Match3 game with PixiJS

Preview demo: https://gamedevland.github.io/match3/

这次我将尝试使用英语来写我的README，虽然这样会比较难理解，不过这是一次锻炼的机会，可惜我英语也不太好，所以如果有什么不对的地方，还请见谅。
Creating Match3 game with PixiJS
In this article, we will create a match3 game using PIXI.

Video version

Additional materials
Complete source code
Preview demo
Before starting the development, we need to perform 2 steps:

1️⃣ Download our PIXI project template. You can start working on the game right now or check out the tutorial on how to create a PIXI project template.

2️⃣ Download the assets pack for our game. Assets provided by the great website kenney.nl

1. Creating the tiles board
   Let's set the first task to create the board with tiles.

1.1. Creating a single field
The first step is to create a board field class and render a single field sprite on the screen.

Create the game/Field.js class:

```
import { App } from "../system/App";

export class Field {
constructor(row, col) {
this.row = row;
this.col = col;
```

```
        this.sprite = App.sprite("field");
        this.sprite.x = this.position.x;
        this.sprite.y = this.position.y;
        this.sprite.anchor.set(0.5);
    }

    get position() {
        return {
            x: this.col * this.sprite.width,
            y: this.row * this.sprite.height
        };
    }
```

}

A field is located in a certain column and in a certain row on the board.

Info

The row and col properties of the Field class define the row and column.
The position getter determines the position of the field on the screen, depending on the position on the board and taking into account the size of the field sprite.
And now we can render a single field on the screen in the Game class:

```
import { Field } from "./Field";
// ...
export class Game {
constructor() {
// ...
const field = new Field(1, 1);
this.container.addChild(field.sprite);
}
}
```

1.2. Creating a fields grid

We have a class for the board field and now we can create the board itself, consisting of a number of fields. Let's create a class src/scripts/game/Board.js:

```
import * as PIXI from "pixi.js";
import { App } from "../system/App";
import { Field } from "./Field";

export class Board {
constructor() {
```

```
this.container = new PIXI.Container();
this.fields = [];
this.rows = App.config.board.rows;
this.cols = App.config.board.cols;
this.create();
}

  create() {
  }

}
```
We will add all elements of the board (fields and tiles) to the board container. And the board container is then added to the scene container.

The board consists of a grid of fields. Let's set all the created fields to the this.fields property.

The number of rows and columns should be configurable. Therefore, let's add these settings to the global project config:

```
// ...
export const Config = {
// ...
board: {
rows: 6,
cols: 6
}
};
```
To create a board, we need to place its fields on a grid of a given size. That is, create field sprites in each column and each row of the board. Let's do this in the create method:

```
export class Board {
// ...
create() {
this.createFields();
}
```

```
  createFields() {
      for (let row = 0; row < this.rows; row++) {
          for (let col = 0; col < this.cols; col++) {
              this.createField(row, col);
          }
      }
  }
  createField(row, col) {
      const field = new Field(row, col);
      this.fields.push(field);
      this.container.addChild(field.sprite);
  }
```

}

Let's bring the board to the stage in the Game class:

```
// ...
export class Game {
constructor() {
// ...
this.board = new Board();
this.container.addChild(this.board.container);
}
// ...
}
```

Now the board is located in the upper left corner. Let's fix this by aligning its position to the center of the screen. We can get the size of the board's field by taking the first element from the generated fields array and reading the width of its sprite: this.fields[0].sprite.width; Knowing the size of one field, we can calculate the size of the entire board by multiplying the number of rows and columns by the size of the field:

```
    this.width = this.cols * this.fieldSize;
    this.height = this.rows * this.fieldSize;
```

Knowing the size of the board and the size of the screen, we can make the right padding for the board container. Subtract the size of the board from the size of the screen and the resulting remaining space will take up 2 indents to the left and right of the board:

```
export class Board {
constructor() {
// ...
this.ajustPosition();
}
// ...
ajustPosition() {
this.fieldSize = this.fields[0].sprite.width;
this.width = this.cols * this.fieldSize;
this.height = this.rows * this.fieldSize;
this.container.x = (window.innerWidth - this.width) / 2 + this.fieldSize / 2;
this.container.y = (window.innerHeight - this.height) / 2 + this.fieldSize / 2;
}
}
```

1.3. Creating a single tile

We have the board fields ready, and now we can create matching tiles in them. As in the case with the fields creation, we will start by creating a class for a single tile.

As we can see in the assets, all tile names refer to their colors. To create a sprite, we need to specify the color we want to give to that tile. We pass the name of the color as a parameter in the constructor and create the corresponding sprite. Create a file src/scripts/games/Tile.js:

```
import { App } from "../system/App";

export class Tile {
constructor(color) {
this.color = color;
this.sprite = App.sprite(this.color);
this.sprite.anchor.set(0.5);
}

  setPosition(position) {
      this.sprite.x = position.x;
      this.sprite.y = position.y;
  }

}
```

We also implement the setPosition method, which will set the sprite to the correct position. Let's add a single tile in the Board class:

```
// ...
import { Tile } from "./Tile";

export class Board {
// ...
create() {
```

this.createFields();
this.createTiles();
}

```
  createTiles() {
      const tile = new Tile("green");
      this.container.addChild(tile.sprite);
  }
```

// ...
1.4. Create a tile in each field
Now we can create a tile in each field of the board. To do this, we can loop through the array of all fields and set our own tile in each field:

```
  create() {
      this.createFields();
      this.createTiles();
  }

  createTiles() {
      this.fields.forEach(field => this.createTile(field));
  }

  createTile(field) {
      const tile = new Tile("green");
      field.setTile(tile);
      this.container.addChild(tile.sprite);
  }
```

Set a tile in each field means to place the tile in the same position on the screen as the given field. Let's implement the setTile method in the Field class:

```
  setTile(tile) {
      this.tile = tile;
      tile.field = this;
      tile.setPosition(this.position);
  }
```

}
It remains to make sure that each field has a tile with a random color. To do this, we will create a factory that will generate a random tile. Let's create the src/scripts/game/TileFactory.js class:
import { App } from "../system/App";

```
import { Tools } from "../system/Tools";
import { Tile } from "./Tile";

export class TileFactory {
static generate() {
const color = App.config.tilesColors[Tools.randomNumber(0, App.config.tilesColors.length - 1)];
return new Tile(color);
}
}
```

Add the colors config to the project global config in src/scripts/game/Config.js:

```
export const Config = {
// ...
tilesColors: ['blue', 'green', 'orange', 'red', 'pink', 'yellow'],
};
```

And add a method that returns a random integer to our special helpers class src/scripts/system/Tools.js:

```
export class Tools {
// ...
static randomNumber(min, max) {
if (!max) {
max = min;
min = 0;
}

        return Math.floor(Math.random() * (max - min + 1) + min);
    }

}
```

And now in the Board class we create a tile using the factory:

```
    createTile(field) {
        const tile = TileFactory.generate();
        // ...
    }
```

2. Moving tiles

   We start developing the functionality for moving tiles.

2.1 Selecting a tile to move

We need to click on a tile to select it. This means tiles must be interactive. Let's add interactivity when creating tiles in the Board class:

```
createTile(field) {
    // ...
    tile.sprite.interactive = true;
    tile.sprite.on("pointerdown", () => {
        this.container.emit('tile-touch-start', tile);
    });
}
```

Let's fire the tile-touch-start event using the capabilities of the PIXI.Container class. And then track and handle this event in the Game class:

```
// ...
export class Game {
constructor() {
// ...
this.board.container.on('tile-touch-start', this.onTileClick.bind(this));
}
// ...
```

Let's run the onTileClick method when the tile-touch-start event fires. In this method, we will handle all three possible scenarios:

$\boxed{1}$ select a new tile to move if no other tile has been selected

$\boxed{2}$ swap tiles if another tile has already been selected and it is next to the current one

$\boxed{3}$ select a new tile if another tile has already been selected, but it is not next to the current one

Right now we are implementing the first point. Now we are implementing the first point. When choosing a new tile, we need to do 2 things:

remember the selected tile
visually highlight the selected field

```
export class Game {
onTileClick(tile) {
if (this.selectedTile) {
// select new tile or make swap
} else {
this.selectTile(tile);
```

}

}

```
  selectTile(tile) {
      this.selectedTile = tile;
      this.selectedTile.field.select();
  }
```

We can highlight the field with the selected tile by showing an additional border in this field. Let's implement the code in the Field class:

export class Field {

constructor(row, col) {

// ...

this.selected = App.sprite("field-selected");

this.sprite.addChild(this.selected);

this.selected.visible = false;

this.selected.anchor.set(0.5);

```
  }

  unselect() {
      this.selected.visible = false;
  }

  select() {
      this.selected.visible = true;
  }
```

2.2 Swapping tiles

In the onTileClick method of the Game class, add swap method call, which implements the movement of tiles:

```
  onTileClick(tile) {
      if (this.selectedTile) {
          this.swap(this.selectedTile, tile);
      } else {
          this.selectTile(tile);
      }
  }
```

Let's define what actions we need to move tiles:

1 Reset fields in moved tiles

2 Reset tiles in the board's fields

3 Place the moved tiles in the positions of the new fields on the screen

We will create tweens animation using gsap for tiles movement. It's also worth locking the board by setting an additional flag to prevent interactivity while the animation is running.

Let's implement these actions in code:

```
swap(selectedTile, tile) {
    this.disabled = true;        // lock the board to prevent tiles from moving again while the
    this.clearSelection();      // hide the "field-selected" frame from the field of the selecte
    selectedTile.sprite.zIndex = 2; // place the selectedTile sprite one layer higher than the t

    selectedTile.moveTo(tile.field.position, 0.2); // move selectedTile to tile position
    // move tile to electedTile position
    tile.moveTo(selectedTile.field.position, 0.2).then(() => {
        // after motion animations complete:
        // change the values of the field properties in the tile objects
        // change the values of the tile properties in the field objects
        this.board.swap(selectedTile, tile);
        this.disabled = false; // unlock the board
    });
}
```

Let's reset the selection in the field in clearSelection method:

```
clearSelection() {
    if (this.selectedTile) {
        this.selectedTile.field.unselect();
        this.selectedTile = null;
    }
}
```

Let's implement the moveTo method in the Tile class using the gsap tween animation:
moveTo(position, duration) {
return new Promise(resolve => {
gsap.to(this.sprite, {
duration,
pixi: {
x: position.x,
y: position.y
},
onComplete: () => {

resolve()
}
});
});
}
Let's add the swap method in the Board class, in which we will change the properties values in the moved objects:

```
swap(tile1, tile2) {
    const tile1Field = tile1.field;
    const tile2Field = tile2.field;

    tile1Field.tile = tile2;
    tile2.field = tile1Field;

    tile2Field.tile = tile1;
    tile1.field = tile2Field;
}
```

Now, when moving tiles, we can swap not only neighboring tiles, but any tiles on the board. But only neighboring tiles should be swapped. And if the player has chosen a non-neighboring tile, then we just need to completely clear the first selection and select the new tile.

Let's update the condition in the onTileClick method in the Game class:

```
onTileClick(tile) {
    if (this.disabled) {
        return;
    }

    if (this.selectedTile) {
        if (!this.selectedTile.isNeighbour(tile)) {
            this.clearSelection(tile);
            this.selectTile(tile);
        } else {
            this.swap(this.selectedTile, tile);
        }
    } else {
        this.selectTile(tile);
    }
}
```

Note

As you can see, we also added a check for the disabled flag, which we set in the last paragraph. Thus, we block the functionality of the onTileClick method while the tiles are moving on the board, in order to avoid possible bugs.

Let's implement the isNeighbour method in the Tile class. A neighbor is a tile located either in an adjacent column or in an adjacent row. This means that the difference between either rows or columns of the checked and current tile modulo must be equal to one:

```
isNeighbour(tile) {
    return Math.abs(this.field.row - tile.field.row) + Math.abs(this.field.col - tile.field.col)
}
```

3. Search for combinations
   After swapping tiles, the board must be checked for combinations.

Note

A combination is considered to be the collection of 3, 4 and 5 identical tiles in a row.

To check for all these combinations, it is enough to compare each tile on the board with the next two tiles in a row and the next two tiles in a column.

Let's add the comparison rules to the game config Config.js:

export const Config = {
// ...
combinationRules: {col: 1, row: 0}, {col: 2, row: 0},], [{col: 0, row: 1}, {col: 0, row: 2},
};
We have added 2 validation rules that show exactly which fields on the board should have the same tiles in relation to the field being checked. That is, for each checked field on the board, it is necessary to check the match of the field in the next two columns, as well as in the next two rows.
We implement the CombinationManager class:

import { App } from "../system/App";

export class CombinationManager {
constructor(board) {
this.board = board;
}

```
  getMatches() {
      let result = [];

      this.board.fields.forEach(checkingField => {
          App.config.combinationRules.forEach(rule => {
              let matches = [checkingField.tile];

              rule.forEach(position => {
                  const row = checkingField.row + position.row;
                  const col = checkingField.col + position.col;
                  const comparingField = this.board.getField(row, col);
                  if (comparingField && comparingField.tile.color === checkingField.tile.color) {
                      matches.push(comparingField.tile);
                  }
              });

              if (matches.length === rule.length + 1) {
                  result.push(matches);
              }
          });
      });

      return result;
  }
```

}

Let's implement the getField method in the Board class:

```
  getField(row, col) {
      return this.fields.find(field => field.row === row && field.col === col);
  }
```

And call its method in the Game class:

import { CombinationManager } from "./CombinationManager";

export class Game {

constructor() {

// ...

this.combinationManager = new CombinationManager(this.board);

}

```
  swap(selectedTile, tile) {
      // ...
      tile.moveTo(selectedTile.field.position, 0.2).then(() => {
          this.board.swap(selectedTile, tile);
          const matches = this.combinationManager.getMatches();
          this.disabled = false;
      });
  }
```

}

4. Processing combinations

4.1 Removing tiles

First of all, we will remove all the tiles in the collected combinations:

export class Game {

// ...

swap(selectedTile, tile) {

// ...

const matches = this.combinationManager.getMatches();

if (matches.length) {

this.processMatches(matches);

}

});

}

```
  processMatches(matches) {
      this.removeMatches(matches);
  }

  removeMatches(matches) {
      matches.forEach(match => {
          match.forEach(tile => {
              tile.remove();
          });
      });
  }
```

}

Let's implement the remove method in the Tile class:

```
remove() {
    if (!this.sprite) {
        return;
    }
    this.sprite.destroy();
    this.sprite = null;

    if (this.field) {
        this.field.tile = null;
        this.field = null;
    }
}
```

If the object no longer has a sprite, then it has already been deleted. Otherwise, we delete the sprite and the reference to the field. At the field itself, we also remove the reference to the current tile.

4.2 Fall of the remaining tiles

After the combination is triggered and the collected tiles are removed, it is necessary to drop the remaining tiles on the board. Add processFallDown method call in processMatches:

```
processMatches(matches) {
    this.removeMatches(matches);
    this.processFallDown();
}
```

Starting from the bottom row of the board, check each field for a tile. If the field is empty, we will shift down the column all the tiles that are above it:

```
processFallDown() {
    return new Promise(resolve => {
        let completed = 0;
        let started = 0;

        // check all fields of the board, starting from the bottom row
        for (let row = this.board.rows - 1; row >= 0; row--) {
            for (let col = this.board.cols - 1; col >= 0; col--) {
                const field = this.board.getField(row, col);

                // if there is no tile in the field
                if (!field.tile) {
                    ++started;

                    // shift all tiles that are in the same column in all rows above
                    this.fallDownTo(field).then(() => {
                        ++completed;
                        if (completed >= started) {
                            resolve();
                        }
                    });
                }
            }
        }
    });
}
```

We implement moving tiles down the column in the fallDownTo method:

```
fallDownTo(emptyField) {
    // checking all board fields in the found empty field column, but in all higher rows
    for (let row = emptyField.row - 1; row >= 0; row--) {
        let fallingField = this.board.getField(row, emptyField.col);

        // find the first field with a tile
        if (fallingField.tile) {
            // the first found tile will be placed in the current empty field
            const fallingTile = fallingField.tile;
            fallingTile.field = emptyField;
            emptyField.tile = fallingTile;
            fallingField.tile = null;
            // run the tile move method and stop searching a tile for that empty field
            return fallingTile.fallDownTo(emptyField.position);
        }
    }

    return Promise.resolve();
}
```

Let's add the fallDownTo method to the Tile class:

```
fallDownTo(position, delay) {
    return this.moveTo(position, 0.5, delay, "bounce.out");
}
```

4.3 Adding and dropping new tiles

After the completion of the fall of the remaining tiles, we need to create new tiles on top of the board so that they fall into the resulting empty fields:

```
processMatches(matches) {
    this.removeMatches(matches);
    this.processFallDown()
        .then(() => this.addTiles())

}
```

To perform the creation and dropping of new tiles, we need to get all the fields on the board that have no tiles left. For each empty field, create a new tile, place it higher than the first row of the board, and start the motion animation on the given empty field:

```
addTiles() {
    return new Promise(resolve => {
        // get all fields that don't have tiles
        const fields = this.board.fields.filter(field => field.tile === null);
        let total = fields.length;
        let completed = 0;

        // for each empty field
        fields.forEach(field => {
            // create a new tile
            const tile = this.board.createTile(field);
            // put it above the board
            tile.sprite.y = -500;
            const delay = Math.random() * 2 / 10 + 0.3 / (field.row + 1);
            // start the movement of the tile in the given empty field with the given delay
            tile.fallDownTo(field.position, delay).then(() => {
                ++completed;
                if (completed >= total) {
                    resolve();
                }
            });
        });
    });
}
```

4.4 Checking for combinations after tiles falling

After the completion of falling new tiles, combinations may appear on the board again.

If there are new combinations, it is also necessary to process them, that is, collect the combination, make a tiles fall and create new tiles. For all these actions, we have already developed functionality in the processMatches method. We will call it recursively until there are no combinations left on the board after the next falling of new tiles:

```
processMatches(matches) {
    this.removeMatches(matches);
    this.processFallDown()
        .then(() => this.addTiles())
        .then(() => this.onFallDownOver());
}

onFallDownOver() {
    const matches = this.combinationManager.getMatches();

    if (matches.length) {
        this.processMatches(matches)
    } else {
        this.disabled = false;
    }
}
```

5. Collect combinations at the start
   Combinations can appear be not only after moving two tiles, but also during the initial placement of tiles after creating the board. Such combinations must be automatically processed without falling animation and replaced with other tiles before showing the starting board to the player.

We already have the functionality needed to handle starting combinations:

1️⃣ Find all combinations on the board using combinationManager.

2️⃣ Remove all founded matches

3️⃣ Create new tiles in empty fields

4️⃣ If combinations appear after adding new tiles, then repeat. Otherwise, let's start the game.

// ...
export class Game {
constructor() {
// ...

this.removeStartMatches();

}

```
  removeStartMatches() {
      let matches = this.combinationManager.getMatches(); // find combinations to collect

      while(matches.length) { // as long as there are combinations
          this.removeMatches(matches); // remove tiles in combinations

          const fields = this.board.fields.filter(field => field.tile === null); // find empty fie

          fields.forEach(field => { // in each empty field
              this.board.createTile(field); // create a new random tile
          });

          matches = this.combinationManager.getMatches(); // looking for combinations again after
      }
  }
```

}

6. Reverse swap

If, after the swap, no combination was formed on the board to collect, it is necessary to perform a reverse swap of tiles. To move tiles on the board, we have already implemented the swap method. We can modify it to perform a reverse move if no combinations are found after the main move. Add the reverse flag as the third parameter to the swap method:

// ...

export class Game {

// ...

```
  swap(selectedTile, tile, reverse) {
      this.disabled = true;
      selectedTile.sprite.zIndex = 2;

      selectedTile.moveTo(tile.field.position, 0.2);

      this.clearSelection();

      tile.moveTo(selectedTile.field.position, 0.2).then(() => {
          this.board.swap(selectedTile, tile);

          // after the swap, check if it was the main swap or reverse
          if (!reverse) {
              // if this is the main swap, then we are looking for combinations
              const matches = this.combinationManager.getMatches();
              if (matches.length) {
                  // if there are combinations, then process them
                  this.processMatches(matches);
              } else {
                  // if there are no combinations after the main swap, then perform a reverse swap
                  this.swap(tile, selectedTile, true);
              }
          } else {
              // in this condition, by the reverse flag, we understand that the swap was reversed,
              // all you need to do is unlock the board, because here the movement is already comp
              this.disabled = false;
          }
      });
  }

}
```