UNIVERSITATEA „ALEXANDRU IOAN CUZA” IAŞI

FACULTATEA DE INFORMATICĂ

LUCRARE DE LICENŢĂ

Dezvoltarea jocurilor Web cu HTML5 și WebGL. Un prototip de joc de tip labirint

propusă de

Ioan Ungurean

Sesiunea: iulie, 2017

Coordonator ştiinţific:

Conf. Dr. Sabin-Corneliu Buraga

UNIVERSITATEA „ALEXANDRU IOAN CUZA” IAŞI

FACULTATEA DE INFORMATICĂ

Dezvoltarea jocurilor Web cu HTML5 și WebGL. Un prototip de joc de tip labirint

Ioan Ungurean

Sesiunea: iulie, 2017

Coordonator ştiinţific:

Conf. Dr. Sabin-Corneliu Buraga

DECLARAŢIE PRIVIND ORIGINALITATEA ŞI RESPECTAREA DREPTURILOR DE AUTOR

Prin prezenta declar că Lucrarea de licenţă cu titlul "Dezvoltarea jocurilor Web cu HTML5 și WebGL. Un prototip de joc de tip labirint" este scrisă de mine şi nu a mai fost prezentată niciodată la o altă facultate sau instituţie de învăţământ superior din ţară sau străinătate. De asemenea, declar că toate sursele utilizate, inclusiv cele preluate de pe Internet, sunt indicate în lucrare, cu respectarea regulilor de evitare a plagiatului:

- toate fragmentele de text reproduse exact, chiar şi în traducere proprie din altă limbă, sunt scrise între ghilimele şi deţin referinţa precisă a sursei;

- reformularea în cuvinte proprii a textelor scrise de către alţi autori deţine referinţa precisă;

- codul sursă, imaginile etc. preluate din proiecte *open-source* sau alte surse sunt utilizate cu respectarea drepturilor de autor şi deţin referinţe precise;

- rezumarea ideilor altor autori precizează referinţa precisă la textul original.

Absolvent **Ioan Ungurean**,

_____

(semnătura în original)

DECLARAŢIE DE CONSIMŢĂMÂNT

Prin prezenta declar că sunt de acord ca Lucrarea de licență cu titlul "Dezvoltarea jocurilor Web cu HTML5 și WebGL. Un prototip de joc de tip labirint", codul sursă al programelor şi celelalte conţinuturi (grafice, multimedia, date de test etc.) care însoţesc această lucrare să fie utilizate în cadrul Facultăţii de Informatică.

De asemenea, sunt de acord ca Facultatea de Informatică de la Universitatea „Alexandru Ioan Cuza" Iași să utilizeze, modifice, reproducă şi să distribuie în scopuri necomerciale programele-calculator, format executabil şi sursă, realizate de mine în cadrul prezentei lucrări de licenţă.

Absolvent **Ioan Ungurean**,

_____

(semnătura în original)

ACORD PRIVIND PROPRIETATEA DREPTULUI DE AUTOR

Facultatea de Informatică este de acord ca drepturile de autor asupra programele-calculator, format executabil şi sursă, să aparţină autorului prezentei lucrări, Ioan Ungurean.

Încheierea acestui acord este necesară din următoarele motive:

Iaşi, *data*

Decan **Adrian Iftene**

(semnătura în original)

Absolvent **Ioan Ungurean**

(semnătura în original)

Cuprins

Introducere

Motivație

Acastă lucrare de licență are ca scop oferirea unui baze arhitecturale în crearea și dezvolatarea multiplelor tipuri de jocuri într-o manieră cât mai simplă, accesibilă și ușor de folosit.

Scurt istoric

Odată cu avansarea tehnologică, calculatoarele performante au devenit tot mai raspândite și accesibile unui public tot mai larg. Numărul de utilizatori de internet a crescut și acesta exponențial, ajungându-se la o cifră de 3,2 miliarde de utilizatori, conform unui studiu făcut în Februarie 2016[1]. În urma acestui studiu se poate deduce că aproape jumătate din populația planetei este conectată la internet. Tot din acest studiu aflăm că numărul utilizatorilor de internet la nivel global s-a trilplat în ultimul deceniu.

Dacă luăm în considerare aceste statistici, este de înțeles de ce a fost nevoie de apariția HTML5 și a WebGL. Un număr mare de utilizatori cu calculatoare performante care erau folosite insuficient, practic o piață de consumatori care era încă într-un stadiu incipient.

Jocuri în browser au existat mereu sub forma minijocurilor în Flash care erau denumite popular și "5 min games", dar nivelul lor de performanță și complexitate era foarte scăzut. Chiar și așa ele erau destul de populare deoarece nu necesitau instalări dificile și puteau fi accesate de oriunde. Tot ce aveai nevoie era un calculator și un browser. Acest tip de joc a devenit o sursă de venit importantă pentru dezvoltatorii mici și mijlocii.

Combinând succesul ridicat al acestor tipuri de jocuri cu tehnologii de ultimă generație, care până la acel moment nu erau disponibile, s-a creeat o piață care a captat atenția atât utilizatorilor ce doresc performanță și grafică de înaltă calitate, cât și acelora care doresc usurință și mobilitate în accesarea jocurilor.

[1] Raportul Băncii Mondiale „Digital Dividends World Development” 2016

Structură

În această secțiune este prezentat un conspect al lucrării cu punctele cheie ale fiecărui capitol și legătura dintre ele. Structura acestei documentații este formată din patru capitole.

În primul capitol vom face o scurtă descriere a tuturor tehnologiilor folosite (limbaje de programare, biblioteci, framework-uri etc.) pentru crearea arhitecturii și a jocului prototip implementat.

- Pentru partea de **server** folosesc:
  - **Node.js** și **Express** pentru partea de management multiplayer
  - **Socket.IO** pentru comunicare și sincronizare în timp real între nodurile din rețea conectate la server

- Pentru partea de **client** folosesc:
  - **JavasSripț**
  - **Socket.IO**
  - **Three.js** folosit pentru a crea și a afișa grafică 3D animată într-un browser web indiferent de mediu / sistem de operare. Biblioteca Three.js utilizează WebGL.
  - **xBEM –** folosit în special pentru elemente grafice (UI)

- Pentru partea de **module bundle** folosesc**:**
  - **Webpack 2**

Urmând ca în capitolul doi să trecem la cea mai mare parte a lucrării descriind arhitectura folosită punând accent pe modularitate, scalabilitate și flexibilitate. De asemenea tot în acest capitol va fi prezentat și jocul prototip construit pe baza acestei arhitecturi. Vom analiza mai întâi modulele de bază. Acestea au fost create pentru a permite o dezvoltare mai rapidă cu o performanță ridicată. Apoi vom trece la partea de redare. Vom discuta despre avantajele și dezavantajele diferitelor metode de redare și de ce o anumită metodă a fost aleasă în raport cu celelalte. Vom discuta modul în care diverse elemente pot descrie o scenă și cum poate fi adăugată o logică personalizată pentru a adăuga elementul de interactivitate unui joc derivat. Apoi, vom analiza managementul conținutului, mai exact modul în care sunt gestionate resursele. Tot în acest capitol vom vorbi despre modul în care funcționează detectarea coliziunilor și

modul în care scena poate fi simulată într-un mod realist, o componentă cheie a oricărui joc.

În capitolul trei vom descrie pe larg și succint modelele arhitecturale și de proiectare folosite în acest proiect cu câteva exemple ce demonstrează ușurința cu care putem schimba obiectivele unui joc și elementele grafice pentru ca în final să creezi un nou joc, total diferit de cel de la care am plecat.

Ultimul capitol va conține concluziile finale asupra arhitecturii implementate, provocările la care ne putem aștepta pe viitor și îmbunătățirile care se pot aduce la această arhitectură.

Capitolul I

Definiția jocurilor

Jocurile sunt o activitate sau un concurs bazat pe reguli. În cadrul oricărui joc jucătorul se focalizează pe îndeplinirea unui rol activ, iar împreună cu jocul în sine are la bază urmatoarele aspecte:

- Activitate (Activity) – Jocul constă într-o activitate definit mai bine ca și un proces sau eveniment.
- Factori de decizie (Decision-makers) – Nevoia jucătorilor de a lua decizii pro-activ
- Obiective (Objectives) – Fiecare joc împarte oferă anumite scopuri de îndeplinit în cadrul lui.
- Limite de context (Limiting context) – Necesitatea unor reguli pentru a limita și structura activitățile.

În funcție de numărul de jucători (solitari sau în echipă), scop (câștig, dobândire de abilități), obiectiv (scor maxim, top 10), tematică (lingvistice, sportive, acțiune), mijloc/echipament (tablă, teren, consolă), categorie de jucători (copii, adulți) jocurile pot fi încadrate în diferite categorii:

- Jocuri de strategie (Strategy board games) obiectivul fiind capturarea adversarilor, câștigarea de teritoriu prin plasarea / mutarea pieselor (ex. Șah, Domino, Monopoly etc.).
- Jocuri de rol (Role-play games) în cadrul acestor jocuri fiecare jucător își asumă rolul unui personaj având caracteristici particulare (ex. Dungeons & Dragons).

Alte tipuri de jocuri clasice: jocuri cu carți (Hearts, Poker, Whist), jocuri de noroc (Loterie, Bingo), jocuri cu zaruri, puzzel-uri etc. [1]

Clasificarea jocurilor electronice

Pentru a facilita prestarea jocurilor cu oponenți umani și/ sau contra calculatorului este nevoie de un dispozitiv electronic (sistem conectat la TV,

computer, dispozitiv mobil). De asemenea și jocurile electronice sunt împărțite în categorii după cum urmează: jocuri de arcadă, jocuri video, jocuri pe calculator, jocuri online, jocuri web. Cele mai populare stiluri/genuri de jocuri electronice[2] sunt: acțiune, aventură, strategie, simulare, puzzle, amuzament, educațional. [1]

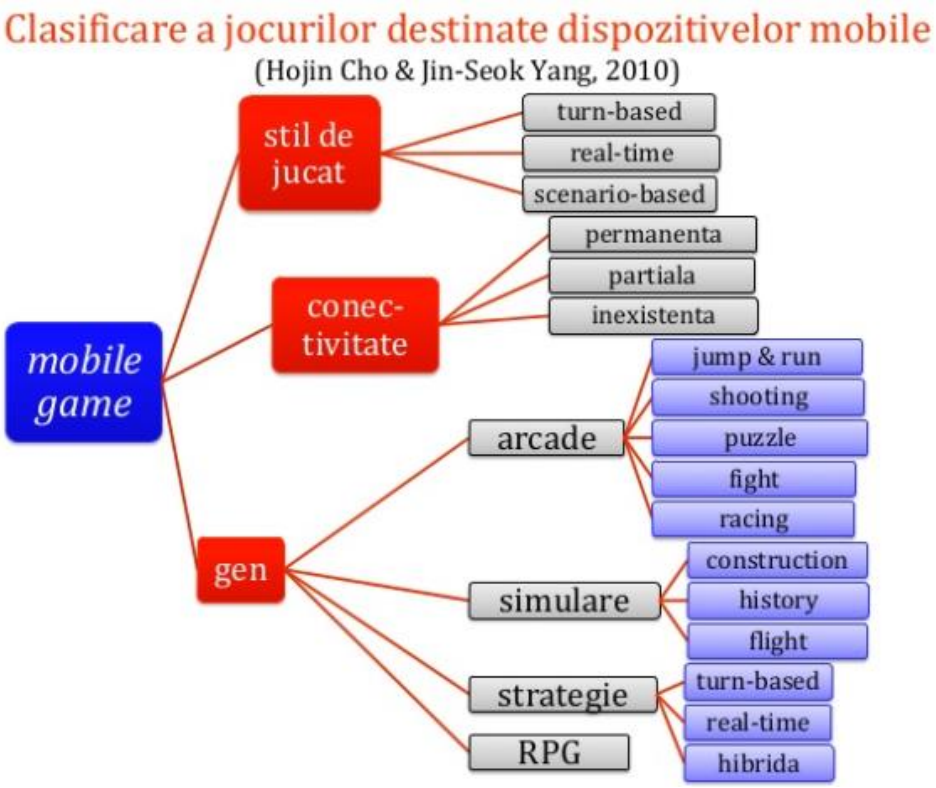

Figura 1. Categorii de jocuri

[**Sursa**: https://www.slideshare.net/busaco/dezvoltarea-jocurilor-web]

Aspecte de interes privitoare la jocurile electronice

Experiența jocului (game experience)

Se caracterizează prin reacție motorie, concentrare, mecanică de bază, narațiune.

Maniera de joc (game play)

Constă în dependența de povestea jocului, reguli, condiții de victorie/ înfrângere implicite sau explicite.

[2] Conform Rollings & Morris, 2004

Tipurile de provocări (challenges)

Tipurile de provocări sunt de tipul explicit sau implicit, provocări logice sau realizarea de deducții.

Personajele

Personajele pot fi personaje ce pot fi jucate de utilizator sau personaje independente de jucător, create prin software (NPCs). Tipurile de personaje ce se regăsesc în jocuri sunt următoarele: animal, fantastic, istoric, preluat, mitic.

Punctele de vedere (POV- points of view)

Sunt de două tipuri:

- jucătorul observă acțiunea prin ochii avatarului (first-person POV)
- utilizatorul poate vedea avatarul pe parcursul acțiunii (third-person POV).

Modelarea personajelor

Are la bază proiectarea vizuală adăugându-se personalitate, postură, costumație, nume, aspectul socio-cultural.

Proiectarea nivelurilor (level design)

Are următoarele ingrediente acțiune, mod de explorare, rezolvarea unui puzzle, narațiune, estetică.

Utilizarea inteligenței artificiale

Trebuie să convingă utilizatorul că entitățile jocului sunt “smart” prin tehnicile AI. [1]

utilizarea inteligenței artificiale
(Rouse, 2005)

provocarea jucătorului	*challenge the player*
modelarea comportamentului NPC	*not do dumb things*
realizarea impredictibilității	*be unpredictable*
suport în derularea narațiunii	*assist storytelling*
crearea unei lumi credibile	*create a living world*

Figura 2. Utilizarea inteleigenței artificiale

Concluzii

În cadrul acestui capitol am prezentat, pe scurt, aspecte din domeniul proiectului curent de licență.

În cele ce urmează voi face o descriere a tehnologiilor folosite în aplicație și modul de utilizare al acestora și de asemena voi prezenta design pattern-urile pe care se bazează această arhitectură web.

Capitolul II

Tehnologii folosite pe partea de server

Node.js

Node.js permite dezvoltarea de aplicații web la nivel de server. Este o platformă construită pe motorul V8 al Chrome și se bazează pe un model bazat pe evenimente, iar operațiile de input/output sunt neblocante bazându-se pe callback-uri.

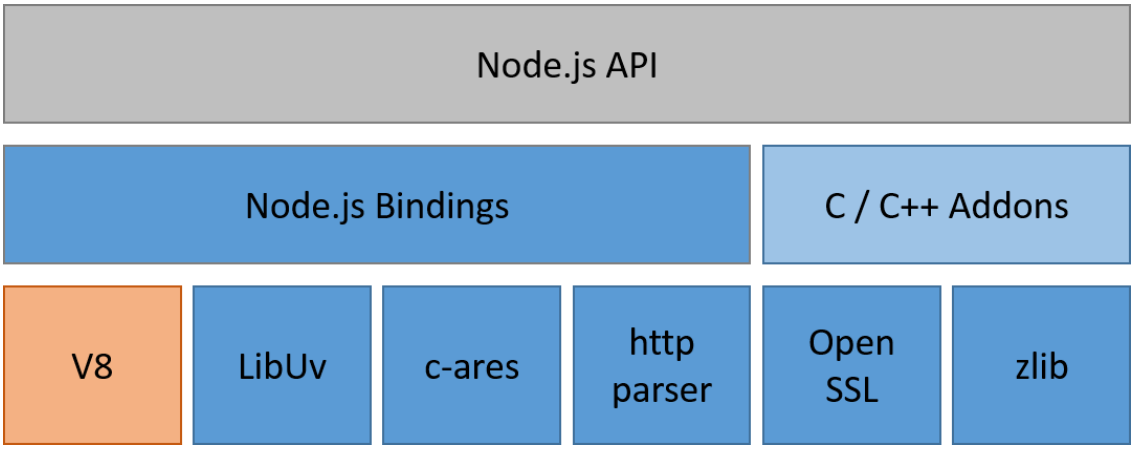

Figura 3. Arhitectură Node.js

[**Sursa**: Imgur, Node.js architecture]

O aplicație Node.js rulează într-un singur proces. Codul JavaScript nu este executat paralel, dar tratează în mod asincron diverse evenimente I/O deosebire esențială față de serverele de aplicații web tradiționale ce recurg la servere multi-process/threaded. [2] [3]

Express

Express este un framework pentru Node.Js minimal, flexibil și cross-platform, ce oferă un set robust de funcționalități pentru aplicații web și mobile. El este creat pentru a construi aplicții web și API-uri. [4]

Socket.IO

Socket.io este o bibliotecă JavaScript pentru aplicații web în timp real. Oferă comunicare bi-direcțională, în timp real între clienții web și servere.

Conține două parți:

- Bibliotecă pentru partea de client ce rulează în browser
- Bibliotecă pentru Node.js pentru partea de server.

Ambele componente folosesc un API aproape identic. Precum Node.js, biblioteca Socket.IO este orientată pe evenimente.

Socket.IO folosește în primul rând protocolul WebSocket cu interogări ciclice pentru opțiunea de rezervă, în timp ce oferă aceeași interfață. Deși poate fi folosit pur și simplu ca un wrapper pentru WebSocket, furnizează multe alte funcționalități, incluzând broadcasting pentru socketi multipli, sortarea datelor asociate fiecărui client și I/O asincron. Poate fi instalat cu ajutorul comenzii **npm** (node packge manager).

Oferă posibilitatea de a implementa analize în timp real, mesaje instante etc. Socket.IO gestionează conexiunea într-un mod transparent. Dacă este posibil va înștiința automat WebSoket asupra unei modificări.

Este o implementare a unui protocol transport în timp real peste alt protocol în timp real. Partea de negociere a protocolului face ca suportul standard WebSocket pentru client să nu poată accesa server-ul Socket.IO, iar un client implementat cu Socket.IO nu poate comunica cu un WebSocket ce nu este bazat pe Socket.IO sau cu un Long Polling Comet server. Prin urmare, Socket.IO solicită utilizarea bibliotecilor Socket.IO și pe partea de client dar și pe partea de server. [5]

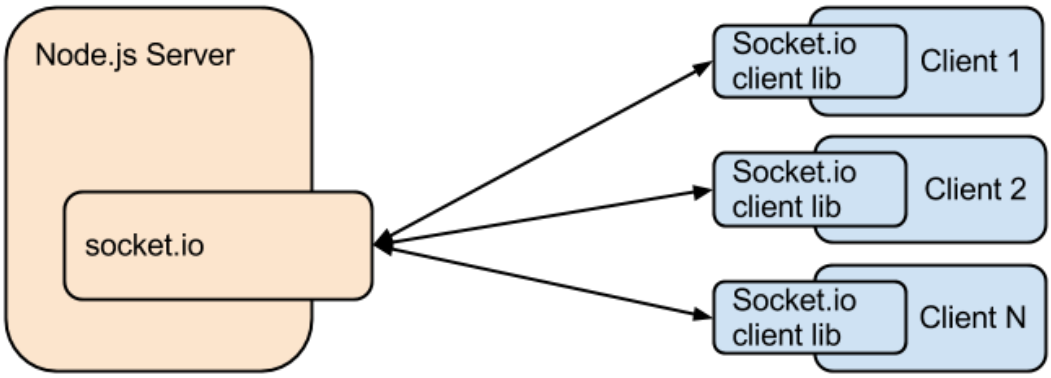

Figura 4. Arhitectura server-ului folosind Socket.IO

[**Sursa**: http://blog.lightstreamer.com/benchmarking-socketio-vs-lightstreamer]

Webpack Middleware

Este un simplu wrapper middleware pentru webpack. Oferă fișierele emise de către webpack printr-un server de tip connect. Acesta are câteva avantaje prin livrarea datelor sub formă de fișiere: fișierele nu sunt scrise pe disc, ele sunt gestionate în memorie; dacă fișierele au fost modificate în modul watch, middleware-ul nu mai deservește vechiul grup de fișiere și încetinește cererile până când compilarea se sfârșește. Vom acoperi **Webpack 2** mai în detaliu în secțiunile ce urmează. [6] [7]

Tehnologii folosite pe partea de client

WebGL

WebGL reprezintă o interfață de programare pentru browser, folosind elementul Canvas din HTML5, pentru a compune scene 3D. Interfața de programare se accesează folosind 2 limbaje: JavaScript execută încărcarea resurselor, compunerea scenei, logica spațiului 3D. GLSL se execută direct pe procesorul grafic, în paralel, o funcție pentru fiecare vector din model și altă funcție pentru fiecare pixel afișat. Datorită conexiunii între WebGL și OpenGL, aplicațiile 3D pe WebGL sunt în același timp foarte performante, și oferă posibilități similare în programarea jocurilor și animațiilor 3D în aplicații separate. Faptul că este construit folosind standardul HTML și JavaScript este avantajul principal pe care WebGL îl are asupra concurenților săi. [8][9]

Mai jos se poate observa arhitectura WebGL:

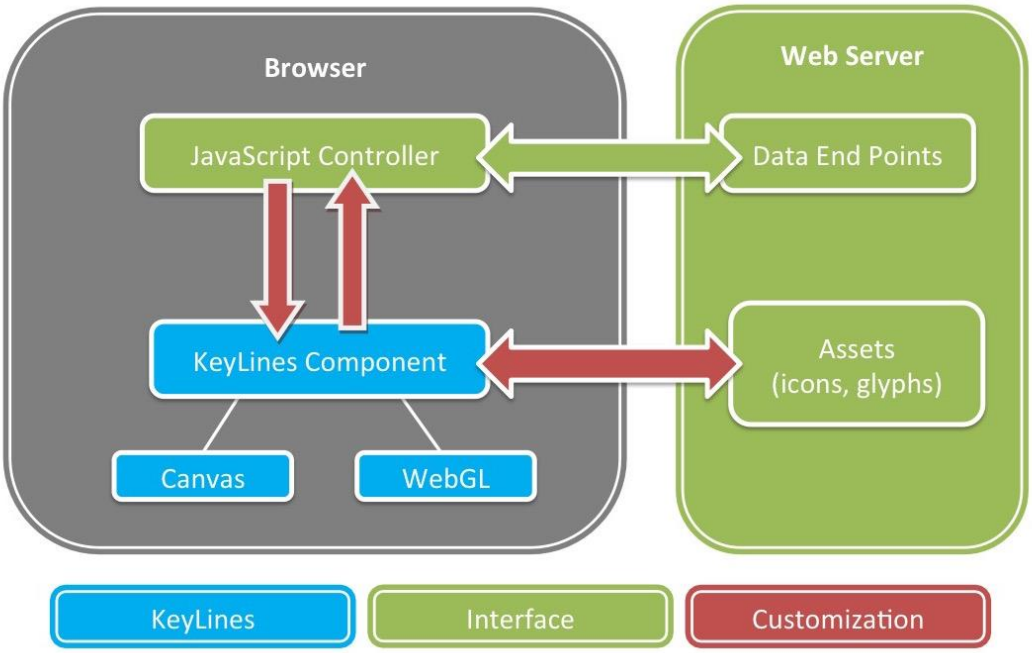

Figura 5. Arhitectură WebGL

[**Sursa**: WebGL Beginner's Guide, by Diego Cantor, Brandon Jones]

WebGL a fost dezvoltat și întreținut de grupul Khronos[3] și a fost implementat cu standardul HTML5. Este susținută de cele mai moderne browsere desktop (Chrome, Safari, Firefox și Opera) și de cele mai multe sisteme de operare desktop (Windows, Linux, Mac OS). În timp ce specificațiile permit ca acesta să ruleze pe dispozitive mobile (fiind construit în jurul specificațiilor OpenGL ES, care se poate executa deja pe dispozitive mobile moderne), acesta nu este încă acceptat de niciun browser oficial mobil (cu excepția programului Opera Mobile). WebGL oferă o gama de avantaje:

- Un API care este bazat pe un standard grafic 3D răspândit şi familiar.
- Compatibilitate între navigatoare (cross-navigator) şi între platforme (cross-platforme).

[3] Grupul Khronos, Inc. este un consorțiu american non-profit, finanțat de membri, cu sediul în Beaverton, Oregon, care se concentrează pe crearea de interfețe de programare pentru aplicații gratuite

- Integrare etanşă cu conținutul HTML, incluzând compoziție pe straturi, interacțiune cu alte elemente HTML şi folosirea mecanismelor standard de tratare a evenimentelor.
- Accelerare hardware 3D în conținutul navigatorului.
- Un mediu de scripting ce facilitează prototipizarea graficei 3D (nu trebuie să compilezi şi să link-eşti înainte de a vedea grafică redată).

Three.js

Este o bibliotecă WebGL menţinută şi dezvoltată de Ricardo Cabello (Mr-Doob). Trăsăturile principale ale acestei biblioteci sunt:

- **Redare:** WebGL, <canvas>, <svg>, CSS3D, DOM.
- **Scene:** adăugare şi ştergere de obiecte la rulare.
- **Camere:** perspectivă şi ortografică; controlere: trackball, FPS şi path.
- **Animaţie:** morph şi keyframe.
- **Lumini:** ambientală, direcțională, spre un punct, spre o zonă şi lumini hemisferice.
- **Materiale:** Basic, Lambert, Phong.
- **Shaders:** acces la toate capabilităţile WebGL.
- **Obiecte:** mesh-uri , particule, sprites, linii etc.
- **Geometri:** plan, cub, sferă, torus, text 3D şi altele.
- **Loadere:** binar, de imagine, JSON şi scenă.
- **Utilitare:** set complet pentru măsurarea timpilor şi funcţionalităţi 3D incluzând frustum[4], lucrul cu quaternion[5], matrice, UV-mapping[6] etc.
- **Export/Import:** utilităţi pentru a crea fişiere JSON compatibile cu Three.js pentru extensiile Blender, CTM, FBX, 3D max şi Obj. [8]

Socket.IO

Socket.io este o bibliotecă JavaScript pentru aplicații web în timp real. Oferă comunicare bi-direcțională, în timp real între clienții web și servere. Mai

[4] În geometrie un frustum este porțiunea care se găsește între două plane paralele ce secționează un poliedru
[5] Numere hipercomplexe non-comutative obținute prin extinderea mulțimii numerelor complexe. Folosiți pentru calcule ce implică rotații tridimensionale.
[6] Procesul de modelare 3D pentru a reprezenta o imagine 2D a unui model 3D.

multe informații se pot găsi în Capitolul II la Tehnologii folosite pe partea de server unde am acoperit mai în detaliu această bibliotecă care este bazată pe protocolul **WebSocket.** [5]

JavaScript

JavaScript adesea abreviat "JS" este un limbaj de programare de nivel înalt, dinamic, netipizat, bazat pe obiecte, multi-paradigmă, interpretat și rapid cu ajutorul căruia se pot dezvolta aplicații incredibile. Alături de HTML și CSS, JavaScript este una din cele trei tehnologii de bază pentru producția de content pentru World Wide Web. Este folosit pentru a crea pagini web interactive și pentru a furniza programe online, incluzând **jocuri video**. Acest limbaj de programare este adoptat de majoritatea site-urilor web, iar toate browser-ele moderne îl suportă fără necesitatea instalării unui plug-in ci cu ajutorul unui engine JavaScript deja implementat în browser.

Cea mai des întâlnită utilizare a JavaScript este în scriptarea paginilor web. Programatorii web pot îngloba în paginile HTML script-uri pentru diverse activități cum ar fi verificarea datelor introduse de utilizatori sau crearea de meniuri și alte efecte animate.

Browserele rețin în memorie o reprezentare a unei pagini web sub forma unui arbore de obiecte și pun la dispoziție aceste obiecte script-urilor JavaScript, care le pot citi și manipula. Arborele de obiecte poartă numele de Document Object Model sau DOM. În practică, însă, standardul W3C pentru DOM este incomplet implementat. Deși tendința browserelor este de a se alinia standardului W3C, unele din acestea încă prezintă incompatibilități majore, cum este cazul Internet Explorer.

O tehnică ce facilitează transferul asincron de date foarte întâlnită este AJAX, abreviere de la „Asynchronous JavaScript and XML". Această tehnică constă în executarea de cereri HTTP în fundal, fără a reîncărca toată pagina web, și actualizarea numai anumitor porțiuni ale paginii prin manipularea DOM-ului. Tehnica AJAX permite construirea unor interfețe web cu timp de răspuns mic, întrucît operația (costisitoare ca timp) de încărcare a unei pagini HTML complete este în mare parte eliminată.

Există o serie de reguli pentru a optimiza codul JavaScript în cadrul aplicațiilor web:

- Plasarea JavaScriptului la sfârșitul paginii, din moment ce motorul de redare HTML rulează înainte de cel pentru JavaScript.

- Codul JavaScript trebuie să fie extern din moment ce este descărcat prima dată și apoi cache-uit pentru folosiri ulterioare.
- Gruparea logică a scripturilor pentru a fi încărcată în paginile care au nevoie de ele.
- A se evita încărcarea de scripturi de dimensiuni foarte mari atunci când se utilizează doar bucați mici de cod din el.
- Compactarea codului duce la performanțe ridicate din moment ce clientul va descărca mai puțin până când va putea stoca în cache un fișier script.
- De asemenea, performanțele JavaScript-ului au legătură și cu interpretorul.
- Aceste motoare sunt specifice navigatoarelor de pe care accesăm aplicația și au diverse avantaje și dezavantaje. [10]

CSS

CSS [11] joacă un rol important în dezvoltarea aplicațiilor web. Similar, cu Javascript sunt o serie de optimizări care pot fi făcute pentru a spori performanțele unei aplicații privind organizarea codului:

- CSS trebuie să fie extern pentru a beneficia de cache-ing.
- Referințele CSS trebuie puse în capătul de sus al paginii.
- Mărimea fișierelor CSS influențează viteza de încărcare.
- Evitarea referințelor circulare sau duplicate.

SASS

SASS [12] este un limbaj de scripting care este interpretat sau compilat în CSS (Cascading Style Sheets). SassScript este limbajul de scripting în sine. Sass este alcătuită din două sintaxe. Sintaxa originală, numită "sintaxa indentată", folosește o sintaxă similară cu cea a lui Haml[7]. Utilizează indentarea pentru a separa blocurile de coduri și caracterele noi pentru a separa regulile. Sintaxa mai nouă, "SCSS", folosește formatarea blocului ca cea a CSS. Utilizează barele pentru a denumi blocurile de cod și punct și virgulă pentru a separa liniile dintr-un bloc. Sintaxa indentată și fișierele SCSS au în mod tradițional extensiile .sass și .scss.

[7] Haml (HTML Abstraction Markup Language)

CSS3 [13] constă dintr-o serie de selectori și pseudo-selectori care grupează regulile care se aplică acestora. Sass (în contextul mai larg al ambelor sintaxe) extinde CSS prin furnizarea mai multor mecanisme disponibile în limbaje de programare mai tradiționale, în special limbaje orientate pe obiecte, dar care nu sunt disponibile pentru CSS3 în sine. Atunci când SassScript este interpretat, acesta creează blocuri de reguli CSS pentru diferiți selectori, așa cum sunt definiți de fișierul Sass. Interpretul Sass traduce SassScript în CSS. Alternativ, Sass poate monitoriza fișierul .sass sau .scss și îl poate traduce într-un fișier .css de ieșire ori de câte ori fișierul .sass sau .scss este salvat. Sass este pur și simplu sugar-syntax pentru CSS.

Implementarea oficială a Sass este open-source și scrisă cu ajutorul limbajului de programare Ruby; Cu toate acestea, există alte implementări, inclusiv PHP, și o implementare de înaltă performanță în C numită libSass. Există, de asemenea, o implementare Java numită Jsass.

SassScript oferă următoarele mecanisme: variabile, nesting, mixins și moștenire bazată pe selectori.

Block Elment Modifier (BEM)

BEM (Block, Element, Modifier) [14] este o abordare bazată pe componente pentru dezvoltare web. Ideea din spatele ei este de a împărți interfața utilizatorului în blocuri independente. Acest lucru face ca dezvoltarea interfeței să fie ușoară și rapidă, chiar și pentru o interfață complexă, și permite refolosirea codului existent fără a exista cod duplicat.

Block. O componentă a paginii independente din punct de vedere funcțional care poate fi refolosită. În HTML, blocurile sunt reprezentate de atributul de clasă. Iată câteva caracteristici:

Element. O parte compusă dintr-un bloc care nu poate fi folosită separat de acesta.

Modifier. O entitate care definește aspectul, starea sau comportamentul unui bloc sau element.

În această aplicație folosim **xBEM**[8] un framework pentru metodologia **BEM.** Mai jos vom prezenta câteva avantaje ale folosirii acestui framework:

- Combinări ușoare și intuitive pentru scrierea claselor B.E.M
- Fișierul CSS de ieșire compilat este optimizat
- Nu este nevoie de mai mult de o clasă per element
- Fără nesting mai mare de nivelul unu

Aceast framework a fost dezvoltat și este întreținut de Bogdan Prisecaru.

Tehnologii folosite pentru partea de dezvoltare (development)

Webpack

Webpack este un **module bundler,** mai exact, un instrument care se ocupă de împachetarea într-un fișier „bundle" a modulelor scrise în procesul de dezvoltare. El rezolvă dependențele dintre **module**, urmând ca rezultatul să fie scris într-un singur fișier. Un **modul** poate **importa** elemente din alte module și poate **exporta** în aceeași manieră către alte module. Ceea ce înseamnă că aplicația devine un graf format din dependențe între module. Această funționalitate este oferită chiar de către JavaScript, prin noua versiune ES2015 (denumită și ES6). [8] [9]

Webpack a devenit unul dintre cele mai importante instrumente pentru dezvoltarea web din zilele nostre. El poate fi configurat pentru a transforma resurse precum HTML, CSS, JS fișiere JSON, imagini etc. Totodată poate oferi un control mai mare asupra numărului de cereri HTTP pe care o aplicație le face. [6][7] În figura de mai jos se poate observa acest lucru într-un mod vizual:

[8] xBEM alternativă pentru SASS mixin pentru Block Element Modifier (BEM)

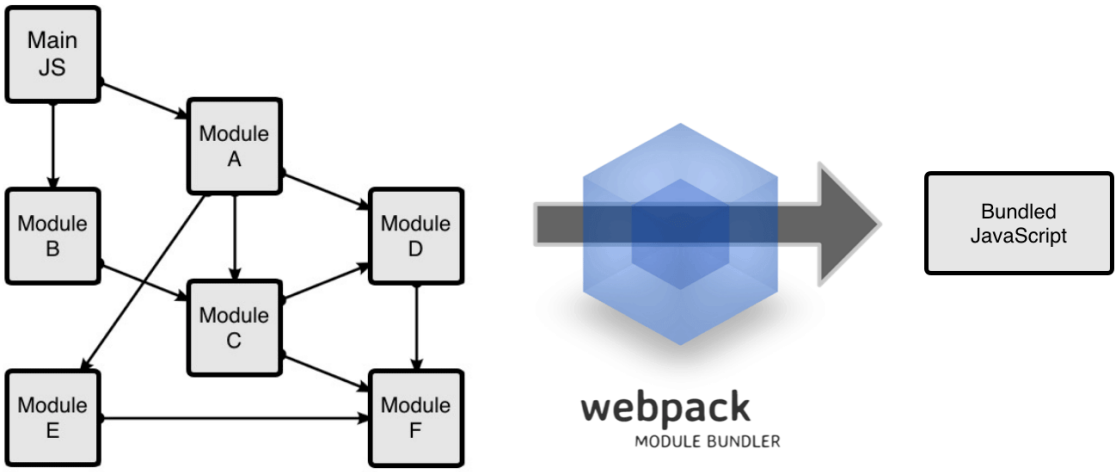

Figura 6. Webpack vizualizare

[**Sursa**: https://webpack.github.io/]

În această lucrare de licență am folosit două fișiere de configurare, unul pentru dezvoltare (development), iar celălalt pentru producție (production). Cel pentru dezvolatare ajută dezvoltatorul să scrie cod cât mai ușor cu putință (hot reload, debugging etc.), iar cel de producție, la build time, scoate fișierele de dimensiuni cât mai mici (denumit ca proces de minificare) și redenumește variabilele, funcțiile, clasele și metodele folosind unul sau cel mult două caractere eliminând spațiile și sfârșitul de linie, în final rezultând doar o singură linie de cod (denumit ca proces de uglify).

```
module.exports = function(config) {
  return {
    plugins: [
      new ExtractTextPlugin({
        filename: 'bundle.css',
        allChunks: true
      }),
      new webpack.optimize.CommonsChunkPlugin({
        name: 'vendor',
        minChunks: function(module) { // this assumes your vendor imports exist in the node_modules
        directory
          return module.context && module.context.indexOf('node_modules') !== -1;
        }
      }),
      new webpack.optimize.OccurrenceOrderPlugin()
    ],
    module: {
      rules: [
        {
          test: /\.(css|scss)$/,
          use: ExtractTextPlugin.extract({
            use: [
              {
                loader: 'css-loader'
              },
              {
                loader: 'resolve-url-loader'
              },
              {
                loader: 'sass-loader',
                options: {
                  sourceMap: true,
                  includePaths: [
                    path.resolve('node_modules/xbem/src/'),
                    path.resolve(`${projectPaths.source}/ui/themes/${config.theme}`),
                    path.resolve(`${projectPaths.source}/ui/themes/${config.theme}/fonts`),
                    path.resolve(`${projectPaths.source}/ui/themes/${config.theme}/patterns`)
                  ]
                }
              }
            ]
          })
        }
      ]
    }
  };
};
```

Figura 7. Fișierul de configurare pentru partea de dezvoltare

```
module.exports = (config) => {
  return {
    entry: {
      app: [
        'webpack-dev-server/client?' + host,
        'webpack/hot/only-dev-server'
      ]
    },
    plugins: [
      new webpack.DefinePlugin({
        DEVELOPMENT: true
      }),
      new webpack.HotModuleReplacementPlugin()
    ],
    stats: 'errors-only',
    devtool: 'source-map', // or 'eval' or false for faster compilation
    devServer: {
      inline: true,
      host: hostname,
      port: port,
      publicPath: assetHost, // Make sure publicPath always starts and ends with a forward slash.
      contentBase: [
        path.join(process.cwd(), projectPaths.source),
        path.join(process.cwd(), projectPaths.dist)
      ],
      clientLogLevel: 'none',
      noInfo: true,
      historyApiFallback: {
        disableDotRule: true,
        proxy: {}
      }
    },
    module: {
      rules: [
        {
          test: /\.scss$/,
          use: [
            {
              loader: 'style-loader'
            },
            {
              loader: 'css-loader'
            },
            {
              loader: 'resolve-url-loader'
            },
            {
              loader: 'sass-loader',
              options: {
                sourceMap: true,
                includePaths: [
                  path.resolve('node_modules/xbem/src/'),
                  path.resolve(`${projectPaths.source}/ui/themes/${config.theme}`),
                  path.resolve(`${projectPaths.source}/ui/themes/${config.theme}/fonts`),
                  path.resolve(`${projectPaths.source}/ui/themes/${config.theme}/patterns`)
                ]
              }
            }
          ]
        }
      ]
    }
  };
};
```

Figura 8. Fișierul de configurare pentru partea de producție

Obiectivele lucrării

Prin realizarea acestui proiect noi ne propunem să construim o bază solidă pentru orice dezvoltator care dorește să creeze un joc de tip grilaj (tile based). Cele mai importante avantaje ale acestei arhitecturi sunt **modularitatea, scalabilitatea și flexibilitatea**. Prin *modularitate* se înțelege faptul că majoritatea componentelor pot fi refolosite pentru cazuri diferite, componentele fiind independente de logica care le definește. Această

arhitectură este *scalabilă* și anume putem adăuga un număr mare de elemente grafice fără a ne afecta performanța. (e.g. această arhitectură se bazează pe prototipuri pe care le clonăm fără a mai fi nevoie să le încărcăm de fiecare dată în pagină). *Flexibilitatea* constă în faptul că putem folosi unele componente din joc în alt scop decât cel definit. (e.g. detectarea coliziunilor, diferite tipuri de algoritmi de generare de labirinturi – condiția este ca respectivul algoritm să returneze o matrice).

Concluzii

În cadrul acestui capitol am prezentat, pe scurt, tehnologiile folosite atât în facilitarea dezvoltării de aplicații web (ex. Webpack) cât și în dezvoltarea propriu-zisă (limbaje de programare, biblioteci, framework-uri etc.). Tot în acest capitol am rezentat și care sunt obiectivele pe care ni le-am propus în realizarea acestei arhitecturi. În capitolul următor vom face o descriere amânunțită a părții de arhitectură și implementare.

Capitolul III

Elemente de arhitectură și implementare

În acest capitol vom prezenta arhitectura generală a aplicației și modul în care au fost implementate cele mai importante componente, printre care se numără partea de redare (modulul cel mai important pentru orice aplicație grafică), gestionarea scenei (cum sunt organizate obiectele în interiorul aplicației), modul în care resursele sunt încărcate și gestionate pentru o performanță maximă, detectarea coliziunilor (din nou, un modul foarte important pentru orice aplicație în timp real) și, ulterior, arta (cum au fost create resursele pentru jocul prototip și modul în care acestea interacționează).

Arhitectura generală

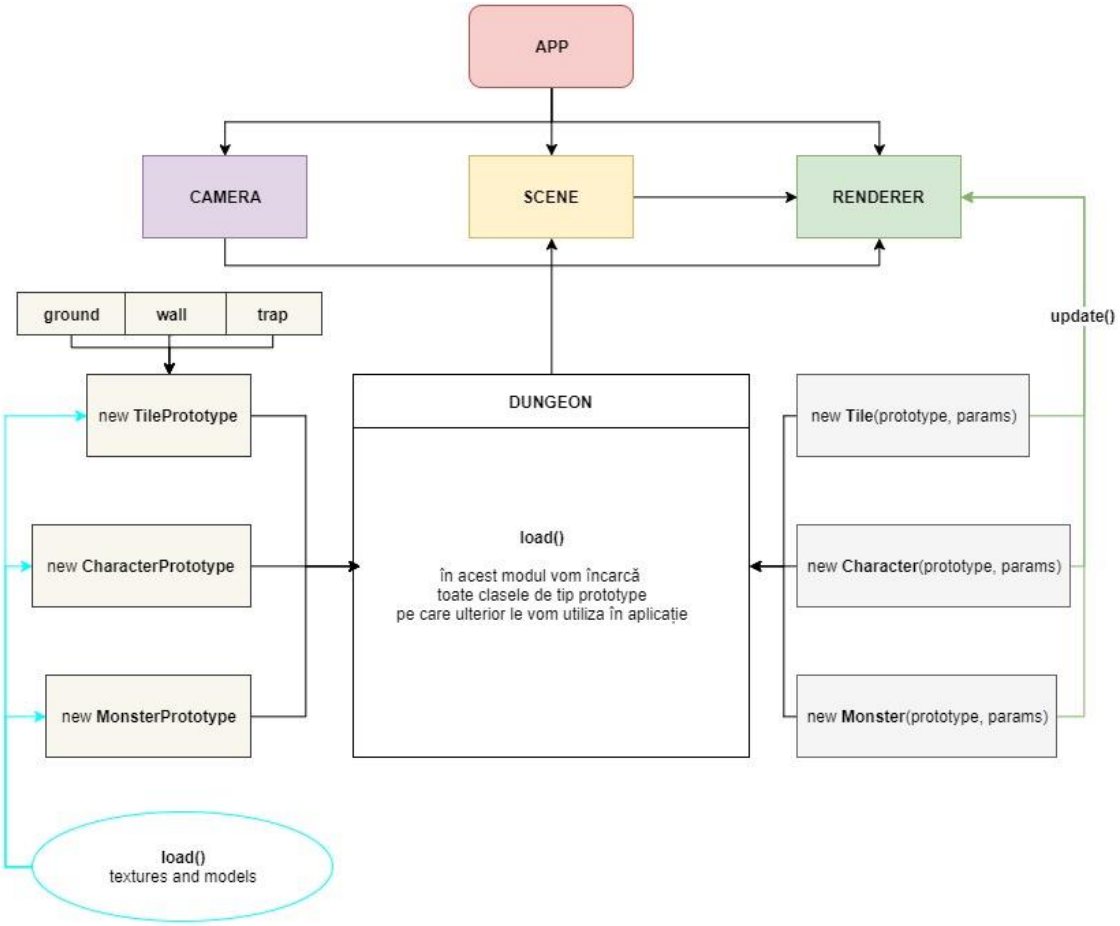

Figura 9. Arhitectura generală a jocului prototip

Această arhitectură este formată din trei module principale și anume: camera, scene și renderer. După cum îi spune și numele, în modulul **Camera** este implementată camera perspectivă pe care o vom folosi în aplicație pentru a naviga cu personajul prin labirint/dungeon. Modulul **Renderer** se ocupă de partea de desenare pe ecran a obiectelor 3D prezente în scenă folosindu-se de o funcție recursivă care se autoapelează de o infinitate de ori. În interiorul buclei respective se apelează funcția *reqestAnimationFrame()*. În final, dar nu în cele din urmă avem modulul **Scene** unde se proiectează ce anume și unde anume vom desena obiectele 3D din scenă. Modulul **Scene** va importa celelalte două module și anume *Camera* și *Renderer*.

Toată funcționalitatea jocului se află în submodulul **Dungeon** în care se importă toate clasele de tip *Prototype* împreună cu dependențele lor. Tot aici sunt importate și clasele care se folosesc de Prototype pentru a clona obiecte, obiecte care mai apoi sunt folosite pentru labirint/dungeon, personaj, NPC și altele. De menționat este faptul că fiecare obiect clonat are metoda lui de update care se autoapelează în momentul în care se redesenează pagina. Acest lucru se întâmplă de 60 de ori pe secundă pentru crea un comfort vizual sporit jucătorului.

Modulele de three.js folosite

Scena sau „Scene”

Este elementul principal al bibliotecii three.js. Scena ne permite să proiectăm ce și unde va fi redat de three.js. Aici este locul unde vom pune camera, luminile și obiectele. [15]

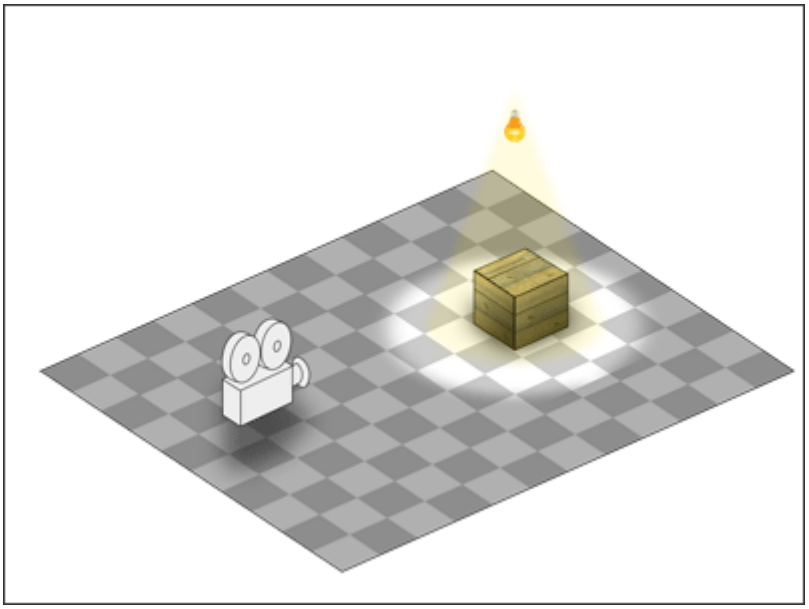

Figura 10. Exemplu scenă

[**Sursa**: https://github.com/yomotsu/figures-to-explain-webgl-threejs]

WebGL Render

Este inima, motorul ce dă viață scenei noastre, asta desigur dacă platforma de pe care îl folosim suportă acest API.

Este de notat că acest renderer are o performanță mult mai înaltă față de o alternativă a sa, și anume CanvasRenderer. [15]

Pentru acest obiect parametrii sunt opționali așa că vom prezenta doar o listă scurtă a acestora:

- Canvas - un canvas (o "pânză") unde render-ul afișează output-ul.
- Precision - precizie shader.Poate fi "highp","mediump" sau "lowp".
- Alpha - porțiune rezervată pentru transparență.

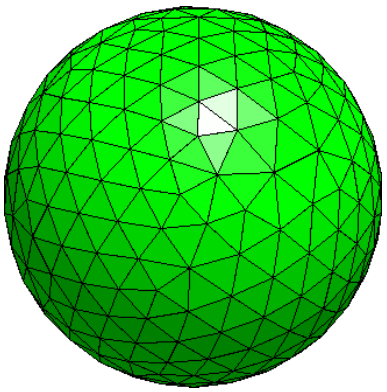

Figura 11. Exemplu de Mesh

[**Sursa**: http://nmag.soton.ac.uk/nmag/0.2/manual/]

- PremultipliedAlpha - metodă de stocare pentru alpha.
- Antialias - contur mai fluent.
- Stencil - este folosit pentru a limita zona de redare.

Camera

În cazul acestui proiect s-a folosit o cameră perspectivă. Distorsiunea perspectivă este determinată de distanţa relativă la care imaginea este captată şi vizionată, şi are legătură cu unghiul de vizualizare al imaginii (în timp ce este captată), fiind fie îngust sau larg față de unghiul de vizualizare al imaginii la care este vizionată. [15]

Parametrii disponibili pentru această camera sunt:

- FOV (field of view) - deschiderea unghiulară a camerei.
- Aspect - raport de aspect între deschiderea unghiulară pe verticală şi cea pe orizontală.
- Far, Near - distanța de la observator la planul apropiat (al imaginii) respectiv la cel depărtat. Aceste plane delimitează volumul de vizualizare al scenei.

Lumină sau „Light”

Reprezintă obiectele care oferă luminozitate pentru scenă.
Parametrii disponibili pentru lumină sunt culoarea, intensitatea și distanța (lumina va fi mai intensă la poziția ei de start și va scădea odată ce ne îndepărtăm). [15]

BufferGeometry

Această clasă este o alternativă mai eficientă față de Geometry, deoarece stochează toate datele, inclusiv pozițiile vârfurilor, indicii de față, normalele, culori, UV-uri și atribute personalizate ca buffere; Acest lucru reduce costul de transmitere a tuturor acestor date către GPU, dar lucrul cu BufferGeometry este mai greu decât ca cel cu Geometry; BufferGeometry se potrivește cel mai bine obiectelor statice în care nu este nevoie să manipulați geometria după o instanțiere. [15]

Următoarele atribute sunt stabilite de diferiți membri ai acesteia:

- Position: Stochează coordonatele x, y și z ale fiecărui vârf în această geometrie.
- Normal: Stochează componentele x, y și z ale vectorului obișnuit sau vertexului normal al fiecărui vârf din această geometrie.
- Color: Stochează canalele roșii, verzi și albastre ale culorii vertexului fiecărui vârf din această geometrie.
- Index: Permite ca reperele să fie reutilizate pe mai multe triunghiuri.

Object3D

Este o clasă pentru obiecte tridimensionale. Aceasta este clasa care o vom utiliza pentru a grupa și controla grupuri de Mesh-uri. [15]

Mesh

Sau mesh poligonal este o colecție de vertecsi, muchii și fețe care definesc forma unui obiect poliedral în grafica 3D pe calculator. Fețele sunt de obicei compuse din triunghiuri, cadrilatere și alte forme convexe, deoarece acest lucru simplifică redarea.

În ce privește acest proiect, totalitatea modelelor personajelor și a obiectelor ce formează ambientul (tile-uri, cutii, monstruleți) sunt reprezentate cu ajutorul acestui concept.

Meshurile sunt compuse din două părți:

- Geometria reprezintă toate datele necesare pentru a descrie un model 3D
- Materialul descrie cum ar trebui să arate acel model 3D.

Aceste proprietăți sunt definite într-un mod independent de instrumentul de redare, deci nu trebuie să rescriem materialele dacă ne decidem să folosim alt instrument. [15]

Skybox

Este o metodă de creare a unui fundal, pentru a da impresia unei scene dintr-un joc pe calculator, că aceasta este mai mare decât este în realitate. Când un skybox este folosit, scena este "închisă" într-un obiect cubic, și elemente ca cerul, munții și alte obiecte din depărtare sunt proiectate pe fețele cubului, astfel creând iluzia unei împrejurimi 3D.

JSON Loader

Este un loader [15] folosit pentru a încărca obiecte în formatul JSON.
Acest loader are trei parametri disponibili:

- url (adresa modelului)
- callback (această funcție va fi apelată odată cu modelul încărcat ca o instanță a geometriei când încărcarea este completă)
- calea texturii (parametru opțional, dacă nu este specificat se presupune că texturile sunt în aceeași locație cu modelul).

Proiectul se folosește de acest încărcător pentru a încărca acele modele statice ce formează ambientul (personajul, terenul, cuburile), modele care n-au nevoie de animație. Deși acest tip de loader oferă suport pentru animație, acesta nu este recomandat deoarece nu oferă aceeași performanță ca alternativele sale.

Modele arhitecturale și de proiectare

Singleton

Modelul Singleton [16] este utilizat pentru a restricționa numărul de instanțieri ale unei clase la un singur obiect. El este în general utilizat în următoarele cazuri:

- Ca parte integrată în modelele Abstract Factory, Builder, Prototype
- Obiectele ce reprezintă stări sunt deseori modele Singleton
- Singleton este preferat variabilelor globale deoarece:
    - Nu poluează namespace-ul global cu variabile nenecesare
    - Permite inițializarea intârziată (lazy initialization) pentru a nu consuma inutil resursele sistemului.

La baza pattern-ului Singleton stă o metodă ce permite crearea unei noi instanțe a clasei dacă aceasta nu există deja. Dacă instanța există deja, atunci întoarce o referință către acel obiect. Pentru a asigura o singură instanțiere a clasei, constructorul trebuie făcut protected (un constructor privat împiedică reutilizarea sa sau accesul unei unități de testare).

Diferența dintre o clasă cu atribute și metode statice și un Singleton este aceea că Singleton-ul permite instanțierea lazy, utilizând memoria doar în momentul în care acest lucru este necesar deoarece instanța se creează atunci când se apelează getInstance(). Încă un avantaj ar fi faptul că o clasă Singleton

poate fi extinsă și metodele ei suprascrise, însă într-o clasă cu metode statice acestea nu pot fi suprascrise (overriden).

În implementarea acestei arhitecturi a fost nevoie de o funcție de socket.io care să fie apelată o singură dată atunci când este importată pentru prima oară într-un modul, iar pentru celelalte cazuri trebuia să se folosească de conexiunea deja creată astfel încât am ales să implementăm o funcție de tip „closure”.

```
1   import io from 'socket.io-client';
2
3   export const socket = (() => {
4     let ws = io('http://localhost:3000/');
5     ws.on('connect', () => {
6       console.log('ws: connected to server');
7
8       ws.on('disconnect', () => {
9         console.log('ws: disconnect from server');
10        ws.close();
11      });
12    });
13
14    return {
15      on: (evname, cb) => {
16        ws.on(evname, (socket) => {
17          cb(socket);
18        });
19      },
20
21      emit: (evname, obj) => {
22        ws.emit(evname, obj);
23      }
24    };
25  })();
26
```

Figura 12. Funcție de tip „closure” peste socket.io

Prototype

Pattern pentru clonarea unor noi instanțe (clone) ale unui prototip existent. Modelul prototype este un model de proiectare creațional în dezvoltarea software. El este folosit atunci când tipul obiectelor ce trebuiesc create sunt determinate de o instanță prototip ce este clonată pentru a produce noi obiecte. [17]

Acest model este folosit pentru a evita subclasele unui creator de obiecte în aplicația client; să evite costurile inerente ale creării unui obiect nou în mod obișnuit (de exemplu, utilizând cuvântul cheie "new") atunci când este prohibitiv de costisitor pentru o anumită aplicație.

Pentru a implementa modelul, trebuie declarată o clasă de bază abstractă care specifică o metodă pur virtuală clone(). Orice clasă care are nevoie de o capacitate numită "constructor polimorf" se deduce din clasa de bază abstractă și implementează operația clone(). Clientul, în loc să scrie cod care invocă operatorul "new" pe un nume de clasă greu codificat, apelează metoda **clone()** a prototipului.

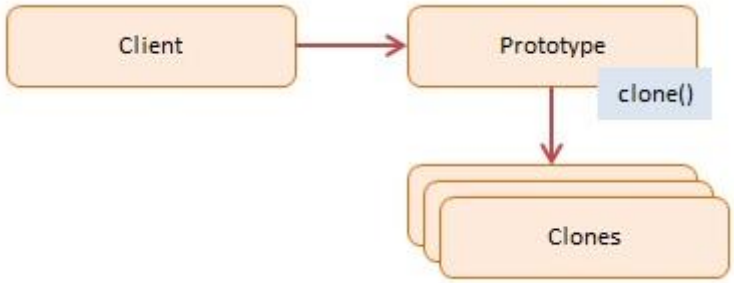

Figura 13. Prototype

[**Sursa**: http://www.dofactory.com/javascript/prototype-design-pattern]

Diviziunea mitotică a unei celule - care are ca rezultat două celule identice - este un exemplu de prototip care joacă un rol activ în copierea în sine și, astfel, demonstrează modelul Prototype. Atunci când o celulă se descompune, rezultă două celule cu genotip identic. Cu alte cuvinte, celula se clonează singură.

Problema: Pentru un labirint de 10 x 10 aveam nevoie să instanțiem aproximativ 50 de cuburi și 50 de plăci fiecare cu dependențele lor grafice (mesh, geometrie etc.). Timpul de încărcare crește exponențial cu dimensiunea scenei de joc. [14]

Soluție: Am creat o clasă de tip Prototype în care am importat un singur cub și o singură placă cu dependențele aferente lor. În momentul în care iterăm prin matricea labirint generată procedural, instanțiem obiecte noi pasând ca parametru prototipul de care avem nevoie (cub, placă, etc.) iar în momentul instanțierii în constructor clonăm prototipul pasat. Acest lucru permițându-ne să extindem jocul oricât de mult dorim și în același timp să avem o modularitate mai mare.

```
1   import { Box3 } from 'three';
2
3   export class Character {
4     constructor(prototype, param, camera) {
5       this.camera = camera;
6       this.object = prototype.clone();
7
8       this.moveKey = {
9         up: false,
10        left: false,
11        down: false,
12        right: false
13      };
14      this.collisionList = [];
15
16      this.object.add(camera);
17      this.keyboardEvents();
18      this.setPosition(param.dungeon);
19      this.setCollisionList(param.dungeon);
20    }
```

Figura 14. Use-case Prototype

Model-View-Controller (MVC)

MVC, sau Model-View-Controller [18] este un șablon arhitectural folosit în industria de software development (inclusiv web development). Această modalitate de lucru reușește cu succes izolarea părții logice de interfața proiectului, rezultând în aplicații extrem de ușor de modificat. În organizarea MVC, modelul reprezintă informația (datele) de care are nevoie aplicația, viewer-ul corespunde cu elementele de interfață, iar controller-ul reprezintă sistemul comunicativ și decizional ce procesează datele informaționale, făcând legătura între model și view.

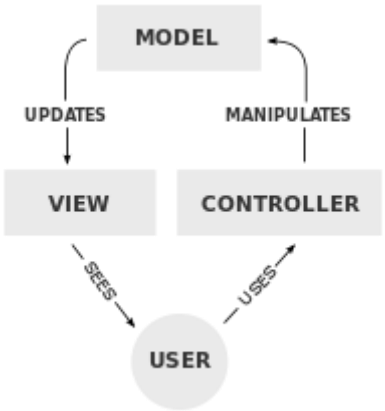

Figura 15. Model View Controller

În această arhitectură partea de model (element 3D) este separată de partea de logică și anume:

- *this.modules(tile)* - partea de logică (Controller)
- *this.object.add(tile.object)* - partea de model 3D

```
72    generateTiles(prototypes) {
73      let tilePrototypes = prototypes.find(prototype => prototype.type === 'tile').prototypes;
74      let layout = DungeonGenerator.generateLayout(this.dataModel.segmentCount);
75
76      for (let i = 0; i < layout.length; i++) {
77        for (let j = 0; j < layout[i].length; j++) {
78          let prototype = this.getTilePrototype(layout[i][j].type, tilePrototypes);
79          let tile = new Tile(prototype, {
80            position: {
81              x: -(this.dataModel.totalSize / 2 - this.dataModel.segmentSize / 2) + this.dataModel.segmentSize * i,
82              z: -(this.dataModel.totalSize / 2 - this.dataModel.segmentSize / 2) + this.dataModel.segmentSize * j
83            }
84          });
85
86          this.modules.push(tile);
87          this.object.add(tile.object);
88        }
89      }
90    }
```

Figura 16. Exemplu concret de MVC

Modulele aplicației

Scene

```
run() {
  let controls = new OrbitControls(this.camera, this.renderer.domElement);
  controls.maxPolarAngle = Math.PI * 0.5;
  controls.minDistance = 5;
  controls.maxDistance = 10000;

  this.load().then((modules) => {
    modules.forEach((object) => {
      this.scene.add(object);
    });
    this.loop();
  });
}

/**
 *
 */
loop() {
  let renderLoop = () => {
    requestAnimationFrame(renderLoop);
    this.renderer.render(this.scene, this.camera);
    this.update();
  };
  renderLoop();
}
```

Figura 17. Scena aplicației

Acest modul se va ocupa de proiectarea și redarea elementelor 3D. În constructor vom instanția clasele:

- Camera - pentru a vizualiza scena
- OrbitControls - cu două atribute și anume camera și obiectul de redare Această instanță este folosită pentru a controla camera folosind mouse-ul sau tastatura.
- Dungeon - modulul în care este implementat logica jocului

Tot în acest modul sunt implementate patru metode:

- `run()` - se ocupă de încărcarea și punerea în scenă a tuturor modelelor 3D din aplicație
- `loop()` - bucla de redare care se autoapelează de aproximativ 60 de ori într-o secundă
- `load()` - returnează un vector de promisiuni cu toate elementele 3D prezente în aplicație
- `update()` – se apelează metoda update pentru fiecare modul 3D care are o metodă de update

Camera

```
1  import { PerspectiveCamera } from 'three';
2
3  export class AppCamera extends PerspectiveCamera {
4    constructor() {
5      super(75, window.innerWidth / window.innerHeight, 0.1, 10000);
6      this.position.set(25, 225, 25);
7    }
8  }
9
```

Figura 18. Modulul Cameră

Un simplu modul de cameră perspectivă care îți permite să vizualizezi scena în momentul jocului.

Renderer

```
1   import { WebGLRenderer } from 'three';
2
3   export class AppRenderer extends WebGLRenderer {
4     constructor() {
5       super({
6         antialias: true,
7         canvas: document.querySelector('#app')
8       });
9       this.setClearColor(0xCCCCCC);
10      this.setSize(window.innerWidth, window.innerHeight);
11
12      window.addEventListener('resize', () => {
13        this.setSize(window.innerWidth, window.innerHeight);
14      }, false);
15    }
16  }
17
```

Figura 19. Modulul de redare

Modulul de redare din three.js care se va ocupa de redarea propriu-zisă în browser a tuturor elementelor 3D.

Dungeon

Mare parte din logica acestui joc este implementată în acest modul.

- `get fetch()` - returnează o promisiune a metodei *generate*
- `generate()` - această metodă rezolvă promisiunea returnată de metoda *load*
- `load()` – crează câte un prototip pentru fiecare obiect 3D care poate fi reutilizat în aplicație *(Tile, Monster, Character)*
- `generateGrid()` și `generateAxis()` - redau elementele ajutătoare în procesul de dezvoltare. Rezultatul se poate vedea în imaginea de mai jos:

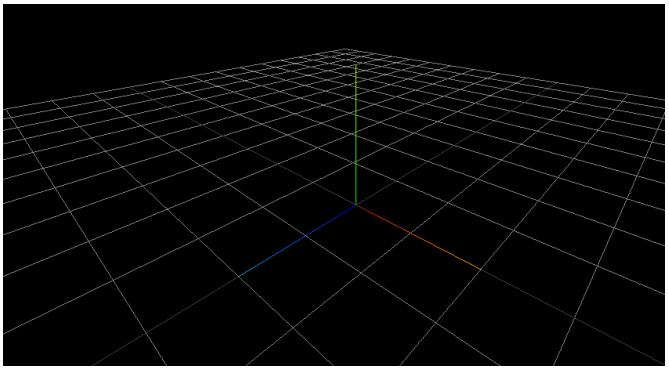

Figura 20. Grid & Axis Three.js

[**Sursa**: https://threejs.org/examples/?q=axis]

- `getTilePrototype(type, prototypeArray)` – caută și returnează din vectorul de prototipuri, prototipul cerut (*ground, wall, start, end*)
- `generateTiles(prototypes)` – iterează prin matricea generată procedural și construiește placă cu placă fiecare părticică din labirint folosindu-se de metoda de mai sus *getTilePrototype*
- `generateMonsters(prototypes)` – caută în vectorul de prototipuri și returnează o clonă a unui monstru
- `generateCharacter(prototypes)` – caută în vectorul de prototipuri și returnează o clonă a unui personaj
- `update()` – iterează prin fiecare obiect clonat și apelează acolo unde este cazul metoda update pentru fiecare în parte

Codul sursă al acestui modul de arhitectură foarte important se poate găsi la sfârșitul acestei lucrări la secțiunea Anexe.

TilePrototype

Clasa *TilePrototype* are în componența sa patru metode. În cele ce urmează le vom explica pe fiecare în parte.

- `get fetch()` – returnează o promisiune a metodei *generate*
- `generate()` – această metodă rezolvă promisiunea returnată de metoda *load* și încarcă obiectul într-un vector de obiecte având unul din tipurile: *ground, wall, start, end*
- `resolveGeometry(type, size)` – pentru un tip pasat ca parametru returnează un modelul de Three.js corespunzător *(PlaneBufferGeometry* pentru placa de tip *ground, BoxBufferGeometry* pentru placa de tip *wall)*
- `load()` – cu ajutorul unui încărcător de date încărcăm *Mesh-ul* corespunzător pentru fiecare obiect 3D.

```
load() {
  let loader = new TextureLoader();
  let assets = [{
    name: 'ground',
    textureUrl: './resources/ground.png',
    geometry: 'plane'
  },{
    name: 'wall',
    textureUrl: './resources/wall.png',
    geometry: 'box'
  },{
    name: 'spawn',
    textureUrl: './resources/circle_a.png',
    geometry: 'plane'
  },{
    name: 'exit',
    textureUrl: './resources/circle_b.png',
    geometry: 'plane'
  }];

  let promises = assets.map((asset) => {
    return new Promise((resolve, reject) => {
      loader.load(asset.textureUrl, (response) => {
        this.map.assets.set(asset.name, {
          geometry: this.resolveGeometry(asset.geometry, 20),
          material: new MeshBasicMaterial({ side: DoubleSide, map: response})
        });

        resolve(response);
      });
    });
  });

  return Promise.all(promises);
}
```

Figura 21. Metoda load() din modulul Tile

MonsterPrototype și ***CharacterPrototype*** se bazează pe același structură ca cea prezentată în modulul *Tile*.

Character

Acest modul instanțiază clasa *CharacterPrototype* de fiecare dată un jucător se conectează. Modulul Character cuprinde următoarele metode:

- `keyboardEvents()`
- `setPosition(dungeon)`
- `setCollisionList(dungeon)`
- `checkMoveDiagonally()`
- `getMoveSpeed()`
- `move(backPeddle)`
- `update()`

Monster

În acest modul se clonează clasa *MonsterPrototype* în funcție de câți inamici sunt setați să apară în scenă. Această clasă cuprinde următoarele metode:

- `setPosition()`
- `setCollisionList(dungeon)`
- `chooseAvailableDirection(dungeon)`
- `checkCollision()`
- `step(backwards)`
- `move()`
- `update()`

LabirintGenerator

Pentru generarea labirintului am folosit algoritmul Growing Three. Acest algoritm are o complexitate de $O(n^2)$.

O scurtă explicație a algoritmului Growing Tree:

1. Fie C o listă de celule, inițial goală. Adăugați o celulă în C, la întâmplare.
2. Alegeți o celulă din C și extrageți un pasaj către orice vecin nevizitat din acea celulă, adăugând și acel vecin în C. Dacă nu există vecini nevizitați, eliminați celula din C.
3. Se repetă Pasul 2 până când lista C este goală.

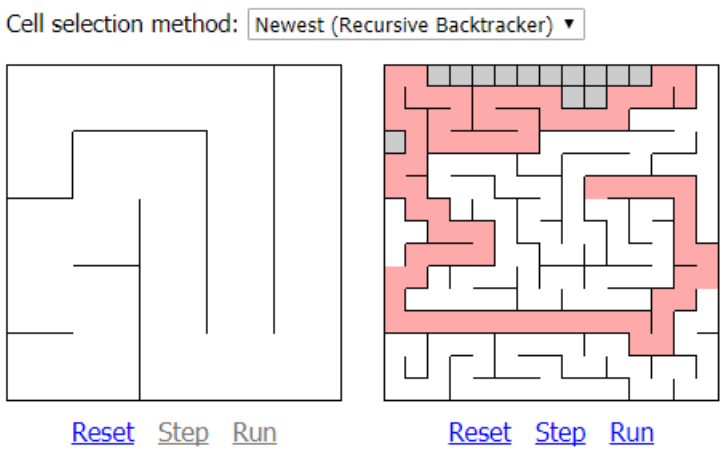

Figura 22. Algoritmul Growing Tree de generare a labirintului

[**Sursa**: http://weblog.jamisbuck.org/2011/1/27/maze-generation-growing-tree-algorithm]

Elemente de interfață

Elementele de interfață prezentate în această lucrare de licență sunt: pagina de înregistrare, scorul și timpul de joc. Pe baza timpului de joc (cât îi ia unui jucător să ajungă de la start la final) se calculează și scorul jucătorului.

Prezentare aplicației

Pentru o mai bună înțelegere a cenceptelor expuse anterior în cele ce urmează voi prezenta jocul întro manieră vizuală cu explicații aferente acolo unde este nevoie.

Pagina de start. Jucătorul introducând numele și apăsând login în spate se creează o conexiune WebSocket la server și se crează instanța de joc.

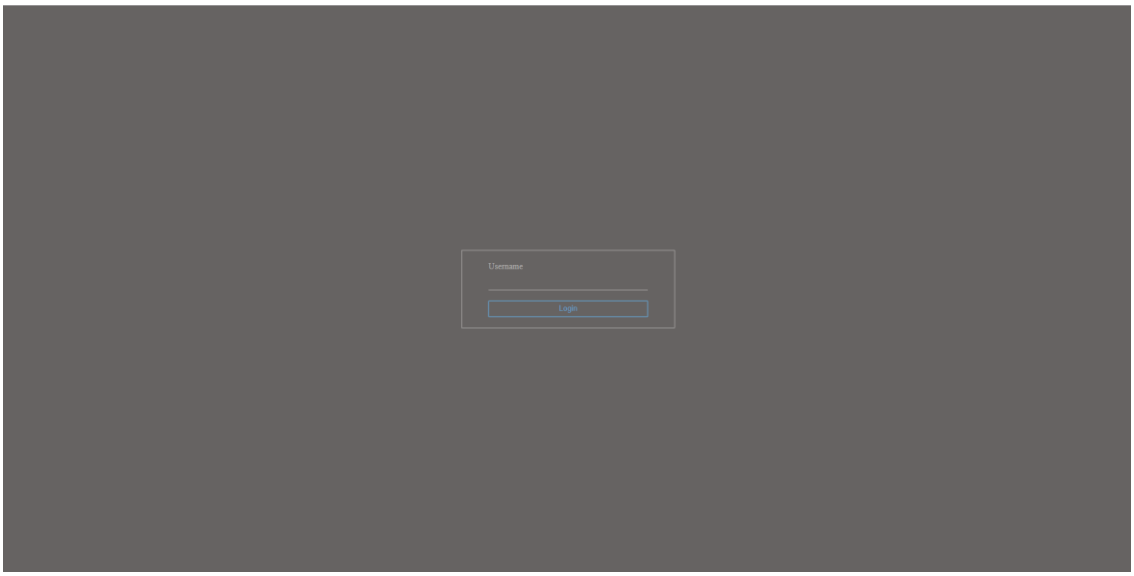

Figura 23. Pagina de start

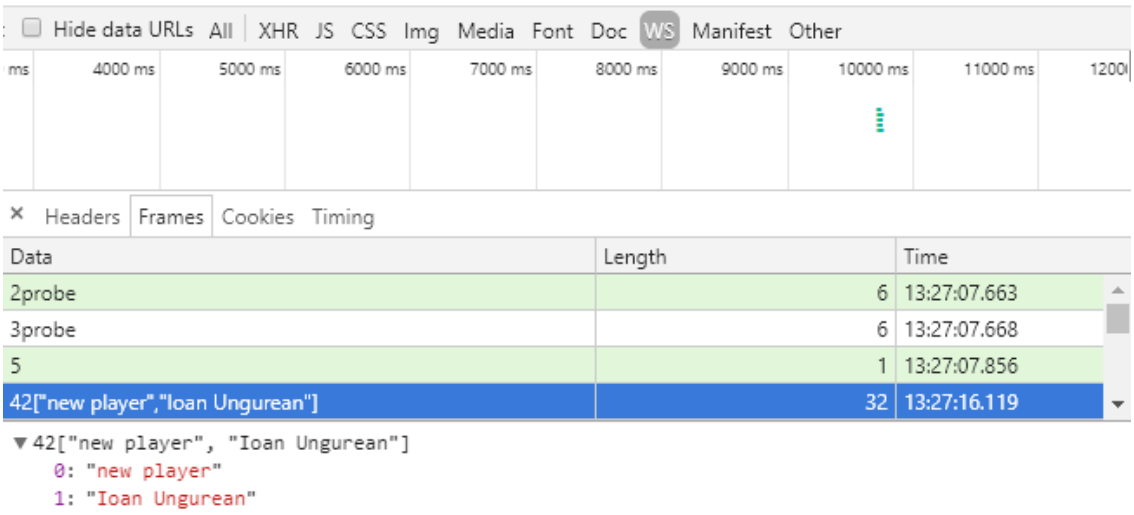

Figura 24. Datele transmise către server la înregistrare

Imediat după înregistrare se instanțiază arena de joc unde vedem personajul plasat la poziția de start, personajele controlate de calculator (NPC-uri). Personajul nu poate vedea decât până la o anumită distanță el fiind limitat de ceața din jurul lui. Prin acest simplu element jocul devine mult mai interesant și creează un sentiment de mister.

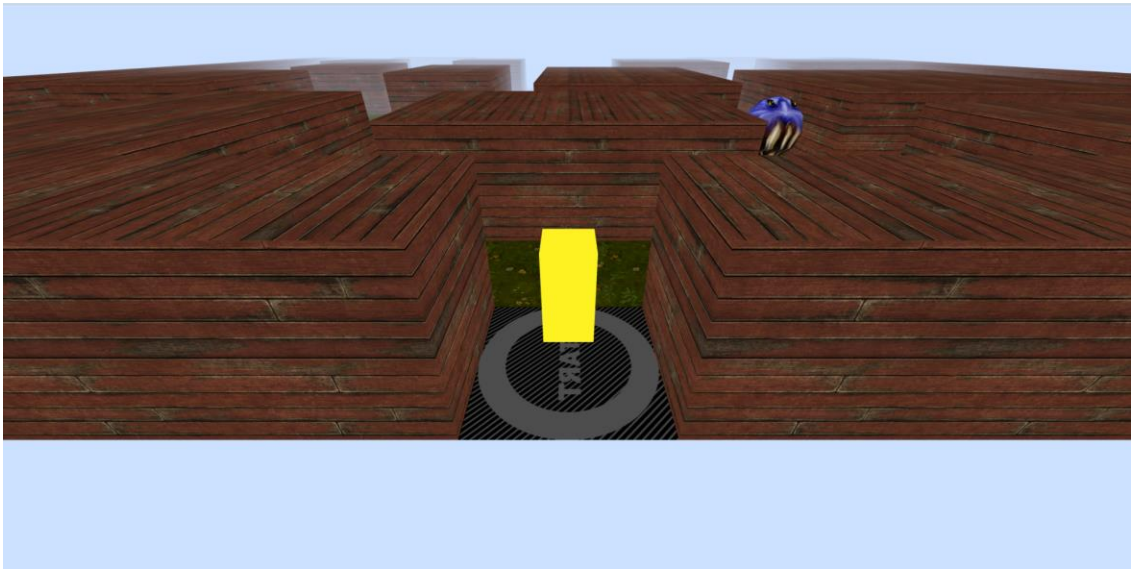

Figura 25. Arena de joc

Figura 26. Elemente din joc

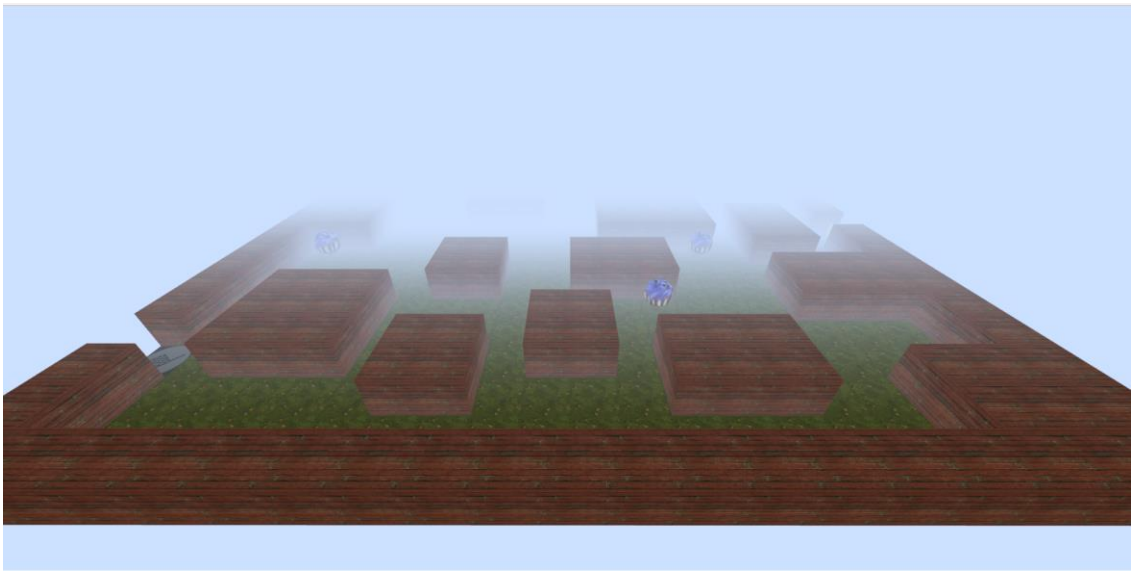

Figura 27. Arena de joc - Imagine de ansamblu

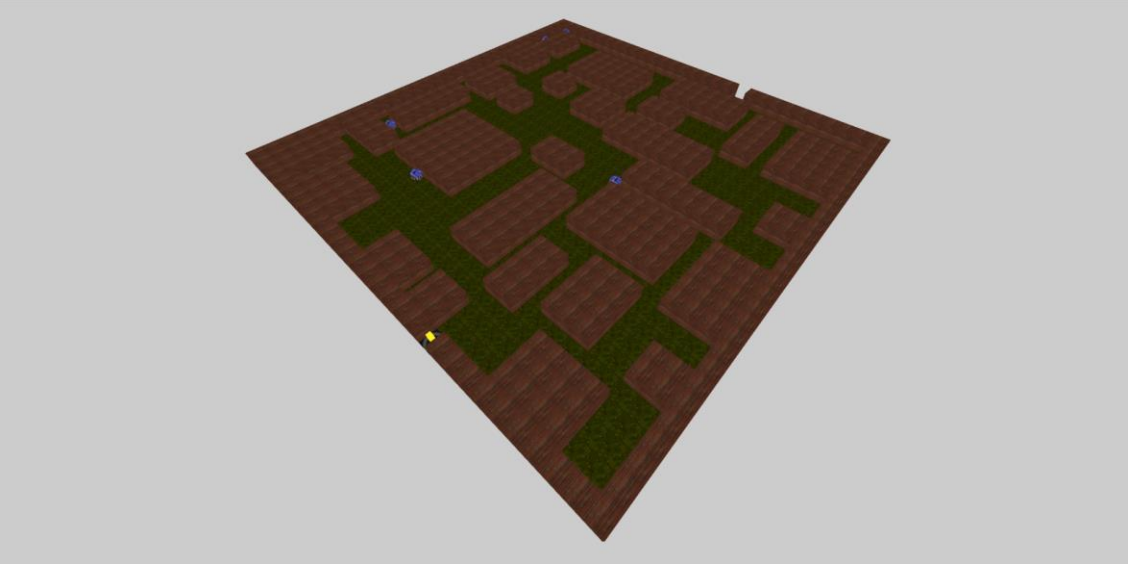

Figura 28. Arena de joc - Imagine de ansamblu

Concluzii

În acest capitol a fost prezentatată cea mai importantă parte a lucrării de licență și anume arhitectura aplicației punând accent pe modularitate, scalabilitate și flexibilitate. Aceste trei elemente sun și cele mai importante atunci când vine vorba despre construirea unei arhitecturi pentru o aplicație, oricare ar fi ea.

Capitolul IV

Concluzii finale

Arhitectura din cadrul proiectului curent de licență, dar și jocul prototip implementat pe acestă arhitectură oferă dezvoltatorului o bază foarte importantă pentru crearea unor jocuri viitoare bazate pe tehnica de grilaj (tile based), iar utilizatorului un mijloc de relaxare.

Contribuții

Prin acest proiect ne-am propus creea unei arhitecturi care să fie modulară, scalabilă și flexibilă astfel încât să putem oferi posibilitatea oricărui dezvoltator să poată să-și fructifice ideile fără un efort prea mare. Jocul prototip dezvoltat pe această arhitectură poate rula în browser, are o interfață 3D, și poate fi jucat fără nici un efort din partea clientului, de descărcare și/sau instalare.

Lucrarea de față prezintă tehnologii relativ noi precum WebGL, HTML5 și folosește o selecție de tehnici riguroase de optimizare pentru a îmbunătăți experiența utilizatorului.

Jocul aduce la viață o scenă în care utilizatorul este nevoit să folosească personajul pentru a captura jucătorii adverși, dar și personajele controlate de calculator (NPC), folosind strategii viabile. Jocul este bazat pe o tehnică de grilaj (tile based).

Îmbunătățiri viitoare

În viitor, jocul se poate extinde pe mai multe planuri, cu mai multe nivele de dificultate, mai mulți inamici, mai multe elemente care să creeze impresia de diversitate.

Este posibilă implementarea unui chat și a unui sistem de clasament online. Această adițională implementare presupune o dezvoltare mai riguroasă a aplicației server.

Se poate oferi de asemenea posibilitatea ca utilizatorii să își selecteze propriile personaje. O adiție complementară acesteia este creșterea numărului de personaje disponibile în joc. Se pot adauga abilitați speciale pentru aceste personaje (e.g înviere, atac dublu, apărare zonală, scut ș.a.) sau se pot extinde atributele deja prezente, un exemplu ar fi ca apărarea să fie bazată și pe materialul folosit în costumul personajului (e.g haine simple, robe, platoșă ș.a.).

Trebuie notat faptul că acestea ar crește de asemenea și timpul de încărcare. Se ajunge astfel la dilema cu care se confruntă dezvoltatorii cel mai des, conținut sau optimizare?

Bibliografie

[1] https://www.slideshare.net/busaco/dezvoltarea-jocurilor-web

[2] https://profs.info.uaic.ro/~busaco/teach/courses/wade/presentations/web-nodejs.pdf

[3] https://nodejs.org/docs/latest-v5.x/api/

[4] https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs

[5] https://socket.io/docs/

[6] https://www.npmjs.com/package/webpack-middleware

[7] https://webpack.js.org/concepts/

[8] https://threejs.org/docs/

[9] http://12devsofxmas.co.uk/2012/01/webgl-and-three-js

[10] https://developer.mozilla.org/en-US/docs/Web/JavaScript

[11] https://developer.mozilla.org/en-US/docs/Web/CSS

[12] http://sass-lang.com/documentation/file.SASS_REFERENCE.html

[13] https://developer.mozilla.org/en-US/docs/Web/CSS/CSS3

[14] https://en.bem.info/methodology/quick-start/

[15] https://threejs.org/docs/index.html#manual/introduction/Creating-a-scene/

[16] https://sourcemaking.com/design_patterns/singleton

[17] https://sourcemaking.com/design_patterns/prototype

[18] http://www.worldit.info/articole/introducere-in-design-patterns-mvc-partea-i/

Anexe

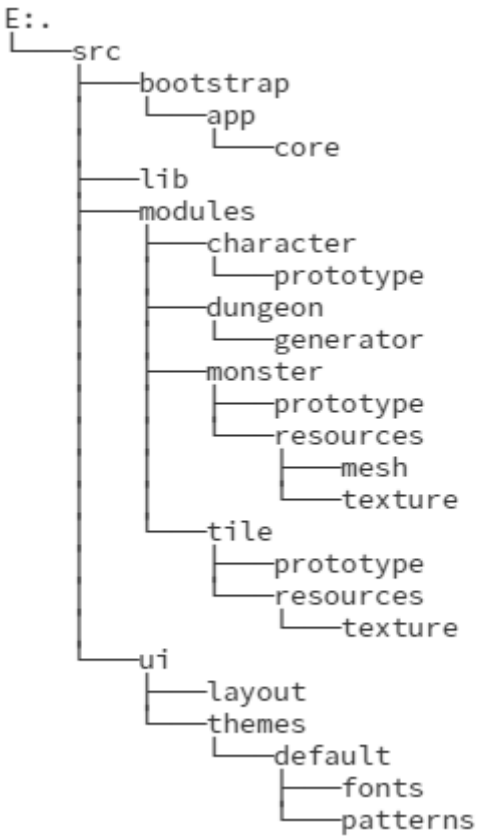

Figura 29. Structura folderelor

```
72  generateTiles(prototypes) {
73    let tilePrototypes = prototypes.find(prototype => prototype.type === 'tile').prototypes;
74    let layout = DungeonGenerator.generateLayout(this.dataModel.segmentCount);
75
76    for (let i = 0; i < layout.length; i++) {
77      for (let j = 0; j < layout[i].length; j++) {
78        let prototype = this.getTilePrototype(layout[i][j].type, tilePrototypes);
79        let tile = new Tile(prototype, {
80          position: {
81            x: -(this.dataModel.totalSize / 2 - this.dataModel.segmentSize / 2) + this.dataModel.segmentSize * i,
82            z: -(this.dataModel.totalSize / 2 - this.dataModel.segmentSize / 2) + this.dataModel.segmentSize * j
83          }
84        });
85
86        this.modules.push(tile);
87        this.object.add(tile.object);
88      }
89    }
90  }
91
92  generateMonsters(prototypes) {
93    let monsterPrototypes = prototypes.find(prototype => prototype.type === 'monster').prototypes;
94    for (let i = 0; i < 45; i++) {
95      let monster = new Monster(monsterPrototypes[0], {
96        dungeon: this.object
97      }); // TODO: other way
98      this.modules.push(monster);
99      this.object.add(monster.object);
100   }
101 }
102
103 generateCharacter(prototypes) {
104   let characterPrototypes = prototypes.find(prototype => prototype.type === 'character').prototypes;
105   let character = new Character(characterPrototypes[0], {
106     dungeon: this.object
107   }, this.camera);
108   this.modules.push(character);
109   this.object.add(character.object);
110 }
111
112 update() {
113   this.modules.forEach((module) => {
114     if (module && module.update) {
115       module.update();
116     }
117   });
118 }
```

Figura 30. Modulul Dungeon

```
generate() {
  return this.load().then((resources) => {
    this.map.assets.forEach((value, key) => {
      let prototype = new Mesh(
        value['geometry'],
        value['material']
      );
      prototype.name = key;
      if (key === 'wall') {
        prototype.position.y = this.size / 2;
      } else {
        prototype.rotation.x = Math.PI * 0.5;
      }

      this.prototypes.push(prototype);
    });

    return {
      type: 'tile',
      prototypes: this.prototypes
    };
  });
}

resolveGeometry(type, size) {
  switch (type) {
    case 'plane':
      return new PlaneBufferGeometry(size, size, 1, 1);
    case 'box':
      return new BoxBufferGeometry(size, size, size);
  }
}

load() {
  let loader = new TextureLoader();
  let assets = [{
    name: 'ground',
    textureUrl: './resources/ground.png',
    geometry: 'plane'
  },{
    name: 'wall',
    textureUrl: './resources/wall.png',
    geometry: 'box'
  },{
    name: 'spawn',
    textureUrl: './resources/circle_a.png',
    geometry: 'plane'
  },{
    name: 'exit',
    textureUrl: './resources/circle_b.png',
    geometry: 'plane'
  }];

  let promises = assets.map((asset) => {
    return new Promise((resolve, reject) => {
      loader.load(asset.textureUrl, (response) => {
        this.map.assets.set(asset.name, {
          geometry: this.resolveGeometry(asset.geometry, 20),
          material: new MeshBasicMaterial({ side: DoubleSide, map: response})
        });

        resolve(response);
      });
    });
  });

  return Promise.all(promises);
}
```

Figura 31. Prototipul Tile