Design Document for Project 5

Team 7:

Tyler Lyczak - tlyczak2@uic.edu

Alex Chomiak - achomi2@uic.edu

Derek Ochal - dochal2@uic.edu

Thor Hawist - thawis2@uic.edu

Quickstart Guide:

In order to start the server, you need to be in the root directory of the folder. Then open up a terminal and run the command:

- node index.js

After you start the server, you need to open a browser and type in:

- http://localhost:9999

This will make a connection with the server. It will request you to type in your name. After that, you need to wait until 3 other players to join to play the game. You can make multiple tabs as needed.

Part 1: Server

To start the server, we need to run *index.js* which has a single line

```
require('./sockets/main')
```

This serves the files to the clients when they connect, giving the the js files they need to run and play the game.

For the server to work, we use *server.js* which uses node, socket.io, and express to run the server. We get the from the node_modules folder. We call them by using

```
const express = require('express')
const socketio = require('socket.io')
```

From this, we send the files to any client that joins the web server with the function

```
app.use(express.static(__dirname + '/public'))
app.get('/',(req,res) => {res.sendFile(path.join('/index.html'))})
```

This gives the client the html page and the js files necessary to communicate with the server. The main server file that runs the logic of the game is *main.js*. In here, we define constants such as

```
const Player = require('./classes/Player')
const PlayerConfig = require('./classes/PlayerConfig')
const PlayerData = require('./classes/PlayerData'
const Bullet = require('./classes/Bullet')
const Pillar = require('./classes/Pillar')
```

to pull each of the needed classes that are needed for the game to run. In *PlayerData.js*, it has variables that track what x and y coordinate, the id, the name, the health, and other various variables needed to track each player in the game. In *PlayerConfig.js*, we have the speed and angular speed of the player. And in *Player.js*, we have both *PlayerData* and *PlayerConfig* stored along with their id. For *Bullet.js* and *Pillar.js*, they store necessary information, such as the x and y coordinate. *Bullet.js* also contains more information, such as who shoot the bullet and the vector of the bullet travel. Moving on in *Main.js*, we have a for-loop that places pillars around the map. It gets random x and y coordinates in the parameter of the map. Then pushes the pillar to an array of them.

```
let x = Math.floor(Math.random() * gameSettings.mapWidth)
let y =  Math.floor(Math.random() * gameSettings.mapHeight)
newPillar = new Pillar(x,y)
pillars.push(newPillar)
```

Also in *Main.js*, we use `setInterval` to update each bullet location. In here, we call a different JavaScript file, *collision.js*, which we use to tell if a bullet collides with a player. The first function in *collision.js* is `playerCollision` which we use to stop players from overlapping each other.

```
return new Promise((resolve, reject) => {
players.forEach((p) => {
       if(p.id === playerID) return;
              if(rectCollideswithOtherRect({x: newX, y:
              newY},tankSize,p.playerData,tankSize)) {
                     resolve(p)
       reject()
```

As we see, it calls another function, `rectCollideswithOtherRect`, to check if the two players are colliding. If they are, it will reject the new x and y coordinates the player if moving to and keep them where they were originally.

Also in *collision.js*, we have `bulletCollidingWithPlayer`, which does basically the same thing as `rectCollideswithOtherRect` but with a bullet. But instead of rejecting the players new x and y coordinates, it will subtract health from the player.

Back in *main.js* under the `setInterval`, we have

```
let player = bulletCollidingWithPlayer(bullet,players,tankSize)
if( player !== null) {
      player.playerData.health -= 10
      if(player.playerData.health <= 0) {
            player.playerData.death = true
            if(player.id !== bullet.ownerID)  {
                  bullet.owner.score += 50
            }
      else {
            if(player.id !== bullet.ownerID) {
                  player.playerData.score -= 10
                  bullet.owner.score += 10
```

In these statements, we do things according to what the bullet is. If the player shot a player, they lose points and maybe will die depending on their health. If the player shoots another player and they die, they get 50 points. If they player shoots a player, they get 10 points and if themselves got shot, they lose 10 points.

If the bullet collides with a wall or pillar, it will call `rectCollision` to check first, then it will change the direction of the bullet with:

```
bullet.xVec = -bullet.xVec
bullet.yVec = -bullet.yVec
```

When the server detects a connection from a client, it will run `io.on('connect', (socket)` which will run various of functions that sets up a new player with the server. First, if the socket of the client is connected, it will run `socket.on('init', async (data)` which will run these commands:

```
let playerConfig = new PlayerConfig()
x = Math.random() * gameSettings.mapWidth;
y = Math.random() * gameSettings.mapHeight
let playerData = new PlayerData(socket.id,data.username,x,y)
player = new Player(socket.id,playerConfig,playerData)
```

These will give the player a random x and y coordinate on the map to start at and gives them their own settings to keep track of. The server will also do a `socket.emit` to send the client these settings:

```
        socket.emit('initialized', {
                bullets,
                players: playersData,
                playerX : playerData.x
                playerY : playerData.y,
                playerAngle : playerData.angle,
                playerTurretAngle : playerData.turretAngle,
                gameSettings,
                pillars
        })
```
This gives the player all the variables they need to keep track of and send back to the server to keep the server, other clients, and game running.

Later on, there is a `setInterval` that will run at the tickrate of the server. In this `setInterval`, the server will emit 'tock', which will send the player these variables:

```
        setInterval( () => {
                socket.emit('tock',{
                        Bullets,
                        players: playersData,
                        playerX: player.playerData.x,
                        playerY: player.playerData.y,
                        death: player.playerData.death,
                        score: player.playerData.score
                })
        }, gameSettings.tickRate)
```
Since the server does all the work of collision and player movement, it needs to keep each player updated on what is happening, so it will send this every tickrate to keep each player in the game updated on what is happening currently.

The opposite of this is `socket.on('tick',(data))`, which is what the client sends to the server every tickrate to update what is going on for each client. In here, it will check if the player made a valid move, what bullets it shot, there health, if they died, if they are colliding with anything, and more. The server will calculate all of this and send it back to the client with `socket.emit('tock')`, which will then in return update the client on what they did, according to the server. These two run in unison to allow for the server to handle everything the clients are doing to make the game fair for the client not to handle many processes going on.

Part 2: Client

When the client connects to the server, its first served the *index.html* in the public folder, then along with it is server everything in the public folder. The client is greeted with a login modal, which ask for a name. When the user inputs a name, *client.js* takes over with a JQuery function that listens for the submit button.

```
$('.name-form').submit((event)=>{
        name = document.querySelector('#name-input').value;
        init()
}
```

After the user click submit, it runs `init()`, which creates a connection to the server

```
function init() {
        c = new Connection(io.connect('http://localhost:9999'))
        draw()
}
```

When it creates a connection with the server, it passes through *connection.js* to initialize the connection. In *connection.js*, it will run `socket.emit('init')` to send the name of the user to the server. It will then listen for `socket.on('initialized', (data))` to get:

```
socket.on('initialized', (data) => {
        player = {
                x: data.playerX,
                y: data.playerY,
                angle: data.playerAngle,
                turretAngle: data.playerTurretAngle
        }
        settings = data.gameSettings
        players = data.players
        pillars = data.pillars
}
```

This will set all the players needed variables that it will send to the server when the server sends a “tock” with `socket.on('tock', (data)`, which will update the players variable on what is happening in the game.

```
socket.on('tock', (data) => {
        player = {
                x: data.playerX,
                y: data.playerY,
                angle: data.playerAngle,
                turretAngle: data.playerTurretAngle
        }
        players = data.players
        bullets = data.bullets
        death = data.death
        socket.emit('tick', {
                left,
                right,
                forwards,
                backwards,
                turretAngle,
                Clicked
        })
```

In response to the server sending it data, the client will send back what is happening on its side. The variables for the different directions are based on what the client is pressing on there keyboard and where there mouse is pointing at to create an angle.

Back in `init()`, it calls the function `draw()`, which will call *canvas.js* to start drawing in the canvas of the html page. In *canvas.js*, it will transform the canvas to display what the server is telling it to. First, it clears the canvas:

```
context.clearRect(0,0,settings.mapWidth + canvas.width, settings.mapHeight +
canvas.height)
```

Then it will set the “camera” on the client, which just makes it zoomed in:

```
const camX = -player.x + canvas.width/2 - tank.width/2
const camY = -player.y + canvas.height/2 - tank.height/2
```

Then it will draw the border and pillars

```
context.strokeRect(0,0, settings.mapWidth + tank.width, settings.mapHeight +
tank.height)
pillars.forEach((pill) => {
        context.drawImage(pillar,pill.x,pill.y)
})
```

Then , it will draw each player from the player array given from the server:

```
        players.forEach((p) => {
                context.drawImage(tank,-tank.width/2,-tank.height/2)
                context.drawImage(turret,-tank.width/2,-tank.height/2)
```

Then it will finally draw each bullet:

```
        bullets.forEach((bullet) => {
                context.beginPath()
                context.arc(bullet.x,bullet.y,5,0,Math.PI *2)
                context.fill()
        }
```

It will then call the function `requestAnimationFrame(draw)` to redraw the whole canvas each frame so the client will always have an up-to-date game.

Also in *canvas.js*, it has variables and functions that will check when a key is being pressed down to move the player and when the mouse is being click to shoot a bullet:

```
        document.onkeydown = checkKeyDown;
        document.onkeyup = checkKeyUp
        document.onmousedown = () => {clicked = true}
        document.onmouseup = () => {clicked = false}
```

These will update the variable of forwards, backwards, left, right, and clicked, which will be sent to the server to update the players movement.

If the client dies in the game, the socket will be closed in *connection.js* in:

```
        if(death) {
                socket.close()
                $('#loginModal').modal('show')
        }
```

After the client dies, the login modal will be shown again, giving the player an option to input a name and play again.

Summary:

When both the client and server establish a connection with Socket.io, they will constantly send each other data of what is happening. The client will be sending what movement it want to do and when it wants to shoot. The server will respond with that movement, that bullet, and the movement and bullets of other players. The client will then redraw what the server sends it to so the client can see what is going on in real time. These two work in unison to keep a steady, fast past action game so the user can have an enjoyable experience.