**ISTITUTO TECNICO INDUSTRIALE GALILEO FERRARIS**

# WWW.MIDGE.XYZ

**A.S 2017/2018**


**Relatore**

**Prof. Campagnaro Leonardo**

**Candidato**

**Bassan Davide**

Indice

## Introduzione

I videogiochi, fin dai loro albori, sono una tra le attività di intrattenimento preferite dai ragazzi: riescono a catturare subito l'attenzione e a coinvolgere i *players*, divertendoli grazie alla loro interattività.

Negli ultimi anni, grazie anche alla facilità di utilizzo di internet, le case produttrici di *videogame* si sono concentrate nello sviluppo *online* con la possibilità di giocare da remoto sfidando qualsiasi persona, e talvolta, da piattaforme diverse. Questa rivoluzione videoludica ha portato alla nascita dei *browser game* dove tutto quello che è richiesto per partecipare è una connessione alla rete ed un *browser* da cui poter accedervi.

Il mio progetto consiste in un videogioco *online* multi-giocatore e tutto ciò che è necessario per giocare è un amico e una connessione ad Internet.

# Parte I

### 1.1 Midge

*Midge* è la traduzione in inglese della parola moscerino; il videogioco ha questo nome perché identifica un essere piccolo, veloce e talvolta fastidioso.

Il giocatore si immedesima nei panni di un quadrato (il moscerino) che ruota in base allo spostamento del *mouse* e che si può muovere all'interno della mappa usando le lettere della tastiera W (sopra), A (sinistra), S (sotto) e D (destra).

Nell'angolo in alto a sinistra del moscerino è sito il pungiglione, identificato da un triangolino bianco, che deve colpire gli avversari nelle parti scoperte, quindi ovunque tranne nel pungiglione.

Lo scopo del gioco è pungere quanti più avversari possibili, evitando di essere colpiti dal pungiglione dei nemici.

### 1.2 Sviluppo

Il videogioco è basato su *NodeJS*, un *framework* del noto linguaggio di programmazione *JavaScript.*

Questa scelta è stata molto ponderata perché esistono innumerevoli librerie informatiche per il sopracitato linguaggio, ma *NodeJS* ha i vantaggi di essere un enorme ecosistema di pacchetti *open source* (utilizzabili gratuitamente) e di avere

un'enorme *community* di sviluppatori. Il *framework* mi ha fornito una solida base per costruire la comunicazione tra *client* e *server* la quale deve essere istantanea tra tutti i giocatori connessi per fornire un *gameplay* fluido.

Per la grafica ho utilizzato *Canvas*, un'estensione di *HTML* che permette il *rendering* di immagini *bitmap* in maniera dinamica attraverso un linguaggio di *scripting*.

La pagina è costruita con *HTML* e ne ho personalizzato lo stile agendo sui fogli di stile (*CSS)* che mi hanno permesso di modellarne la grafica rendendo l'utilizzo del sito più piacevole.

### 1.3 Client

Il *client* è il dispositivo che si connette all'indirizzo internet del sito web e utilizza l'applicativo.

Per migliorare la sicurezza e per ridurre la possibilità di *cheaters* (bari) ho preferito lasciare al *client* solo la visualizzazione dell'elaborazione del *server*, senza far eseguire alcun calcolo significativo al dispositivo dell'utilizzatore.

### 1.4 Server

Il *server* è ciò che fisicamente contiene il Midge. Nel mio caso il server svolge un ruolo fondamentale; infatti gestisce tutto il videogioco, a partire dal *login* per finire con la collisione e quindi l'eliminazione del giocatore sconfitto.

Questo calcolatore riceve i dati inviati dal *client,* ad esempio la pressione di un tasto da parte di un giocatore, ne elabora il contenuto e lo inoltra a tutti i dispositivi connessi per notificargli che c'è stato un aggiornamento nella mappa.

### 1.5 Database

Il *database* gestisce la persistenza dei dati che appartengono ai giocatori, alcuni esempi sono il nome utente, la *password* o il miglior punteggio registrato da quel giocatore.

La base di dati è creata tramite il *DBMS (Database Management System)* non relazionale *MongoDB*, ho deciso di utilizzarlo proprio perché è *NoSQL* (non relazionale) e quindi, nel caso dovesse contenere molti dati, l'estrazione risulterà molto più veloce rispetto ai comuni *DBMS* quali *MySQL* o *Microsoft SQL Server.*

Un altro vantaggio che ha favorito la mia scelta verso *MongoDB* è stato il fatto che è completamente *open source* e quindi non necessita l'acquisto di licenze per il suo utilizzo.

Interrogare questo tipo di *database* è semplice, ma non seguendo il linguaggio *SQL* vi sono delle differenze. Qui sono riportati alcuni esempi di *query* per *MongoDB:*

Inserimento:

```
database.tabella.insert({
 attributo: valore
})
```

Cancellazione di un solo *record*:

```
database.tabella.removeOne({
chiave: valore
})
```

Cancellazione di più *record*:

```
database.tabella.remove({
attributo: { condizione }
})
```

Selezione:

```
database.tabella.find({
attributo: valore
})
```

Aggiornamento:

```
database.tabella.update({
 { attributo: valore },
 {$set{
     attributo: valore
 }}
})
```

### 1.6 Comunicazione

Nei giochi multi-giocatore è richiesta, come spiegato in precedenza, una adeguata comunicazione tra *client* e *server*, che in *NodeJS* viene gestita dal modulo *Socket.io* permettendo l'invio e la ricezione di pacchetti (*socket*) in *real time.*

Ogni *socket* può avere 4 stati:

- in connessione
- in disconnessione
- connesso
- disconnesso

La trasmissione a sua volta può trovarsi in altre 4 differenti fasi:

- in apertura
- in chiusura
- aperta
- chiusa

All'avvio di una trasmissione si inizializza un *handshaking* (stretta di mano) tra *client* e *server* il quale, nel caso di successo, porterà al *client* le informazioni necessarie per proseguire la comunicazione in quel canale. La comunicazione quindi passa dallo stato di "in apertura" a quello di "aperta" e, fino a che la connessione sarà aperta, i *socket* potranno essere trasmessi sempre attraverso il metodo dell'*handshaking*.
Al momento della disconnessione di una delle due parti scatta il *timeout* che terminerà l'*handshaking* e tramuterà la fase della comunicazione da "in disconnessione" a "disconnesso".

Le funzioni che permettono lo scambio di *socket* tra il *client* e il *server* sono:

Invio:

```
socket.emit('nomePacchetto',{
  nome: valore,
  cognome: valore
  …
});
```

Ricezione:

```
socket.on('nomePacchetto',
 function('data'){
   nome = data.nome;
   cogn = data.cognome;
});
```

# Parte II

### 2.1 Authentication System

La registrazione e l'autenticazione permettono al *server* di memorizzare i dati di un giocatore.

Al momento della connessione alla piattaforma il *browser* genererà due *box* di *input* da compilare con il proprio nome utente e la *password*; inoltre due pulsanti "*Sign In"* e "*Sign Up"* che corrispondono all'*autenticazione* e alla *registrazione*.

I due *box* sono due *tag* di *input HTML*, il nome utente di tipo "*text*" e la *password* di tipo "*password*", entrambi hanno un *id* (codice identificativo univoco) che al momento della pressione del bottone "*Sign In"* o "*Sign Up"*, viene richiamato dalla funzione *Javascript,* la quale provvede ad inviare il pacchetto contenente il nome utente e la *password* al *server*.

*Form* per l'autenticazione:

```
<input type="text" id="signDiv-username" placeholder="Username" />
<input type="password" id="signDiv-password" placeholder="Password" />
<button type="button" id="signDiv-signIn">Sign In</button>
<button type="button"id="signDiv-signUp">Sign Up</button>
```

Invio del *socket* di autenticazione al *server :*

```
signDivSignIn.onclick = function(){
  socket.emit('signIn',{
    username: signDivUsername.value,
    password: signDivPassword.value
  });
}
```

Invio del *socket* di registrazione al *server:*

```
signDivSingUp.onclick = function(){
  socket.emit('signUp',{
    username: signDivUsername.value,
    password: signDivPassword.value
  });
}
```

Una volta emessi i *socket* il *server* li riceve e ne elabora il contenuto controllando con una *query* se lo *username* inserito corrisponde effettivamente alla *password* digitata o nel caso di *Sign up* provvede a inserire un nuovo utente.

Funzione per il controllo della *password* inserita:

```
var isValidPassword = function(data,cb){
  db.account.find({
    username: data.username,
    password: data.password
  },
  function(err,res){
    if(res.length > 0)
      cb(true);
    else
      cb(false);
  });
}
```

Ricezione ed inoltro del *socket* di autenticazione:

```
socket.on('signIn', function(data){
  isValidPassword(data,function(res){
    if(res){
      [...]
      socket.emit('signInResponse',{success:true});
    }
    else
      socket.emit('signInResponse',{success:false});
  });
});
```

Controllo del nome utente:

```
var isUsernameTaken = function(data,cb){
  db.account.find({
  username: data.username
  },
  function(err,res){
    if(res.length > 0)
      cb(true);
    else
      cb(false);
  });
}
```

Aggiunta di un nuovo giocatore nel *database:*

```
var addUser = function(data,cb){
  db.account.insert({
    username: data.username,
    password: data.password
  },
  function(err){
    cb();
  });
}
```

Ricezione ed inoltro del *socket* di registrazione:

```
socket.on('signUp', function(data){
  isUsernameTaken(data,function(res){
    if(res)
      socket.emit('signUpResponse',{success:false});
    else
      addUser(data,function(){
        Player.onConnect(socket);
        socket.emit('signUpResponse',{success:true});
      });
  });
});
```

Se la registrazione o l'accesso hanno avuto esito positivo in contemporanea si avviano due procedure: la creazione di un nuovo giocatore e la visualizzazione della schermata di gioco.

### 2.2 Player Creation

Ogni giocatore all'interno del server è definito come un «oggetto»; gli attributi più rilevanti sono:

- id: identificatore univoco
- x: posizione X
- y: posizione Y
- rotation: grado rotazione
- pressingRight: tasto D
- pressingLeft: tasto A
- pressingUp: tasto W
- pressingDown: tasto S
- spdX: velocità X
- spdY: velocita Y
- score: punteggio
- lose

Il *server*, dopo la corretta autenticazione, provvede ad inviare al *client* una copia dell'«oggetto» *player*, che lo disegnerà a schermo.

L'«*oggetto*» *Player* ha a disposizione molteplici «*metodi*» che permettono il funzionamento del gioco, richiamati dentro ad «*update*», una funzione ricorsiva che aggiorna i movimenti dei giocatori e le collisioni.

*Update:*

```
self.update = function(){
 self.updatePosition();
}

[...]

var super_update = self.update;
  self.update = function(){
    self.updateSpd();
    self.checkCollision();
  super_update();
}
```

Per risalire ad ogni giocatore il sistema memorizza il codice identificativo della trasmissione che il *server* ha con ogni *client* e crea un vettore (lista) di giocatori.
La sintassi per richiamare un giocatore è quindi:

```
var giocatore = Player(socket.id);
giocatore.score += 1;
```

### 2.3 Chat

Il sistema di messaggistica istantanea in un videogioco *online* è molto importante poiché riesce a coinvolgere i giocatori anche nei "tempi morti" presenti durante una partita.

La *chat* consiste in un *form* contenente un solo *tag* di *input*, il giocatore digita ciò che vuole condividere con gli altri giocatori e alla pressione del tasto «invio» viene eseguito un controllo sul contenuto. In caso di esito positivo si attiva una procedura che invia il *socket* contenente il messaggio al *server* che lo inoltra a tutti i dispositivi connessi.

*Chat box:*

```
<div class="contain" id="gameDiv" style="display: none;">
  <canvas id="ctx"></canvas>
  <div class="overlay chat">
    <div class="messages" id="chat-text">
      <p class="msg">Server: Welcome!</p>
    </div>
    <form id="chat-form">
      <input class="typesend" id="chat-input" type="text"></input>
    </form>
  </div>
</div>
```

Nella *chat* è integrato (solo per gli *utenti admin*) la possibilità di eseguire interrogazioni al *database;* infatti se il messaggio di un utente inizia con il carattere «/» il *server* lo interpreta come una richiesta alla sua base di dati.

In *Javascript* «*preventDefault()*» è un metodo utilizzabile sugli eventi, adibito a prevenire il *refresh* della pagina alla pressione del tasto «invio» e quindi che l'utente venga rimandato alla schermata di login.

Invio del messaggio al *server:*

```
chatForm.onsubmit = function(event){
  event.preventDefault();
  if(chatInput.value[0] === '/')
    socket.emit('evalServer', chatInput.value.slice(1));
  else
    socket.emit('sendMsgToServer', chatInput.value);
  chatInput.value = '';
}
```

Una volta ricevuto il pacchetto, il *server* ne analizza il contenuto trasmettendolo a tutti i giocatori connessi che lo visualizzeranno a schermo.

Trasmissione del *socket:*

```
socket.on('sendMsgToServer', function(data){
  var playerName = ("" + socket.id).slice(2,7);
  for(var i in SOCKET_LIST){
    SOCKET_LIST[i].emit('addToChat',playerName + ': ' + data);
  }
});

socket.on('evalServer', function(data){
```

```
  var res = eval(data);
  socket.emit('evalAnswer', res);
});
```

Ricezione e visualizzazione del messaggio:

```
var chatText = document.getElementById('chat-text');
var chatInput = document.getElementById('chat-input');
var chatForm = document.getElementById('chat-form');

socket.on('addToChat', function(data){
  chatText.innerHTML += '<p [...]>' + data + '</p>';
    chatText.scrollTop = chatText.scrollHeight;
  });

socket.on('evalAnswer', function(data){
 console.log(data);
});
```

### 2.4 Movement Translation

Come è descritto nella introduzione (pag. 4), i giocatori possono muovere il proprio *avatar* in orizzontale e in verticale con i tasti «W»,«A»,«S», «D» e ruotarlo rispetto al loro centro con il mouse.

Anche in questo caso viene analizzato, grazie alle funzioni *Javascript onkeydown* e *onkeyup,* quando un giocatore preme un tasto e quando lo rilascia.
Queste funzioni attendono la pressione di un tasto e, quando questo viene premuto o rilasciato, ritornano un valore che lo identifica tramite la tabella "*unicode*".
Quando il codice restituito corrisponde ad uno dei tasti predisposti per il movimento del proprio "*Midge*" il *client* invia un *socket* contentente l'azione (*up, left, right, down*) e lo stato dell'azione (premuto, rilasciato).

Pressione di un tasto:

```
document.onkeydown = function(event){
  if(event.keyCode === 68)
    socket.emit('keyPress',{inputId:'right',state: true});
  if(event.keyCode === 83)
    socket.emit('keyPress',{inputId:'down',state: true});
  if(event.keyCode === 65)
    socket.emit('keyPress',{inputId:'left',state: true});
  if(event.keyCode === 87)
    socket.emit('keyPress',{inputId:'up',state: true});
}
```

Rilascio di un tasto:

```
document.onkeyup = function(event){
  if(event.keyCode === 68)
    socket.emit('keyPress',{inputId:'right',state: false});
  if(event.keyCode === 83)
    socket.emit('keyPress',{inputId:'down',state: false});
  if(event.keyCode === 65)
    socket.emit('keyPress',{inputId:'left',state: false});
  if(event.keyCode === 87)
    socket.emit('keyPress',{inputId:'up',state: false});
}
```

A questo punto il *server* lo analizza e si occupa di aggiornare la velocità del giocatore corrispondente; infatti all'aumentare del tempo in cui un tasto rimane premuto maggiore sarà la velocità di spostamento del giocatore.

Ricezione del *socket*:

```
socket.on('keyPress',function(data){
  if(data.inputId === 'left')
    player.pressingLeft = data.state;
  if(data.inputId === 'right')
    data.pressingRight = data.state;
  if(data.inputId === 'up')
    data.pressingUp = data.state;
  if(data.inputId === 'down')
    data.pressingDown = data.state;
 });
```

Aggiornamento della velocità:

```
self.updateSpd = function(){
  if(self.pressingRight)
    self.spdX += self.maxSpd;
  else{
    if(self.pressingLeft)
      self.spdX -= self.maxSpd;
    else
      self.spdX = 0;
  }
  if(self.pressingUp)
    self.spdY -= self.maxSpd;
  else{
    if(self.pressingDown)
```

```
      self.spdY += self.maxSpd;
    else self.spdY = 0;
  }
}
```

Aggiornamento della posizione:

```
self.updatePosition = function(){
  self.x += self.spdX;
  self.y += self.spdY;
}
```

Per quanto riguarda la rotazione il procedimento è leggermente diverso. Il *client* ad ogni movimento del *mouse* invia il punto centrale dello schermo e la posizione, quindi il *server* ne calcola l'angolo con la funzione della libreria *Math* «atan2» e ne imposta l'angolo di rotazione.

Invio delle coordinate del *mouse* e del punto centrale dello schermo:

```
document.onmousemove = function(event){
  socket.emit('mouseDown',{xw: c.width/2, yw:c.height/2,
                           xm:event.x, ym:event.y
  });
}
```

Calcolo angolo e ridefinizione della rotazione:

```
socket.on("mouseDown",function(data){
  player.rotation = Math.atan2(data.yw - data.ym, data.xw - data.xm);
});
```

### 2.5 Drawing

La visualizzazione del gioco è la fase che concretizza l'elaborato del *server*, e viene eseguita dal *client* tramite l'estensione di *HTML Canvas.*
Per rendere più intuitivo il gioco, il *midge* controllato da ogni utente è fisso al centro dello schermo e apparentemente non si muove ma, in realtà, muove tutta la schermata; questo accade anche per la rotazione, per la quale non viene ruotato solo il singolo elemento ma tutta la schermata.
Al completamento della visualizzazione di un elemento sullo schermo viene reimpostata la corretta rotazione della mappa e si procede con l'elemento successivo.

Visualizzazione del gioco:

```
self.draw = function(){
  var x = self.x - Player.list[selfId].x + c.width/2;
  var y = self.y - Player.list[selfId].x + c.height/2;
  ctx.save();
  ctx.beginPath();
  ctx.translate(x,y);
  ctx.rotate(self.rotation);
  ctx.fillRect(50/-2,50/-2,50,50);
  ctx.fillStyle='white';
  ctx.strokeStyle="white";
  ctx.strokeRect(47/-2,47/-2,47,47);
  ctx.moveTo(0,-25); ctx.lineTo(-25,-25);
  ctx.lineTo(-25,0); ctx.lineTo(0,-25);
  ctx.fill();
  ctx.restore();
}
```

### 2.6 Collision Detection

La fase conclusiva del gioco avviene quando un giocatore riesce a toccare un altro giocatore con il suo pungiglione. Per calcolare quando ciò accade ho usato la libreria fornita all'interno della *community* di *NodeJS* denominata *"SAT"* (*Separating Axis Theorem*), (spiegazione del suo funzionamento al video https://www.youtube.com/watch?v=Ap5eBYKlGDo).
Grazie a questa libreria tutto il necessario per controllare lo scontro sono le quattro coordinate del moscerino.
Quando avviene lo scontro, quindi, il *server* controlla se è avvenuto tra un pungiglione o tra un angolo normale e ne valuta la risposta, cancellando il giocatore nel caso abbia perso, aumentandone il punteggio in caso di vittoria oppure simulando un rimbalzo tra i due interessati.

Questo processo avviene tra tutti i giocatori connessi.

Controllo della collisione:

```
var collide = sat.testPolygonPolygon(slf,pl);

if(collide){
  var tr_to_tr = sat.testPolygonPolygon(tr_slf,tr_pl);
  if(tr_to_tr){
    self.x += Math.random() * (70 - (-150)) + (-150);
    self.y += Math.random() * (70 - (-150)) + (-150);
    p.x += Math.random() * (70 - (-150)) + (-150);
```

```
    p.y += Math.random() * (70 - (-150)) + (-150);
  }
  else{
    var slf_to_pl = sat.testPolygonPolygon(tr_slf,pl);
     if(slf_to_pl){
       self.score += 1;
       p.lose = true;
     }
     else{
       var pl_to_slf = sat.testPolygonPolygon(tr_pl,slf);
       if(pl_to_slf){
         p.score += 1;
         self.lose = true;
       }
       else{
         self.x += Math.random() * (70 - (-150)) + (-150);
         self.y += Math.random() * (70 - (-150)) + (-150);
         p.x += Math.random() * (70 - (-150)) + (-150);
         p.y += Math.random() * (70 - (-150)) + (-150);
       }
     }
  }
}
```

### 2.7 Aggiornamenti Futuri

Il gioco così per com'è ora non è commercializzabile perché necessita di alcuni cambiamenti e alcune migliorie. Qui ne è riportata una lista dove sono citati i maggiori miglioramenti che devono essere fatti.

- Criptazione e salvataggio dei dati nel D*ataBase*
- Ottimizzazione della grafica 2D e layout responsive
- Miglioramento nella fisica delle collisioni
- Aggiunta di una classifica globale

## Fonti bibliografiche e sitografia


NodeJS Docs
https://nodejs.org/it/docs/

W3Schools
https://www.w3schools.com/

MongoDB Docs
https://docs.mongodb.com/manual/

HTML.it
http://www.html.it/

RipMat
http://www.ripmat.it/

dyn4j
http://www.dyn4j.org/2010/01/sat/

npmjs
https://www.npmjs.com/package/sat/

heroku
https://www.heroku.com/