



**UNIVERSITÀ
DI PARMA**

Relazione progetto tecnologie internet

Corso a cura del professore Michele Amoretti

Giuseppe Ricciardi 306330, Helmi Zammali 310312

Ingegneria IET, luglio 2022

Indice

1	Introduzione	2
1.1	Architettura Client-Server e utilizzo delle socket	2
1.2	Linguaggi, programmi e librerie utilizzate	3
1.3	La struttura del progetto	4
2	La logica di gioco	5
2.1	Phaser.game, connessione e disconnessione del giocatore	5
2.2	Movimento e rotazione	7
2.3	La meccanica di shooting	10
3	Mondo di gioco e collisioni	13
3.1	Struttura della mappa	13
3.2	Gestione collisioni con la mappa	14
4	Gestione degli utenti tramite MongoDB	15
4.1	Login	15
4.2	Registrazione	17
4.3	Classifica	17

1 Introduzione

Lo scopo di questa relazione è quello di illustrare e analizzare il processo di realizzazione del progetto per il corso di *Tecnologie internet*.

Si tratta di un gioco top-down shooter multigiocatore con architettura client-server che prende il nome di *CSUO: Counter Strike Unipr Offensive*.

L'idea nasce dalla passione di entrambi per il noto gioco *Counter Strike Global Offensive* e dalla piena condivisione del pensiero: "s'impara soltanto divertendosi." del poeta francese Anatole France.



Figure 1: Gameplay del gioco

1.1 Architettura Client-Server e utilizzo delle socket

Il gioco si basa su un'*architettura client-server*, il **client** ha il compito di gestire gli input del giocatore, mostrare tutti gli altri giocatori e le loro azioni nel mondo di gioco. Ogni client scambia i messaggi con un **server autoritario** che gestisce la *logica del gioco* (cap.2): movimento dei giocatori, collisioni con il mondo di gioco, meccanica di fuoco etc.

La comunicazione tra server e client avviene tramite **socket** che gestisce una comunicazione event-based, low-latency e full duplex basata su una singola connessione TCP.

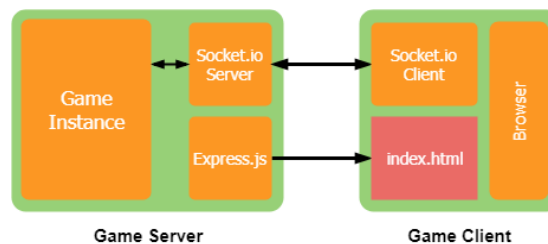


Figure 2: Schema riassuntivo dell'architettura del gioco

N.B: l'utilizzo di un server autoritario permette l'installazione della libreria *phaser* usata per la logica di gioco direttamente sul server per evitare che un giocatore possa alterare del codice che gestisce meccaniche di gioco in suo favore.

1.2 Linguaggi, programmi e librerie utilizzate

I linguaggi utilizzati per la realizzazione del progetto sono: **HTML**, **JS** e **CSS**. Per la classifica è stato utilizzato il database NoSQL **MongoDB**. La condivisione del codice è stata fatta tramite una directory condivisa sul sito *Github* e l'utilizzo del software **GIT**.

Sono state utilizzate diverse librerie:

- **Node.js e Express:** per la realizzazione del server che gestisce tutta la logica del gioco, express è un framework che permette il rendering degli static-file lato client.
- **Phaser:** è un 2D game framework usato per creare giochi HTML5 per desktop e dispositivi mobile, ha vari physics engine integrati (e ben documentati) ed è stato molto utile per vari aspetti relativi alla creazione del gioco.
- **jsdom, node-canvas, datauri:** usate per far funzionare correttamente *phaser: headless* sul server autoritario. Jsdom è usato per ricreare molte delle API del Javascript DOM dal browser, permette di caricare ed interagire con file HTML. Node-canvas è un'implementazione dell'API canvas su Node.js, è richiesto da Phaser per la sua corretta esecuzione. Datauri permette a jsdom di supportare il metodo URL.createObjectURL.
- **dotenv, mongoose** per la connessione al database MongoDB.
- **bcrypt** per criptare tramite hash le password inserite dagli utenti.
- **passport** per gestire l'autenticazione degli utenti.
- **body-parser e cookie-parser** per realizzare operazioni di GET e POST sul database.
- **bootstrap;** framework per fornire rapidamente lo stile ai form di login e registrazione tramite CSS.

1.3 La struttura del progetto

app.js inizializza il client ed il server autoritario chiamando i file *index.html* (root pages) delle rispettive cartelle, inoltre stabilisce la connessione con il database MongoDB.

La cartella *auth* contiene *auth.js* che gestisce l'autenticazione dell'utente (registrazione e login), *user-Model.js* definisce la definizione del modello utente presente all'interno del database, *routes* ha tutti gli endpoints necessari per registrare/loggare gli utenti (*main.js*) e aggiornare/ottenere gli scores per la leaderboard (*secure.js*). *Middleware* contiene *asyncMiddleware.js* per gestire le eccezioni delle varie routes che usano funzioni asincrone del tipo "wait" o "async".

authoritative_server include tutti i file necessari alla logica del gioco lato server, la cartella *assets* (presente anche nel client) comprende tutti i file multimediali da importare (suoni,immagini,spritesheet etc.). *js* include tutti i file javascript utilizzati lato server: *game.js* gestisce la logica di gioco, *utilites.js* definisce i metodi utilizzati da *game.js* e *actors.js* contiene le classi.

Per il lato client l'*index.html* è leggermente più complesso: esso infatti comprende la sessione di login dell'utente ed ha collegamenti per accedere a *signup.html* nel caso l'utente non fosse già registrato nel database, e *game.html* che gestisce l'avvio del gioco lato client una volta che il login sia avvenuto con successo. *main.css* gestisce la grafica di *index.html* e *signup.html*.

Anche il client possiede i file *js game,actors,utilites*, inoltre sono presenti *menu.js* e *leaderboard.js* che sono due classi *scena* di Phaser (anche *game.js* lo è) e costituiscono rispettivamente il menu e la classifica del gioco. *main.js* gestisce tutte le scene di gioco lato client, e *refreshToken.js* serve per verificare ciclicamente l'identità dell'utente.

```

  game
  auth
  JS auth.js
  authoritative_server
  assets
  js
  JS actors.js
  JS game.js
  JS utilites.js
  index.html
  client
  assets
  css
  # main.css
  js
  game.html
  index.html
  signup.html
  middleware
  JS asyncMiddleware.js
  models
  JS userModel.js
  routes
  JS main.js
  JS secure.js
  JS app.js
```

2 La logica di gioco

In questo capitolo verranno analizzate le principali meccaniche di gioco, per comodità sono state riportate solo le parti di codice più importanti.

2.1 Phaser.game, connessione e disconnessione del giocatore

L'oggetto *game* è la classe principale di Phaser da cui partono tutti gli altri oggetti, come le *scene*. Una **scena** è una classe che può essere modificata per i propri usi personali, ad ogni scena viene assegnato nel costruttore un *id* che viene utilizzato per essere chiamato dalle altre scene, i metodi principali sono:

- *Preload()*: gestisce il caricamento dei file multimediali nel gioco.
- *Create()*: gestisce tutte le operazioni che devono essere svolte appena l'istanza game viene creata.
- *Update()*: svolge le operazioni definite al suo interno con un framerate di 60 frames/secondi, è utilizzata per aggiornare la posizione dei giocatori e mandare al server gli input del giocatore.

Nel nostro progetto sono presenti 3 scene lato *client*, definite in *client/js/menu.js*:

1. **Mainmenuscene**: scena che definisce il menu.
2. **Gamescene**: scena che rappresenta il gioco vero e proprio, accessibile dal menu.
3. **Highscore**: scena che specifica la classifica, accessibile anch'essa dal menu.

Nel *server* è presente solo **Gamescene**.

Quando il giocatore clicca dal menu il tasto "play" viene chiamata la scena *Gamescene*:

```
1 if (this.buttons[this.selectedButtonIndex] == this.playButton)
2 {
3     this.menusn.stop()           //stop menu soundtrack
4     this.scene.start('game')    //transit to the new scene
5 }
```

Nel *preload()* della Gamescene vengono caricate le spritesheet del personaggio e la mappa ,che in realtà è soltanto un'immagine PNG (cap. 3). In *scene.create()* viene inizializzata la **connessione** con il server mediante socket con:

```
1 this.socket = io(); //initialization of socket connection
```

Tramite la chiamata alla funzione *io()* della libreria *socket.io* oltre ad inizializzare la socket viene anche mandato un messaggio al server intitolato 'connection' e l'ID della socket che ha effettuato la connessione.

Al server viene mandato anche un messaggio 'username' contenente l'username del giocatore memorizzato tramite l'accesso alla cache della sessione corrente.

```

1  var username = " ";
2
3  function getname(){ //get username of current player
4      username = sessionStorage.getItem("name")
5  }
6  getname();
7
8  // ... CODICE TAGLIATO
9
10 this.socket.emit('username', current_player.name);

```

current_player è una variabile globale utilizzata per gestire il player che l'utente controlla.

Il server gestisce il messaggio in questo modo:

```

1  //on connection of a player
2  io.on('connection', function (socket) {
3      const sessionID = socket.id;
4      socket.on('username', function(name){
5          username = name;
6          //spawn player in a point (x,y) without walls.
7          // ... CODICE TAGLIATO
8          // create a new player and add it to players object
9          // ... CODICE TAGLIATO
10         addPlayer(self, players[socket.id]); //defined in utilites.js
11         handlePlayerCollisions(self); //defined in utilites.js
12         socket.emit('current_player', players[socket.id]);
13         // send the players object to the new player
14         socket.emit('currentPlayers', players);
15         // update all other players of the new player
16         socket.broadcast.emit('newPlayer', players[socket.id]);

```

Crea un nuovo giocatore prendendo username e socket.id dal client che ha effettuato la connessione e lo inserisce nel gruppo players e aggiunge l'oggetto player alla fisica del mondo (cap. 3), infine il server manda un messaggio 'current_player' alla socket per associare l'ID della socket attuale al current_player (non è possibile fare questa operazione direttamente da client perchè l'ID viene associato alla socket solo dopo aver mandato il messaggio 'connection'), trasmette l'intera lista dei giocatori a tutti i client connessi tramite 'currentPlayers' e infine manda un messaggio ai client per aggiungere il giocatore tramite 'socket.broadcast.emit('newPlayer', players[socket.id])'. La **disconnessione** è un evento che viene chiamato in automatico nel momento in cui l'utente termina la sessione (ovvero chiude la scheda) oppure può essere chiamato tramite *socket.disconnect()*, nel nostro caso quando l'utente ritorna al menu tramite il tasto 'ESC'. Se l'evento viene triggerato manda un messaggio 'disconnect' al server.

```

1  //ESC key for going back to the menu
2  this.input.keyboard.on('keydown_ESC', function (event) {
3      if(confirm("Press OK to go back to menu ")){
4          self.socket.disconnect();
5          self.scene.stop();
6          self.scene.start('menu');
7      }
8  }, 0, this);

```

Quando il server riceve il messaggio di disconnessione elimina il giocatore associato e trasmette agli altri client un messaggio 'remove' con l'id del player da eliminare, così facendo tutti i client elimineranno dal gioco l'utente disconnesso

```
1 //SERVER SIDE(authoritative_server/js/game.js):
2 socket.on('disconnect', function ()
3 {
4     // remove player from server
5     removePlayer(self, socket.id); //defined in utilites.js
6     // remove this player from our players group
7     delete players[socket.id];
8     // emit a message to all players to remove this player
9     io.emit('remove', socket.id);
10 });
11
12 //CLIENT SIDE (client/js/game.js):
13 this.socket.on('remove', function (playerId) {
14     self.players.getChildren().forEach(function (player) {
15         if (playerId === player.playerId) {
16             player.hp.bar.destroy();
17             player.destroy();
18             player.userHUD.destroy();
19         }
20     });
21 });
```

Nella prossima sezione verrà analizzata la gestione del movimento del giocatore.

2.2 Movimento e rotazione

Tutti gli **input** del giocatore (pressione o rilascio di un tasto sulla tastiera, il click del mouse etc.) sono gestiti dal client. Nel metodo create() vengono inizializzate le variabili tramite una funzione di phaser:

```
1 //Player input initialization
2 this.keyA = this.input.keyboard.addKey(Phaser.Input.Keyboard.
3     KeyCodes.A);
4 this.keyS = this.input.keyboard.addKey(Phaser.Input.Keyboard.
5     KeyCodes.S);
6 this.keyD = this.input.keyboard.addKey(Phaser.Input.Keyboard.
7     KeyCodes.D);
8 this.keyW = this.input.keyboard.addKey(Phaser.Input.Keyboard.
9     KeyCodes.W);
10 this.leftKeyPressed = false;
11 this.rightKeyPressed = false;
12 this.upKeyPressed = false;
13 this.downKeyPressed = false;
```

In update() viene fatto un controllo costante del valore delle variabili, nel caso l'utente digitasse una delle key inizializzate, la variabile cambia valore (da "true" a "false", o viceversa) e viene mandato un messaggio 'playerinput' al server con i valori delle variabili (left,right,up,down):

```
1 var left = this.leftKeyPressed;
2 var right = this.rightKeyPressed;
3 var up = this.upKeyPressed;
```

```

4  var down = this.downKeyPressed;
5
6  //checking player input
7  if (this.keyA.isDown) {
8      this.leftKeyPressed = true;
9  }
10 else{this.leftKeyPressed = false;}
11 if (this.keyD.isDown) {
12     this.rightKeyPressed = true;
13 }
14 else {this.rightKeyPressed = false;}
15 if (this.keyW.isDown) {
16     this.upKeyPressed = true;
17 }
18 else {this.upKeyPressed = false;}
19 if(this.keyS.isDown){
20     this.downKeyPressed = true;
21 }
22 else {this.downKeyPressed = false;}
23
24 //if something is changed send player input to the server
25 if (left !== this.leftKeyPressed || right !== this.rightKeyPressed
    || up !== this.upKeyPressed || down !== this.downKeyPressed) {
26     this.socket.emit('playerInput', { left: this.leftKeyPressed ,
27         right: this.rightKeyPressed, up: this.upKeyPressed,
28         down: this.downKeyPressed});

```

Il server quando riceve il messaggio "playerinput" dalla socket esegue la funzione *handlePlayerInput* in *create()*:

```

1  function handlePlayerRotation(self,rotation,playerId){
2      self.players.getChildren().forEach((player) => {
3          if (playerId === player.playerId) {
4              players[player.playerId].rotation = rotation;
5          }
6      });
7  }

```

Questa funzione scorre tutti i players e imposta l'input del giocatore dal quale ha ricevuto il messaggio tramite socket, in *update()* il server controlla il valore delle variabili input di ogni giocatore e imposta le loro velocità:

```

1  this.players.getChildren().forEach((player) => { //updating
2      players velocity
3      const input = players[player.playerId].input;
4      if (input.left) {
5          player.setVelocityX(-250);
6          if(input.right)
7          {
8              player.setVelocityX(0);
9          }
10     }
11     else if (input.right) {
12         player.setVelocityX(250);
13         if(input.left)
14         {
15             player.setVelocityX(0);

```



```

16     }
17     else {
18         player.setVelocityX(0);
19     }
20     if (input.up) {
21         player.setVelocityY(-250);
22         if(input.down)
23         {
24             player.setVelocityY(0);
25         }
26     } else if(input.down){
27         player.setVelocityY(250);
28         if(input.up)
29         {
30             player.setVelocityY(0);
31         }
32     }
33     else {
34         player.setVelocityY(0);
35     }
36     players[player.playerId].x = player.x;
37     players[player.playerId].y = player.y;
38     players[player.playerId].velocity_x = player.body.velocity.x;
39     players[player.playerId].velocity_y = player.body.velocity.y;
40 });
41 io.emit('playerUpdates', players);

```

Infine imposta le posizioni di ogni giocatore, ed emette il messaggio 'playerUpdates' con le posizioni aggiornate. Il client gestisce l'evento 'playerUpdates' impostando delle variabili target (target.x,target.y) con i valori ottenuti dal server, queste variabili verranno poi usate per stabilire la posizione dei giocatori in update(). La scelta di salvare momentaneamente le posizioni dei giocatori per poi gestire il **movimento** effettivo in update() è stata presa per rendere più fluida l'esperienza di gioco.

```

1  //IN CREATE() FUNCTION
2  this.socket.on('playerUpdates', function (players) {
3      Object.keys(players).forEach(function (id) {
4          self.players.getChildren().forEach(function (player) {
5              //other players saw by current_player
6              if (players[id].playerId === player.playerId) {
7                  player.targetRotation = players[id].rotation;
8                  player.targetX = players[id].x;
9                  player.targetY = players[id].y;
10             }
11         //... CODICE TAGLIATO
12         //setting rotation of current player (working on player)
13         if (player.playerId === current_player.playerId) {
14             player.rotation = Phaser.Math.Angle.Between(
15                 current_player.x, current_player.y, self.reticle.x,
16                 self.reticle.y);
17             current_player.rotation = player.rotation;
18         }
19     });
20 }

```

Nella parte finale del codice appena visto è presente il calcolo della **rotazione** del giocatore, effettuato tramite la funzione *Phaser.Math.Angle.Between* che

calcola l'angolo tra la posizione (x,y) del giocatore e la posizione del puntatore del mouse, la rotazione viene poi passata al server tramite l'emit di un messaggio 'rotation', emesso ogni volta che il valore della rotazione cambia, con il valore già calcolato per evitare di appesantire ulteriormente il server.

```
1 //IN UPDATE() FUNCTION
2 this.players.getChildren().forEach(function(player){
3     player.setRotation(player.targetRotation);
4     player.setPosition(player.targetX, player.targetY);
5     //... CODICE TAGLIATO
6 })
```

N.B: il cursore (reticle) viene gestito interamente nel client.

2.3 La meccanica di shooting

Con i tasti del mouse il giocatore può bloccare lo schermo oppure sparare (solo nel caso in cui lo schermo sia bloccato), un trigger controlla l'eventuale click da parte dell'utente e manda un messaggio al server se il giocatore ha almeno un colpo a disposizione

```
1 //CREATE() DEL CLIENT
2 this.input.on('pointerdown', function () {
3     if (!game.input.mouse.locked){
4         game.input.mouse.requestPointerLock();
5     }
6     else{
7         if (current_player.ammo!=0)
8         {
9             self.socket.emit('shot', self.reticle)
10            current_player.ammo --;
11        }
12    }
13 });
```

Il server quando riceve questo messaggio crea un nuovo oggetto di classe Bullet (definita in authoritative_server/js/actor.js), e lo aggiunge alla fisica del mondo di gioco mediante la funzione *shotbullet()* definita in authoritative_server/js/utilities.js. Anche il server controlla se il giocatore abbia munizioni, in tal caso procede a sparare il proiettile mediante un metodo della classe Bullet, emette un messaggio 'fire' per confermare effettivamente che il colpo è stato sparato. Nel caso le munizioni fossero esaurite allora il server manda un messaggio 'reload' al client che, una volta ricevuto, effettua l'animazione di ricarica del giocatore con id mandato dal server e viene ripristinato il valore di default del numero di colpi.

```
1 //CREATE() DEL SERVER
2 socket.on('shot', function(reticle){
3     const bullet = new Bullet(self);
4     shotbullet(self, bullet); //defined in utilites.js
5     shotby = players[socket.id].playerId;
6     //shooting
7     if(players[socket.id].ammo != 0){
8         bullet.fire(players[socket.id], reticle);
9         players[socket.id].ammo--;
10        io.emit('fire', bullet, bullet_id);
11    }
12    //...CODICE TAGLIATO
13 })
```

```

12     if(players[socket.id].ammo === 0)
13     {
14         io.emit('reload', shotby);
15         players[socket.id].ammo = 5; //5 is default values
16     }
17 }

```

Il client gestisce così il messaggio 'fire':

```

1 this.socket.on('fire', function(bullet, bullet_id, shotby){
2     bullet.shotby = shotby;
3     bullet.id = bullet_id;
4     displayBullet(self,bullet,'bullet');//def. in utilities.js
5     self.pum.play();
6 })

```

La funzione *displayBullet* aggiunge una sprite 'bullet' (caricata nel preload()) nella posizione passata dal server; da notare che nel server il Bullet è un GameOb-
ject di phaser mentre per il client si tratta solo di un'immagine.

Dopo aver mandato un messaggio per l'emissione del colpo il server controlla la collisione tra proiettile e player:

```

1 self.players.getChildren().forEach((player) => {
2     if(socket.id !== player.playerId){
3         self.physics.add.collider(player, bullet, () => {
4             if (bullet.active === true && player.active === true){
5                 player.health = player.health - 20;
6                 io.emit('hit', player.playerId);
7                 if (player.health == 0){
8                     //RESPAWN PLAYER
9                     //... CODICE TAGLIATO
10                    io.emit('death', player, player.playerId,players[
11                        player.playerId].name);
12                    io.emit('update-scores',players[socket.id].playerId
13                        , players[socket.id].name);
14                }
15                //Destroy bullet
16                handleBulletCollision(bullet);

```

Nel momento in cui avviene una collisione tra player e bullet viene decremen-
tata la vita del giocatore colpito e trasmesso un messaggio al client con il suo
id, il client gestirà la parte visiva del colpo modificando l'oggetto Healthbar
del giocatore colpito. Nel caso in cui il giocatore abbia vita pari a 0 allora il
server procederà con il respawn del giocatore: genera nuovamente una posizione
dove verrà spostato il giocatore e resetterà gli hp, inoltre verranno mandati due
messaggi: 'death' che contiene tutte le informazioni riguardo il giocatore ucciso,
utilizzate dal client per il respawn e per la modifica della leaderboard (cap. 4),
ed il messaggio 'update-scores' utilizzato dal client per aggiornare il numero di
uccisioni fatte da chi ha ucciso il giocatore, infine viene distrutto il proiettile.
La posizione del proiettile viene continuamente aggiornata tramite il metodo
update() della classe Bullet, che manda un emit al client con la posizione del
proiettile;

```

1 //Class Bullet in authoritative_server/js/actors.js
2

```

```

3 update: function (delta) //delta provides access to delta values
  between the Game Objects current and previous position.
4 {
5
6     this.x += this.xSpeed * delta;
7     this.y += this.ySpeed * delta;
8     this.born += delta;
9     if (this.born > 1800)
10    {
11        this.setActive(false);
12        this.setVisible(false);
13    }
14    if(this.valid === true)
15        io.emit('bulletUpdate', this, this.id, this.shotby);
16
17 }

```

Il client quando riceve il messaggio aggiorna la posizione del proiettile da cui riceve il messaggio (ogni proiettile oltre ad avere l'attributo shotby che rappresenta il giocatore che lo ha sparato, contiene un id proprio generato tramite time()):

```

1  this.socket.on('bulletUpdate', function(bullet, bullet_id, shotby)
2  {
3      bullet.id = bullet_id;
4      bullet.shotby = shotby;
5      self.bullets.getChildren().forEach(function (displayed_bullet)
6      {
7          if (bullet.id === displayed_bullet.id) {
8              displayed_bullet.setRotation(bullet.rotation);
9              displayed_bullet.setPosition(bullet.x, bullet.y);
10          }
11      })
12  })

```

Dopo aver analizzato le fasi più importanti del gioco occorre precisare alcune componenti non ancora state menzionate presenti nel client:

- La classe **Healthbar**, presente in client/js/actors.js rappresenta la vita dei giocatori, viene stanziato nel metodo *displayPlayers*.
- La variabile **ammoHUD**, un testo che indica il numero di proiettili disponibili per il giocatore.

Queste componenti non sono state inserite nel server perché non influenzano in alcun modo la logica del gioco, avrebbero soltanto speso memoria ed inciso negativamente nella fluidità del gioco.

3 Mondo di gioco e collisioni

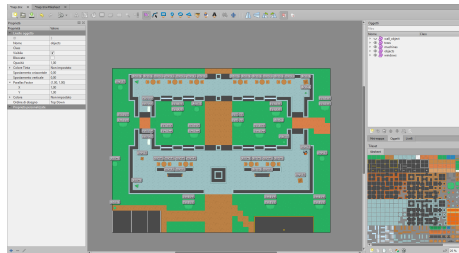
Tutte le sprites utilizzate provengono dall'autore Kenney, sono copyright free sia per uso personale che commerciale.



Figure 3: Preview delle sprites

3.1 Struttura della mappa

L'applicazione *Tiled* è stata usata per la progettazione della mappa, la sua caratteristica principale è quella di esportare le tilemaps in formato *.json*, supportato nativamente dalla libreria Phaser.



L'utilizzo di mappe in formato json facilita molto le operazioni da effettuare con essa, come ad esempio l'aggiunta delle collisioni. La mappa è costruita su più layer sovrapponibili tra loro (simile al funzionamento di photoshop/GIMP/etc.), distinguiamo due tipi di layer:

- *Tile layer*: nel nostro caso è lo sfondo della mappa (tutte le caselle rappresentanti il prato, pavimento e l'asfalto).
- *Object layer*: tutti gli oggetti per il quale vogliamo aggiungere della fisica, estraibili direttamente con Phaser dopo aver importato la mappa. Nella mappa sono presenti un layer per gli alberi, uno per i muri ed uno per gli altri oggetti.

3.2 Gestione collisioni con la mappa

La mappa viene caricata nel `preload()` della Gamescene del server, nel `create()` vengono estratti i vari object layer. Una volta estratti, essi vengono iterati per ottenere ogni singola tile oggetto per aggiungerla alla fisica del mondo e per implementare le collisioni, il tutto viene fatto con poche righe di codice:

```
1 //adding objects layer to the map with physics
2 this.objects = map.createFromObjects('objects', 'object1');
3 this.objects.forEach(object => {
4     this.physics.world.enable(object);
5     object.body.immovable = true;
6 })
```

Lo stesso procedimento viene fatto per gli altri object layer, in alcuni layer sono state modificate le dimensioni delle tiles tramite phaser per avere una collisione migliore.

```
1 //adding trees layer to the map with physics
2 this.trees = map.createFromObjects('trees', 'tree');
3 this.trees.forEach(tree => { //editing for better collision
4     tree.width = 35;
5     tree.height = 35;
6     this.physics.world.enable(tree);
7     tree.body.immovable = true;
8 })
```

4 Gestione degli utenti tramite MongoDB

Tutte le informazioni sui giocatori vengono salvate su un database MongoDB hostato tramite Atlas cloud service. L'URI per la connessione al database è salvato in un file `.env`, viene utilizzato per il collegamento in `app.js`. Ogni utente è caratterizzato dai seguenti attributi definiti in `userModels.js`:

- **Username:** è una stringa che funge da ID del DB e rappresenta l'username con il quale l'utente si registra, l'username ha anche il vincolo d'unicità: due utenti non possono registrarsi con lo stesso username.
- **Password:** stringa utilizzata dall'utente per accedere, durante la registrazione viene criptata tramite hash.
- **Kills:** intero che costituisce il numero di uccisioni fatte dal giocatore.
- **Deaths:** intero per il numero di morti del giocatore.

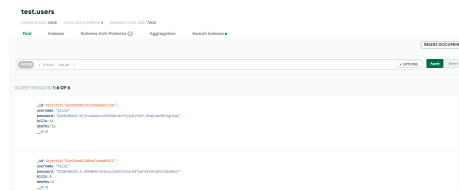


Figure 4: View del database da MongoDB Cloud

4.1 Login

Quando l'utente accede all'applicazione tramite localhost viene indirizzato alla pagina `client/index.html`, dove è definito la *form* per il **login**. Una volta inseriti username e password viene eseguita la funzione `signIn` che prova a fare un POST per accedere alla route `/login`.

```
1 function signIn() {
2   var data = {
3     username: document.forms[0].elements[0].value,
4     password: document.forms[0].elements[1].value
5   };
6   var name = data.username;
7   $.ajax({
8     type: 'POST',
9     url: '/login',
10    data,
11    success: function (data) {
12      sessionStorage.setItem("name", name)
13      window.location.replace('/game.html');
14    },
15    error: function (xhr) {
16      window.alert('incorrect username or password');
17      window.location.replace('/index.html');
18    }
19  });
20 }
```

La route `/login` è definita in `routes/main.js`, in questa route è stato aggiunto il middleware `passport.authenticate` impostato per utilizzare la configurazione di login del passport (definita in `auth.js`). Nella funzione di callback, controlliamo prima se si è verificato un errore o se un oggetto `user` non è stato restituito dal middleware del passport. Se questo controllo è vero, creiamo un nuovo errore e lo passiamo al middleware successivo. Se il controllo è falso, chiamiamo il metodo di login che è esposto sull'oggetto della request. Questo metodo viene aggiunto automaticamente dal passport. Quando chiamiamo questo metodo, passiamo l'oggetto `user`, un oggetto `options` e una funzione di callback come argomenti. Nella funzione di callback, creiamo due token web JSON utilizzando la libreria `'jsonwebtoken'`. Per i JWT, includiamo l'ID e l'username dell'utente nel payload JWT e impostiamo la scadenza del token principale in cinque minuti e la scadenza del `refreshToken` in un giorno. Infine entrambi i token vengono archiviati nell'oggetto `response` chiamando il metodo `cookie`.

```
1 router.post('/login', async (req, res, next) => {
2   passport.authenticate('login', async (err, user, info) => {
3     try {
4       if (err || !user) {
5         const error = new Error('An Error occurred');
6         return next(error);
7       }
8       req.login(user, { session: false }, async (error) => {
9         if (error) return next(error);
10        const body = {
11          _id: user._id,
12          username: user.username
13        };
14        const token = jwt.sign({ user: body }, 'top_secret', {
15          expiresIn: 300 });
16        const refreshToken = jwt.sign({ user: body }, '
17          top_secret_refresh', { expiresIn: 86400 });
18        // store tokens in cookie
19        res.cookie('jwt', token);
20        res.cookie('refreshJwt', refreshToken);
21        // store tokens in memory
22        tokenList[refreshToken] = {
23          token,
24          refreshToken,
25          username: user.username,
26          _id: user._id
27        };
28        //Send back the token to the user
29        return res.status(200).json({ token, refreshToken });
30      } catch (error) {
31        return next(error);
32      }
33    }
34  })(req, res, next);
```


4.2 Registrazione

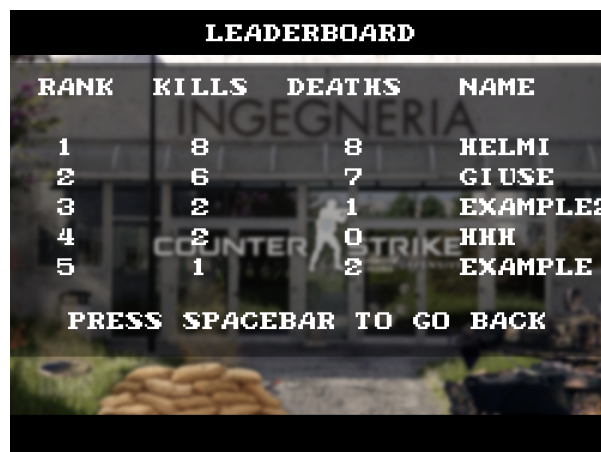
Per la **registrazione** il giocatore deve compilare il form presente in *signup.html*, una volta fatto il submit dei dati verrà eseguito un POST sulla route */signup*, definita anch'essa in *routes/main.js*, il middleware *passport.authenticate* che utilizza la configurazione di *signup* definita in *auth.js*

```
1
2 // handle user registration
3 passport.use('signup', new localStrategy({
4   usernameField: 'username',
5   passwordField: 'password',
6   passReqToCallback: true
7 }, async (req, username, password, done) => {
8   try {
9     const { username } = req.body;
10    const user = await UserModel.create({username,password});
11    return done(null, user);
12  } catch (error) {
13    done(error);
14  }
15 }));
```

I **token** vengono utilizzati per verificare costantemente l'identità degli utenti connessi ed evitare accessi non autorizzati, nel caso un utente non autorizzato tentasse di giocare riceverebbe un alert a riguardo e verrebbe reindirizzato alla pagina di login.

4.3 Classifica

L'utente può accedere alla **classifica** dal menu, è definita in *leaderboard.js* come una scena di Phaser. Nel *preload()* viene caricato il *bitmaptext* usato come font per le scritte, mentre nel *create()* viene effettuato un GET della route */scores*, in caso di success viene stampata la classifica:



RANK	KILLS	DEATHS	NAME
1	8	8	HELMİ
2	6	7	GIUSE
3	2	1	EXAMPLE2
4	2	0	HHH
5	1	2	EXAMPLE

PRESS SPACEBAR TO GO BACK

La route **scores** è definita in routes/secure.js, interroga il DB per ottenere i 5 utenti con più uccisioni in ordine decrescente.

```
1 router.get('/scores', asyncMiddleware(async (req, res, next) => {
  // find method on the UserModel to search for documents in the
  // database.
2 const users = await UserModel.find({}, 'username kills deaths -_id'
  ).sort({ kills: -1}).limit(10);
3 res.status(200).json(users);
4 }));
```

L'**update** delle uccisioni e delle morti dei giocatori viene eseguito in client/game.js ogni volta che il server manda i messaggi "death" ed "update-scores", entrambi inviati quando un player viene ucciso. Il client esegue due POST in base all'attributo da modificare:

```
1 this.socket.on('death', function(dead_player,id,name){
2   //... CODICE TAGLIATO
3   if(id == current_player.playerId) //updating leaderboard from
    killed player client
4   {
5     $.ajax({
6       type: 'POST',
7       url: '/submit-deaths',
8       data : {
9         username: name
10      },
11      //...CODICE TAGLIATO
12    })
13
14    this.socket.on('update-scores', function(id,name){
15      if(current_player.playerId==id) //updating leaderboard from
        killer client
16      {
17        //...CODICE TAGLIATO -> RICHIESTA AJAX UGUALE A QUELLA
          PRECEDENTE, CAMBIA SOLO 'URL'
18      }
19    })
20  }
```

Le due routes eseguono un incremento unitario degli attributi *deaths* e *kills* dell'utente con username passato dalla POST (req.body).

```
1 router.post('/submit-scores', asyncMiddleware(async (req, res, next)
  ) => {
2   const { username, } = req.body;
3   await UserModel.updateOne({ username }, {$inc: {kills: 1}}); //
    $inc == kills++
4   res.status(200).json({ status: 'ok' });
5 }));
6
7 router.post('/submit-deaths', asyncMiddleware(async (req, res, next)
  ) => {
8   const { username } = req.body;
9   await UserModel.updateOne({ username }, {$inc: {deaths: 1}}); //
    $inc == deaths++
10  res.status(200).json({ status: 'ok' });
11 }));
```