



UNIVERSITATEA DIN
BUCUREȘTI

FACULTATEA DE
MATEMATICĂ ȘI
INFORMATICĂ



SPECIALIZAREA INFORMATICĂ

Lucrare de licență

PLANIFICATOR PENTRU DESIGNUL INTERIOR AL LOCUINȚEI

Absolvent

Ciașu Nicoleta

Coordonator științific

Conf. dr. Mihai Prunescu

București, iunie 2022

Rezumat

Deși numărul **proprietarilor** care apelează la un specialist în arhitectura de interior este în creștere, marea majoritate a oamenilor încă aleg să își decoreze pe cont propriu locuința, apelând la procedee tradiționale, precum pixul și hârtia, deoarece aplicațiile software dedicate proiectării, precum SketchUp sau AutoCAD, sunt destinate uzului profesionist.

Aplicația propusă vine în sprijinul acestui segment de piață, propunând o platformă care să ușureze etapa de planificare pentru cei ce aleg să își amenajeze singuri locuința. **Prin aplicație**, în prima etapă, utilizatorii vor stabili structura locuinței la nivel de pereți și împărțire în camere, având posibilitatea inclusiv de a crea etaje. În continuare, aceștia pot alege dintr-o colecție variată de reprezentări simbolice ale obiectelor de mobilier uzuale, ale căror dimensiuni, poziții, sau rotații pot fi modificate. În final, utilizatorii pot alege să printeze pe suport hârtie planurile lor, sau să le salveze într-un format **incarcabil** și reeditabil ulterior. Implementată cu tehnologii web, aplicația este la îndemâna oricui, indiferent de specificațiile calculatorului, sistemul de operare, sau de cunoștințe în domeniul arhitecturii în interior.

Abstract

Even though the percentage of people which hire an interior designer when planning their homes is on the rise, the vast majority of people choose to design their rooms by themselves. More often than not, people end up having to resort to traditional planning methods, such as the pen and paper, as most design software, such as SketchUp or AutoCAD, is targeted towards professionals and comes with a steep learning curve.

Arcada is an application which aims to ease the process of home planning, by proposing a platform which enables home users to design the interiors of their homes efficiently. Users are free to employ the layout of their homes, by drawing the structure of the walls and dividing it into rooms. Afterwards, they can choose from a varied collection of usual furniture representations, which they are free to scale, move, or rotate as they see fit. Finally, they can choose to print their floor plans, or save them in a format which allows future loading and modification. Emphasis was put towards accessibility by building the application using a web stack, thus making it accessible to anyone with an internet connection on their computers.

Cuprins

1	Introducere	5
1.1	Contextul problemei	5
1.2	Scopul și ideea proiectului	5
1.3	Structura proiectului	6
1.4	Structura lucrării	6
2	Tehnologii utilizate	7
2.1	Alegerea tehnologiilor	7
2.2	Typescript	8
2.3	Document Object Model	8
2.4	PixiJS	9
2.5	React.js	11
2.6	Zustand	12
2.7	Express.js	13
2.8	MongoDB	14
2.9	Mongoose	14
3	Dezvoltarea aplicației	16
3.1	Arhitectura proiectului	16
3.2	Modulul interfață	16
3.3	Modulul editor	20
3.3.1	Manipularea etajelor	21
3.3.2	Adăugarea mobilei	21
3.3.3	Stocarea informațiilor despre pereți	23
3.3.4	Aplicarea transformărilor asupra obiectelor din plan	27
3.3.5	Proportionalitatea elementelor din plan	30
3.3.6	Vizualizarea detaliată a planului	30
3.4	Modulul de serializare	31
3.4.1	Salvarea. Serializare	33
3.4.2	Încărcarea. Deserializare	34
3.5	Server-ul	36

3.5.1	Comunicarea între client și server	38
3.6	Comunicarea între modulele aplicației	38
4	Ghid de utilizare al aplicației	40
4.1	Adăugarea elementelor în plan	41
4.1.1	Desenarea pereților	41
4.1.2	Modificarea pereților	43
4.1.3	Adăugarea mobilierului	45
4.1.4	Modificarea mobilierului	47
4.1.5	Adăugarea elementelor structurale	48
4.1.6	Ștergerea elementelor din plan	49
4.2	Gestionarea etajelor	49
4.3	Unelte ajutătoare	50
4.3.1	Vizualizarea planului: zoom si panning	51
4.3.2	Măsurarea dimensiunilor din plan	51
4.3.3	Facilitarea alinierii elementelor	51
4.3.4	Afișarea și ascunderea dimensiunii obiectelor	51
4.3.5	Modul ajutor	51
4.4	Salvarea și încărcarea planului	51
5	Concluzii	54
	Bibliografie	55

Capitolul 1

Introducere

1.1 Contextul problemei

Arhitectura de interior este ocupația aflată la granița dintre știință și artă ce se ocupă cu planificarea și amenajarea tuturor spațiilor închise, de la case, apartamente, până la clădiri de birouri și restaurante.

Deși majoritatea spațiilor comerciale beneficiază de sprijinul specialiștilor, numărul celor care aleg să apeleze la un serviciu profesionist de design interior este foarte mic. Un sondaj al Houzz Home din 2015 sugerează că numai 17% dintre cei 430.000 respondenți au ales să folosească un astfel de serviciu în scopul amenajării locuințelor proprii [15].

Cum activitățile precum renovarea sau decorarea spațiului propriei locuințe au loc frecvent[3], apare în mod natural întrebarea: ce mijloace are utilizatorul casnic la dispoziție pentru a-și planifica locuința?

1.2 Scopul și ideea proiectului

Având în minte această observație, am decis să dezvolt Arcada, o aplicație ce aduce utilizatorului casnic posibilitatea de a-și crea, organiza și plănui schița locuinței proprii. Arcada se deosebește de majoritatea aplicațiilor existente pe piață care tratează acest subiect prin orientarea către accesibilitate, funcționalități esențiale, intuitivitate, trăsături care facilitează obținerea unor schițe utile într-un timp scurt.

Arcada permite utilizatorilor să creeze întreagă configurație a casei lor, indiferent de numărul de etaje, și permite construirea majorității formelor și dispunerilor ale camerelor, atât timp cât sunt definibile prin poligoane. După ce utilizatorul îndeplinește așezarea și ajustarea pereților în plan, acesta poate alege dintr-o colecție ce conține numeroase reprezentări ale obiectelor de mobilier, amplasându-le, apoi modificându-le dimensiunile și rotația. Sistemul de măsurători la scară se asigură de faptul că atât obiectele, cât și pereții, respectă dimensiunile reale, din teren, pentru a garanta că planul poate fi

executat în practică. Deoarece planificarea unei locuințe este un proces de lungă durată, ce presupune multe schimbări și îmbunătățiri pe parcurs, există posibilitatea de salvare a planului, respectiv încărcarea sa pentru vizualizare sau reeditare.

Scopul proiectului este de a oferi acestei grupe de utilizatori o alternativă viabilă la aplicațiile tradițional folosite în designul interior, precum SketchUp[26] ori alte soluții tip CAD¹, aplicații care, deși foarte puternice, prezintă o curbă de învățare înaltă, fiind adresate utilizării în mediul profesional, fapt ce reprezintă o piedică în calea obținerii unor rezultate satisfăcătoare de către utilizatorii casnici.

1.3 Structura proiectului

Există patru module în componența Arcada:

- Modulul ce constă în zona principală a aplicației, numită în continuare „editor”, unde utilizatorul poate interacționa cu elementele constitutive ale planului în lucru.
- Interfața auxiliară editorului, ce înglobează toate modurile prin care utilizatorul îl poate manipula. Această se prezintă sub forma bării de unelte, localizată în stânga editorului. Prin această interfață, utilizatorul poate alege să adauge, editeze, sau șteargă diferitele elemente ale etajului curent, să parcurgă etajele, să încarce sau să salveze planul activ.
- Modulul ce gestionează serializarea și deserializarea părților constitutive ale planului, ce are ca scop formatarea informației prezentă în memoria de lucru a aplicației într-un format ce poate fi salvat și apoi reconstituit. Acest modul este folosit în cadrul funcționalităților de salvare și încărcare.
- Backend-ul aplicației, ce furnizează colecția de obiecte care pot fi prezente în planuri, și permite adaugarea, modificarea, sau ștergerea lor.

1.4 Structura lucrării

Voi începe prin a prezenta parcursul prin care am ales suita de tehnologii folosite în cadrul dezvoltării aplicației, punând în evidență metodele actuale prin care dezvoltatorii web pot crea experiențe interactive 2D. După aceea, va urma o descriere a tehnologiilor, urmând că mai apoi să prezint arhitectura aplicației, dedicând câte un subcapitol fiecărui modul și funcționalitate, evidențiind pe parcurs aspectele teoretice ale informaticii aplicate în proiect. După aceea, voi compune un ghid al aplicației, oferind scenarii de utilizare. În final, voi vorbi despre posibilitățile de extindere ale proiectului și impresiile personale.

¹CAD este un acronim pentru Computer Assisted Graphics, ce descrie grupul aplicațiilor care permit generarea de conținut grafic 2D sau 3D cu ajutorul calculatorului.

Capitolul 2

Tehnologii utilizate

2.1 Alegerea tehnologiilor

După identificarea scopului și al obiectivelor, a urmat o etapă de cercetare, în care am studiat plaja de tehnologii folosite în prezent în dezvoltarea de aplicații. În această etapă, am luat decizii cu privire la stack-ul software, la platforma țintită, urmând ca apoi să stabilesc arhitectura aplicației.

În concordanță cu scopul de a aduce aplicația cât mai aproape de utilizatorii casnici, am ales să dezvolt **Arcadă** folosind o suită de tehnologii pentru web. Aplicațiile web sunt foarte flexibile: pot fi rulate din browser, pot fi împachetate cu Electron¹, pentru a obține software executabil pe platformele desktop cele mai importante (Linux, Windows, macOS), sau cu Capacitor/Cordova², obținând build-uri pentru Android/iOS. Astfel, folosind aceeași bază de cod, aplicația are potențialul de a ajunge la orice utilizator care folosește un dispozitiv modern.

Programarea web este una dintre cele mai dinamice arii din industria software, iar structura unei aplicații web moderne diferă în mod fundamental de felul în care acestea erau dezvoltate acum 10-15 ani. Spre deosebire de mediile tradiționale de dezvoltare, unde metodele de dezvoltare s-au stabilizat, în web, acest lucru încă nu a avut loc[18], web-ul fiind unul dintre cele mai fragmentate medii din punct de vedere al adopției tehnologiilor.

Din toată plaja această variată de posibilități, se conturează, totuși, câteva stack-uri populare, compuse din tehnologii care se completează bine împreună. În urma acestui studiu, am ales stack-ul MERN (MongoDB, Express, React, Node.js) la care am adăugat PixiJS pentru partea de randare a elementelor grafice ce au nevoie de actualizări dese.

¹Electron este un framework care permite încapsularea aplicațiilor web în scopul generării de binare executabile pentru platformele desktop. <https://www.electronjs.org/>

²Cordova (<https://cordova.apache.org/>), și succesorul sau, Ionic Capacitor (<https://capacitorjs.com/>), sunt framework-uri pentru dezvoltarea de aplicații mobile, care încapsulează codul sursă scris cu tehnologii web în aplicații ce pot rula pe aceste platforme.

2.2 Typescript

TypeScript este un limbaj de programare ce extinde JavaScript, formând un superset al acestuia. Printre îmbunătățirile aduse de TypeScript se numără posibilitatea de a folosi tipuri statice, accesorii, interfețe, sau alte funcționalități utile pentru dezvoltarea aplicațiilor web. [11]

Prin aceste adăugări, pot fi detectate mai multe erori în etapa de pre-compilare, fapt ce reduce bug-urile, greșelile din timpul dezvoltării. Succesul acestei abordări este evidențiat de faptul că în prezent TypeScript este unul dintre cele mai populare limbaje folosite în dezvoltarea web, conform studiului 2020 State of JS efectuat de StackOverflow. [17]

TypeScript este un limbaj compilat, spre deosebire de JavaScript, care este unul interpretat. Browsersle nu suportă fișierele TypeScript în mod nativ. Pentru a permite utilizarea TypeScript pe client-side, fișierele sursă ale unei aplicații web sunt transcompilate în limbajul JavaScript folosindu-se TypeScript Compiler (tsc). [11]

Un alt avantaj compilării este că permite dezvoltatorilor să se folosească de funcționalitățile JavaScript-ului la standard modern (ES6) în cadrul codului lor sursă, urmând **că** mai apoi compilatorul să genereze cod de JavaScript compatibil cu standarde mai vechi (ES5, ES3), asigurând compatibilitate cu browserele vechi fără vreun efort în plus din partea dezvoltatorului.

Totuși, TypeScript nu este fără de cusur. De exemplu, una dintre limitările pe care am întâlnit-o în timpul dezvoltării aplicației este că limbajul nu suportă supraîncărcarea funcțiilor sau a operatorilor. Acest lucru este o consecință a transcompilării dintr-un limbaj strongly-typed într-unul weakly-typed. Informațiile despre tipurile de date ale variabilelor sunt folosite doar la compilare, la runtime apărând erori, deoarece interpreterul nu știe să diferențieze între semnăturile funcțiilor.

În cadrul Arcada, am folosit TypeScript pentru a dezvolta frontend-ul aplicației.

2.3 Document Object Model

Document Object Model (prescurtat „DOM”) este o interfață care reprezintă elementele unei pagini web sub forma unui arbore, permițând astfel operații de manipulare a acestora. Fiecărui element din HTML **ii** este asociat un nod, iar relațiile părinte-copil sunt reprezentate prin muchii. Prin implementarea acestei interfețe, orice limbaj de programare poate fi extins pentru a permite interacțiunea cu elementele prezente într-o pagină web.

În prezent, marea majoritate a dezvoltatorilor web folosesc implementarea din JavaScript a DOM API. Pentru a accesa DOM-ul, un dezvoltator poate apela metodele obiectului `document`, disponibil la nivel global în codul sursă. Prin intermediul acestei implementări, dezvoltatorul accesează referințe către nodurile din DOM ce corespund ele-

mentelor HTML din pagină. [22]

Prin această implementare, se pun la dispoziție selectori, ce pot returna referințe către unul sau mai multe noduri ale arborelui, se pot asocia anumitor elemente metode care să se apeleze la declanșarea unui eveniment (procedeu numit „event listening”), iar modificările efectuate asupra nodurilor se propagă către elementele corespunzătoare din HTML.

Astfel, paginile web au devenit interactive, primii programatori web interacționând în mod direct cu DOM API pentru a selecta elemente și a le adăuga funcționalități. În timp, odată cu creșterea în complexitate a paginilor web, au fost observate dezavantajele dezvoltării prin manipularea directă a DOM-ului, paginile scrise astfel fiind foarte greu de întreținut[21]. Acest fapt a condus la crearea framework-urilor web moderne precum Angular, React.js sau Vue, ce sunt folosite în mod frecvent astăzi.

Înțelegerea Document Object Model-ului a fost integrală pentru a putea folosi în mod eficient toate celelalte tehnologii de front-end din aplicație. Atât React, cât și PixiJS, biblioteci folosite pentru modulele de interfață, respectiv editor, extind ideea de DOM în moduri diferite:

- React reține în memorie ierarhia componentelor sale sub formă unui DOM virtual, iar atunci când au loc modificări la nivel de model în aplicație, actualizează elementele afectate și le rerendează.
- PixiJS memorează un arbore al elementelor din scenă, al cărui funcționalitate este inspirată de către DOM API, atât **că** ierarhie, cât și ca sistem de gestionare a evenimentelor.

În continuare, voi prezenta aceste două tehnologii, urmând ca apoi să descriu comunicarea între ele.

2.4 PixiJS

Cea mai importantă decizie luată din punct de vedere a tehnologiilor a fost alegerea bibliotecii pe care o voi folosi pentru implementarea editorului grafic.

Specificațiile de funcționare ale editorului sunt neobișnuite pentru mediul de dezvoltare web, fiind nevoie de o componentă grafică puternică, care trebuie să țină evidența obiectelor desenate, pozițiilor acestora, să poată **aplica** transformări asupra lor, și să răspundă la evenimente. Un astfel de site web, folosind doar tehnologii native, nu ar fi putut exista acum 10-15 ani, **singură** opțiune viabilă a vremurilor de atunci fiind Adobe Flash.[12]

Odată cu lansarea HTML5, a fost introdus elementul `<canvas />`, care definește o **zona** a paginii web care poate fi manipulată de către JavaScript la nivel de pixeli[9], conținutul acestui element nerespectând principiile DOM API. Zona de tip canvas poate fi manipulată fie cu API-ul WebGL, fie cu cel specific Canvas.

Am dorit să folosesc o bibliotecă JavaScript care să extindă funcționalitățile de desene a obiectelor din Canvas, oferind un set minimal de funcționalități, peste care să pot construi motorul editorului. Majoritatea acestor biblioteci, precum PhaserJS[23] sau ImpactJS[14], sunt orientate spre dezvoltarea de jocuri, fiind nepotrivite pentru un proiect tip software utilitar. Utilizarea unei astfel de biblioteci ar fi crescut în mod semnificativ timpul de încărcare al aplicației, în mod nenecesar, fiind folosit un subset restrâns al funcționalităților oferite de acestea.

În final, am ales PixiJS, o bibliotecă lightweight ce oferă un motor de randare, suport pentru evenimente, și un sistem de gestionare a asset-urilor încărcate. Peste acest schelet, am dezvoltat engine-ul Arcada.

PixiJS este un motor de randare 2D care oferă un strat de abstractizare peste WebGL, permițând dezvoltarea facilă de conținut interactiv cross-platform. Acesta este potrivit pentru orice componentă a unei aplicații web care presupune elemente numeroase, cu care se interacționează des, și care trebuie actualizate periodic (prin intermediul buclei de randare). Așadar, PixiJS este potrivit pentru vizualizări de date, jocuri, simulări, sau orice altă formă de conținut interactiv.

PixiJS memorează obiectele din scenă sub forma unui graf. Pentru ca un obiect să fie randat, el va fi atașat fie rădăcinii grafului scenei (denumit „root”), fie unui alt obiect care are ca strămoș root-ul. La momentul randării, PixiJS parcurge graful folosind căutarea în adâncime (i.e. Breadth-First Search) și desenează elementele găsite. Stabilirea relației părinte-copil se efectuează prin apelarea metodei `parinte.addChild(copil)`.

Detașarea unui nod din arborele de randare duce atât la dispariția sa, cât și a subarboarelui acestuia, de pe ecran.

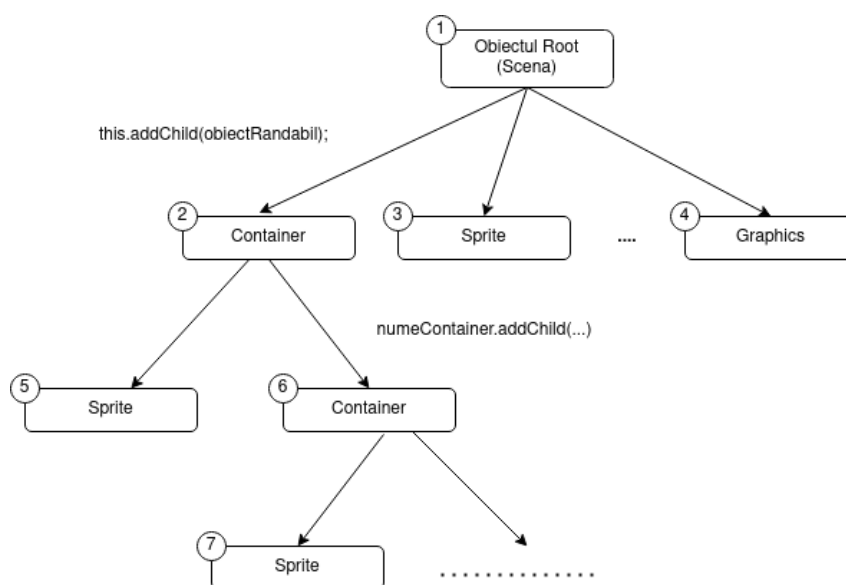


Figura 2.1: Diagramă a arborelui de randare utilizat de PixiJS pentru ținerea în evidență a obiectelor ce vor fi desenate pe ecran. Obiectele din arborele dat exemplu vor fi randate în ordinea numerotării

Coordonatele poziției unui obiect sunt calculate relative la poziția părintelui, fiind văzută de acesta drept originea sistemului de axe. Transformările aplicate părinților se propagă tuturor copiilor[6]. De exemplu, pentru un obiect aflat la coordonatele globale (100, 100), ce are un copil aflat la poziția (50,150), poziția globală a copilului este (150, 250).

Container, Sprite, Graphics sunt câteva dintre tipurile de obiecte atașabile arborelui de randare. Un Sprite este compus dintr-o imagine ce va fi afișată pe ecran (numită textură, sau în engleză Texture) și diverse proprietăți precum dimensiunile curente, scara, unghiul la care este rotit obiectul. [5]

Pe lângă modulul de randare, PIXIJS oferă și suport pentru gestionarea și reacționarea la evenimente. Acestea au fost proiectate astfel încât să funcționeze similar cu evenimentele DOM-ului. Fiecare obiect marcat ca interactiv va declanșa și capta evenimente la acțiuni precum click stânga sau dreapta, mișcarea mouse-ului, atingerea obiectului, cât și evenimente pentru acțiunile specifice ecranelor cu touchscreen.

Faptul că PIXIJS oferă funcționalitățile esențiale pentru randarea de conținut interactiv, lăsând la dispoziția dezvoltatorului implementarea logicii aplicației sale, performanța bibliotecii, cât și faptul că asigură în mod implicit compatibilitatea cu dispozitive vechi, oferind posibilitatea de a randa cu Canvas API atunci când detectează că WebGL nu este disponibil[4], au făcut ca PIXIJS să fie alegerea ideală pentru proiectul Arcada.

2.5 React.js

React.js este o bibliotecă de JavaScript specializată pe proiectarea de interfețe grafice. Dezvoltarea aplicațiilor folosind React se bazează pe crearea de componente reutilizabile, astfel reducându-se cantitatea de cod redundant din aplicație și îmbunătățindu-se modularitatea proiectului. În prezent, React este cea mai folosită bibliotecă de frontend, conform sondajului Stack Overflow Development Survey din 2021[16].

Componentele pot fi create sub formă de funcție sau de clasă, și sunt definite folosind paradigma de programare declarativă. [7]. Un exemplu de astfel de componentă utilizată în aplicația Arcada arată astfel:

```
1 function add(item:IFurnitureData) {
2   let action = new AddFurnitureAction(item.data);
3   action.execute();
4 }
5
6 export function FurnitureItem(item:IFurnitureData) {
7   let data = item.data;
8   return (
9     <Card onClick={() => add(item)} shadow="sm" p="lg">
10      <Card.Section style={{ height: 120, padding: 5 }}>
```

```

11     <Center>
12         <Image src={`${endpoint}/2d/${data.imagePath}`} fit="contain"
height={115} alt={data.name} />
13     </Center>
14 </Card.Section>
15 <Card.Section>
16     <Text align={"center"} weight={500}>{data.name}</Text>
17 </Card.Section>
18 </Card>
19 )
20 }

```

În această secvență de cod am descris aspectul și comportamentul unei componente care va primi detaliile unei piese de mobilier în parametrul `item:IFurnitureData` și va returna un chenar ce conține imaginea obiectului și informații despre acesta. La apăsarea pe componentă, se va executa funcția `add()`, care va porni acțiunea de adăugare a obiectului în plan.

Această componentă se folosește, la rândul ei, de componentele `<Card/>`, `<Image/>`, `<Text/>` ce aparțin bibliotecii de componente Mantine[24]. Programand declarativ, este suficient ca dezvoltatorul să descrie rezultatul final, lăsând în seama React sarcina de a transforma acest cod în pașii imperativi ce trebuie aplicați DOM-ului pentru a desena componentă în pagină.

Ierarhia componentelor unei pagini este menținută în memorie sub forma unui DOM virtual, biblioteca ReactDOM fiind responsabilă de a sincroniza DOM-ul virtual cu cel „real”, mai precis DOM-ul corespunzător paginii HTML, randat de browser[8].

Sincronizarea se face prin compararea versiunii actuală a DOM-ului virtual cu o versiune anterioară, de dinaintea efectuării unei schimbări, și actualizării în DOM-ul real doar a acelor elemente identificate ca fiind modificate. Pentru a putea efectua acest procedeu în mod eficient, ReactDOM se folosește proprietatea obiectelor imutabile de a fi ușor de comparat. Astfel, pentru bună funcționare a bibliotecii, componentele definite de dezvoltator trebuie să respecte principiul imutabilității. [13]

Deoarece React este doar o bibliotecă, nu un framework, pentru a putea crea o aplicație web mai complexă, este necesară folosirea unui manager pentru stări („state manager”), ce are rolul de a centraliza starea globală a aplicației. Rămâne la latitudinea dezvoltatorului să aleagă tehnologia care se potrivește cel mai bine cu cerințele și dimensiunea proiectului său. În cazul aplicației Arcada, am ales state manager-ul Zustand.

2.6 Zustand

Deoarece există o stare comună la nivelul întregii aplicații, cu informații necesare pentru funcționarea mai multor componente ale aplicației, pentru a respecta principiile

de **incapsulare** ale React, și totodată asigură accesul la date în toate componentele care depind de acestea, am avut următoarele opțiuni:

- Fie definesc aceste date comune în rădăcina aplicației, și le pasez în arborele de componente, până ajung la subcomponentele care au nevoie de ele (procedeu numit „prop drilling”, nerecomandat în dezvoltarea cu React, fiind considerat un anti-pattern[28])
- Fie folosesc React Context API pentru a evita situația de mai sus și a conecta componentele direct la datele dorite, procedeu vulnerabil la probleme de performanță[20], și cu suport limitat pentru accesarea stării dinafară React
- Fie aleg să folosesc un state manager extern, iar fiecare clasă care are nevoie să cunoască starea globală, să se conecteze la acesta, cu dezavantajul introducerii unei noi dependențe în proiect

Punând în balanță aspectele pozitive și negative ale fiecărei opțiuni, am optat, în final, pentru a introduce un state manager în proiect, al cărui responsabilitate este să memoreze toate informațiile utilizate inter-modular într-o singură sursă comună, respectând principiul singurei surse de adevăr (i.e. „single source of truth”), astfel prevenind probleme precum duplicarea datelor sau probleme de concurență.

Zustand este un state manager minimalist, ce oferă toate funcționalitățile de bază necesare pentru dezvoltarea unei aplicații. Este făcut să complementeze React, având un mecanism de setare și încărcare a datelor compatibil cu acesta. Spre deosebire de Redux, unul dintre cele mai folosite managere de stare din prezent, Zustand se remarcă prin simplitate și eficiență de scriere, având foarte puțin cod-șablon (i.e. boilerplate code), și fiind foarte ușor de integrat într-o arhitectură deja existentă.

2.7 Express.js

Partea de server a proiectului este scrisă în Express.js, un framework de NodeJS lansat în anul 2010 care permite dezvoltarea rapidă și eficientă a backend-urilor. Express oferă funcții ajutătoare pentru cele mai uzuale operații care au loc pe backend, precum preluarea request-urilor, autentificarea, conectarea la o bază de date, gestionarea cookie-urilor. Totodată, framework-ul este extensibil, fiind compatibil cu alte module sau importuri[25].

Express a fost alegerea naturală pentru acest proiect, oferind posibilitatea dezvoltării cu ușurință a unui API folosit în cadrul aplicației.

2.8 MongoDB

MongoDB este un tip de bază de date NoSQL. Spre deosebire de bazele de date clasice, ale căror scop este administrarea eficientă a datelor tabelare, punând la dispoziție, în acest scop, limbajul SQL pentru plasarea interogărilor, bazele de date tip NoSQL organizează datele sub structura unor colecții ce conțin documente.

Filozofia MongoDB apropie metoda de stocare a datelor în baza de date cu principiile programării orientate pe obiecte. Documentele sunt stocate sub formatul BSON (i.e. Binary Representation of JSON, reprezentare binară a unui document JSON), pot conține un număr variabil de câmpuri, și pot fi îmbricate. [27]

Alegerea sistemului de baze de date potrivit depinde întotdeauna de cerințele și specificațiile proiectului. În cazul Arcada, structura datelor cu care lucrează proiectul se pretează foarte bine pe modelul bazat pe documente și colecții, așadar decizia de a alege o bază de date NoSQL a venit de la sine.

Deși MongoDB pune la dispoziție MongoDB Query Language (MQL) pentru efectuarea interogărilor, mulți dezvoltatori aleg să apeleze la biblioteci tip Object-Document Mapping (ODM) pentru a introduce un strat de abstractizare peste baza de date, lucrând doar cu reprezentarea documentelor sub formă de obiecte în memorie, profitând astfel de apropierea dintre aspectul documentelor și principiile programării orientate pe obiecte.

2.9 Mongoose

Mongoose este o bibliotecă pentru NodeJS și MongoDB care asigură o mapare între documentele bazei de date și obiecte instanțiate în cod (î.e. Object-Document Mapper). Pentru a crea aceste asocieri, Mongoose se folosește de două concepte: Schema, respectiv Modelul. Schema unei colecții desemnează o structură generală a unui document din colecție. Modelul este obiectul cu care se poate interacționa în cod, funcționând ca un strat intermediar între baza de date și aplicația server. Pentru a putea fi creat, un model primește o schemă, un nume de colecție, și o conexiune la baza de date. [19]

```
1 const mongoose = require("mongoose");
2 const Category = require("./category");
3
4 const Furniture = new mongoose.Schema({
5   name: {
6     type: String,
7     required: true
8   },
9   width: {
10    type: Number,
11    required: true
12  },
```

```

13     height: {
14         type: Number,
15         required: true
16     },
17     imagePath: {
18         type: String,
19         required: true
20     },
21     category: {
22         type: mongoose.Schema.Types.ObjectId,
23         ref: Category }
24 })
25
26 module.exports = mongoose.model("furniture", Furniture);

```

Dupa crearea modelului, dezvoltatorul poate rula căutări, efectua modificări asupra elementelor, toate operațiunile valide pe baza de date, în final salvând modificările pe BD prin apelarea metodei `.save()`.

```

1 const Category = require("../models/category")
2 // ...
3 for (let category of categories) {
4     try {
5         const newCategory = new Category({ "name": category.name, "visible"
6         : category.visible })
7         await newCategory.save()
8     }
9 }

```

Capitolul 3

Dezvoltarea aplicației

3.1 Arhitectura proiectului

Arhitectura aplicației Arcada respectă modelul client-server, cu mențiunea că marea majoritate a logicii aplicației are loc în client.

Clientul este punctul central al aplicației, și este împărțit în trei module principale: modulul editor, în care se poate interacționa în mod direct cu planul aflat în lucru, modulul interfeței de control, prin care se pot trimite comenzi către editor, și modulul de serializare, ce iterează prin elementele vizibile pe ecran și extrage datele necesare, compunând un fișier care poate fi salvat și apoi încărcat.

Scopul serverului este acela de a servi date statice, resursele comune tuturor planurilor de interior, precum imaginile corespunzătoare pieselor de mobilier, dimensiunile lor standard, și categoriile în care se încadrează. Separarea datelor statice de client are ca scop micșorarea timpilor de încărcare și **consumului de bandă** al aplicației. Am dorit ca utilizatorii să încarce doar acele resurse pe care intenționează să le folosească în planul lor, nu întreaga bibliotecă de componente, în timpul rulării aplicației.

3.2 Modulul interfață

Am apelat la biblioteca React pentru a construi interfața auxiliară editorului. Diagrama UML a acestei funcționalități arată astfel:

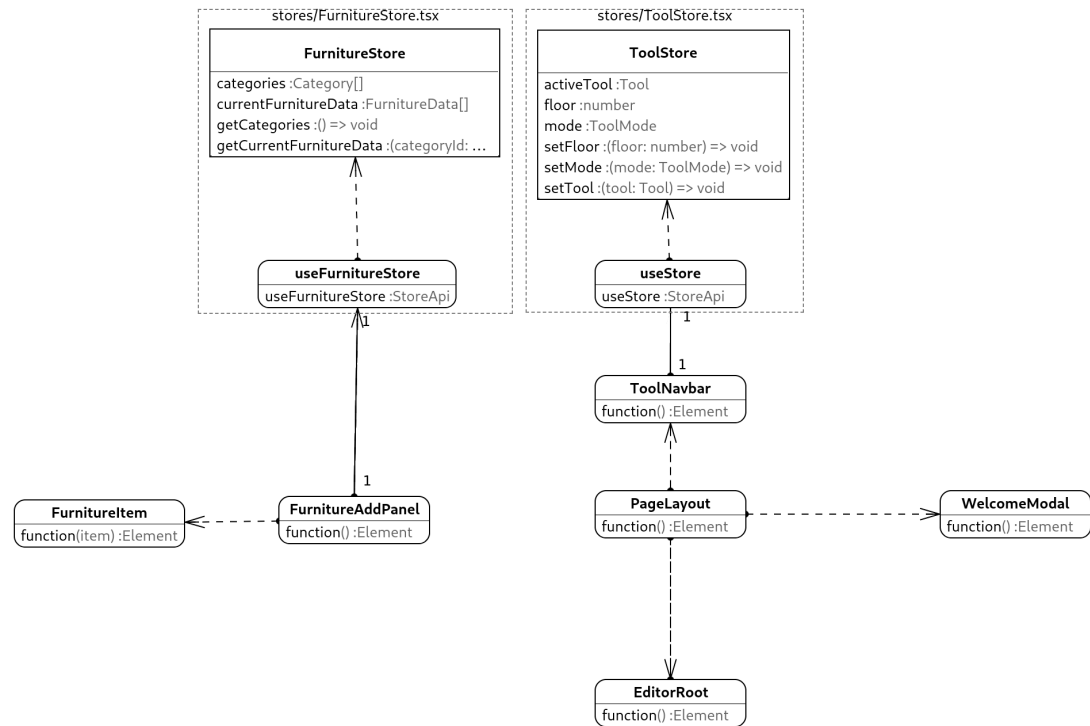


Figura 3.1: Diagrama UML a interfeței

Pentru a dezvolta acestui modul, am creat mai multe componente funcționale, reprezentând unități de sine stătătoare ale interfeței. Componenta funcțională `PageLayout` reprezintă punctul de plecare al aplicației, ce definește așezarea în pagină a celorlalte elemente, apoi le încarcă.

```

1 export function PageLayout() {
2   return (<>
3     <WelcomeModal /> // Meniu de bun-venit
4     <Grid grow gutter={0}>
5       <ToolNavbar /> // Bara de unelte
6       <Grid.Col span={9}>
7         <EditorRoot /> // Editor-ul
8       </Grid.Col>
9     </Grid>
10  </>);
11 }

```

Componenta funcțională `ToolNavbar` reprezintă bara de unelte aflată în stânga editorului. Aceasta comunică prin hook-ul `useStore` cu store-ul `EditorStore`, al cărui scop este de a memora starea curentă a aplicației. Modulul Editor poate accesa valorile stocate în store-uri, și se folosește de ele pentru a își gestiona activitatea.

Pentru a exemplifica acest comportament, voi descrie funcționalitatea panoului de

adăugare mobilă. Construcția acestui panou are loc în componenta FurnitureAddPanel:

```
1 export function FurnitureAddPanel() {
2   const { classes } = useStyles();
3   const [category, setCategory] = useState('');
4   const [availableCategories, setAvailableCategories] = useState([]);
5   const { categories, currentFurnitureData, getCurrentFurnitureData } =
useFurnitureStore();
6   const [cards, setCards] = useState([]);
7
8   // atunci cand user-ul selecteaza o categorie, incarca elementele de
mobila ce ii apartin
9   useEffect(() => {
10     if (category) {
11       getCurrentFurnitureData(category)
12     }
13   }, [category])
14
15   // atunci cand a fost actualizata lista mobilierului, creeaza chenare
pentru fiecare obiect si adauga-le pe ecran
16   useEffect(() => {
17     setCards(currentFurnitureData.map((item) =>
18       (
19         <FurnitureItem data={item} key={item._id}></FurnitureItem>
20       )
21     ))
22   }, [currentFurnitureData])
23
24   // la prima incarcare a categoriilor, seteaza sa fie afisata prima
categorie
25   useEffect(() => {
26     setCategory(categories[0]._id)
27   }, [categories])
28
29   return (<>
30     <Navbar.Section>
31       <Select className={classes.mb} value={category} onChange={
setCategory}
32       data={categories.map(cat => {
33         return { value: cat._id, label: cat.name };
34       })} />
35     </Navbar.Section>
36     <Navbar.Section grow component={ScrollArea} mx="-xs" px="xs">
37       <SimpleGrid style={{ padding: 5 }} cols={2}>
38         {cards}
39       </SimpleGrid>
40     </Navbar.Section>
41   </>)
```

42 }

Pentru a funcționa, panoul menține în cadrul stării interne informații despre categoria curent selectată, respectiv lista card-urilor care prezintă informații despre elementele de mobilă ale categoriei respective. Totodată, panoul are acces la store-ul `FurnitureStore`, în care sunt stocate detaliile obiectelor încărcate în urma request-ului făcut către API. Odată ce au fost încărcate obiectele, hook-ul `useEffect()`, linia 16, va genera card-urile asociate fiecărui obiect tip `FurnitureItem`, ce vor fi afișate într-un grid, sub selectorul de categorii.

```
1 function add(item:IFurnitureData) {
2     let action = new AddFurnitureAction(item.data);
3     action.execute();
4 }
5
6 export function FurnitureItem(item:IFurnitureData) {
7     let data = item.data;
8     return (
9         <Card onClick={() => add(item)} shadow="sm" p="lg">
10             <Card.Section style={{ height: 120, padding: 5 }}>
11                 <Center>
12                     <Image src={`/${endpoint}/2d/${data.imagePath}`} fit="contain"
13                         height={115} alt={data.name} />
14                 </Center>
15             </Card.Section>
16             <Card.Section>
17                 <Center>
18                     <Text align="center" weight={500}>{data.name}</Text>
19                 </Center>
20             </Card.Section>
21         </Card>
22     )
23 }
```

`FurnitureItem` este o componentă care preia un obiect de tipul `IFurnitureData` și generează chenarul pe care utilizatorul apasă pentru a adăuga un astfel de obiect în editor. Funcția `add()` creează o nouă acțiune de tipul „Add” și apoi o execută. Astfel, se asigură asincronicitatea, iar interfața nu va fi blocată până când se îndeplinește adăugarea în editor. Toate celelalte elemente ale interfeței funcționează pe principii similare.

3.3 Modulul editor

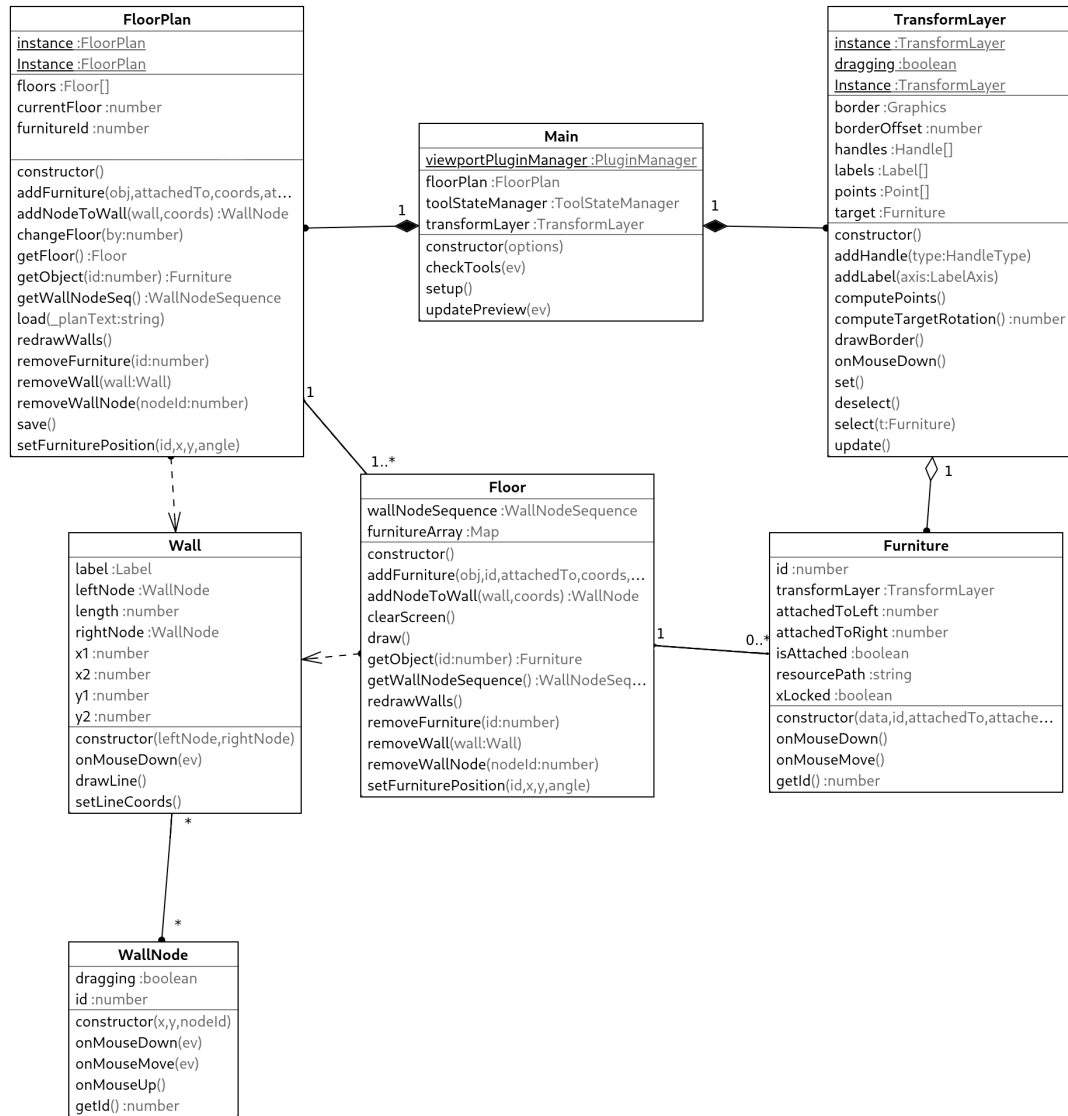


Figura 3.2: Diagrama UML a modulului editor

Toate celelalte module ale aplicației interacționează cu planul prin intermediul fațadei oferite de clasa **FloorPlan**. Aceasta ascunde detaliile de implementare ale planului, simplificând interacțiunea cu planul și reducând din inter-dependența claselor.

Că funcționalități de sine stătătoare, clasa permite controlul etajelor. În plus, această expune metode pentru adăugarea, ștergerea, editarea pereților, adăugarea sau ștergerea mobilierului, salvarea și încărcarea datelor din aplicație.

Deoarece un singur plan poate fi încărcat în aplicație, am decis să organizez **FloorPlan** conform design pattern-ului Singleton. **FloorPlan** extinde clasa **Container**, și face parte din

ierarhia de obiecte randate de PIXIJS.

3.3.1 Manipularea etajelor

Un plan este compus din mai multe etaje („floors”). Un singur etaj poate fi activ și editabil la un moment dat, indicele acestuia fiind reținut în variabila `currentFloor`. Metodă `changeFloor()` gestionează această funcționalitate, apelând metodele `addChild()`, `removeChild()`, moștenite din clasa `Container`, pentru a schimba care dintre etaje este atașat grafului scenei, deci desenat de PIXIJS.

```
1 // FloorPlan.ts
2 public changeFloor(by: number) {
3
4     this.removeChild(this.floors[this.currentFloor]);
5     this.currentFloor += by;
6     // daca nu exista etajul, creeaza-l
7     if (this.floors[this.currentFloor] === null) {
8         this.floors[this.currentFloor] = new Floor();
9     }
10    this.addChild(this.floors[this.currentFloor])
11 }
```

Toate celelalte clase din Arcada aplică modificări asupra etajului aflat în lucru. Lista etajelor (`floors: Floor[]`) este un membru privat al clasei `FloorPlan`, și este accesibilă doar în cadrul acesteia. Pentru a permite celorlalte clase să interacționeze cu etajul, sunt expuse metode care mediază interacțiunea, funcționând ca un wrapper. Funcționalitatea lor efectivă este implementată în clasa `Floor`.

Clasa `Floor` conține membrii următori, ai căror funcționalitate o voi descrie în continuare:

```
1 public furnitureArray: Map<number, Furniture>;
2 private wallNodeSequence: WallNodeSequence;
```

3.3.2 Adăugarea mobilei

În map-ul `furnitureArray` sunt stocate referințe către elementele de mobilier prezente în plan. Adăugarea obiectelor se face prin metoda `addFurniture()`, ce apelează constructorul clasei `Furniture`, creând un obiect nou, adăugându-l în plan, și îl atașează unui perete, în cazul ușilor sau a ferestrelor. De asemenea, se păstrează o referință către obiect, care este mapata id-ului corespunzător obiectului, în `furnitureArray`.

```
1 public addFurniture(obj: FurnitureData, id: number, attachedTo?: Wall,
2     coords?: Point, attachedToLeft?: number, attachedToRight?: number) {
3
4     let object = new Furniture(obj, id, attachedTo, attachedToLeft,
5         attachedToRight)
```

```

4   this.furnitureArray.set(id, object);
5
6   if (attachedTo !== undefined) {
7       attachedTo.addChild(object)
8       object.position.set(coords.x, coords.y)
9   } else {
10      this.addChild(object)
11      object.position.set(FURNITURE_START_POS_X, FURNITURE_START_POS_Y)
12  }
13  return id;
14 }

```

În metoda de mai sus, se remarcă prezența parametrilor opționali în coadă definiției funcției. TypeScript nu permite supraîncărcarea metodelor, motiv pentru care am procedat în acest fel pentru a diferenția între obiectele care sunt atașate de pereți și cele care nu.

Furniture

Este clasa care reprezintă o piesă de mobilă din plan. Pentru a putea fi desenată, această moștenește clasa Sprite, devenind astfel compatibilă cu motorul de randare. Proprietățile cele mai importante ale acesteia sunt dimensiunile (lungimea, lățimea), imaginea obiectului, unghiul rotirii, și id-ul.

Obiectele pot fi de sine stătătoare (au **că** părinte root-ul planului) sau pot fi atașate altor obiecte din plan (de exemplu, o fereastră este asociată unui perete, iar transformările aplicate peretelui se propagă și ferestrei), această funcționalitate fiind controlată prin transmiterea parametrului opțional **attachedTo** în constructorul obiectului.

Fiecărui obiect de tip mobilă îi sunt atașate două funcții pe event-urile ce se activează atunci când utilizatorul da click pe un obiect sau mișcă mouse-ul pe acesta.

```

1  // Furniture.ts
2  switch (useStore.getState().activeTool) {
3      case Tool.FurnitureEdit: {
4          const action = new EditFurnitureAction(this);
5          action.execute();
6          break;
7      }
8      case Tool.FurnitureRemove: {
9          const action = new DeleteFurnitureAction(this.id);
10         action.execute();
11         break;
12     }
13 }

```

La apăsarea pe un obiect, se verifică dacă unealta activă în editor permite interacțiunea cu obiectul. În funcție de unealtă selectată, se execută acțiunea corespunzătoare. Acțiunile posibile în editor sunt definite conform design pattern-ului Command.

3.3.3 Stocarea informațiilor despre pereți

WallNodeSequence este clasa în care se memorează configurația pereților unui etaj. Are în componentă următoarele atribute:

```
1 // graful reprezentat prin lista de adiacenta
2 private wallNodes: Map<number, WallNode>;
3
4 // mapare intre indicele nodului si obiectul care contine detalii
  despre nod
5 private wallNodeLinks: Map<number, number[]>
6
7 // lista ce retine referinte catre reprezentarile grafice ale
  muchiilor grafului
8 private walls: Wall[];
9
10 // ultimul id alocat
11 private static wallNodeId: number = 0;
```

Pentru a reprezenta structura încăperilor într-o formă manipulabilă computațional, am aplicat concepte atât din teoria grafurilor, cât și din geometria computațională.

Definiție. Un graf G este definit ca perechea (V, E) unde V este mulțimea nodurilor, iar $E = \{(x, y) | x, y \in V, x \neq y\}$ lista muchiilor. ([1] pg. 1)

Definiție. Un graf G este numit *graf planar* dacă există o reprezentare grafică a sa în plan astfel încât niciuna dintre muchiile sale nu se intersectează. O astfel de reprezentare a unui graf se mai numește și *încorporare în plan a grafului*. ([1] pg. 3)

Definiție. O *încorporare în plan a grafului* G (eng. „graph embedding”) este dată de două funcții de corespondență: una care asociază fiecărui nod din graf un punct din plan, și alta care asociază fiecărei muchii o curbă plană. ([1] pg. 7)

Teorema lui Fáry. Pentru orice graf planar simplu există o încorporare care se poate realiza reprezentând toate muchiile ca segmente de dreaptă. ([1] pg. 13)

Am decis să reprezint planul încăperilor că pe un graf orientat, unde muchiile reprezintă pereții iar nodurile reprezintă colțuri sau puncte unde are loc întâlnirea mai multor pereți. Fiecărui nod îi este asociat poziția sa în sistemul de coordonate (X, Y) . Fie E o muchie între nodurile A și B , reprezentând faptul că există un perete între aceste două noduri. Ponderea acestei muchii va fi egală cu distanța euclidiană dintre coordonatele asociate celor două noduri. Planul pereților unui etaj este, de fapt, o încorporare în plan a grafului, folosindu-se o mapare între noduri și puncte din sistemul de coordonate al ecranului.

E important de notat faptul că în grafica pe calculator sistemul de coordonate folosit este similar sistemului cartezian de coordonate, existând totuși câteva diferențe notabile.

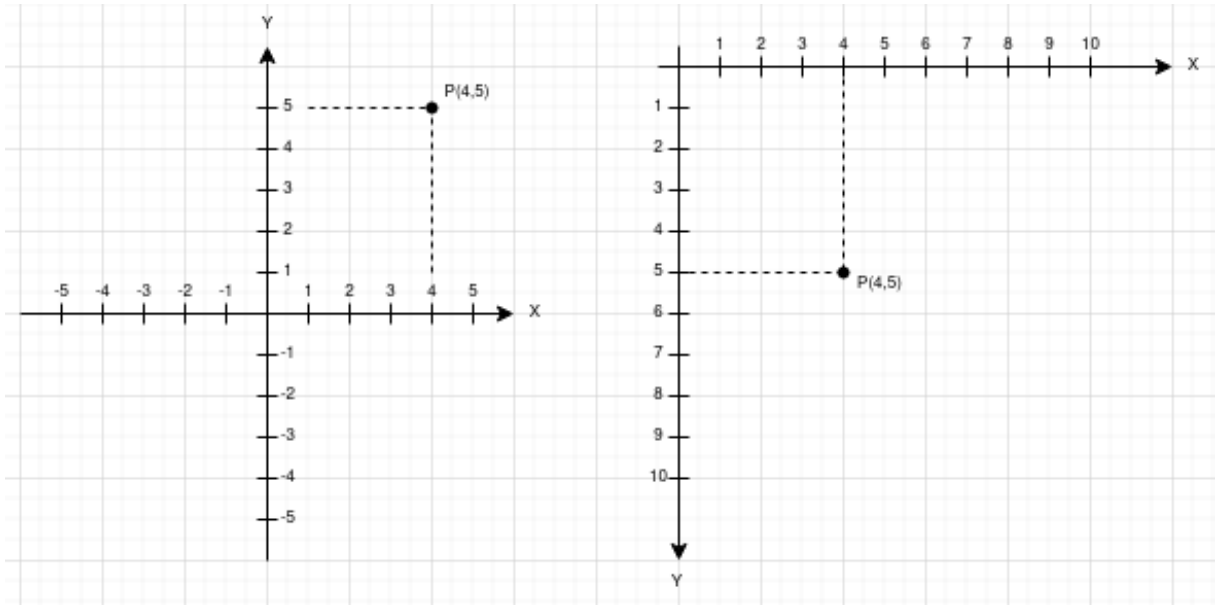


Figura 3.3: Reprezentarea punctului $P(4,5)$ în sistemul de coordonate cartezian, respectiv în coordonate specifice graficii pe calculator

Principala diferență între aceste două sisteme este că în sistemul utilizat în grafică, axa X este oglindită. Această diferență are origini istorice, fiind cauzată de faptul că primele monitoare funcționau pe baza tehnologiei de tub catodic, și desenau imaginea pe ecran pornind din colțul stânga-sus, secvențial, oprindu-se în colțul dreapta-jos al ecranului [2]. Primul cadran se află în dreapta-jos, iar punctul $(0,0)$ reprezintă pixelul aflat în colțul de stânga-sus a ecranului.

Arcada memorează în `wallNodes` o mapare între eticheta nodului și un obiect de tip `WallNode`, ce conține informații despre nod, cel mai important detaliu fiind poziția. `wallNodeLinks` reprezintă lista de adiacență a grafului, sub forma unei mapări de la o eticheta a unui nod la lista nodurilor înspre care există muchie.

Exemplu. Fie graful G definit prin lista de adiacență:

$$V = \{A, B, C, D, E, F, G, H, I, J, K, L, M, N\}$$

$A \rightarrow B$
 $B \rightarrow C, H$
 $C \rightarrow D, E$
 $D \rightarrow F$
 $E \rightarrow F, I$
 $F \rightarrow J$
 $G \rightarrow H, K$
 $H \rightarrow L$
 $I \rightarrow J, M$
 $J \rightarrow N$
 $K \rightarrow L$
 $L \rightarrow M$
 $M \rightarrow N$
 $N \rightarrow B, G$

Încorporarea în plan a acestui graf arată astfel:

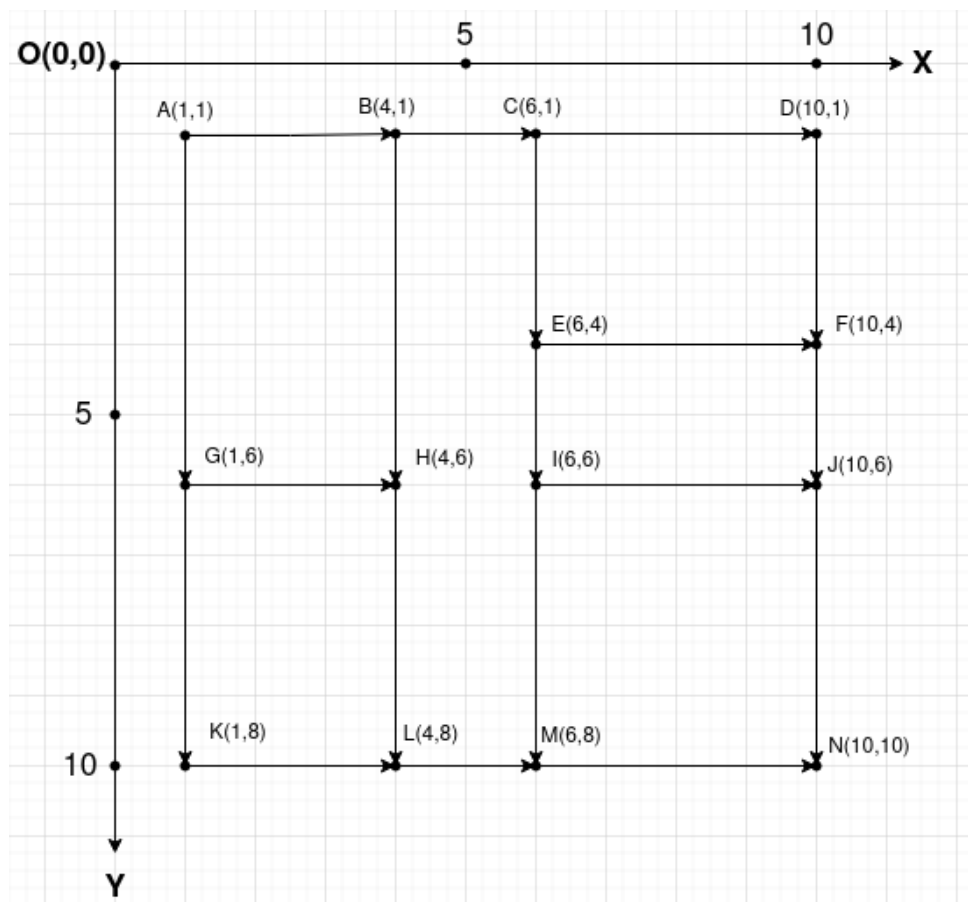


Figura 3.4: Încorporarea în plan a grafului orientat G

Această încorporare generează, în aplicația Arcada, următoarea dispunere a camerelor:

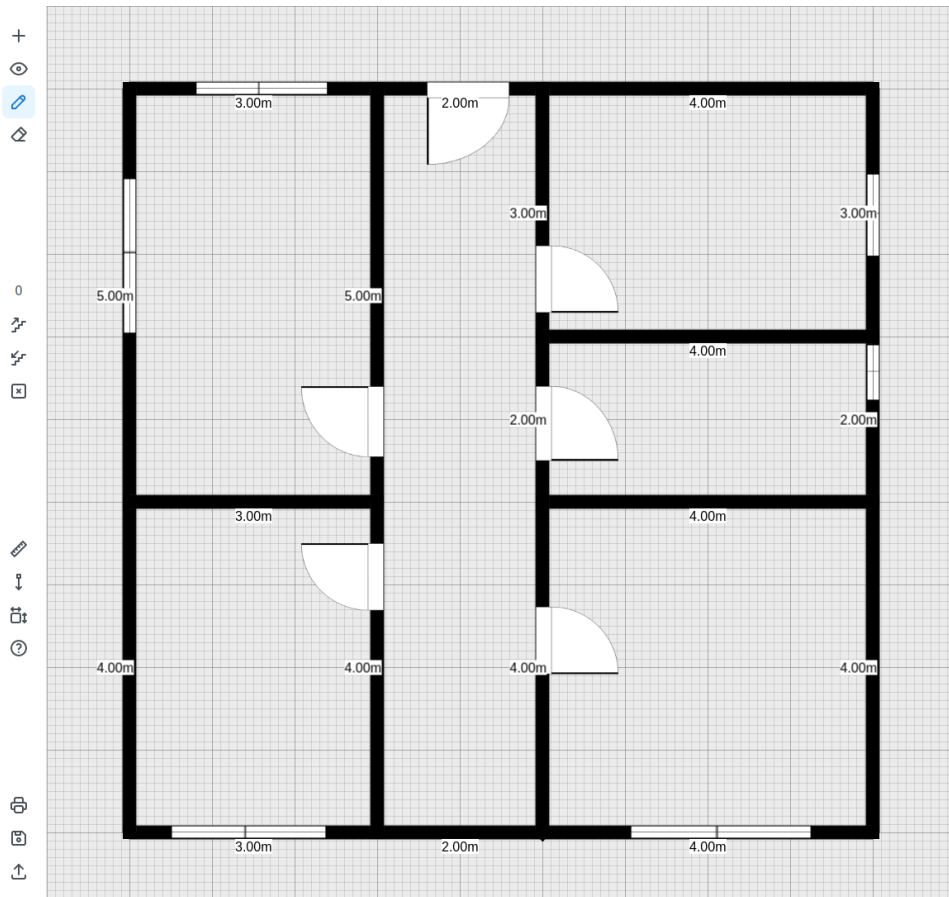


Figura 3.5: Planul **încaperilor** generat de încorporarea în plan a grafului G dată

Unelte puse la dispoziție în cadrul aplicației, la nivelul pereților, au rolul de a aplica modificări asupra grafului memorat și a încorporării sale. De exemplu, adăugarea unui nod se face prin metoda `addNode`, unde `x,y` sunt coordonatele din plan alese de utilizator prin apăsarea pe ecran:

```

1 public addNode(x: number, y: number, id?: number) {
2     let nodeId;
3     if (id) {
4         nodeId = id;
5     } else {
6         nodeId = this.getNewNodeId();
7     }
8     // creeaza un obiect nou ce reprezinta nodul in plan, la pozitia (x,y),
9     // si asociaza-l etichetei nodeId
10    let newNode = new WallNode(x, y, nodeId);
11
12    // retine in maparea wallNodes o referinta de la eticheta nodului la
13    // reprezentarea sa grafica
14    this.wallNodes.set(nodeId, new WallNode(x, y, nodeId));
15
16    // adauga in lista de adiacenta noul nod
17    this.wallNodeLinks.set(nodeId, []);

```

```

16
17 // ataseaza nodul etajului curent, pentru a fi desenat atunci cand etajul
    este activ
18 this.addChild(newNode)
19 return newNode;
20 }

```

3.3.4 Aplicarea transformărilor asupra obiectelor din plan

Dorind să extind funcționalitatea aplicației pentru a permite utilizatorilor să mute, redimensioneze, sau rotească obiectele de mobilier, am creat o clasă numită `TransformLayer` ce atasează unui obiect de mobilier, și randează instrumente care permit manipularea dimensiunilor acestuia.

Clasa este implementată sub forma unui Singleton, astfel garantând faptul că nu vor **exista** situații în care mai multor obiecte să le fie afișate controalele de editare simultan.

Chenarul pentru editare este compus din margine, puncte care pot fi manevrate (denumite „handles”), și chenare de text ce afișează dimensiunile obiectului (lungimea, lățimea).

Selectarea unui obiect salvează o referință către obiect în variabilă `target`, calculează pozițiile punctelor chenarului, și le randează.

```

1 public select(t: Furniture) {
2     if (useStore.getState().activeTool !== Tool.Edit) {
3         return;
4     }
5     if (this.target !== null) {
6         this.deselect();
7         return;
8     }
9
10    // seteaza obiectul curent, calculeaza in functie de dimensiunile
    acestuia pozitiile handle-urilor, si le afiseaza pe ecran
11    this.target = t;
12    this.computePoints();
13    this.draw();
14
15    this.visible = true;
16    this.interactive = true;
17 }

```

La rândul lor, fiecare Handle cunoaște obiectul al cărei dimensiuni le va manipula, cât și axa pe care va face transformările. Enum-ul `HandleType` listează toate funcționalitățile posibile ale unui Handle:

```

1 export enum HandleType {
2     Horizontal, // modifica latimea unui obiect
3     Vertical, // modifica lungimea unui obiect

```

```

4   HorizontalVertical, // modifica ambele dimensiuni ale obiectului,
    pastrand proportiile
5   Rotate, // permite rotirea
6   Move // permite mutarea
7 }

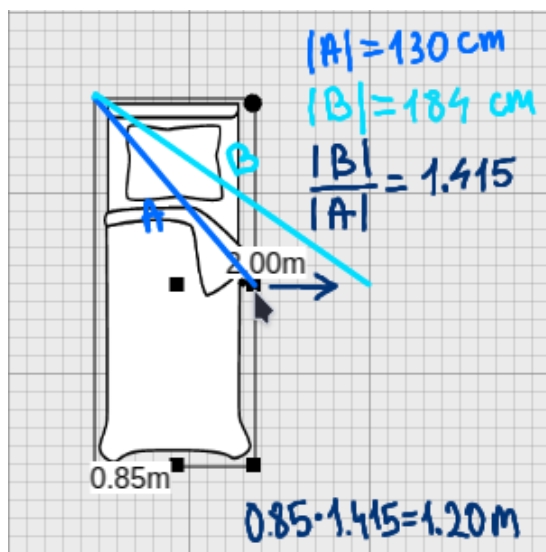
```

La apăsarea pe orice Handle, se memorează detalii despre starea inițială a obiectului, precum dimensiunile, poziția, rotația, cât și coordonatele în plan unde a fost făcut click. De asemenea, `active` este setat pe `True`.

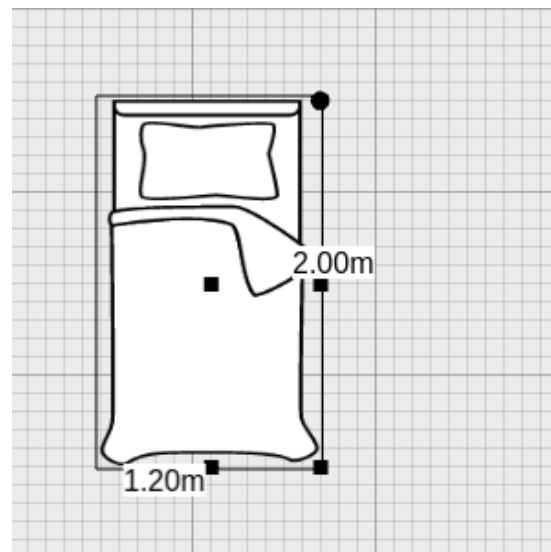
Atunci când cursorul este mișcat și flagul `active` este `True`, se calculează în `mouseEndPoint` nouă poziție a mouse-ului, apoi se fac calcule specifice efectului dorit.

Modificarea dimensiunilor

Pentru a schimba dimensiunile obiectului, la apăsarea pe handle-ul corespunzător, se calculează distanța euclidiană de la colțul stânga sus la poziția de start a mouse-ului, respectiv la poziția curentă a mouse-ului, și se efectuează raportul acestora. Acest raport reprezintă factorul cu care dimensiunea obiectului va fi modificată.



(a) creșterea lățimii unui obiect



(b) rezultat

Figura 3.6: Redimensionarea unui obiect pe axa X. Procedeul este identic pentru axa Y

Mutarea obiectelor în plan

Pentru mutarea obiectelor în plan, se calculează distanța pe ambele axe dintre poziția nouă a mouse-ului și poziția originală. Valorile obținute sunt adunate poziției originale a obiectului, obținându-se astfel efectul de translație.

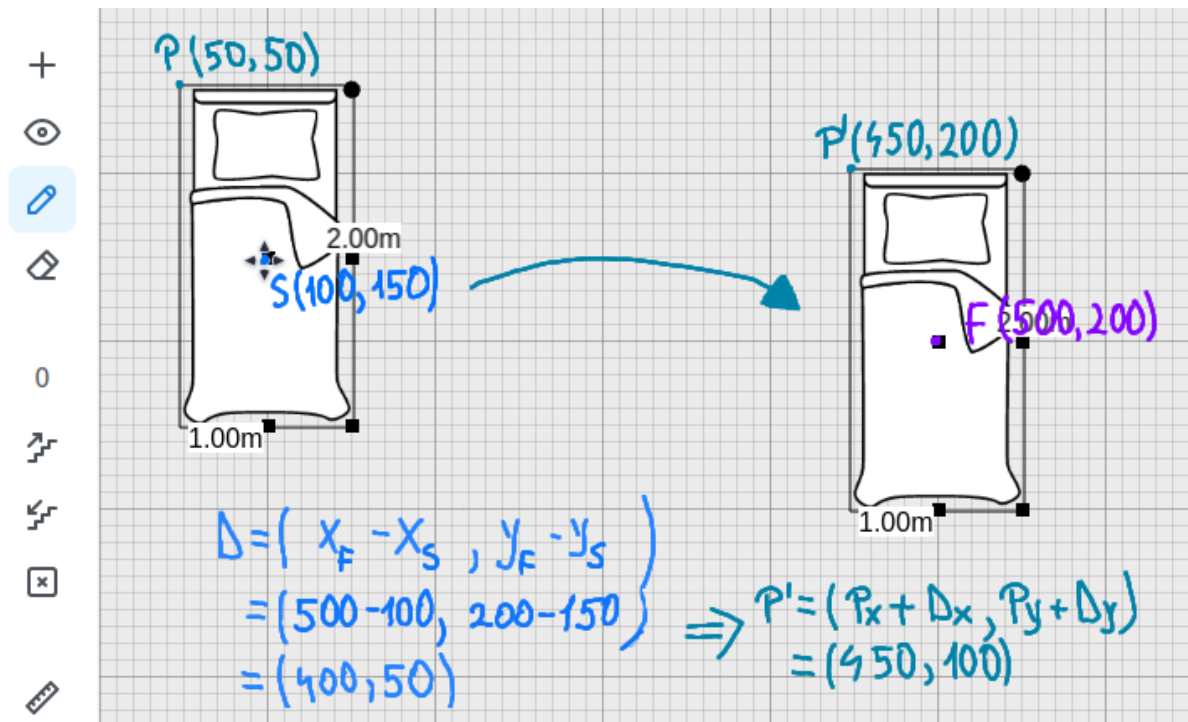


Figura 3.7: Mișcarea mouse-ului din punctul S în punctul F determină mutarea obiectului din punctul P în punctul P'

Rotirea

Cum motorul PixiJS **ofera** suport pentru rotirea sprite-urilor, pentru a putea aplica **rotatii** asupra unor obiecte, am decis sa calculez, **luand** ca punct de **referinta** originea sistemului de axe, unghiul rotirii astfel:

- Fie d_1 dreapta **data** de punctele $(0,0)$ si coordonatele mouse-ului la momentul de start al rotirii
- Fie d_2 dreapta **data** de punctele $(0,0)$ si coordonatele mouse-ului ob**ținute in** urma **miscării** sale
- Fie α, β unghiurile dintre dreptele A, B si axa OX
- Diferenta $\Delta = \beta - \alpha$ reprezintă unghiul cu care a fost mișcat mouse-ul de la punctul de start la punctul actual. Am modificat rotația obiectului cu această valoare.

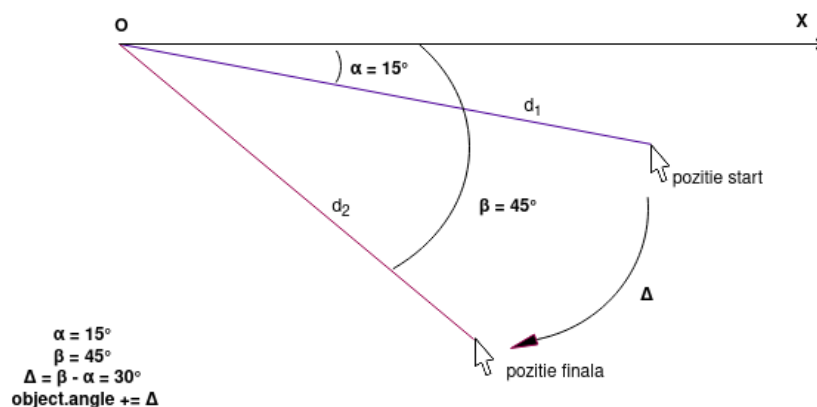


Figura 3.8: Calcularea unghiului cu care să se modifice **rotatia** obiectului. Pentru un obiect cu o rotire inițială de 15° , deplasarea mouse-ului la un unghi de 45° față de axa OX va crește rotația obiectului cu 30°

Pentru a putea calcula unghiurile α, β , am folosit funcția `Math.atan2(x,y)` ce **primește** ca parametru un punct din plan și **calculează** unghiul dintre axa OX și segmentul dat de punctele $(0,0)$ și (x,y) .

3.3.5 Proportionalitatea elementelor din plan

Pentru a mă asigura de faptul că atât obiectele, cât și pereții sunt proporționali, am definit constanta `METER`, egală cu 100 pixeli din spațiul planului. Toate calculele de redimensionare a obiectelor sunt făcute ținând cont de această constantă. De exemplu, în cazul adăugării unui nou obiect, avem:

```
1 this.width = data.width * METER;
2 this.height = data.height * METER;
```

unde `data.width`, `data.height`, dimensiunile standard ale obiectului, sunt exprimate în metri.

În plus, în întreaga interfață sunt trasate la interval de 10 pixeli (echivalent a 10 centimetri) linii ajutătoare, menite să ușureze amplasarea obiectelor. La interval de 1 metru, sunt trasate linii îngroșate.

3.3.6 Vizualizarea detaliată a planului

Atât lungimea, cât și lățimea maximă a unui etaj, este 25 de metri. Pentru a putea permite utilizatorului să vizualizeze întregul plan, am implementat, utilizând pachetul `Pixi-Viewport`[10], posibilitatea de a **mari**, micșora, și naviga prin suprafața planului, astfel încât doar o anumită parte din plan să fie vizibilă pe ecran. Acest lucru este util deoarece permite utilizatorilor să creeze planuri ale căror suprafețe sunt mai mari decât dimensiunile ecranului de lucru, cât și pentru a permite un mai bun control asupra manipulării obiectelor.

Toate calculele în care apar coordonate în plan sunt trecute prin funcțiile ajutătoare `viewportX`, `viewportY` ce primesc coordonate de tip ecran, și returnează coordonate în spațiul planului. Coordonatele sunt calculate astfel:

1. Coordonata este împărțită la factorul de zoom, pentru a obține valoarea ei reală în screen-space.
2. Valoarea obținută este translatată cu distanța dintre colțul stânga-sus al hărții și colțul stânga-sus al zonei vizibile din plan.

Astfel, se obține coordonata din plan corespunzătoare poziției apăsării mouse-ului, în urmă adăugării funcționalității de `viewfinder`.

3.4 Modulul de serializare

Utilizând tehnici de serializare a obiectelor, am transformat elementele vizibile în plan într-un format care poate fi salvat într-un fișier și stocat. Prin încărcarea fișierului, se va reconstitui starea originală a planului. Astfel, am implementat o metodă prin care utilizatorii pot obține persistența datelor.



Figura 3.9: Diagrama UML a claselor principale ale modului de serializare

3.4.1 Salvarea. Serializare

Fiecare element constituent al planului implementează metoda `serialize()`, ce returnează obiectele în formă lor serializabilă (care nu conține referințe). La apăsarea butonului de salvare se apelează funcția `serialize()` a clasei `Serializer`, ce apelează la rândul ei metoda de pregătire a datelor pentru fiecare etaj. Apoi, etajele parcurg fiecare nod, perete, și piesă de mobilier, și fac copii ale parametrilor esențiali reconstituirii.

```
1 public serialize(): FloorSerializable {
2     let plan = new FloorSerializable();
3     // extrage informatii despre punctele de legatura dintre pereti
4     let wallNodes = this.wallNodeSequence.getWallNodes();
5     for (let node of wallNodes.values()) {
6         plan.wallNodes.push(node.serialize());
7     }
8     // extrage informatii despre muchiile grafului - pereti
9     plan.wallNodeLinks = Array.from(this.wallNodeSequence.
10    getWallNodeLinks().entries());
11
12     // itereaza prin elementele de mobila, adaugand in lista obiectele
13    in forma lor serializabila
14     let serializedFurniture = []
15     for (let furniture of this.furnitureArray.values()) {
16         serializedFurniture.push(furniture.serialize())
17     }
18     plan.furnitureArray = serializedFurniture;
19     return plan;
20 }
```

Rezultatul final este compus într-un obiect de tipul `FloorPlanSerializable`, care este transformat în text compatibil cu JavaScript Object Notation (JSON) apelând funcția `JSON.stringify()`.

```
1 public serialize(floors: Floor[], furnitureId: number) {
2     let floorPlanSerializable = new FloorPlanSerializable();
3     for (let floor of floors) {
4         let floorSerializable = floor.serialize();
5         floorPlanSerializable.floors.push(floorSerializable)
6     }
7     floorPlanSerializable.furnitureId = furnitureId;
8     floorPlanSerializable.wallNodeId = floors[0].getWallNodeSequence().
9    getWallNodeId();
10    let resultString = JSON.stringify(floorPlanSerializable)
11    return resultString
12 }
```

3.4.2 Încărcarea. Deserializare

Odată ce utilizatorul încarcă, folosind funcția „Load”, un fișier salvat anterior, datele acestuia vor fi preluate și vor fi transformate într-un obiect de tipul `FloorPlanSerializable` folosindu-se funcția `JSON.parse()`, ce conține informații despre elementele tuturor etajelor planului. În continuare, se vor crea etaje respectând datele încărcate din fișier. Constructorul clasei `Floor` acceptă ca parametru opțional un obiect de tip `FloorData`. Dacă acesta este precizat, constructorul va parcurge membrii obiectului și va genera etajul conform specificațiilor.

Fie un utilizator ce își dorește să își amenajeze camera de cămin. Acesta desenează în aplicație camera în starea ei actuală, apoi o salvează. Conținutul fișierului rezultat ar putea fi similar cu următorul:

```
1 {
2   "floors": [
3     {
4       "furnitureArray": [
5         {
6           "x": 110,
7           "y": 110,
8           "height": 2,
9           "width": 1.6,
10          "id": 3,
11          "texturePath": "king-size-bed",
12          "rotation": 0
13        },
14        {
15          "x": 226,
16          "y": 0,
17          "height": 1,
18          "width": 1,
19          "id": 4,
20          "texturePath": "door",
21          "rotation": 0,
22          "attachedToLeft": 1,
23          "attachedToRight": 2
24        },
25        {
26          "x": 370,
27          "y": 270,
28          "height": 0.6,
29          "width": 0.6,
30          "id": 5,
31          "texturePath": "chair-1",
32          "rotation": 0
33        },
34      ]
35    }
36  ]
37 }
```

```

34     {
35         "x":340,
36         "y":310,
37         "height":0.8,
38         "width":1.2,
39         "id":6,
40         "texturePath":"table",
41         "rotation":0
42     }
43 ],
44     "wallNodes":[
45         { "id":1, "x":100, "y":100 },
46         { "id":2, "x":500, "y":100 },
47         { "id":3, "x":500, "y":400 },
48         { "id":4, "x":100, "y":400 }
49     ],
50     "wallNodeLinks":[[1,[2,4]],[2,[3]],[3,[4]],[4,[]]]
51 },
52     "furnitureId":6,
53     "wallNodeId":4
54 }

```

Ulterior, atunci când va dori să continue lucrul asupra planului, el va putea încărca fișierul în aplicație și va obține:

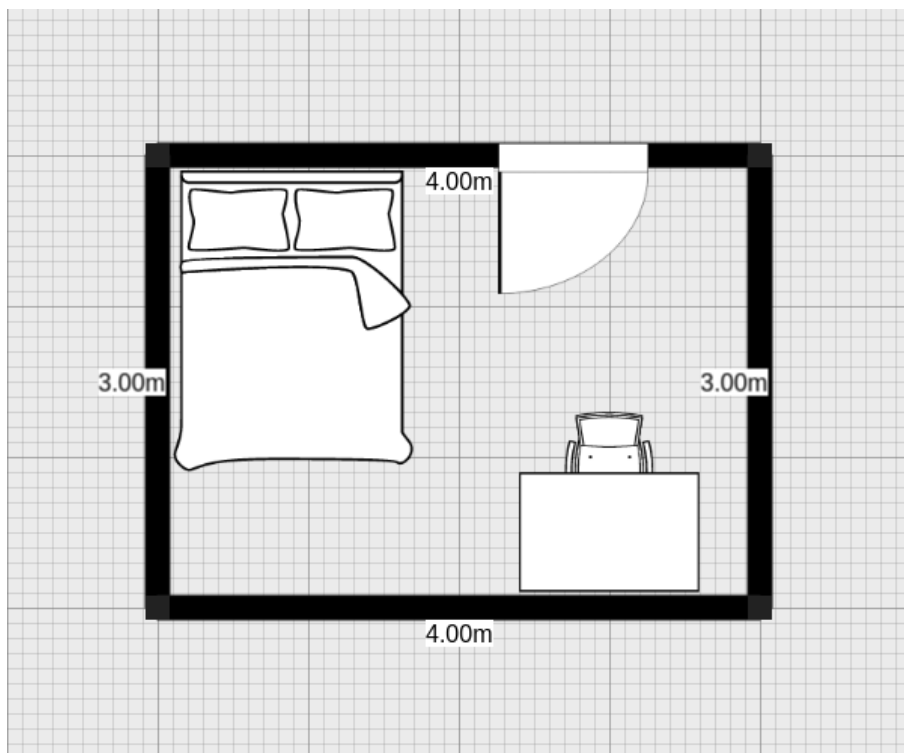


Figura 3.10: Planul unei camere de cămin, obținut prin încărcarea în aplicație a fișierului anterior

Metoda `load()` execută pașii opuși celei de `save()`, **iterand** prin elementele serializate și deserializându-le. Constructorul etajului poate primi aceste date ca parametru opțional, caz în care va crea nouă **instanța** respectând specificațiile date. Mai întâi va fi reconstituită structura pereților, urmând că apoi să fie create noi obiecte și să fie atașate etajului.

```
1      if (floorData) {
2          let nodeLinks = new Map<number, number[]>(floorData.
wallNodeLinks)
3
4          this.wallNodeSequence.load(floorData.wallNodes, nodeLinks);
5          for (let fur of floorData.furnitureArray) {
6              let furnitureData: FurnitureData = {
7                  width: fur.width,
8                  height: fur.height,
9                  imagePath: fur.texturePath
10             };
11             let attachedTo = this.wallNodeSequence.getWall(fur.
attachedToLeft, fur.attachedToRight)
12             let object = new Furniture(furnitureData, fur.id,
attachedTo)
13             this.furnitureArray.set(fur.id, object);
14
15             if (attachedTo !== undefined) {
16                 attachedTo.addChild(object)
17             } else {
18                 this.addChild(object)
19             }
20             object.position.set(fur.x, fur.y)
21             object.rotation = fur.rotation;
22         }
23         return;
24     }
```

3.5 Server-ul

Componenta de server a proiectului Arcada are ca responsabilitate oferirea accesului utilizatorilor la colecția elementelor de mobilă. În acest scop, ea expune următoarele endpoint-uri de API:

- `GET /categories`: returnează un obiect JSON conținând lista tipurilor de mobilă existente în baza de date.
- `GET /category/:categoryId`: returnează un obiect JSON conținând lista tuturor elementelor de mobilier care aparțin categoriei `categoryId`, pentru a putea fi afișate utilizatorului

- GET /furniture/:furnitureId: returnează un obiect JSON conținând detaliile unui obiect de mobilă, precum numele, dimensiunile, și calea către grafica sa
- GET /2d/:filename: dat fiind un nume al unei resurse grafice (obținut prin endpoint-ul de mai sus), returnează imaginea asociată acesteia, sub formă de blob (șir de octeti).

Adăugarea, editarea detaliilor, sau ștergerea obiectelor de pe back-end poate fi făcută doar de administrator, prin intermediul endpoint-urilor de API accesibile doar cu credențialele de administrator:

- POST /login cu body conținând user-ul și parolă contului de administrator: dacă autentificarea are loc cu succes, returnează un token JWT care permite accesul la funcționalitățile de administrare. Tokenul este salvat local și transmis ulterior în fiecare request pentru autentificare.
- POST /category cu body { name:string, visible:bool }: creează o nouă categorie cu numele și flag-ul de vizibilitate dat, returnează id-ul categoriei nou create
- POST /category/add cu body {name:string, width:float, height:float, categoryId:string, textureName:string}: creează un nouă piesă de mobilier și o adaugă categoriei date. Returnează id-ul obiectului nou creat
- POST /furniture/image: salvează imaginea uploadată de administrator în folderul de resurse.
- PUT /category: permite modificarea detaliilor categoriei specificate în body prin id.
- PUT /furniture: permite modificarea detaliilor piesei de mobilier specificate în body prin id.
- DELETE /category: șterge atât categoria precizată în body prin id, cât și toate elementele de mobilă din acea categorie.
- DELETE /furniture: șterge piesa de mobilier cu id-ul dat în body.

În implementarea comunicării dintre server și baza de date am apelat la Mongoose, care este o **biblioteca** ce efectuează o mapare între obiecte și documente (eng. „Object Document Mapper”). Astfel, în loc să rulez în mod direct interogări asupra bazei de date, am conectat Mongoose la MongoDB, și am folosit obiectele create de aceasta pentru a efectua modificări. La final, am scris schimbările folosind operațiunea de tip „commit”.

3.5.1 Comunicarea între client și server

Folosind Fetch API, client-ul Arcada trimite cereri către server, în funcție de resursele dorite. Rezultatele acestor cereri vor fi salvate în store-urile corespunzătoare. De exemplu, cererea pentru lista categoriilor arată astfel:

```
1 export function getCategoriesRequest() {  
2   return fetch(endpoint + "/categories")  
3 }
```

unde `endpoint` este calea către server. La deschiderea aplicației, vor fi inițializate locurile unde se stochează informația, denumite store-uri. Prin apelarea `getCategories()`, datele prezente în store despre categorii vor fi actualizate.

```
1 export const useFurnitureStore = create<FurnitureStore>(set => ({  
2   categories: [],  
3   getCategories: async () => {  
4     let res = await(await getCategoriesRequest()).json()  
5     if (!res.err) {  
6       set(() => ({  
7         categories: res  
8       }));  
9     }  
10  })  
11  })))
```

3.6 Comunicarea între modulele aplicației

În crearea aplicației, am aplicat design pattern-ul structural Command, transformând fiecare acțiune pe care userul o poate face asupra planului etajului într-o clasă. Prin acest design pattern, am separat logica editorului de cea a interfeței, evitând să apelez direct funcțiile din `FloorPlan`.

Command design pattern este caracterizat de încapsularea datelor necesare efectuării acțiunii corespunzătoare într-un obiect. Clasele care urmăresc acest pattern expun o metodă de `execute()`, prin care se poate declanșa acțiunea.

Prin acest mod de a reprezenta acțiunile ce au loc în editor, se pot înfăptui sisteme care permit navigarea prin istoricul acțiunilor.

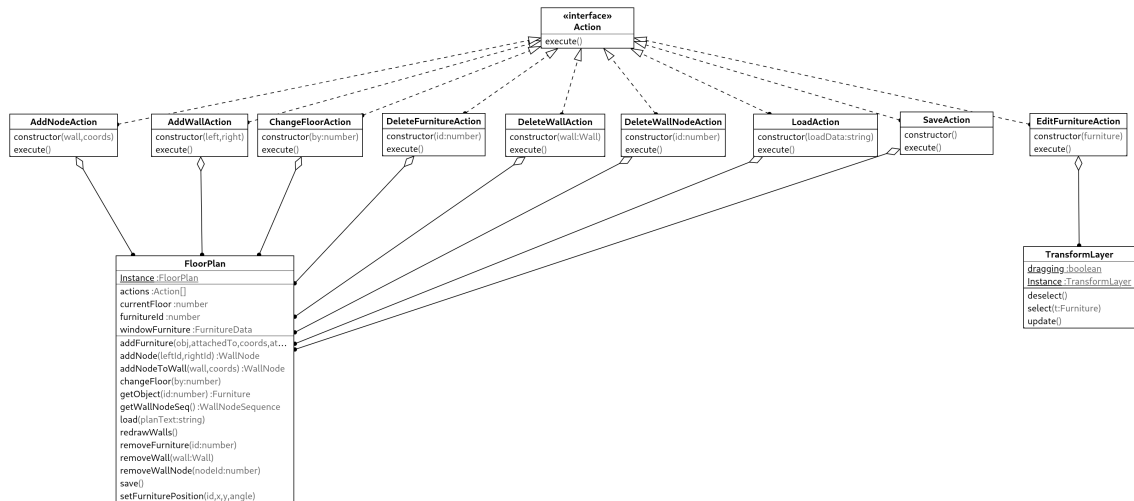


Figura 3.11: Diagrama UML a comenzilor

În plus, am decis că variabilele ale căror valori sunt utilizate atât de modulul editor cât și modulul interfață să fie reținute într-o zonă separată, accesibilă amândurora. Pentru a îndeplini acest scop, am definit store-ul **EditorStore**, în care se rețin **informații** despre unealta curent selectată din editor și despre modul curent de lucru. La apăsarea butoanelor din interfață, valorile vor fi setate. Unelele din editor vor funcționa doar dacă valorile setate se potrivesc.

Capitolul 4

Ghid de utilizare al aplicației

Atunci când un utilizator accesează **pagina** Arcada, el va fi întâmpinat de panoul de bun-venit, care îi oferă opțiunea de a crea un plan nou sau de a încărca unul existent.

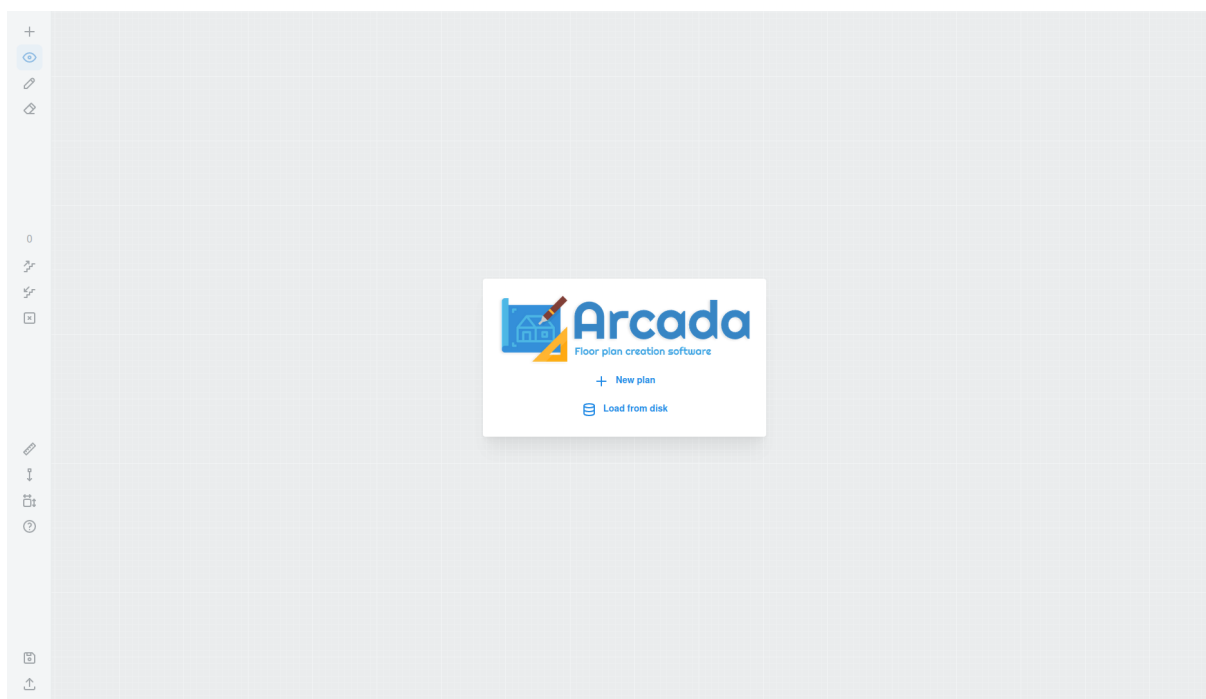
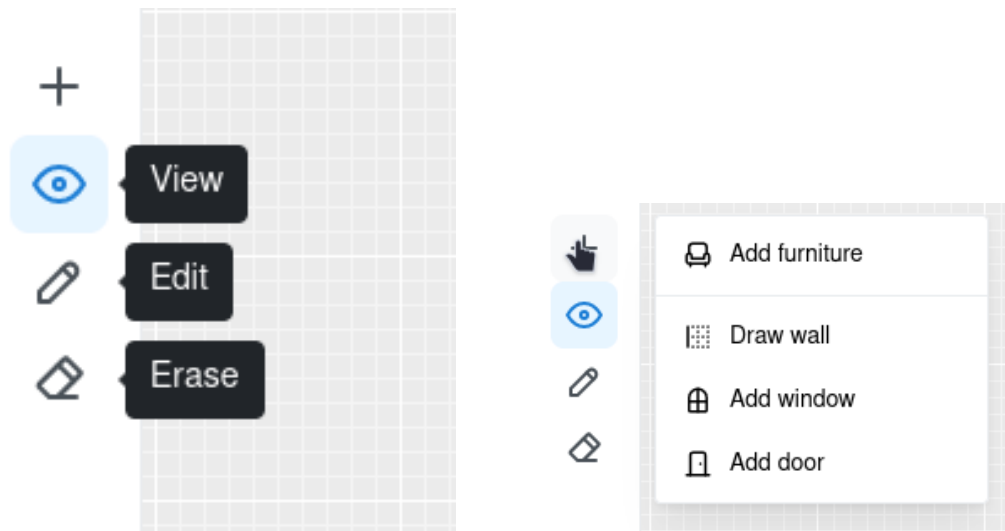


Figura 4.1: Panoul de start al aplicației

Selectarea oricărei dintre cele două opțiuni va debloca accesul utilizatorului asupra **barii** de unelte, aflată în partea stânga. Uneltele sunt organizate în patru grupuri, în funcție de scopul lor, și pot fi selectate apăsând pe butoanele corespunzătoare. La așezarea mouse-ului pe un buton, va fi vizibil numele uneltei.

Prima zonă permite utilizatorului să adauge, modifice, sau șterge elementele din plan. **Această** este compusă din modurile de adăugare, vizualizare, editare, și ștergere. Meni-ul de adăugare permite inserarea obiectelor de mobilier, desenarea pereților, ușilor sau ferestrelor.



(a) modurile de interacționare cu planul

(b) meniu adăugare obiecte

Figura 4.2: Uneltele cu care se pot manipula elementele din plan

4.1 Adăugarea elementelor în plan

4.1.1 Desenarea pereților

Selectând opțiunea „Draw wall” din meniul de adăugare, utilizatorul poate începe să deseneze pereții încăperilor sale.

Desenarea pereților se face prin apăsarea pe spațiul de lucru. Prima apăsare va seta coordonatele de început ale peretelui. Mișcând cursorul, utilizatorul poate determina lungimea dorită a peretelui, cât și orientarea acestuia, ei putând fi orizontali, verticali sau oblici. La o apăsare ulterioară, peretele este salvat, iar utilizatorul poate continua să deseneze un alt perete care să fie conectat cu acesta, sau poate să se oprească din desenat prin dublu-click.

Dacă utilizatorul face primul click pe un capăt al unui perete, și al doilea click pe un alt capăt, peretele nou desenat va fi conectat acestora.

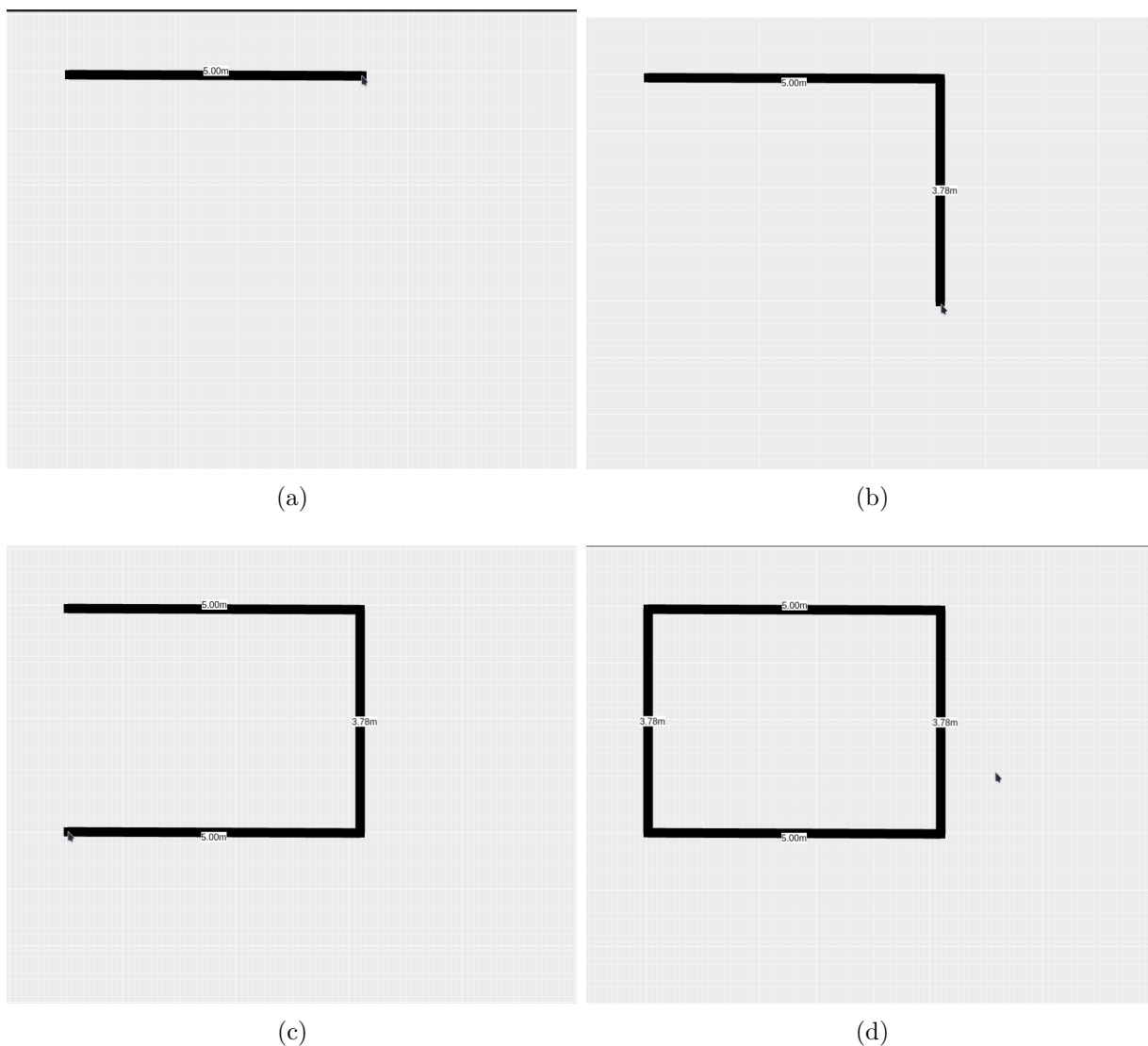


Figura 4.3: Pașii prin care se desenează o cameră în plan

Dacă utilizatorul se află în modul de adăugare pereți și apasă pe un perete existent, zidul nou adăugat va fi conectat de acesta, iar toate modificările, redimensionările ulterioare se vor **aplica** tuturor pereților conectați. Prin acest procedeu, spațiul se poate delimita în camere.

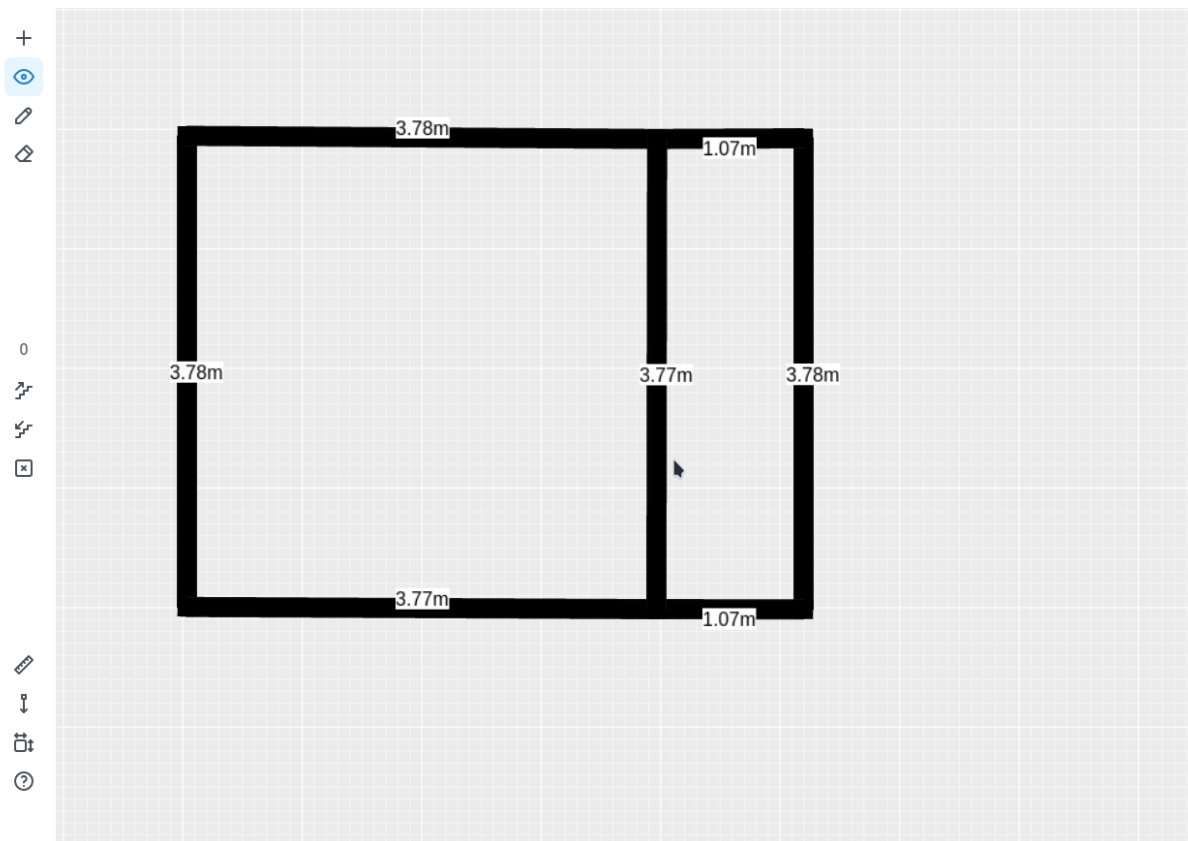


Figura 4.4: Delimitarea spațiului din plan în mai multe camere

4.1.2 Modificarea pereților

Folosind modul de editare, desemnat în bara de unelte cu o pictogramă a unui creion, utilizatorul poate face corecturi asupra dimensiunilor, pozițiilor pereților deja aflați în plan. De asemenea, el poate desemna pereții exteriori ai clădirii.

În mod implicit, pereții desenați sunt considerați pereți interiori. La apăsarea click dreapta pe un perete, acesta va fi transformat în perete exterior. Pereții exteriori se diferențiază de cei prin interiori atât vizual, prin grosime, cât și prin comportament.

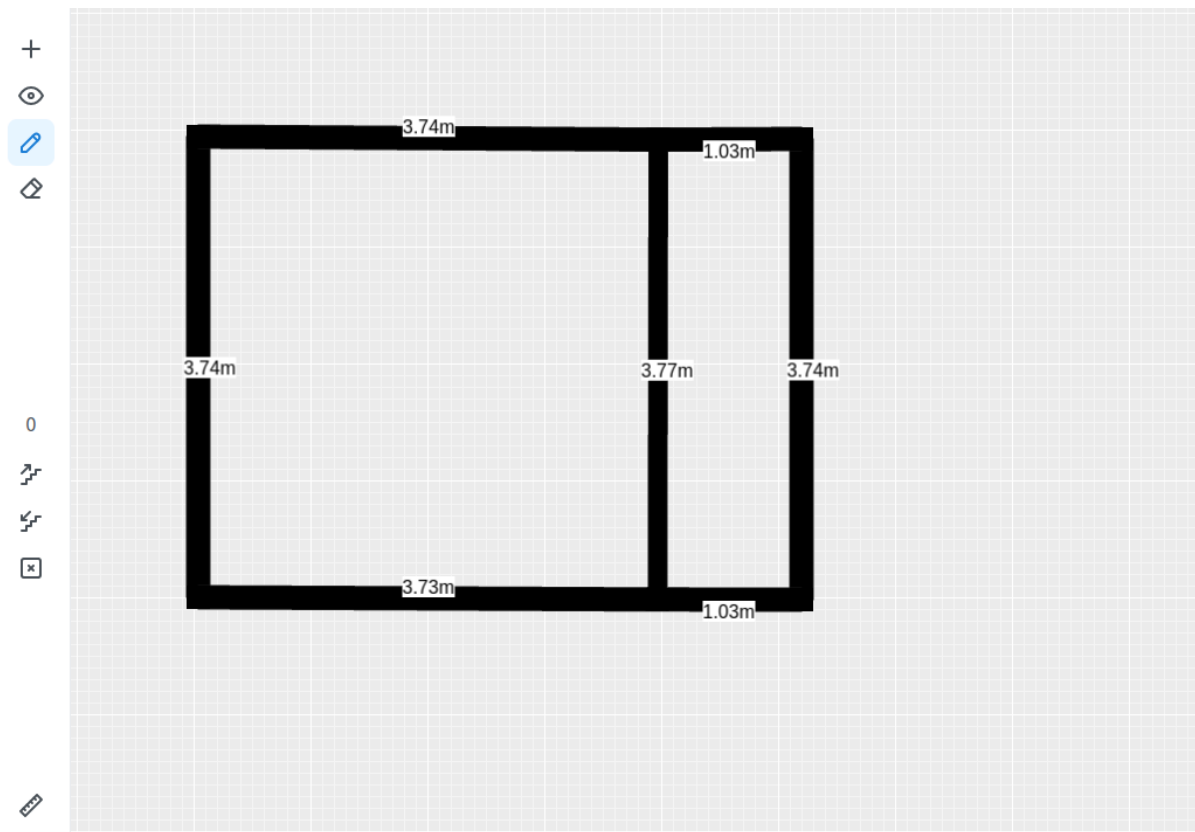


Figura 4.5: Setarea pereților exteriori

La adăugarea unui nou etaj planului existent, doar pereții exteriori ai etajului precedent se vor păstra, deoarece compartimentarea în camere a unui etaj superior este, în mod uzual, diferită de cea a etajului inferior.

Apăsând pe extremitățile pereților și mișcând mouse-ul, aceștia pot fi modificați. Dacă este mutat un punct în care mai mulți pereți se întâlnesc, modificările se aplică tuturor pereților conectați, iar utilizatorul poate vedea acest lucru în timp real.

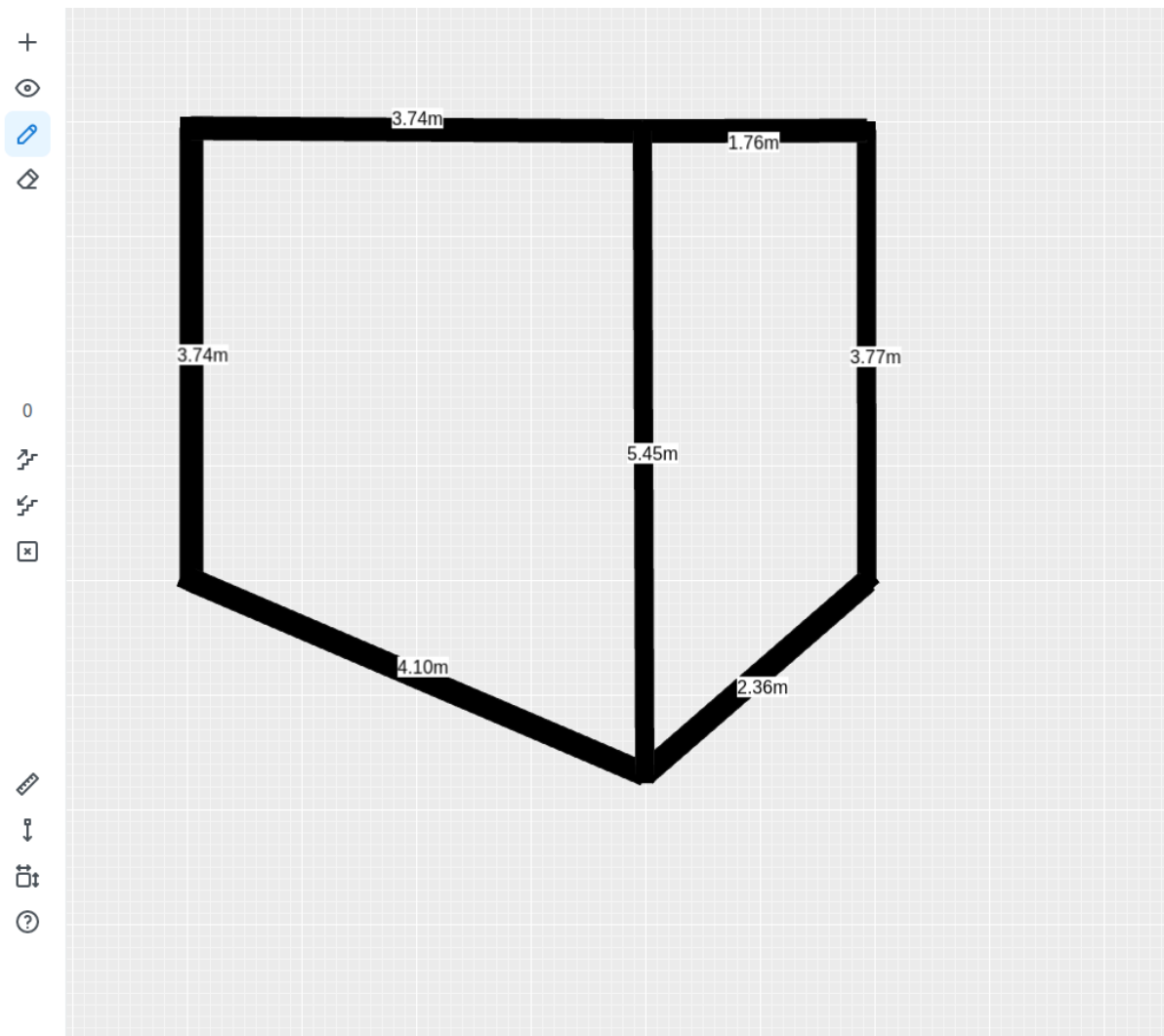


Figura 4.6: Mutarea unui punct de întâlnire a mai multor pereți. Astfel, se pot reprezenta încăperi ale căror dimensiuni sunt neobișnuite

4.1.3 Adăugarea mobilierului

Pentru a adăuga mobilă în plan, utilizatorul trebuie să acceseze meniul „Add furnitu-re”, ilustrat în figura 5.2 (b). La apăsarea acestui buton, în partea dreapta a ecranului se va deschide colecția de elemente de mobilier disponibile pentru adăugare.

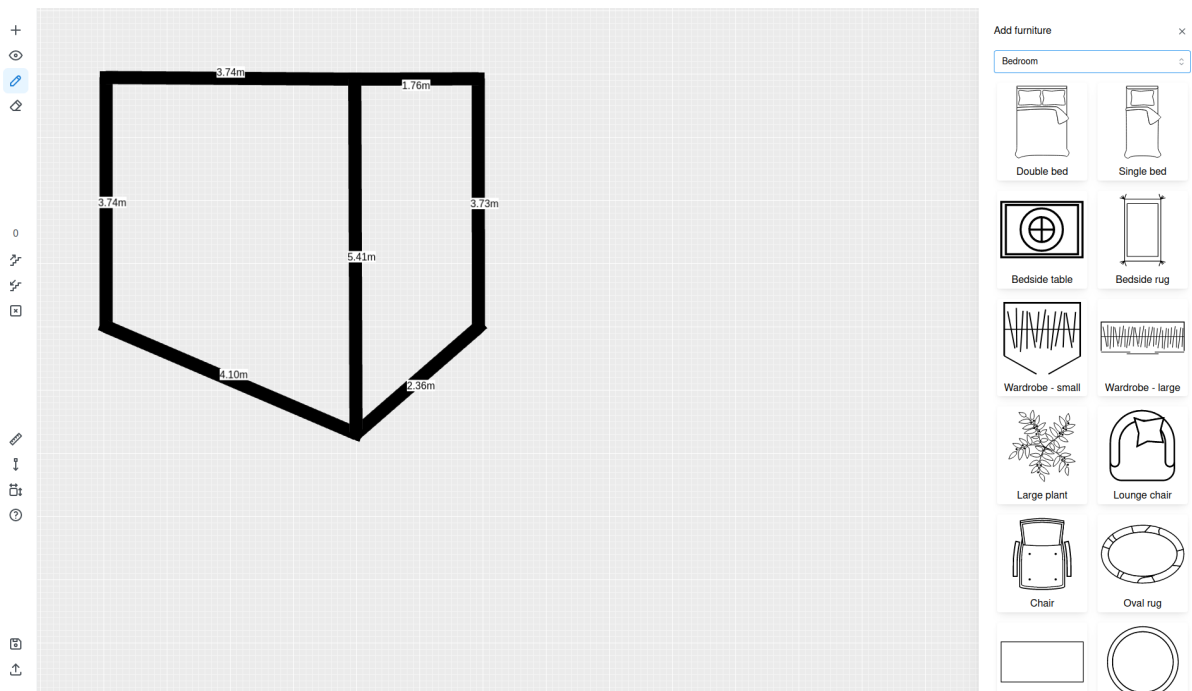


Figura 4.7: Panoul care conține elementele de mobilier disponibile într-o încăpere

Pentru o utilizare facilă a aplicației, elementele sunt grupate în categorii, în funcție de camera unde sunt folosite în mod uzual.

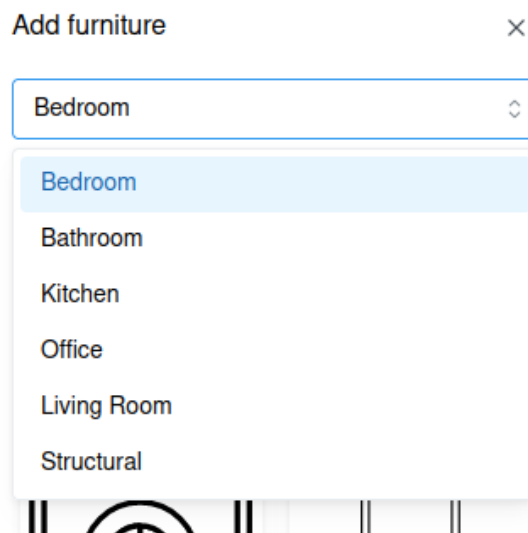


Figura 4.8: Selectarea unei alte categorii de mobilier

Utilizatorul poate apăsa pe oricare dintre elementele din panou pentru a le adăuga în plan.

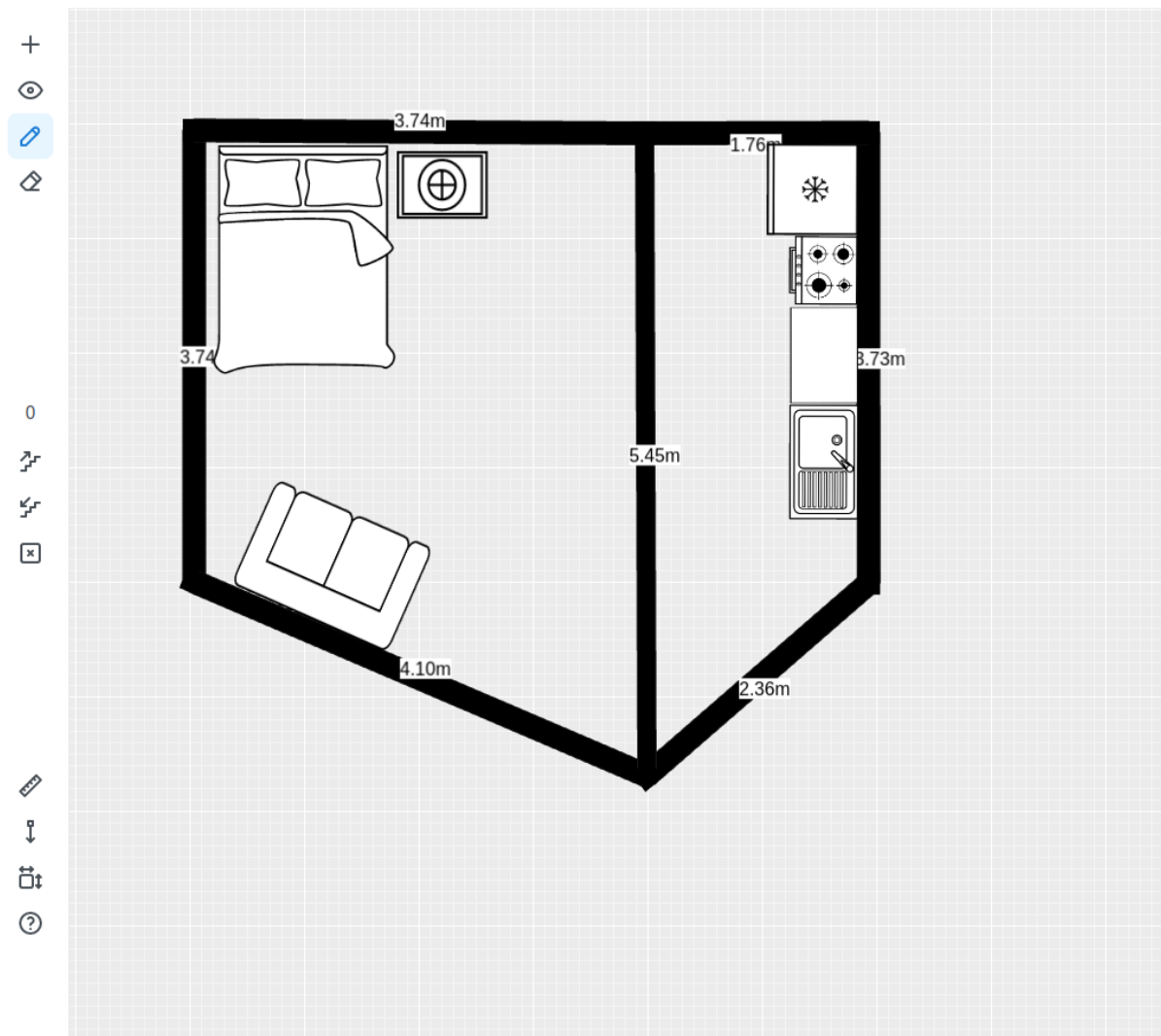


Figura 4.9: Planul după adăugarea a câtorva elemente de mobilier

4.1.4 Modificarea mobilierului

Atunci când modul Edit este selectat, utilizatorul poate apăsa pe orice element de mobilier pentru a îl edita. Obiectul va fi înconjurat de un chenar și îi vor fi afișate dimensiunile. Prin interacțiunea cu butoanele apărute pe marginea chenarului, utilizatorul poate redimensiona obiectul. Butonul rotund permite rotirea obiectului, iar cel poziționat central, deplasarea sa în plan.

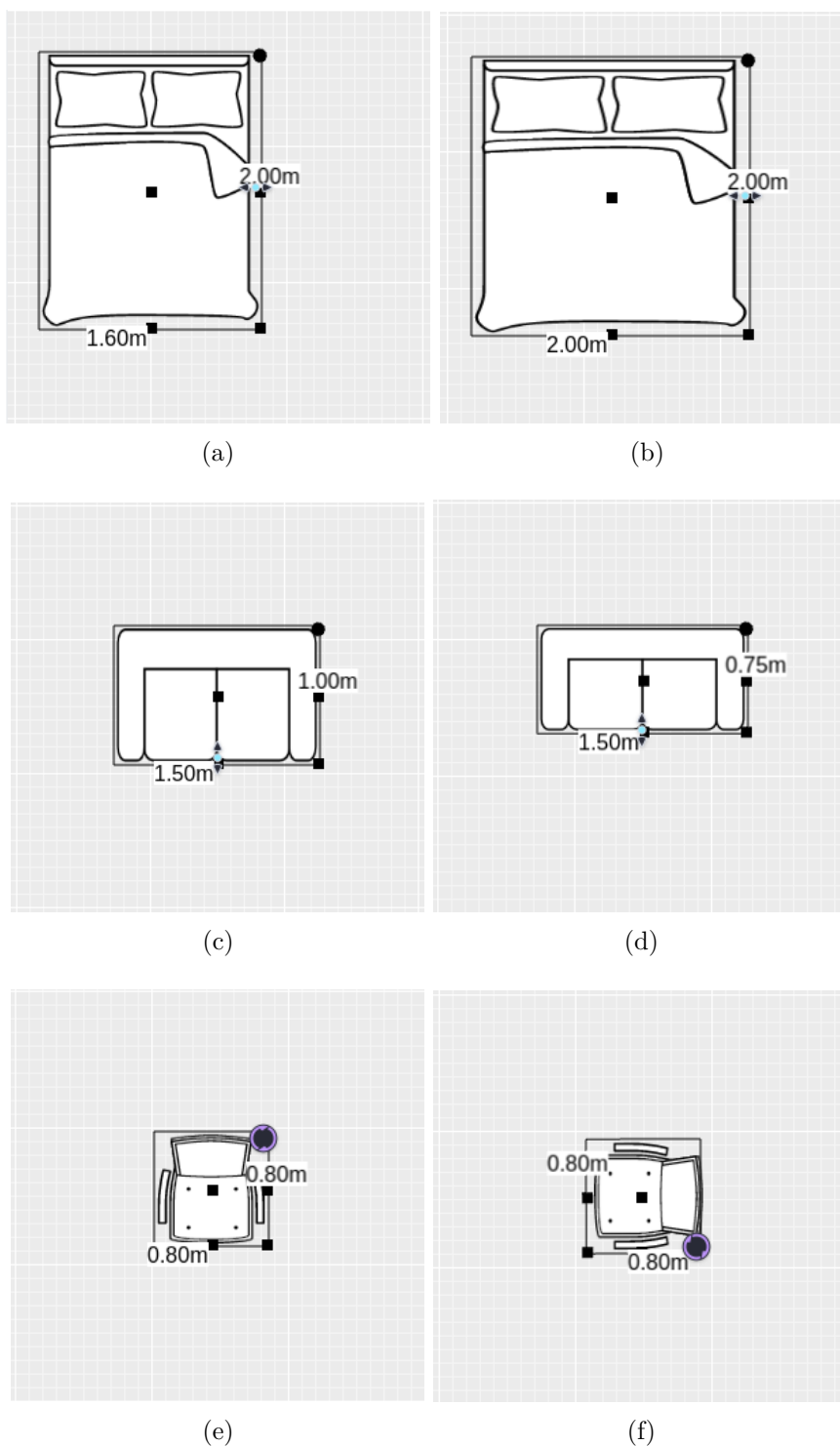


Figura 4.10: Transformări aplicate asupra a diferite obiecte

4.1.5 Adăugarea elementelor structurale

În completarea elementelor de mobilier și a pereților, apar elemente precum ușile, ferestrele. Utilizatorul poate alege să adauge aceste elemente accesând opțiunile „Add window”, respectiv „Add door”, din meniul ilustrat în Figura 5.2 (b). Ușile, ferestrele,

pot fi adăugate prin apăsarea pe perețele unde se doresc a fi amplasate, ele neputând exista în plan fără a fi atașate unui perete. Poziția balamalelor ușii poate fi modificată prin apăsarea butonului **de** click dreapta pe ușa dorită.

Modificările aplicate asupra pereților se vor propaga și elementelor atașate acestora.

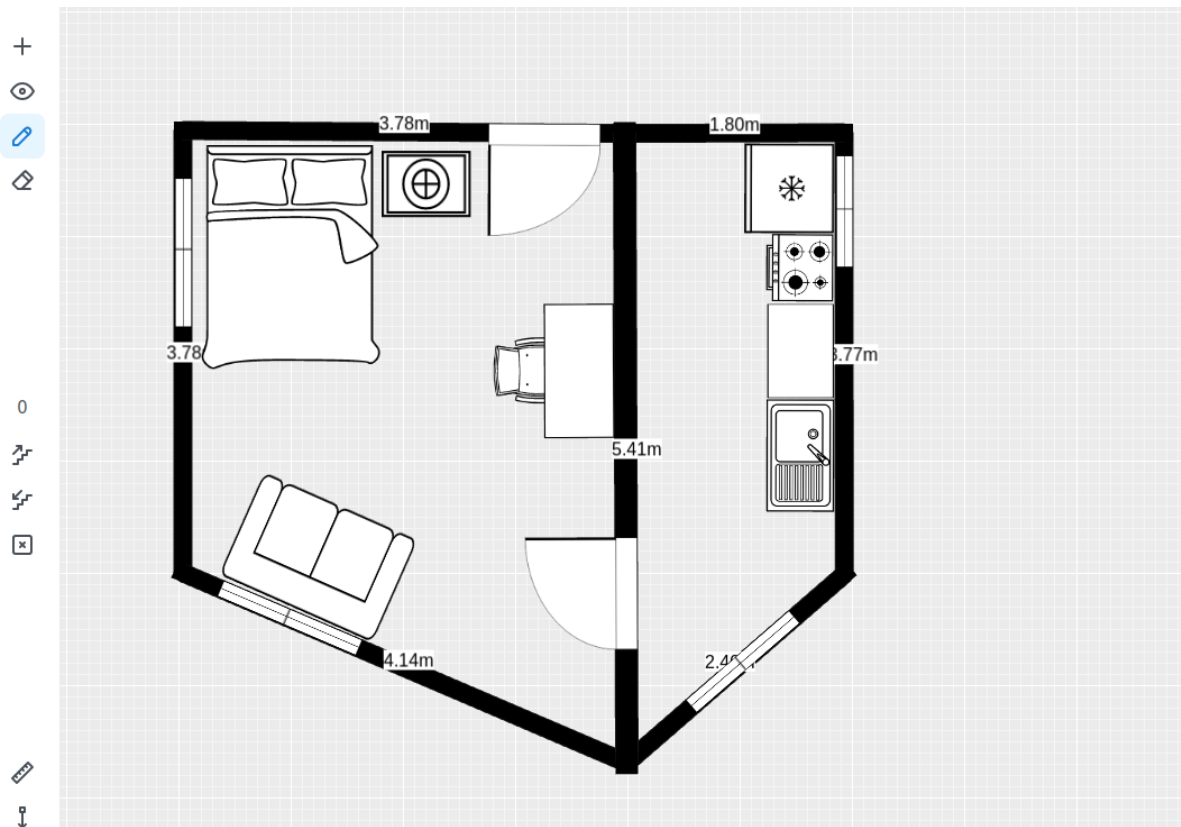


Figura 4.11: Aspectul planului după adaugarea ușilor și a ferestrelor

4.1.6 Ștergerea elementelor din plan

Modul „Erase”, accesibil prin meniul ilustrat în figura 5.2 (b), funcționează precum o radieră. Având acest mod selectat, utilizatorul poate apăsa pe elementele din plan pentru a le șterge.

4.2 Gestionarea etajelor

În a doua secțiune a panoului din stânga ecranului se află opțiunile cu privire la controlul etajelor.

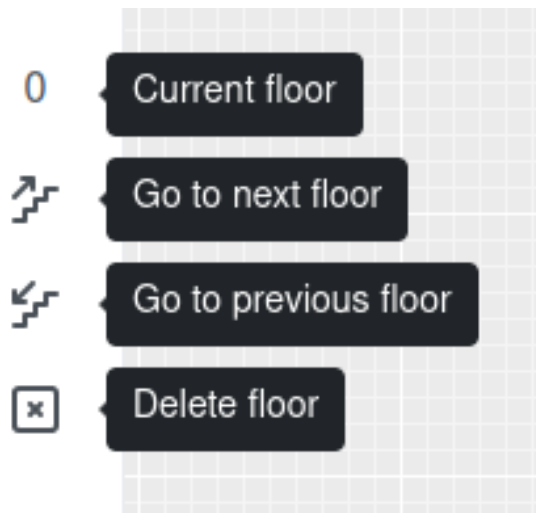


Figura 4.12: Meniul de control al etajelor

Butoanele „Go to next floor”, „Go to previous floor” îi permit utilizatorului să vizualizeze etajele din care este compus din planul său. Dacă utilizatorul încercă să navigheze la un etaj care nu există, acesta va fi creat. Pereții marcați în etajul precedent ca fiind exteriori vor fi copiați în noul etaj.

Butonul „Delete floor” are ca scop ștergerea etajului curent. După obținerea confirmării utilizatorului, etajul pe care se află acesta va fi șters. Orice etaj poate fi șters, cu excepția parterului.

4.3 Unelte ajutătoare

A treia zona a panoului din interfață întrunește uneltele auxiliare, care au scopul de a ajuta utilizatorul pe parcursul creării planului său interior.

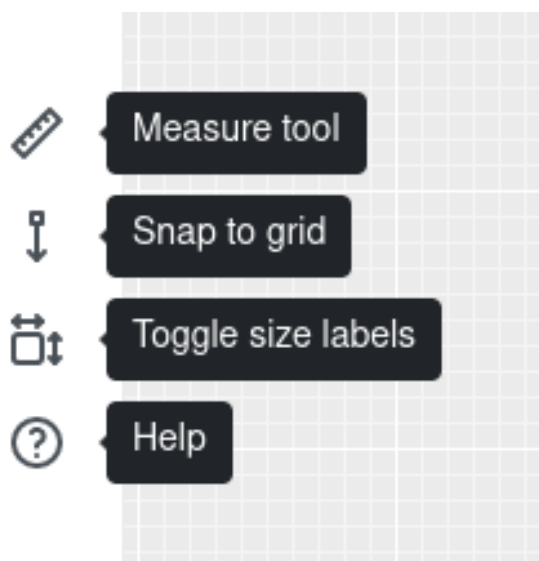


Figura 4.13: Meniul uneltelor ajutătoare

4.3.1 Vizualizarea planului: zoom si panning

Mișcând roțița mouse-ului, utilizatorul are opțiunea de a mări sau micșora planul, permițând astfel vizualizarea în detaliu a unei zone de pe ecran. Prin apăsarea pe o porțiune a ecranului fără vreun obiect amplasat, apoi mutarea cursorului, se poate naviga prin spațiul planului.

4.3.2 Măsurarea dimensiunilor din plan

Pentru a putea verifica distanțele dintre obiecte, utilizatorul are la dispoziție o unealtă care permite verificarea distanțelor dintre oricare două puncte. Astfel, el se poate asigura că dimensiunile din planul său sunt corecte.

4.3.3 Facilitarea alinierii elementelor

Funcționalitatea de grid vine în ajutorul utilizatorului permițând alinierea ușoară a obiectelor. În mod implicit, atât mobilierul, cât și pereții planului vor fi aliniindu-se cu liniile ajutătoare aflate la interval de 10 centimetri.

Prin apăsarea butonului etichetat „Snap to grid”, această funcționalitate poate fi activată sau dezactivată.

4.3.4 Afișarea și ascunderea dimensiunii obiectelor

Al treilea buton din zona de unelte ajutătoare oferă utilizatorului de a afișa sau ascunde etichetele cu dimensiunile pereților.

4.3.5 Modul ajutor

Acest buton afișează în colțul dreapta-sus informații cu privire la modul de lucru, unealta curent activă, cu scopul de a ușura utilizarea aplicației. Fereastra poate fi închisă prin apăsarea butonului marcat cu „x” în colțul din dreapta-sus. Atunci când utilizatorul schimbă unealtă activă, fereastra, dacă este activă, se va actualiza automat.

4.4 Salvarea și încărcarea planului

Ultima secțiune din panoul de unelte are ca scop salvarea, încărcarea sau printarea planurilor.

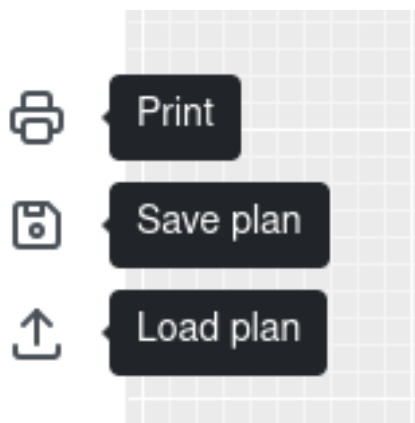


Figura 4.14: Meniul de salvare, încărcare, exportare

Primul buton, cel de „Print”, exportă planul curent sub formă de imagine, apoi deschide dialogul de imprimare. Utilizatorul va putea astfel fie să imprime etajul curent pe suport hârtie, fie să îl salveze în format PDF sau PNG.

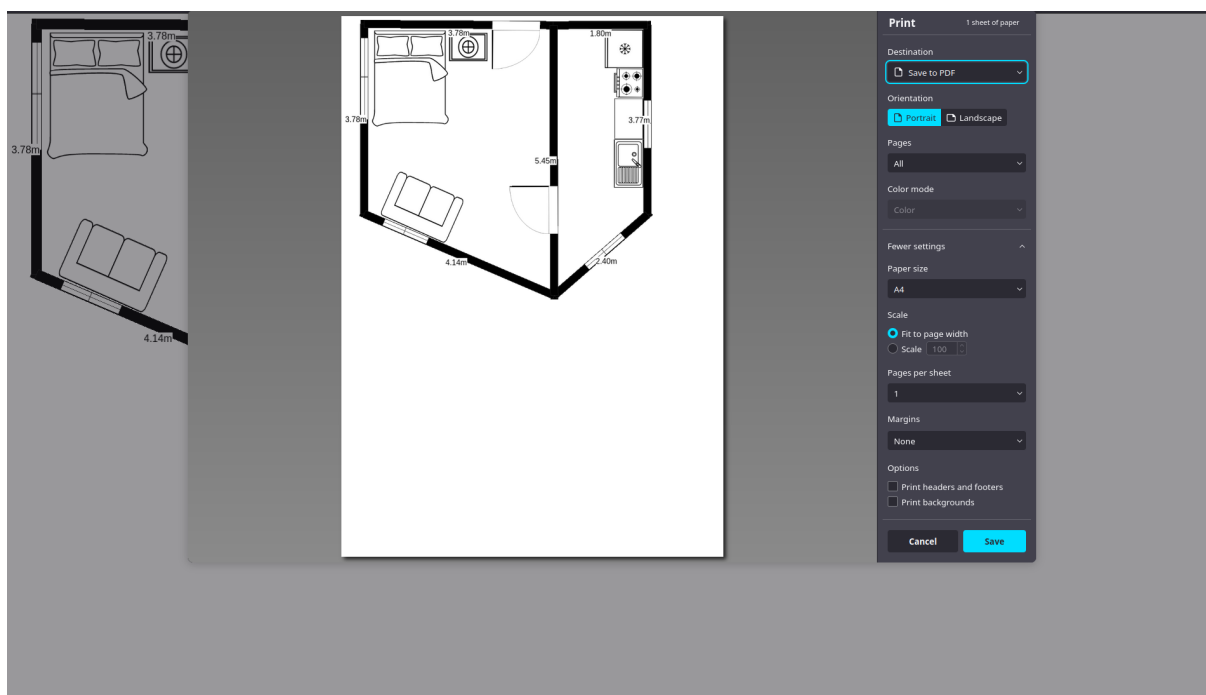


Figura 4.15: Imprimarea unui plan

La apăsarea butonului „Save”, utilizatorul va fi întâmpinat cu o fereastră în care va putea alege unde să salveze planul creat, într-un format care permite încărcarea și editarea ulterioară a planului. Informațiile care compun planul sunt extrase, serializate, și salvate în format JSON.

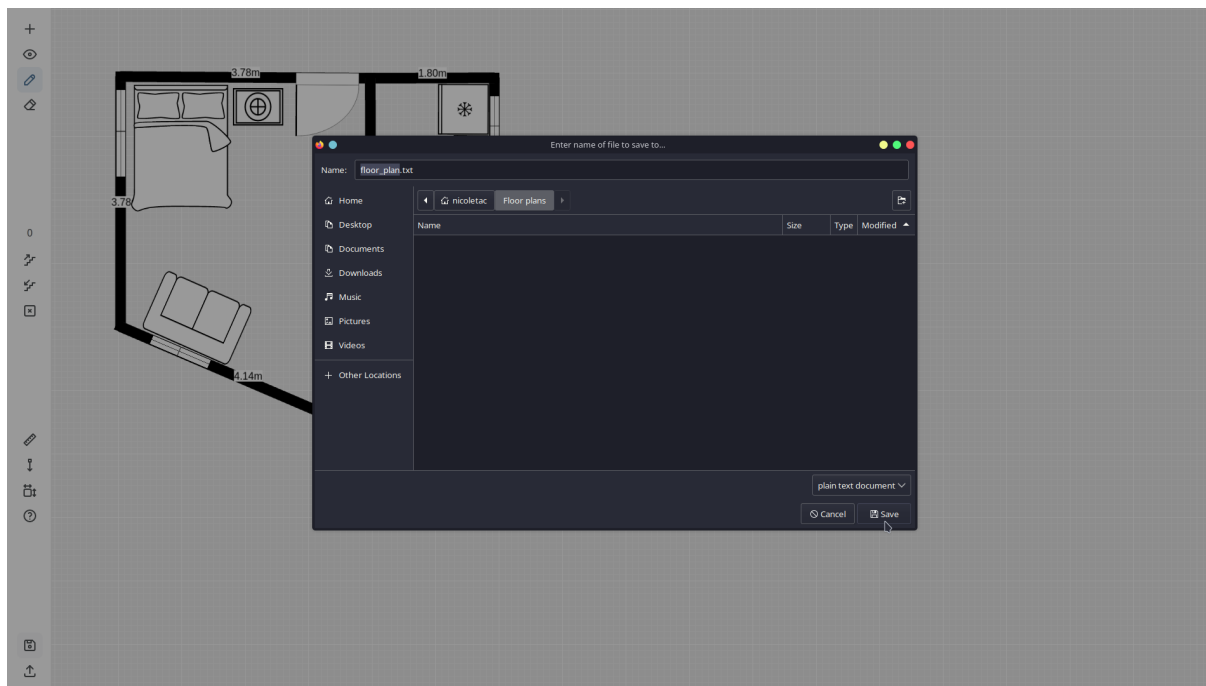


Figura 4.16: Salvarea unui plan

Dacă utilizatorul apasă butonul „Load”, va fi întâmpinat cu o fereastră similară celei din cadrul opțiunii „Save”, cu excepția că acum el va selecta un fișier-plan ale cărui informații vor fi deserializate și încărcate în aplicație.

Capitolul 5

Concluzii

Proiectul Arcada propune o **aplicatie** ce ținteste un segment de piață unde nu există concurență semnificativă în momentul de față. Majoritatea aplicațiilor din ziua de azi au ca scop final ușurarea diferitelor îndeletniciri din viața de zi cu zi, iar platforma respectă acest principiu. Utilitatea ei constă în ajutorul pe care îl oferă utilizatorilor în a crea **schite** ale locuințelor lor, mai rapid, mai estetic, și cu risc diminuat de a greși.

Există mai multe oportunități de dezvoltare a aplicației, atât pe parte tehnică, cât și la nivel de produs în sine. Fiind un proiect modular, Arcada poate fi personalizat cu dataset-uri conținând reprezentări ale obiectelor ale unor producători de mobilă, și poate fi inclus în pagini web existente. Astfel, companii de dimensiuni mici și medii pot opta să includă un planificator pe paginile lor, fără a fi nevoiți să își creeze propria soluție de management de la zero. Prin adaptarea backend-ului la cerințele firmelor, există chiar posibilitatea de a genera colecțiile de obiecte automat, folosindu-se bazele de date pe care companiile le au deja setate.

La nivel tehnic, **extindere a** aplicației ar putea **constă** în adăugarea opțiunii de a randa 3D planul locuinței. Asociind fiecărui obiect în 2D un model 3D, utilizatorul va avea posibilitatea de a vizualiza planul locuinței sale, pentru a își face o idee mai bună asupra deciziilor pe care le va lua. Alte idei ar putea fi extinderea colecției de obiecte cu mai multe variante ale aceleiași piese de mobilier sau posibilitatea modificării podelelor.

Bibliografie

- [1] Ahcène Bounceur, Madani Bezoui și Reinhardt Euler, *Boundaries and Hulls of Euclidean Graphs: From Theory to Practice*, Iul. 2018, pp. 1, 7, 13, ISBN: 9781138048911, DOI: [10.1201/9781315169897](https://doi.org/10.1201/9781315169897).
- [2] Carmen Carmackeff și Jeff Tyson, *How Computer Monitors Work*, 2022, URL: <https://computer.howstuffworks.com/monitor7.html> (accesat în 13.3.2022).
- [3] Statista Research Department, *Frequency of renovations among homeowners who renovated 2015-2020*, 2021, URL: <https://www.statista.com/statistics/449541/frequency-of-renovations-among-homeowners-who-renovated/> (accesat în 25.3.2022).
- [4] PixiJS Official Documentation, 2022, URL: <https://pixijs.com/> (accesat în 19.5.2022).
- [5] PixiJS Official Documentation, *PixiJS Guides - Working with Sprites*, 2022, URL: <https://pixijs.io/guides/basics/sprites.html> (accesat în 19.5.2022).
- [6] PixiJS Official Documentation, *The Scene Graph*, URL: <https://pixijs.io/guides/basics/scene-graph.html> (accesat în 20.5.2022).
- [7] ReactJS Official Documentation, *Components and Props*, 2021, URL: <https://reactjs.org/docs/components-and-props.html> (accesat în 15.4.2022).
- [8] ReactJS Official Documentation, *Virtual DOM and Internals*, 2021, URL: <https://reactjs.org/docs/faq-internals.html> (accesat în 11.4.2022).
- [9] Rob Ferguson, *HTML5 Canvas or SVG?*, URL: <https://robferguson.org/blog/2015/12/30/html5-canvas-or-svg/> (accesat în 12.5.2022).
- [10] David Figatner, 2018, URL: <https://davidfig.github.io/pixi-viewport/> (accesat în 12.3.2022).
- [11] Josh Fruhlinger, *What is TypeScript? Strongly typed JavaScript*, 2020, URL: <https://www.infoworld.com/article/3538428/what-is-typescript-strongly-typed-javascript.html> (accesat în 11.4.2022).

- [12] Ana Gregurec, *Web Animation In The Post-Flash Era*, 2018, URL: <https://www.toptal.com/designers/web/animating-the-web-in-the-post-flash-era> (accesat în 29.5.2022).
- [13] Esteban Herrera, *Immutability in React: There's nothing wrong with mutating objects*, 2018, URL: <https://blog.logrocket.com/immutability-in-react-eb55253a1cc/> (accesat în 11.3.2022).
- [14] *ImpactJS*, URL: <https://impactjs.com/> (accesat în 11.3.2022).
- [15] Houzz Home Inc., *Overview of U.S. Renovation, Custom Building Decorating in 2014*, 2015, URL: <https://web.archive.org/web/20150727042141/http://info.houzz.com/rs/houzz/images/HouzzAndHomeReportJune2015.pdf> (accesat în 20.3.2022).
- [16] Stack Overflow Inc., *Most popular web frameworks among developers worldwide 2021*, 2021, URL: <https://insights.stackoverflow.com/survey/2021> (accesat în 29.3.2022).
- [17] Stack Overflow Inc., *What is MongoDB? Introduction, Architecture, Features Example*, 2020, URL: <https://2020.stateofjs.com/en-US/>.
- [18] Mehdi Jazayeri, „Some Trends in Web Application Development”, în *Future of Software Engineering (FOSE '07)*, 2007, pp. 6–7, DOI: [10.1109/FOSE.2007.26](https://doi.org/10.1109/FOSE.2007.26).
- [19] Nick Karnik, *Introduction to Mongoose for MongoDB*, 2022, URL: <https://www.freecodecamp.org/news/introduction-to-mongoose-for-mongodb-d2a7aa593c57/> (accesat în 29.5.2022).
- [20] Vladimir Klepov, *How to destroy your app performance using React contexts*, 2021, URL: <https://blog.thoughtspile.tech/2021/10/04/react-context-dangers/> (accesat în 11.3.2022).
- [21] Dillion Megida, *Why Does React Exist*, 2020, URL: <https://egghead.io/blog/why-does-react-exist-in-the-first-place> (accesat în 11.5.2022).
- [22] Mozilla Developer Network, *Introduction to the DOM*, 2022, URL: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction (accesat în 11.4.2022).
- [23] *PhaserJS*, URL: <https://phaser.io/> (accesat în 11.3.2022).
- [24] Vitaly Rtishchev, *Mantine Component Library*, 2021, URL: <https://mantine.dev/> (accesat în 15.4.2022).
- [25] Simplilearn, *What Is Express JS In Node JS*, 2022, URL: https://www.simplilearn.com/tutorials/nodejs-tutorial/what-is-express-js?source=sl_frs_nav_playlist_video_clicked (accesat în 11.3.2022).

- [26] *SketchUp*, URL: <https://www.sketchup.com/> (accesat în 11.3.2022).
- [27] Anne Michelle Tabirao, *What is MongoDB? Introduction, Architecture, Features Example*, 2022, URL: <https://ubuntu.com/blog/what-is-mongodb> (accesat în 29.5.2022).
- [28] John Tucker, *React Anti-Pattern: Prop-Drilling*, 2018, URL: <https://codeburst.io/react-anti-pattern-prop-drilling-54474d5236bd> (accesat în 19.3.2022).