutom tillgång till att redigera spelets dialoger på admin-sidan vilket visas i figur 4.14.

4.4.2.1 Databas

För lagring av speldata implementerades en ny databasstruktur med tabellen `game`, baserad på en tidigare etablerad kodbas.

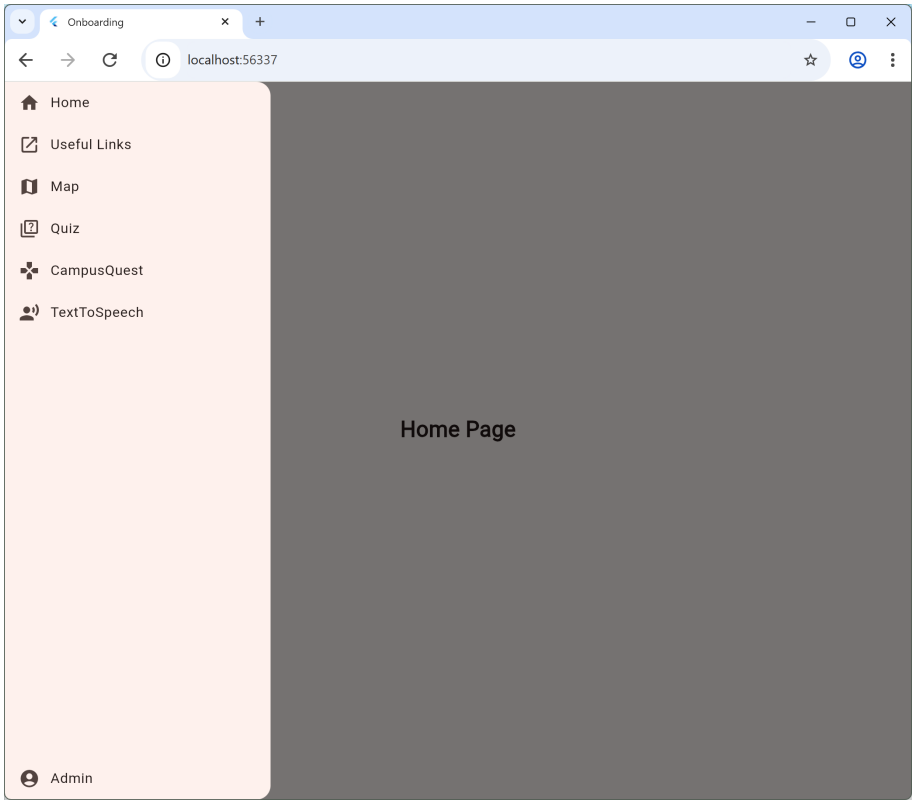
Arbetet följde den befintliga arkitekturen, med tydlig uppdelning mellan klient, server och databaslager. Kommunikation med backend sker via HTTP POST-anrop genom en hjälpfunktion i Flutter:

```
final data = await DatabaseWebApi.postRequest('/getGame', {'email':
email});
```

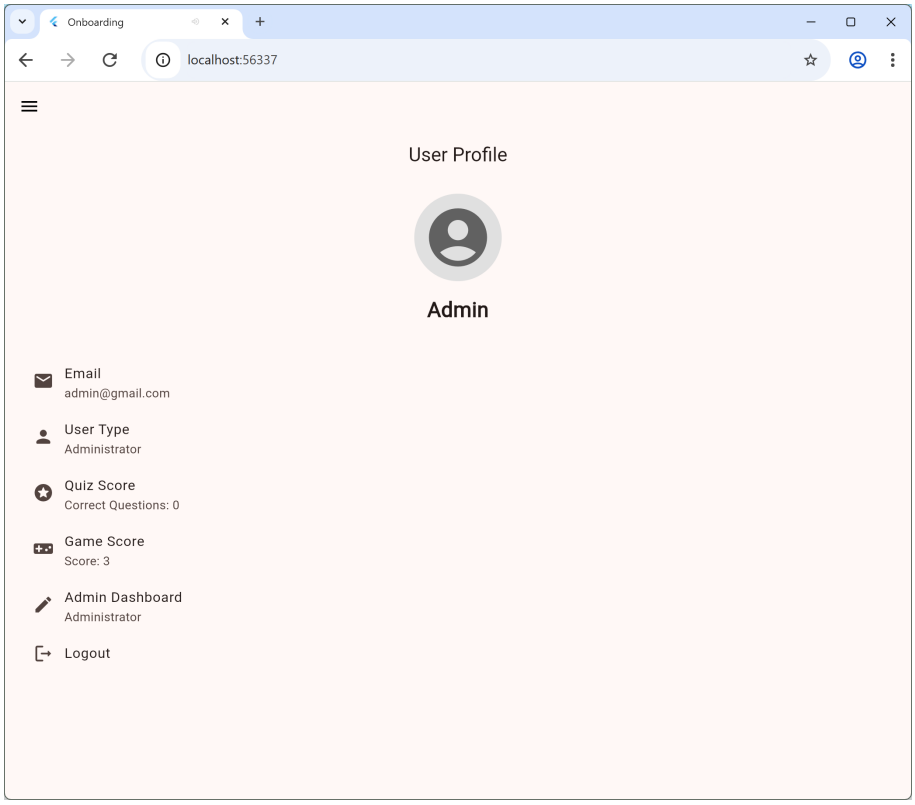
`DatabaseWebApi` är en existerande klass som erbjuder statiska metoder för att skicka HTTP-förfrågningar och hantera JSON-data. Den fungerar som gränssnitt mellan klienten och backend-servern.

Servern är uppbyggd med Alfred Dart-ramverket. Samtliga API-rutter registreras i `BackendServer`. Ett exempel på en sådan rutt är `/getGame`, som används för att hämta information om ett spel:

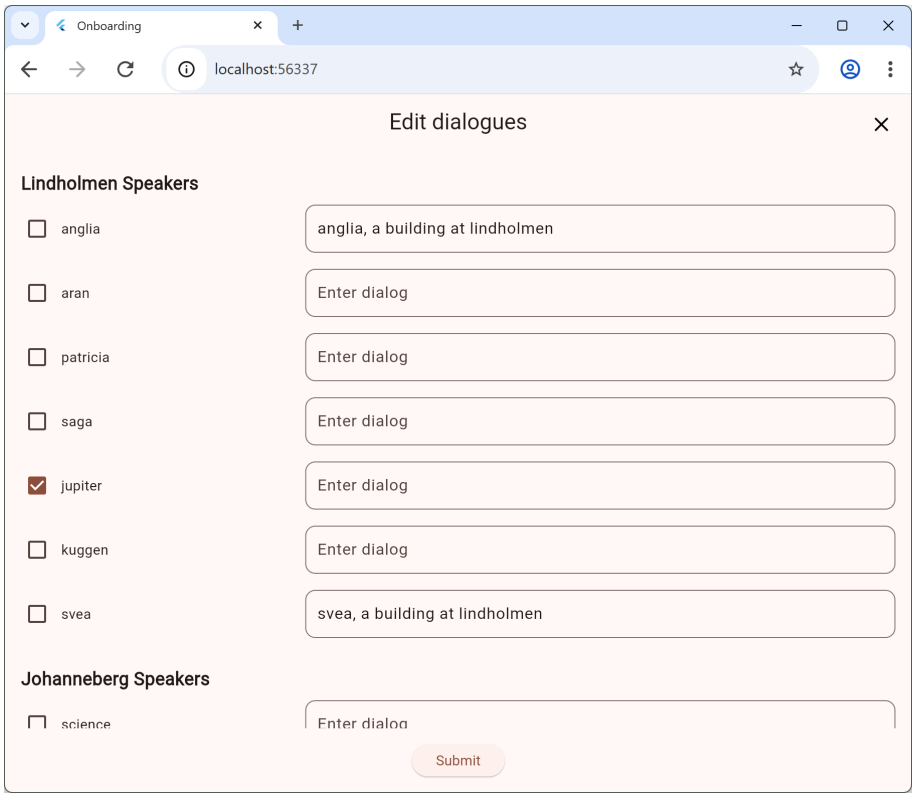
```
app.post('/getGame', GameController.getGame);
```



Figur 4.19: På sidomenun kan man som inloggad användare välja mellan start-sidan, en sida för användbara länkar, en sida för kartan, quiz-sida, “CampusQuest”-sidan eller TTS-sidan



Figur 4.20: Profilsidan för inloggade användare visar poäng för CampusQuest



Figur 4.21: Inloggade administratörer kan redigera spelets dialoger via adminsidan

När denna rutt anropas, hanterar `GameController.getGame` begäran enligt följande:

1. Innehållet i POST-förfrågan extraheras (t.ex. `email`).
2. En databasanslutning etableras med `PostgresDB.create()`.
3. Metoden `getGame(email)` anropas på databasinstansen. I denna metod körs SQL förfrågan `SELECT * FROM game WHERE email = @email`
4. Anslutningen stängs via `conn.closeConnection()`.
5. Ett svar returneras till klienten.

Övriga API-anrop registreras och hanteras på ett liknande sätt vilket skapar ett modulärt och återanvändbart flöde för datahantering, där information passerar från klienten via backend till databasen och tillbaka, utan förändringar i den övergripande arkitekturen.

4.5 Kommunikation mellan Flutter och det inbäddade Godot-spelet

För att möjliggöra tvåvägskommunikation mellan Flutter-applikationen och det inbäddade Godot-spelet i en `iframe` användes `postMessage`-API:t. `PostMessage` fungerar mellan olika system som Flutter och Godot och fungerade bi-direktionellt, det vill säga från Flutter till Godot och från Godot till Flutter.

I Flutter skapades klassen `GodotBridge` för att kommunicera mellan Godot, databasen och andra widgets. I Godot skapades klassen `FlutterBridge` för att kommunicera med Flutter och andra noder. Detaljerad implementering av `GodotBridge` och `FlutterBridge` återfinns i Appendix C och Appendix B.

4.5.1 Meddelandekonventioner

För att hålla kommunikationen organiserad och förutstägbar användes en namngivningskonvention:

- `request_` för anrop som förväntade svar (t.ex. `request_dialog`).
- `*_response` för svar på anrop.

4.5.2 JavaScriptBridge

För att kommunicera med Flutter behövde `FlutterBridge`:

- använda funktionen `postMessage`.
- använda JavaScript för att kunna använda `postMessage`.
- förlita sig på Godots `JavaScriptBridge` för att kunna använda JavaScript.

JavaScriptBridge är ett inbyggt globalt skript i Godot Engine som möjliggör användning av JavaScript-kod direkt från GDScript. Metoden `JavaScriptBridge.eval(js, true)` exekverar en sträng med JavaScript-kod [14].

4.5.3 Godot skickar meddelande vid spelstart

När spelet laddas i webbläsaren använder `FlutterBridge` JavaScriptBridge-funktionaliteten för skicka det första meddelandet till Flutter:

```
JavaScriptBridge.eval("window.parent.postMessage({ type: '
    godot_request' }, '*');", true)
```

Detta skickar ett `godot_request`-meddelande till föräldrafönstret (i detta fall `GodotBridge`).

4.5.4 Flutter tar emot och skickar meddelanden

`GodotBridge` lyssnar på meddelanden med hjälp av JavaScript-interop:

```
web.window.addEventListener('message', _onMessageEvent.toJS);
```

Om meddelandet innehåller `type: 'godot_request'`, svarar `GodotBridge` med ett `flutter_response`-meddelande tillbaka till `iframe`:en via:

```
void _send(Object msg) {
final win = _iframe.contentWindow;
if (win != null && msg is Map) {
win.postMessage(msg.jsify(), web.window.origin.toJS);
}
}
```

4.5.5 Godot tar emot meddelanden från Flutter

Innan `FlutterBridge.gd` kan ta emot meddelanden måste en referens till Godot-metoden som hanterar meddelanden läggas till. För att möjliggöra detta skapas objektet `window.godotBridge` i JavaScript. Objektet har två metoder `setCallback` och `sendMessage`. `setCallback` används från GDScript för att spara en callback och `sendMessage` kallar på callbacken:

```
window.godotBridge = {
    callback: null,
    setCallback: function(cb) { this.callback = cb; },
    sendMessage: function(msg) {
        if (this.callback) this.callback(msg);
    }
};
```

I GDScript hämtas först en referens till `godotBridge`-objektet med hjälp av:

```
var godot_bridge = JavaScriptBridge.get_interface("
    godotBridge")
```

Därefter skapas en JavaScript-callback från funktionen `_on_js_message()`:

```
js_callback = JavaScriptBridge.create_callback(_on_js_message
    )
```

Med `godotBridge`-objektet och `js_callback` kan man nu sätta den GDScript funktion man vill använda när `flutterBrdige` tar emot ett medelande.

```
godot_bridge.setCallback(js_callback)
```

En callback är en funktion som skickas som argument till en annan funktion. Den kan senare anropas (kallas tillbaka), t.ex. när ett meddelande tas emot. I det här fallet innebär det att Godot skickar med en funktion till JavaScript. Genom detta vet JavaScript vad den ska göra när den får ett meddelande från Flutter, det vill säga, den ska ropa på Godots funktion `_on_js_message()`.

4.5.6 Godot hanterar Flutter:s startmeddelande

När `FlutterBridge.gd` tar emot ett meddelande av typen `flutter_response` betyder det att Flutter-applikationen är redo att kommunicera. Hanteringen i GDScript ser ut på följande sätt:

- Variabeln `flutter_ready` sätts till `true` för att indikera att Flutter är initierad och redo att ta emot meddelanden.
- En signal `flutter_readied` skickas ut till de noder som väntar på att Flutter ska bli redo.

```
"flutter_response":
        flutter_ready = true
        emit_signal("flutter_readied")
```

4.5.7 Kompatibilitet och struktur

För att kommunikationen mellan Godot och Flutter ska fungera korrekt krävs att:

- meddelandetyperna matchar (`dialog_request`, `dialog_response` etc).
- datan är serialiserbar (Flutter stöder endast JSON-kompatibla typer).
- nycklar som `type"`, `speaker"`, och `data"` används i meddelanden.
- Eftersom endast primitiva typer och JSON-objekt stöds i `postMessage` måste Flutter-objekt som skickas serialiseras med `.toJson()`

4.5.8 Användning av dialogsystemet

Godot-noder kan nu begära dialogdata från Flutter med:

```
var dialog := await FlutterBridge.request_dialog("Fysik")
```

Förutom dialogsystemet kan Godot hämta speldata och spara spelet via Flutter, avsluta spelet och spela upp text-till-tal. Alla funktioner använder samma logik som dialogsystemet: först säkerställer de att variabeln `flutter_ready` är satt till `true`. Om Flutter ännu inte är redo avvaktar metoderna asynkront tills signalen `flutter_readied` sänds ut, vilket indikerar att Flutter är redo och kan kommunicera. Därefter kan ett request-meddelande skickas till Flutter genom postMessage och JavaScriptBridge. I detta fall ett `dialoge_request`-meddelande. Sedan väntar metoden på signalen, i detta fall `dialog_received`, som utlöses när ett response meddelande, `dialog_response`, kommer fram till FlutterBridge. Request metoden fortsätter efter att ha blivit väckt av signalen och returnerar dialog texten.

Kommunikationen mellan Godot och Flutter bygger på enkel, förutstägbar och asynkron logik baserad på `postMessage`. Godot skickar begäran, Flutter svarar, och

Godot behandlar svaret. Denna arkitektur är modulär och kan enkelt anpassas till andra interaktiva system mellan Godot spel och Flutter applikation.

4.6 Spara och ladda

I spelet kan spelaren, som ovan beskrivet, välja hur huvudkaraktären ska se ut under spelets gång. Spelaren kan också samla poäng genom att prata med olika NPCs, samla på sig olika objekt och utföra olika uppdrag. Spelaren kanske väljer att inte spela klart hela spelet i en omgång, utan stänger ner och startar upp spelet några gånger. Av denna anledning behöver de ovan nämnda aspekterna, samt annan logik sparas mellan spelomgångarna. På så vis behöver spelaren inte börja om varje gång spelet startar igen. Funktionerna att spara och ladda spelet igen sker i ett globalt skript (`gameState.gd`).

För att spara och ladda använder gameState.gd metoden från `flutterBridge`, `saveGame` och `getGame`. Dessa metoder skickar ett medelande till flutter på samma sätt som diskuterades i kommunikation sektionen med en medelande typ `save_request` och `get_request` respektive.

4.7 Skillnad mellan PC- och mobilversionen

Eftersom den redan befintliga appen spelet ska ingå i finns i en version för mobiltelefoner samt en version för PC, behövde även spelet vara anpassat för båda. Inledningsvis utvecklades den version som skulle användas för PC och själva grunden lades för spelet. Det som nämnts hittills i detta kapitel är det som utvecklats för PC. När grunden var lagd och de flesta av spelets funktioner var implementerades påbörjades arbetet med att anpassa spelet för att även kunna spelas på en mobiltelefon. För att veta vilken av versionerna som ska startas när spelaren startar spelet används en variabel (`is_mobile`) i det globala skriptet `GameState.gd`. När spelet startar ser skriptet efter huruvida spelet körs på en Android-enhet. Om den gör det sätts `is_mobile` till sann och andra omständigheter kommer gälla än om den är falsk.

En av de större förändringarna är hur kameran som följer huvudkaraktären är placerad. Eftersom det på en mobil inte går att styra hur kameran är vinklad med hjälp av musen, valdes istället att ha en ovanifrån-vinkel i mobilversionen. Detta illustreras i figur 4.22 Detta innebär att spelare endast ser karaktärerna och rekvisita ovanifrån, men det innebär också en mer pedagogisk överblick över campus. När det gäller kameran som används vid interaktion med NPC är det samma som i PC-versionen och spelaren får då se en NPC framifrån. Vilken kamera som ska vara aktiv bestäms baserat på variabeln `is_mobile` och styrs från huvudkaraktärens egna skript.

Som en konsekvens av att ha kameran helt ovanifrån, behövde kameran också höjas upp högre än i versionen för PC. Detta krävdes för att ge spelaren ett bättre perspektiv över respektive campus. När kameran höjdes upp upplevdes alla karaktärer och rekvisita som betydligt mindre än de gör i PC-versionen och det kan upplevas svårt att hitta dem. Detta löstes genom att skala upp samtliga NPC,

huvudkaraktären och rekvisita precis när spelet börjar. Om spelet inte spelas på en mobiltelefon behålls den ursprungliga storleken på allt.

Figur 4.22: Mobilversionen

När en spelare spelar CampusQuest på en mobiltelefon är tangentbordet inte längre ett alternativ att använda för att styra huvudkaraktären eller interagera med NPC och rekvisita. Det går heller inte längre att trycka på `esc` för att nå pausmenyn. För att lösa problemet med hur karaktären ska styras användes inledningsvis en typ av Joystick. Efter att ha testat detta sätt fram och tillbaka ansågs det inte vara ett särskilt smidigt sätt. Istället prövades att styra karaktären med `hålla och dra` (press and drag). Detta innebär att spelaren håller på skärmen och drar dit spelaren vill att huvudkaraktären ska gå. Detta styrs i huvudkaraktärens egna skript. Om variabeln `is_mobile` är sann lyssnar skriptet efter tryckningar och dragningar på skärmen, istället för tryck på tangentbordet. Denna metod visade sig vara ett bättre sätt för att styra huvudkaraktären och är den metod som i slutändan implementerades.

När spelaren öppnar spelet för första gången är det inte nödvändigtvis intuitivt hur huvudkaraktären styrs. För att hjälpa spelaren komma igång dyker en text upp på skärmen när spelaren precis har valt campus. Texten säger `press and drag to move`. Bredvid texten är en bild på en hand och pilar som är animerade att röra sig i en fyrkant för att göra det extra tydligt. Texten och bilden ses i figur 4.23. Så fort spelaren följer instruktionerna försvinner texten och animeringen. Den dyker sedan inte upp så länge spelaren fortsätter på samma sparade spel, utan bara när ett nytt spel startas.

Figur 4.23: Förklaring av hur huvudkaraktären styrs

Istället för att använda tangentbordet för att interagera med omgivningen och navigera till pausmenyn används knappar i mobilversionen. Dessa är endast synliga då `is_mobile` är sann. I PC-versionen av spelet dyker det upp text vid rekvisita för att tydliggöra att det går att interagera med dem. Detta blev inte lika pedagogiskt i mobilversionen, så den texten togs bort. Dock uppstår då problemet att det inte alltid är tydligt vad som går att interagera med. För att lösa detta är knappen för att interagera semi-transparent när det inte finns något att interagera med. När det istället går att interagera med något i omgivningen blir knappen helt opak och talar om för spelaren att det är ett alternativt att interagera med något huvudkaraktärens omgivning.

5
Resultat

Syftet med detta arbete var att skapa ett interaktivt spel för nyanställda på Chalmers. Målet med spelet var att det skulle vara ett överskådligt och inspirerande sätt för de nyanställda att få en insyn om Chalmers. Genom att kombinera lärande och spelifiering har spelet CampusQuest utvecklats, med syfte att på ett engagerat sätt hjälpa nyanställda lära känna Chalmers och dess två campus Lindholmen och Johanneberg. Spelet finns att nå genom den redan befintliga appen Chalmers Onboarding och kan i nuläget spelas på en dator. Spelet finns även i en version som går att spela på mobilen, genom Chalmers Onboarding. Inledningsvis beskrivs spelet med utgångspunkt att det spelas på datorn och efter detta beskrivs hur mobilversionen skiljer sig från datorversionen. Till sist beskrivs hur spelet har inkorporerats i den redan befintliga appen Chalmers Onboarding.

5.1 Spelets datorversion

När spelaren startar spelet möts denne av en startmeny. Spelaren kan exempelvis välja att gå in i inställningarna och bestämma om `Text-To-Speech` ska vara på under spelet. Det går även att ställa in volymen. Vidare kan spelaren också välja `Controls` där det beskrivs hur spelaren i spelet styr huvudkaraktären, interagerar med omvärlden i spelet och hur spelaren tar sig till pausmenyn. För att starta spelet finns två olika alternativ. Har spelaren redan startat ett spel sedan tidigare och vill fortsätta så väljer spelaren `Continue`, väljer vilket campus denne vill ta sig till och fortsätter sitt gamla spel. Om spelaren vill påbörja ett nytt spel väljer spelaren `Start new game`. Spelaren tas då till en meny där det går att anpassa huvudkaraktären som spelaren kommer att styra under spelet. I en artikel av Strmečki, Bernik och Radošević, nämnd i Teorikapitlet, lyftes flera viktiga element som kan tas i beaktning när lärande och spelifiering kombineras. Ett av dessa element var att kunna anpassa karaktärerna i spelet och har av bland annat denna anledning implementerats i spelet. Genom att kunna anpassa karaktären får spelaren aktivt vara med och bestämma delar av spelet, vilket kan bidra till högre engagemang och en upplevelse av delaktighet.

När spelaren känner sig nöjd med sin karaktär får spelaren välja vilket av campusen denne vill börja med att utforska, Lindholmen eller Johanneberg. Efter detta är spelaren fri att välja mellan flera olika saker att göra. Spelaren kan gå runt på campus och få en överblick över hur det ser ut. Den kan också välja att prata med olika karaktärer, NPCer, som representerar Chalmers olika institutioner. Där kan de välja att lära sig mer om institutionen eller hjälpa NPCn med ett uppdrag. Att ha

med uppdrag och att ha ett poängsystem är också två av elementen som nämndes i artikeln av Strmečki, Bernik och Radošević. De hjälper spelaren att ta sig framåt i spelet och motiverar att fortsätta spelet. Om spelaren lär sig mer om institutionen eller utför ett uppdrag får spelaren poäng. Uppdragen kan gå ut på att lämna över ett föremål till annan NPC på det andra campuset, eller hitta ett borttappat föremål nära en specifik byggnad.

Under tiden som spelaren spelar har denne tillgång till en pausmeny, med flera alternativ att välja. Spelaren kan välja att se uppdragsloggen, där alla aktiva uppdrag är presenterade. Spelaren kan också se i sitt förråd vilka objekt, relaterade till uppdragen, som denne har samlat ihop. Det går också att se nuvarande poäng. Till sist finns det alternativet att ta upp en stor karta över nuvarande campus. Här får spelaren en överblick över följande:

- Relevanta byggnaders namn.
- Var huvudkaraktären befinner sig.
- Var olika NPC befinner sig.

Utöver den stora kartan i pausmenyn finns det även en liten karta som går att se under tiden spelaren springer runt på campus, som är till för att hjälpa spelaren navigera sig.

5.2 Spelets mobilversion

Eftersom det är relativt stor skillnad att spela ett spel på datorn och på en mobiltelefon behövdes vissa förändringar göras för mobilversionen av spelet. Exempelvis går det inte att styra karaktären med hjälp av tangentbordet och det går inte att styra kameran med hjälp av musen. Det går heller inte att interagera med omgivningen med hjälp av tangentbordet. För att göra mobilversionen mer pedagogisk för spelaren valdes att ha en ovanifrån-vinkel, där kameravinkeln är helt fast. Det är då lättare att navigera huvudkaraktären på campus och få en bra överblick. För att styra huvudkaraktären får spelaren trycka på skärmen och dra fingret i den riktning som huvudkaraktären ska gå. För att lösa problemet med interaktion finns det i mobilversionen en knapp spelaren trycker på för att interagera med omgivningen. Denna knapp är semitransparent när inget finns att interagera med, för att bli helt opak när alternativet att interagera finns. Det finns även en knapp för att komma till pausmenyn, istället för att spelaren ska trycka på `esc`.

5.3 Spelets integration i appen Chalmers Onboarding

Ett av projektets mål var att integrera ett interaktivt spel i en redan existerande applikation utan att störa den befintliga arkitekturen. Detta gjordes genom att följa den redan existerande strukturen i ChalmersOnboarding projektet. Eftersom widgets och databasen blev kodade på samma sätt, både estetiskt och funktionellt kunde CampusQuest sömlöst interageras i ChalmersOnboarding appen.

5.4 Kommunikation mellan Flutter och Godot

Implementeringen av tvåvägskommunikation mellan Flutter och det inbäddade Godot-spelet resulterade i en lösning där båda systemen kan utbyta information på ett modulärt och asynkront sätt. Följande mål uppnåddes:

- **Tvåvägskommunikation etablerades:** Det blev möjligt att skicka meddelanden både från Godot till Flutter och från Flutter till Godot genom användning av Flutters `GodotBridge` och Godots `JavaScriptBridge`.
- **Flutter och Godot kunde synkronisera tillstånd:** När spelet initierats kunde Godot identifiera när Flutter var redo att ta emot meddelanden genom en `flutter_response`-signal, vilket möjliggjorde kontrollerad initiering.
- **Ett modulärt dialogsystem implementerades:** Godot-noder kunde be om dialogdata från Flutter på ett standardiserat sätt med `request_dialog`, vilket visar att kommunikationen kan användas för spelrelaterad funktionalitet.
- **Ytterligare funktionalitet som sparning, uppspelning av text-till-tal och avslutning av spel möjliggjordes:** Dessa funktioner använder samma kommunikationsmönster med `GodotBridge` och `JavaScriptBridge`.

Sammantaget visar resultaten att den implementerade kommunikationslösningen är både stabil och flexibel nog för att användas i Chalmers Onboarding och även framtida projekt. Arkitekturen är modulär, och det är enkelt att lägga till fler funktioner i framtiden med samma struktur.

6
Slutsats

Syftet med detta arbete var att skapa ett lärorikt och engagerande spel för nyanställda på Chalmers. Spelet skulle ge dem en inblick i Chalmers, dess campus och institutioner. Vad som nu har utvecklats är ett interaktivt spel, där den nyanställde får styra en karaktär och upptäcka Chalmers två campus, Johanneberg och Lindholmen. Genom att spela spelet får spelaren en möjlighet att bekanta sig med relevanta byggnader och de kan få information om Chalmers alla institutioner. Dessa moment var grundkraven för spelen och bidrar till att den nyanställde bitvis kan ta del av information om Chalmers i egen takt.

De tydligt lärorika delarna med spelet har kombinerats med spelifiering. Forskning har visat att spelifiering och lärande tillsammans kan leda till bättre inlärningsförmåga och bättre kritiskt tänkande. Genom att inkorporera vissa element i spelet, kan det bidra till ökat engagemang. Exempel på sådana moment som ingår i spelet är poängsystem och uppdrag. När poäng introduceras i ett spel kan det öka motivationen att spela. Även uppdrag kan leda till ökad motivation och även hjälpa spelaren ta sig framåt i spelet. Alla uppdrag i spelet har på något sätt att göra med att hitta på Chalmers två campus, vilket hjälper spelaren lära känna området.

6.1 Vidareutveckling

Något som i framtiden kan implementeras i spelet är en social faktor. Exempelvis skulle en topplista kunna införas, där en spelare kan se andra spelares poäng. Detta kan tilltala den tävlingsinriktade spelaren. Det skulle också kunna implementeras någon form av meddelandefunktion där spelare kan prata med varandra, eller en möjlighet för spelare att mötas i spelet.

I detta arbete fanns det inte tidsmässigt utrymme för att låta andra testa spelet på ett systematiskt och kontrollerat sätt. I framtiden hade det varit fördelaktigt att forma någon form av grupp som får testa spelet och komma med åsikter. På så vis kan spelet anpassas efter målgruppen och bli bättre.

Litteraturförteckning

[1] O. Carson and P. D. Advocate, "Comparing popular game engines." PubNub blog, Aug. 2024. Accessed: 2025-06-16.

[2] Godot Engine Contributors, "Gdscript basics — godot engine (4.3) documentation," n.d. Accessed: 2025-06-16.

[3] Creative Bloq Contributors, "The best 3d modelling software." Creative Bloq, n.d. Accessed: 2025-06-16.

[4] L. R. Murillo-Zamorano, J. Á. López Sánchez, A. L. Godoy-Caballero, and C. Bueno Muñoz, "Gamification and active learning in higher education: is it possible to match digital society, academia and students' interests?," *International Journal of Educational Technology in Higher Education*, vol. 18, pp. 1–27, 2021.

[5] D. Strme&269;ki, A. Bernik, and D. Rado&353;evi&263;, "Gamification in e-learning: Introducing gamified design elements into e-learning systems," *Journal of Computer Science*, vol. 11, pp. 1108–1117, Feb 2016.

[6] J. Zeng, D. Sun, C.-K. Looi, and A. C. W. Fan, "Exploring the impact of gamification on students' academic performance: A comprehensive meta-analysis of studies from the year 2008 to 2023," *British Journal of Educational Technology*, vol. 55, no. 6, pp. 2478–2502, 2024.

[7] Godot Engine Contributors, "Introduction — godot engine (stable) documentation," 2024. Accessed: 2025-05-08.

[8] Godot Engine Contributors, "Key concepts overview — godot engine (stable) documentation," 2024. Accessed: 2025-05-08.

[9] Godot Engine Contributors, "Script — godot engine class reference," 2024. Accessed: 2025-05-08.

[10] Blender Foundation, "Blender features," 2024. Accessed: 2025-05-08.

[11] Blender Foundation, "License — blender," 2024. Accessed: 2025-05-08.

[12] A. Andrew, "Another efficient algorithm for convex hulls in two dimensions," *Information Processing Letters*, vol. 9, no. 5, pp. 216–219, 1979.

[13] Godot Engine Contributors, "Exporting for the web — godot engine (4.3) documentation," n.d. Accessed: 2025-05-26.

[14] Godot Engine Contributors, "Javascriptbridge — godot engine (4.3) documentation," n.d. Accessed: 2025-05-26.

A

Databastabell för speldata

Nedan visas SQL-skriptet för att skapa tabellen `game`, som lagrar all spelrelaterad information kopplad till varje användare:

```
CREATE TABLE game (
    id SERIAL PRIMARY KEY,
    -- Auto-incrementing ID

    email VARCHAR(255) NOT NULL UNIQUE,
    -- Email must be unique

    score INTEGER NOT NULL,
    -- Integer score

    bug_state JSONB NOT NULL,
    -- Bug quest status (Map<String, dynamic>)

    talked_to_npcs JSONB NOT NULL,
    -- NPC dialog state (Map<String, bool>)

    inventory JSONB NOT NULL,
    -- Inventory (Map<String, dynamic>)

    picked_up_items JSONB NOT NULL,
    -- Picked up items (List)

    food_orders JSONB NOT NULL,
    -- Food orders (List)

    player_appearance JSONB NOT NULL,
    -- Player appearance (Map<String, int>)

    tts_enabled BOOLEAN NOT NULL DEFAULT FALSE,
    -- TTS state

    FOREIGN KEY (email)
        REFERENCES accounts(email)
        ON DELETE CASCADE
);
```

B

Godot: Kommunikation med Flutter via `FlutterBridge`

Beskrivning

Följande GDScript används i Godot för att kommunicera med det inbäddade Flutter webbgränssnittet via `JavaScriptBridge`. Det möjliggör funktioner som dialoghämtning, speldataöverföring, TTS-uppspelning och sparfunktioner:

Källkod

```
extends Node

var web_mode := OS.has_feature("web")
var flutter_ready := false
var tts_active := false

signal extra_info_ended
var running_extra_info := false

signal flutter_readied
signal game_saved(success: bool)
signal dialog_received(speaker: String, dialogues: String)
signal game_received(game_data: Dictionary)

var js_callback

func _ready():
if Engine.is_editor_hint():
return
_setup_js_bridge()

func _setup_js_bridge():
if not web_mode:
print("[Godot]  JavaScriptBridge not available.")
return
```

```
var js_code := """
window.godotBridge = {
callback: null,
setCallback: function(cb) { this.callback = cb; },
sendMessage: function(msg) {
if (this.callback) this.callback(msg);
}
};

window.addEventListener('message', function(event) {
const message = event.data;
if (!message?.type) return;

switch (message.type) {
case 'dialog_response':
case 'flutter_response':
case 'speakers_request':
case 'game_response':
case 'save_response':
if (window.godotBridge) {
const json = JSON.stringify(message);
window.godotBridge.sendMessage(json);
}
break;
}
});
"""
JavaScriptBridge.eval(js_code, true)

var godot_bridge = JavaScriptBridge.get_interface("godotBridge")
js_callback = JavaScriptBridge.create_callback(_on_js_message)
godot_bridge.setCallback(js_callback)

JavaScriptBridge.eval(
"window.parent.postMessage({ type: 'godot_request' }, '*');", true)
print("[Godot]  Godot bridge is ready!")

func request_dialog(speaker: String) -> String:
if not web_mode:
print("[Godot]  Web platform not available.")
return ""

if not flutter_ready:
await flutter_readied

var js := "window.parent.postMessage({ type: 'dialog_request',
```

```
    speaker: '" + speaker + "' }, '*');"
JavaScriptBridge.eval(js, true)

var args = await dialog_received
var result = args[1] if args != null else ""
return result

func request_game() -> Dictionary:
if not web_mode:
print("[Godot]  Web platform not available.")
return {}

if not flutter_ready:
await flutter_readied

var js := "window.parent.postMessage({ type: 'game_request' }, '*');"
JavaScriptBridge.eval(js, true)

var data = await game_received
return data

func save_game(data: Dictionary) -> bool:
if not web_mode:
print("[Godot]  Web platform not available.")
return false

if not flutter_ready:
await flutter_readied

var payload = {
"type": "save_request"
}
payload.merge(data, true)

var json_str := JSON.stringify(payload)
var js := "window.parent.postMessage(" + json_str + ", '*');"
JavaScriptBridge.eval(js, true)

var result = await game_saved
return result

func quit_game():
if not web_mode:
print("[Godot]  Web platform not available.")
return false
```

```
if not flutter_ready:
await flutter_readied

var js = "window.parent.postMessage({ type: 'quit_game'}, '*');"
JavaScriptBridge.eval(js)

func play_tts(text: String, speaker: String) -> void:
if not tts_active:
return

if not web_mode:
print("[Godot]  Web platform not available.")
return

if not flutter_ready:
return

var js := "window.parent.postMessage({ type: 'play_tts',
    text: '" + text + "' ,
    speaker: '" + speaker + "' }, '*');"
JavaScriptBridge.eval(js, true)

var greeting_variation := 0
var greetings := ["hello", "hi", "hey there", "Hi friend", "Greetings"]
func play_greeting(speaker: String) -> void:
greeting_variation = (greeting_variation + 1) % greetings.size()
play_tts(greetings[greeting_variation], speaker)

func _on_js_message(args: Array):
if args.size() == 0:
print("[Godot]  No arguments passed from JS")
return

var json_str: String = args[0]
var parsed = JSON.parse_string(json_str)

if parsed is Dictionary:
match parsed.get("type", ""):
"flutter_response":
print("[Godot]  Flutter is ready!")
flutter_ready = true
emit_signal("flutter_readied")
"dialog_response":
var dialog_data: String = parsed.get("data", "")
var speaker: String = parsed.get("speaker", "Unknown")
if dialog_data is String:
```

```
print("[Godot]  Dialog received for speaker:", speaker)
emit_signal("dialog_received", speaker, dialog_data)
"game_response":
var game_data = parsed.get("data", {})
if game_data is Dictionary:
print("[Godot] User data received")
emit_signal("game_received", game_data)
"save_response":
var success = bool(parsed.get("success", false))
print("[Godot] Save game ACK received:", success)
emit_signal("game_saved", success)
else:
print("[Godot]  Failed to parse JSON")
```

C

Flutter: Kommunikation med Godot via GodotBridge

Beskrivning

Denna appendix visar källkoden för klassen `GodotBridge`, som fungerar som bryggan mellan Flutter-appen och det inbäddade Godot-spelet (HTML5-export). Klassen hanterar initiering av iframe-elementet för spelet, mottagning och tolkning av meddelanden från Godot via `postMessage`-API:t samt skickar svar eller data tillbaka. Den används bland annat för att hantera dialoger, hämta och spara speldata, samt text-till-tal-funktionalitet.

Källkod

```
import 'dart:async';
import 'dart:js_interop';
import 'package:web/web.dart' as web;
import 'package:flutter/foundation.dart';

import '../../backend-webserver/db_web_api.dart';
import '../../main.dart';
import '../../models/game_model.dart';
import '../../models/npc_model.dart';
import '../text_to_speech_manager.dart';

///to use this you simply import
///import 'widgets/godot_bridge.dart';
///and then use
///GodotBridge().requestSpeakers()
///or
///GodotBridge().fetchNPC(speaker)
class GodotBridge {
  static final GodotBridge _instance = GodotBridge._internal();
  late TextToSpeechManager _ttsManager;

  factory GodotBridge() {
    return _instance;
```

```
}

GodotBridge._internal();

late final web.HTMLIFrameElement _iframe;
Completer<List<String>>? _speakersCompleter;
Completer<void>? _shutdownCompleter;

bool _isInitialized = false;

void init({required TextToSpeechManager ttsManager}) {
  _ttsManager = ttsManager;
  print('[flutter godotbridge] Initializing GodotBridge...');
  if (!kIsWeb) {
    print('[flutter godotbridge] Not running on web platform');
    return;
  }
  if (_isInitialized) {
    print('[flutter godotbridge] Already initialized');
    return;
  }
  _isInitialized = true;

  print('[flutter godotbridge] Creating iframe element');
  _iframe = web.HTMLIFrameElement()
    ..src = 'assets/godot/index.html'
    ..style.border = 'none'
    ..style.width = '100%'
    ..style.height = '100%';

  print('[flutter godotbridge] Adding message event listener');

  web.window.addEventListener('message', _onMessageEvent.toJS);

  print('[flutter godotbridge] Initialization complete');

}

web.HTMLIFrameElement get iframe => _iframe;

Future<void> _speak(String text) async {
  //this can be expanded to support voices for each character
  Map voice = {
    "name": "Microsoft Hazel - English (United Kingdom)",
    "locale": "en-GB"
  };
```

```
  _ttsManager.setVoice(voice);
  _ttsManager.startVoice(text);
  await Future.delayed(Duration(milliseconds: 200));
}

void _onMessageEvent(web.Event ev) async{
  print('[flutter godotbridge] Received message event: ${ev.type}');

  if (ev is web.MessageEvent && ev.data is JSObject) {
    print('[flutter godotbridge] Processing MessageEvent with JSObject data');
    final map = (ev.data as JSObject).dartify() as Map;
    final type = map['type'];
    print('[flutter godotbridge] Message type: $type, data: $map');
    ///use request_* only for commands that expect a reply.
    /// Use *_response for replies.
    switch (type) {
      case 'godot_request':
        print('[flutter godotbridge] Godot is ready, sending flutter_response');

        _send({'type': 'flutter_response'});

        break;
      case 'speakers_response':
        print('[flutter godotbridge] Received speakers response');
        final List<dynamic> arr = map['data'] ?? [];
        print('[flutter godotbridge] Speakers list length: ${arr.length}');
        _speakersCompleter?.complete(List<String>.from(arr));
        break;
      case 'dialog_request':
        final speaker = map['speaker'] ?? 'default';
        print('[flutter godotbridge] Received dialog request for
        speaker: $speaker');
        _sendDialog(speaker);
        break;
      case 'game_request':
        print('[flutter godotbridge] Received game request');
        _sendGame(loggedInUser!.email);
        break;
      case 'save_request':
        print('[flutter godotbridge] Received save_request signal');
        _saveGame(map, loggedInUser!.email);
        break;
      case 'quit_game':
        print('[flutter godotbridge] Received quit_game signal');
        _shutdownCompleter?.complete();
```

```
      case 'play_tts':
        final text = map['text'] ?? 'default';
        print('[flutter godotbridge] Received TTS message: $text');
        await _speak(text);
        break;

      default:
        print('[flutter godotbridge] Unknown message type: $type');
    }
  } else {
    print('[flutter godotbridge] Non-MessageEvent or non-JSObject data');
  }
}

void _send(Object msg) {
  print('[flutter godotbridge] Preparing to send message: $msg');
  final win = _iframe.contentWindow;
  if (win != null && msg is Map) {
    print('[flutter godotbridge] Sending message to iframe');
    win.postMessage(msg.jsify(), web.window.origin.toJS);
  } else {
    print('[flutter godotbridge] Cannot send message);
  }
}

Future<List<String>> requestSpeakers() {
  print('[flutter godotbridge] Requesting speakers list');
  if (_iframe.contentWindow == null) {
    print('[flutter godotbridge] Error: Iframe not ready');
    throw StateError('[GodotBridge] Iframe not ready');
  }
  _speakersCompleter = Completer<List<String>>();
  _send({'type': 'speakers_request'});
  return _speakersCompleter!.future;
}

Future<void> _sendDialog(String speaker) async {
  print('[flutter godotbridge] Sending dialog for speaker: $speaker');
  var dialog = await _fetchDialog(speaker);
  if (dialog == null || dialog.isEmpty) {
    print('[flutter godotbridge] No dialog found for speaker: $speaker');
    dialog = "You are now standing before a building on Chalmers campus.
    Its purpose is not immediately clear,
    but something about it suggests there's more to discover
    if you take a closer look around";
  }
```

```
  final message = {
    'type': 'dialog_response',
    'speaker': speaker,
    'data': dialog,
  };
  print('[flutter godotbridge] Sending dialog response: $message');
  _iframe.contentWindow?.postMessage(message.jsify(), web.window.origin.toJS);
}

Future<String?> _fetchDialog(String speaker) async {
  print('[flutter godotbridge] Fetching dialog for speaker: $speaker');
  try {
    final raw = await DatabaseWebApi.postRequest('/getNPC',
    {'speaker': speaker});
    if (raw is List && raw.isNotEmpty) {
      final npc = NPCObject.fromJson(raw.first);
      return npc.dialog;
    }
    print('[flutter godotbridge] No NPC data found for speaker: $speaker');
  } catch (e) {
    print('[flutter godotbridge] Error fetching dialog: $e');
  }
  return null;
}

Future<void> _sendGame(String email) async {
  Map<String, dynamic> message;

  if (email == null) {
    print('[flutter godotbridge] Cannot send game - email is null');
    message = {
      'type': 'game_response',
      'status': 'no_user_error',
    };
  }
  else {
    print('[flutter godotbridge] Sending game for email: $email');
    final game = await _fetchGame(email);
    if (game == null) {
      print('[flutter godotbridge] No game found for email: $email');
      message = {
        'type': 'game_response',
        'status': 'no_save',
      };
    }
```

```
    else {
      message = {
        'type': 'game_response',
        'data': game.toJson(),
      };
    }
  }

  print('[flutter godotbridge] Sending game response: $message');
  _iframe.contentWindow?.postMessage(message.jsify(), web.window.origin.toJS);
}


Future<GameObject?> _fetchGame(String email) async {
  print('[flutter godotbridge] Fetching game');
  try {

    final raw = await DatabaseWebApi.postRequest('/getGame', {'email': email});

    if (raw is List && raw.isNotEmpty) {
      final game = GameObject.fromJson(raw.first);
      return game;
    }
    print('[flutter godotbridge] No game data found for email: $email');
  } catch (e) {
    print('[flutter godotbridge] Error fetching dialog: $e');
  }
  return null; //if there is no game
}

Future<void> _saveGame(Map map, String email) async {
  try {
    print('[flutter godotbridge] Starting saveGame...');

    final score = map['score'];
    if (score is! int) throw Exception('Invalid or missing score');

    final isInformed = map['is_informed'];
    if (isInformed is! bool) throw Exception('Invalid is_informed flag');

    final bugState = map['bug_state'];
    if (bugState is! Map || bugState.keys.any((k) => k is! String)) {
      throw Exception('Invalid bug_state structure');
    }

    final talkedToNpcs = map['talked_to_npcs'];
```

```
if (talkedToNpcs is! Map || talkedToNpcs.keys.any((k) => k is! String)) {
  throw Exception('Invalid talked_to_npcs list');
}

final inventory = map['inventory'];
if (inventory is! Map || inventory.keys.any((k) => k is! String)) {
  throw Exception('Invalid inventory structure');
}

final pickedUpItems = map['picked_up_items'];
if (pickedUpItems is! List || pickedUpItems.any((e) => e is! String)) {
  throw Exception('Invalid picked_up_items list');
}

final foodOrders = map['food_orders'];
if (foodOrders is! List || foodOrders.any((e) => e is! String)) {
  throw Exception('Invalid food_orders list');
}

final startedQuests = map['started_quests'];
if (startedQuests is! List || startedQuests.any((e) => e is! String)) {
  throw Exception('Invalid started_quests list');
}

final finishedQuests = map['finished_quests'];
if (finishedQuests is! List || finishedQuests.any((e) => e is! String)) {
  throw Exception('Invalid finished_quests list');
}

final playerAppearance = map['player_appearance'];
if (playerAppearance is! Map) {
  throw Exception('Invalid player_appearance map');
}

final requiredKeys = ['shirt', 'hair', 'skin', 'pant', 'shoes'];
for (final key in requiredKeys) {
  if (!playerAppearance.containsKey(key) || playerAppearance[key] is! int) {
    throw Exception('player_appearance missing or invalid: $key');
  }
}

final ttsEnabled = true; //TODO remove hardcode

final saveData = {
  'email': email,
  'score': score,
```

```
        'is_informed': isInformed,
        'bug_state': bugState,
        'talked_to_npcs': talkedToNpcs,
        'inventory': inventory,
        'picked_up_items': pickedUpItems,
        'food_orders': foodOrders,
        'started_quests': startedQuests,
        'finished_quests': finishedQuests,
        'player_appearance': {
          'shirt': playerAppearance['shirt'],
          'hair': playerAppearance['hair'],
          'skin': playerAppearance['skin'],
          'pant': playerAppearance['pant'],
          'shoes': playerAppearance['shoes'],
        },
        'tts_enabled': ttsEnabled,
      };

      print('[flutter godotbridge] Valid game data, sending to
      backend: $saveData');

      final response = await DatabaseWebApi.postRequest('/saveGame', saveData);
      print('[flutter godotbridge] Response from backend: $response');

      _send({'type': 'save_response', 'success': true});
    } catch (e, stack) {
      print('[flutter godotbridge] Error parsing or saving game
      data: $e\n$stack');
      _send({'type': 'save_response', 'success': false, 'error': e.toString()});
    }
  }
```

```
  Future<void> waitForGodotShutdown() async {
    // If a shutdown is already being awaited (not yet completed)
    // return that Future.
    // This prevents multiple listeners from creating multiple Completers.
    if (_shutdownCompleter != null && !_shutdownCompleter!.isCompleted) {
      return _shutdownCompleter!.future;
    }
    else {
      // Create a new Completer which will be completed externally
      // (e.g., when a "shutdown complete" signal is received from Godot).
```

```
      _shutdownCompleter = Completer<void>();
    }
    // Await the Completer's future - execution will pause here
    // until _shutdownCompleter!.complete() is called from somewhere else.
    await _shutdownCompleter!.future;
    // After the shutdown is confirmed,
    send a response to Godot.


    // Function returns here
    // once the shutdown is complete
    // and the response is sent.
  }
}
```

INSTITUTIONEN FÖR DATATEKNIK
CHALMERS TEKNISKA HÖGSKOLA
Göteborg, Sverige
www.chalmers.se

CHALMERS