Lance.gg Pong Game Tutorial

http://docs.lance.gg/develop/tutorial-MyFirstGame.html

Brooke Hedrick

brooke.hedrick@gmail.com

Summer 2018

Setting up the Environment

The creation of a new game starts by copying boilerplate code.

```
Install nodejs from node-v8.11.3-x64.msi

Unzip lancegame-master-with-modules-v1.zip to
C:\Users\<username>
```

You now have the basic directory structure of a game. Look around. The boilerplate includes an index.html file, which will be served to the clients, and a main.js file, which is the entry point of the node.js server. The game code is inside the src directory, divided into sub-directories client, server, and common.

Take a look at webpack.config.js which shows how the game is packaged, and .babelrc shows how the game loads the lance library.

Original code location

```
git clone https://github.com/lance-gg/lancegame.git
```

Step 1: Create the Game Object Classes

There are two kinds of objects in every Pong game: the paddle and the ball. These files extend the DynamicObject class, but are quite simple. The boilerplate includes a sample game object class, in the file src/common/PlayerAvatar.js

Create the following two classes in the src/common directory:

* src/common/Paddle.js

The Paddle class is similar to the sample PlayerAvatar.js object, except that it is called Paddle and it asks the renderer to create a sprite.

```
'use strict';

import DynamicObject from
'lance/serialize/DynamicObject';

export default class Paddle extends DynamicObject {

    constructor(gameEngine, options, props) {
        super(gameEngine, options, props);
        if (props && props.playerId)
            this.playerId = props.playerId;
        this.class = Paddle;
    }

    onAddToWorld(gameEngine) {
        if (gameEngine.renderer) {
            gameEngine.renderer.addSprite(this,
'paddle');
        }
    }
}
```

* src/common/Ball.js

The Ball class is only slightly more complicated than the Paddle class. It adds two getters, which define "bending" properties. The bending properties below indicate that the client object's position should gradually bend towards the server object's position at a rate of 80% each time the server sends position updates. The client object's velocity should not bend at all, because the ball's velocity can change suddenly as it hits a wall or a paddle. We also give the Ball an initial velocity when it is created.

```
'use strict';

import DynamicObject from
'lance/serialize/DynamicObject';

export default class Ball extends DynamicObject {

    get bendingMultiple() { return 0.8; }
    get bendingVelocityMultiple() { return 0; }

    constructor(gameEngine, options, props) {
        super(gameEngine, options, props);
        this.class = Ball;
        this.velocity.set(2, 2);
    }

    onAddToWorld(gameEngine) {
        if (gameEngine.renderer) {
            gameEngine.renderer.addSprite(this,
'ball');
        }
    }
}
```

Step 2: Implement the MyGameEngine class

The game engine class runs on both the server and the client, and executes the game's logic. The client runs the game engine to predict what will happen, but the server execution is the true game progress, overriding what the clients might have predicted.

For Pong, we will need to bounce the ball around the board, and check if it hit a paddle. We will also need to respond to the user's up/down inputs.

^ src/common/MyGameEngine.js

The MyGameEngine class implements the actual logic of the game. First add the objects we created, and some constants, at the top of the file:

```
import TwoVector from 'lance/serialize/TwoVector';
import Paddle from './Paddle';
import Ball from './Ball';
const PADDING = 20;
const WIDTH = 400;
const HEIGHT = 400;
const PADDLE_WIDTH = 10;
const PADDLE_HEIGHT = 50;
```

- **start(): registers the game logic to run as a post-step function, and keep references to the game objects. Modify the start method to match the following:**

```
start() {

    super.start();

    this.on('postStep', () => {
this.postStepHandleBall(); });
    this.on('objectAdded', (object) => {
        if (object.class === Ball) {
            this.ball = object;
        } else if (object.playerId === 1) {
            this.paddle1 = object;
        } else if (object.playerId === 2) {
            this.paddle2 = object;
        }
    });
}
```

- **registerClasses: register all the game objects on the serializer. Add this function:**

```
registerClasses(serializer) {
    serializer.registerClass(Paddle);
    serializer.registerClass(Ball);
}
```

- **processInput: handle user inputs by moving the paddle up or down. Modify the processInput method to match the following:**

```
processInput(inputData, playerId) {

    super.processInput(inputData, playerId);

    // get the player paddle tied to the player
socket
    let playerPaddle =
this.world.queryObject({ playerId });
    if (playerPaddle) {
        if (inputData.input === 'up') {
            playerPaddle.position.y -= 5;
        } else if (inputData.input === 'down') {
            playerPaddle.position.y += 5;
        }
    }
}
```

- **initGame: create two paddles, a ball, and add these objects to the game world. This method will be called only on the server. Add the following initGame() method:**

```
initGame() {

    // create the paddle objects
    this.addObjectToWorld(new Paddle(this, null,
{ position: new TwoVector(PADDING, 0), playerId:
1 }));
    this.addObjectToWorld(new Paddle(this, null,
{ position: new TwoVector(WIDTH - PADDING, 0),
playerId: 2 }));
    this.addObjectToWorld(new Ball(this, null,
{ position: new TwoVector(WIDTH /2, HEIGHT /
2) }));
}
```

- postStepHandleBall: this method is executed after the ball has moved. It contains all the core pong game logic: it checks if the ball has hit any wall, or any paddle, and decides if a player has scored.

```
postStepHandleBall() {
    if (!this.ball)
        return;

    // CHECK LEFT EDGE:
    if (this.ball.position.x <= PADDING + PADDLE_WIDTH &&
        this.ball.position.y >= this.paddle1.y &&
this.ball.position.y <= this.paddle1.position.y +
PADDLE_HEIGHT &&
        this.ball.velocity.x < 0) {

        // ball moving left hit player 1 paddle
        this.ball.velocity.x *= -1;
        this.ball.position.x = PADDING + PADDLE_WIDTH +
1;
    } else if (this.ball.position.x <= 0) {

        // ball hit left wall
        this.ball.velocity.x *= -1;
        this.ball.position.x = 0;
        console.log(`player 2 scored`);
    }
```

** Continues on next page **

```
    // CHECK RIGHT EDGE:
    if (this.ball.position.x >= WIDTH - PADDING -
PADDLE_WIDTH &&
        this.ball.position.y >= this.paddle2.position.y
&& this.ball.position.y <= this.paddle2.position.y +
PADDLE_HEIGHT &&
        this.ball.velocity.x > 0) {

        // ball moving right hits player 2 paddle
        this.ball.velocity.x *= -1;
        this.ball.position.x = WIDTH - PADDING -
PADDLE_WIDTH - 1;
    } else if (this.ball.position.x >= WIDTH ) {

        // ball hit right wall
        this.ball.velocity.x *= -1;
        this.ball.position.x = WIDTH - 1;
        console.log(`player 1 scored`);
    }

    // ball hits top
    if (this.ball.position.y <= 0) {
        this.ball.position.y = 1;
        this.ball.velocity.y *= -1;
    } else if (this.ball.position.y >= HEIGHT) {
        // ball hits bottom
        this.ball.position.y = HEIGHT - 1;
        this.ball.velocity.y *= -1;
    }

}
```

Step 3: Extend the MyServerEngine Class

The server engine will initialize the game engine when the game is started, and handle player connections and "disconnections".

^ src/server/MyServerEngine.js

```
'use strict';

import ServerEngine from 'lance/ServerEngine';
import PlayerAvatar from '../common/PlayerAvatar';

export default class MyServerEngine extends ServerEngine
{

    constructor(io, gameEngine, inputOptions) {
        super(io, gameEngine, inputOptions);
    }

    start() {
        super.start();

        this.gameEngine.initGame();

        this.players = {
            player1: null,
            player2: null
        };
    }
```

** Continued on next page **

```
    onPlayerConnected(socket) {
        super.onPlayerConnected(socket);

        // attach newly connected player an available
paddle
        if (this.players.player1 === null) {
            this.players.player1 = socket.id;
            this.gameEngine.paddle1.playerId =
socket.playerId;
        } else if (this.players.player2 === null) {
            this.players.player2 = socket.id;
            this.gameEngine.paddle2.playerId =
socket.playerId;
        }
    }

    onPlayerDisconnected(socketId, playerId) {
        super.onPlayerDisconnected(socketId, playerId);

        if (this.players.player1 == socketId) {
            console.log('Player 1 disconnected');
            this.players.player1 = null;
        } else if (this.players.player2 == socketId) {
            console.log('Player 2 disconnected');
            this.players.player2 = null;
        }
    }
}
```

Step 4: the Client Code

The client-side code must implement a renderer, and a client engine.
The renderer, in our case, will update HTML elements created for each paddle and the ball:

^ src/client/MyRenderer.js

Change the draw() method and add the addSprite() methods as shown here:

```
draw(t, dt) {
    super.draw(t, dt);

    for (let objId of Object.keys(this.sprites)) {
        if (this.sprites[objId].el) {
            this.sprites[objId].el.style.top =
this.gameEngine.world.objects[objId].position.y + 'px';
            this.sprites[objId].el.style.left =
this.gameEngine.world.objects[objId].position.x + 'px';
        }
    }
}

addSprite(obj, objName) {
    if (objName === 'paddle') objName += obj.playerId;
    this.sprites[obj.id] = {
        el: document.querySelector('.' + objName)
    };
}
```

Step 4: the Client Visuals

The client visuals code are simple HTML so we don't discuss them in detail. In file index.html, change the <body> DOM element contain just the following snippet.

```
<div style="width: 400px; height: 400px;
background: black">
    <div
style="position:absolute;width:10px;height:50px;bac
kground:white" class="paddle1"></div>
    <div
style="position:absolute;width:10px;height:50px;bac
kground:white" class="paddle2"></div>
    <div style="position:absolute;width:5px;
height:5px;background:white" class="ball"></div>
</div>
```

Step 5: Running the Game

Once everything has been put together the end result should look like the pong branch of the repository.

To run the game you must first build the JavaScript bundle. The npm install command above already did this for you, but we changed the code, so you must rebuild by executing:

```
Open a command prompt
C:
cd \Users\<username>\lancegame-master
npm run build
```

To run the game, type:

```
npm start
```

Open two browser windows and point them to the local host. The URL is http://127.0.0.1:3000/on windows, and http://localhost:3000/ on a Mac.

NOTE: If you prefer to get a clean working copy, you can run:

```
git clone https://github.com/lance-gg/lancegame.git
pong
cd pong
git checkout pong
npm install
```