README.md

Phaser 3 Webpack Project Template

A Phaser 3 project template with ES6 support via Babel 7 and Webpack 4 that includes hot-reloading for development and production-ready builds.

Loading images via JavaScript module `import` is also supported.

Requirements

Node.js is required to install dependencies and run scripts via `npm`.

Available Commands

Command	Description
`npm install`	Install project dependencies
`npm start`	Build project and open web server running project
`npm run build`	Builds code bundle with production settings (minification, uglification, etc..)

Writing Code

After cloning the repo, run `npm install` from your project directory. Then, you can start the local development server by running `npm start`.

After starting the development server with `npm start`, you can edit any files in the `src` folder and webpack will automatically recompile and reload your server (available at `http://localhost:8080` by default).

Customizing Template

Babel

You can write modern ES6+ JavaScript and Babel will transpile it to a version of JavaScript that you want your project to support. The targeted browsers are set in the `.babelrc` file and the default currently targets all browsers with total usage over "0.25%" but excludes IE11 and Opera Mini.

```
"browsers": [
  ">0.25%",
  "not ie 11",
  "not op_mini all"
]
```

Webpack

If you want to customize your build, such as adding a new webpack loader or plugin (i.e. for loading CSS or fonts), you can modify the `webpack/base.js` file for cross-project changes, or you can modify and/or create new configuration files and target them in specific npm tasks inside of `package.json'.

Deploying Code

After you run the `npm run build` command, your code will be built into a single bundle located at `dist/bundle.min.js` along with any other assets you project depended.

If you put the contents of the `dist` folder in a publicly-accessible location (say something like `http://mycoolserver.com`), you should be able to open `http://mycoolserver.com/index.html` and play your game.

src/base_classes/Audio.js.md

Audio Class Documentation

Table of Contents

Audio Class

The `Audio` class is responsible for managing all audio assets used in the game. It handles loading, configuring, and playing sounds.

Constructor

The constructor initializes the `Audio` class with a reference to the current scene. It also calls the `loadAudio()` method to load all required audio assets.

```
//Class Audio
export default class Audio {
    constructor(scene) {
        this.scene = scene;
        this.loadAudio();
    }

    // ...
}
```

loadAudio() Method

The `loadAudio()` method loads and configures all audio assets used in the game. The method uses the Phaser `scene.sound.add()` method to add each sound to the scene's sound manager. It also sets specific properties for each sound, such as `loop`, `volume`, etc.

```
    loadAudio() {
        // Loads and configures background music
        this.musicBackgroundDefault = this.scene.sound.add('backgroundDefault', {
            loop: true, // Sets the music to loop continuously
            volume: 1.5 // Sets the music volume to 1.5
        });

        // Loads and configures sound for reels spinning
        this.audioReels = this.scene.sound.add('reels');

        // Loads and configures sound for reels stopping
        this.audioReelStop = this.scene.sound.add('reelStop');

        // Loads and configures sound for winning
        this.audioWin = this.scene.sound.add('win', { loop : true }); // Sets the win sound to loop continuously

        // Loads and configures sound for button clicks
        this.audioButton = this.scene.sound.add('button');

        // Loads and configures sound for losing
        this.audioLose = this.scene.sound.add('lose', { volume: 2.5 }); // Sets the losing sound volume to 2.5

        // Loads and configures default music
        this.musicDefault = this.scene.sound.add('musicDefault', {
            loop: true,
            volume: 2
        });
    }
```

This method demonstrates a common approach for managing audio assets in Phaser games. It provides a clear and organized way to load and configure all necessary sounds, making them easily accessible for later use in the game.

src/base_classes/AutoSpin.js.md

AutoSpin Class Documentation

Table of Contents

AutoSpin

The `AutoSpin` class manages the automatic spinning feature in the game. It provides functionality for starting and stopping automatic spins, adjusting the bet

amount, and executing the spinning process at the specified speed.

Constructor

```
constructor(scene) {
    this.scene = scene;
    this.autoSpin();
}
```

The constructor initializes the `AutoSpin` object with a reference to the game scene. It calls the `autoSpin` method to set up the initial UI elements and event listeners for the auto spin feature.

autoSpin

```
autoSpin() {
    this.buttonAuto = new Sprite(this.scene, Config.width - 110, Config.height - 50, 'bgButtons', 'btn-info.png');
    this.txtAutoSpin = this.scene.add.dynamicBitmapText(Config.width - 155, Config.height - 70, 'txt_bitmap', Options.txtAu
    this.txtAutoSpin.setDisplayCallback(this.scene.textCallback);
    this.buttonAuto.on('pointerdown', () => {
        if (!Options.checkClick) {
            this.buttonAuto.setScale(0.9);
            this.playSpeedAuto();
        }
    });
    this.buttonAuto.on('pointerup', () => this.buttonAuto.setScale(1));
}
```

The `autoSpin` method creates the initial UI elements for the auto spin feature:

- **`this.buttonAuto`:** A sprite representing the "Auto Spin" button, positioned at the bottom right of the screen.
- **`this.txtAutoSpin`:** A dynamic bitmap text displaying the current auto spin status ("AUTO" or "STOP").

Event listeners are attached to the "Auto Spin" button:

- **`pointerdown`:** When the button is pressed down, it scales down to 0.9 and calls the `playSpeedAuto` method.
- **`pointerup`:** When the button is released, it scales back to 1.

playSpeedAuto

```
playSpeedAuto() {
    if(Options.txtAutoSpin === 'STOP') {
        //set text auto
        Options.txtAutoSpin = 'AUTO';
        this.txtAutoSpin.setText(Options.txtAutoSpin);
        //remove timer event
        if(this.txtSpeed && this.timer) {
            this.txtSpeed.destroy();
            this.timer.remove();
        }
    } else {
        //set text auto
        Options.txtAutoSpin = 'STOP';
        this.txtAutoSpin.setText(Options.txtAutoSpin);
        //play audio button
        this.scene.audioPlayButton();

        this.bgAuto = new Sprite(this.scene, Config.width / 2, Config.height / 2,
            'autoSpin', 'bg_auto.png');
        this.auto = new Sprite(this.scene, Config.width / 2, Config.height / 2 - 100,
            'bgButtons', 'btn-spin.png');

        this.txtAuto = this.scene.add.text(Config.width / 2 - 5, Config.height / 2 - 115,
            Options.txtAuto, { fontSize : '35px', color : '#fff', fontFamily : 'PT Serif' });

        this.setXAuto();
        this.plus();
        this.minus();
        this.play();
        this.exit();
    }
}
```

The `playSpeedAuto` method handles the logic for starting and stopping the auto spin:

1. **Toggle Auto Spin Status:** It checks the current auto spin status (`Options.txtAutoSpin`) and toggles it between "AUTO" and "STOP".
2. **Remove Previous Timer:** If the auto spin is being stopped, it removes the existing speed timer (`this.timer`) and destroys the speed display text (`this.txtSpeed`).
3. **Start Auto Spin UI:** If the auto spin is being started, it creates the following UI elements:
    - **`this.bgAuto`:** A background image for the auto spin UI.
    - **`this.auto`:** A button for triggering the spin.
    - **`this.txtAuto`:** A text display for the current bet amount.

4. **Setup UI Functions:** It calls other methods to handle the UI elements:
    - **`setXAuto`:** Sets the correct X position for the bet amount text based on its value.
    - **`plus`:** Creates a button for increasing the bet amount.
    - **`minus`:** Creates a button for decreasing the bet amount.
    - **`play`:** Creates a button for initiating a spin.
    - **`exit`:** Creates a button for exiting the auto spin UI.

plus

```
plus() {
    this.btnPlus = new Sprite(this.scene, Config.width / 2 - 100, Config.height / 2 - 100,
        'autoSpin', 'btn_plus_bet.png');
    this.btnPlus.on('pointerdown', () => {
        //play audio button
        this.scene.audioPlayButton();
        if(Options.txtAuto < 100) {
            this.btnMinus.clearTint();
            this.btnPlus.setScale(0.9);
            Options.txtAuto += 5;
            //set text x auto
            Options.txtAuto < 100 ? this.txtAuto.x = 620 :
                this.txtAuto.x = 610;
            this.txtAuto.setText(Options.txtAuto);
        }
        if(Options.txtAuto === 100) {
            this.btnPlus.setTint(0xa09d9d);
        }
    });
    this.btnPlus.on('pointerup', () => this.btnPlus.setScale(1));
}
```

The `plus` method creates a button for increasing the bet amount:

1. **Create Button:** It creates a sprite representing the "+" button, positioned at the left of the bet amount text.
2. **Event Listeners:** It attaches event listeners to the button:
    - **`pointerdown`:** When the button is pressed down, it scales down to 0.9, plays an audio effect, increases the bet amount (`Options.txtAuto`) by 5, updates the bet amount text, and adjusts its position. If the bet amount reaches 100, it tints the button gray.
    - **`pointerup`:** When the button is released, it scales back to 1.

minus

```
minus() {
    this.btnMinus = new Sprite(this.scene, Config.width / 2 + 100, Config.height / 2 - 100,
        'autoSpin', 'btn_minus_bet.png');
    this.btnMinus.on('pointerdown', () => {
        //play audio button
        this.scene.audioPlayButton();
        if(Options.txtAuto > 5) {
            this.btnPlus.clearTint();
            this.btnMinus.setScale(0.9);
            Options.txtAuto -= 5;
            //function set text x auto
            this.setXAuto();
            this.txtAuto.setText(Options.txtAuto);
        }
        if(Options.txtAuto === 5) {
            this.btnMinus.setTint(0xa09d9d);
        }
    });
    this.btnMinus.on('pointerup', () => this.btnMinus.setScale(1));
}
```

The `minus` method creates a button for decreasing the bet amount:

1. **Create Button:** It creates a sprite representing the "-" button, positioned at the right of the bet amount text.
2. **Event Listeners:** It attaches event listeners to the button:
    - **`pointerdown`:** When the button is pressed down, it scales down to 0.9, plays an audio effect, decreases the bet amount (`Options.txtAuto`) by 5, updates the bet amount text, adjusts its position, and calls the `setXAuto` method to reposition the bet amount text. If the bet amount reaches 5, it tints the button gray.
    - **`pointerup`:** When the button is released, it scales back to 1.

play

```
play() {
    this.btnPlay = new Sprite(this.scene, Config.width / 2, Config.height / 2 + 100,
        'bgButtons', 'btn_play.png').setScale(0.9);
    this.btnPlay.on('pointerdown', () => {
        //play audio button
        this.scene.audioPlayButton();
        //function remove image auto
```

```
        this.removeImgAuto();
        if(this.scene.valueMoney >= Options.coin * Options.line)
            this.speedPlay(Options.txtAuto);
        else
            this.setTextAuto();
    });
}
```

The `play` method creates a button for initiating a spin:

1. **Create Button:** It creates a sprite representing the "Play" button, positioned below the bet amount text.
2. **Event Listeners:** It attaches an event listener to the button:
    - **`pointerdown`:** When the button is pressed down, it plays an audio effect, removes the auto spin UI elements by calling the `removeImgAuto` method, and checks if the player has enough money to place the bet. If enough money is available, it calls the `speedPlay` method to start the spinning process with the specified speed. If not enough money is available, it sets the auto spin status back to "AUTO" by calling the `setTextAuto` method.

exit

```
exit() {
    this.btnExit = new Sprite(this.scene, Config.width - 30 ,
        Config.height - 635,
        'bgButtons', 'btn_exit.png').setScale(0.9);
    this.btnExit.on('pointerdown', () => {
        //play audio button
        this.scene.audioPlayButton();
        //function remove image auto
        this.removeImgAuto();
        //set text auto
        this.setTextAuto();
    });
}
```

The `exit` method creates a button for exiting the auto spin UI:

1. **Create Button:** It creates a sprite representing the "Exit" button, positioned at the bottom right of the screen.
2. **Event Listeners:** It attaches an event listener to the button:
    - **`pointerdown`:** When the button is pressed down, it plays an audio effect, removes the auto spin UI elements by calling the `removeImgAuto` method, and sets the auto spin status back to "AUTO" by calling the `setTextAuto` method.

speedPlay

```
speedPlay(speed) {
    //set text speed
    let width;
    speed > 5 ? width = Config.width - 150 :  width = Config.width - 130;

    this.txtSpeed = this.scene.add.dynamicBitmapText(width, Config.height / 2 - 350, 'txt_bitmap', speed, 80);
    this.txtSpeed.setDisplayCallback(this.scene.textCallback);
    this.timer = this.scene.time.addEvent({
        delay: 500,
        callback: function() {
            //set delay
            this.timer.delay = 4500;
            if(speed > 0 && this.scene.valueMoney >=
                Options.coin * Options.line) {
                //set color
                this.scene.baseSpin.setColor();
                //set check click = true
                Options.checkClick = true;
                //detroys line array
                this.scene.baseSpin.destroyLineArr();
                //funtion remove text win
                this.scene.baseSpin.removeTextWin();
                //save localStorage
                this.scene.baseSpin.saveLocalStorage();
                this.tweens = new Tween(this.scene);
                speed -- ;
                this.txtSpeed.setText(speed);
            } else {
                Options.checkClick = false;
                this.timer.remove(false);
                this.txtSpeed.destroy();
                //set text auto
                this.setTextAuto();
            }
        },
        callbackScope: this,
        loop: true
    });
}
```

The `speedPlay` method handles the automatic spinning process at the specified speed:

1. **Create Speed Display:** It creates a dynamic bitmap text (`this.txtSpeed`) to display the current speed, positioned at the top of the screen.
2. **Start Timer:** It creates a timer (`this.timer`) that triggers a callback function every 500 milliseconds.
3. **Timer Callback:** The callback function handles the following:
    - **Set Delay:** Sets the timer delay to 4500 milliseconds (4.5 seconds) after the first execution.
    - **Check Conditions:** It checks if the speed is greater than 0 and the player has enough money to bet.
    - **Perform Spin:** If both conditions are met, it executes the following:
        - **Set Color:** Calls a method to set the colors for the spinning symbols.
        - **Enable Click:** Sets `Options.checkClick` to `true` to enable click interactions.
        - **Destroy Lines:** Destroys any existing win lines.
        - **Remove Win Text:** Removes any existing win text.
        - **Save Local Storage:** Saves the game data to local storage.
        - **Create Tween:** Creates a new tween object to handle animation.
        - **Decrement Speed:** Decreases the speed by 1 and updates the speed display.
    - **Stop Spin:** If either condition is not met, it stops the auto spin:
        - **Disable Click:** Sets `Options.checkClick` to `false` to disable click interactions.
        - **Remove Timer:** Removes the timer.
        - **Destroy Speed Text:** Destroys the speed display text.
        - **Set Auto Text:** Sets the auto spin status back to "AUTO" by calling the `setTextAuto` method.

setTextAuto

```
setTextAuto() {
    Options.txtAutoSpin = 'AUTO';
    this.txtAutoSpin.setText(Options.txtAutoSpin);
}
```

The `setTextAuto` method sets the auto spin status text back to "AUTO". It updates the `Options.txtAutoSpin` variable and sets the text of the auto spin status display (`this.txtAutoSpin`) to "AUTO".

setXAuto

```
setXAuto() {
    if(Options.txtAuto >= 100)
        this.txtAuto.x = 610;
    else if(Options.txtAuto >= 10)
        this.txtAuto.x = 620;
    else
        this.txtAuto.x = 635;
}
```

The `setXAuto` method adjusts the X position of the bet amount text (`this.txtAuto`) based on its value:

- **100 or more:** Sets the X position to 610.
- **10 to 99:** Sets the X position to 620.
- **Less than 10:** Sets the X position to 635.

removeImgAuto

```
removeImgAuto() {
    this.bgAuto.destroy();
    this.btnPlus.destroy();
    this.btnMinus.destroy();
    this.auto.destroy();
    this.txtAuto.destroy();
    this.btnPlay.destroy();
    this.btnExit.destroy();
}
```

The `removeImgAuto` method destroys all the UI elements related to the auto spin feature:

- **`this.bgAuto`:** The background image.
- **`this.btnPlus`:** The "+" button.
- **`this.btnMinus`:** The "-" button.
- **`this.auto`:** The spin button.
- **`this.txtAuto`:** The bet amount text.
- **`this.btnPlay`:** The "Play" button.
- **`this.btnExit`:** The "Exit" button.

src/base_classes/BaseSpin.js.md

BaseSpin Class Documentation

Table of Contents

BaseSpin Class

This class represents the spin button functionality in the game. It handles the visual elements of the spin button, manages user interaction, and triggers game logic upon clicking the button.

Constructor

```
constructor(scene) {
  this.scene = scene;
  this.addSpin();
}
```

The constructor initializes the `BaseSpin` instance with a reference to the `scene` where it belongs. It calls the `addSpin()` method to create the visual components of the spin button.

addSpin()

```
addSpin() {
  // Create the background image for the spin button.
  this.bgSpin = new Sprite(this.scene, Config.width - 275, Config.height - 50, 'bgButtons', 'btn-spin.png');

  // Create the text label for the spin button.
  this.txtSpin = this.scene.add.dynamicBitmapText(Config.width - 315, Config.height - 70, 'txt_bitmap', Options.txtSpin, 38
  // Set the display callback for the bitmap text.
  this.txtSpin.setDisplayCallback(this.scene.textCallback);

  // Add event listeners for pointer interaction with the spin button.
  this.bgSpin.on('pointerdown', this.playTweens, this);
  this.bgSpin.on('pointerup', () => this.bgSpin.setScale(1));
}
```

This method creates the visual elements of the spin button:

- **`this.bgSpin`:** A sprite object representing the background image of the button.
- **`this.txtSpin`:** A `dynamicBitmapText` object displaying the text "SPIN" on the button.

It also adds event listeners for pointer interaction:

- **`pointerdown`:** Triggers the `playTweens()` method when the button is pressed.
- **`pointerup`:** Resets the scale of the button to 1 when the pointer is released.

playTweens()

```
playTweens() {
  if (!Options.checkClick && this.scene.valueMoney >= (Options.coin * Options.line) && Options.txtAutoSpin === 'AUTO') {
    // Destroy the line array (resets lines on the reels).
    this.destroyLineArr();

    // Set the tint for the spin button and related elements.
    this.setColor();

    // Set the click flag to true.
    Options.checkClick = true;

    // Scale down the button.
    this.bgSpin.setScale(0.9);

    // Remove the winning text from the screen.
    this.removeTextWin();

    // Save the current money value to localStorage.
    this.saveLocalStorage();

    // Create a new Tween instance to handle animation.
    this.tweens = new Tween(this.scene);
  }
}
```

This method is called when the spin button is pressed. It performs the following actions:

- **Checks conditions:**
  - **`!Options.checkClick`:** Ensures the button is not already pressed.
  - **`this.scene.valueMoney >= (Options.coin * Options.line)`:** Verifies enough money is available to spin.
  - **`Options.txtAutoSpin === 'AUTO'`:** Checks if auto-spin is enabled.
- **If all conditions are met:**
  - **`destroyLineArr()`:** Removes the previous lines on the reels.
  - **`setColor()`:** Tints the button and other elements.
  - **`Options.checkClick = true`:** Prevents further clicks until the spin is complete.
  - **`this.bgSpin.setScale(0.9)`:** Scales down the button for visual feedback.
  - **`removeTextWin()`:** Clears any previous winning text.
  - **`saveLocalStorage()`:** Saves the current money value.
  - **`this.tweens = new Tween(this.scene)`:** Creates a new `Tween` instance to handle the spin animation.

destroyLineArr()

```
destroyLineArr() {
  if (Options.lineArray.length > 0) {
    for (let i = 0; i < Options.lineArray.length; i++) {
      Options.lineArray[i].destroy();
    }
    Options.lineArray = [];
  }
}
```

This method clears the `Options.lineArray` which holds references to the lines displayed on the reels. It iterates through the array and destroys each line object.

removeTextWin()

```
removeTextWin() {
  // Play the button sound effect.
  this.scene.audioPlayButton();

  if (this.scene.audioMusicName === 'btn_music.png') {
    // Stop the winning audio if it's playing.
    this.scene.audioObject.audioWin.stop();

    // Play the reel audio.
    this.scene.audioObject.audioReels.play();
  }

  // Subtract the bet amount from the player's money.
  this.scene.valueMoney -= (Options.coin * Options.line);

  // Update the displayed money value.
  this.scene.txtMoney.setText(this.scene.valueMoney + '$');

  // Destroy any existing winning text.
  if (this.scene.txtWin) {
    this.scene.txtWin.destroy();
  }
}
```

This method handles the audio and money management after a spin is triggered:

- **Play the button sound effect.**
- **Stop the winning audio and play the reel audio if needed.**
- **Subtract the bet amount from the player's money.**
- **Update the displayed money value.**
- **Destroy any existing winning text.**

setColor()

```
setColor() {
  this.bgSpin.setTint(0xa09d9d);
  this.scene.autoSpin.buttonAuto.setTint(0xa09d9d);
  this.scene.maxBet.maxBet.setTint(0xa09d9d);
  this.scene.coin.coin.setTint(0xa09d9d);
  this.scene.btnLine.btnLine.setTint(0xa09d9d);
  this.scene.btnMusic.setTint(0xa09d9d);
  this.scene.btnSound.setTint(0xa09d9d);
}
```

This method sets a specific tint color (0xa09d9d) to the spin button and several other UI elements.

saveLocalStorage()

```
saveLocalStorage() {
  if (localStorage.getItem('money')) {
    localStorage.removeItem('money');
    localStorage.setItem('money', this.scene.valueMoney);
  }
  localStorage.setItem('money', this.scene.valueMoney);
  this.scene.setTextX(this.scene.valueMoney);
  this.scene.txtMoney.setText(this.scene.valueMoney + '$');
}
```

This method saves the player's current money value to localStorage. It first checks if a previous money value exists in localStorage and removes it before saving the new value. It also updates the text display of the player's money.

src/base_classes/Coin.js.md

Coin Class Documentation

Table of Contents

1. Introduction

The `Coin` class represents the coin element in the game, responsible for managing the player's coin balance and handling coin-related actions.

2. Constructor

The `constructor` initializes a new `Coin` object.

Parameter	Type	Description
`scene`	Object	The scene where the coin is added.

Code:

```
constructor(scene) {
    this.scene = scene;
    this.addCoin();
}
```

Explanation:

- The constructor takes the `scene` as an argument and stores it in the `this.scene` property.
- It calls the `addCoin()` method to create and initialize the coin element.

3. addCoin() Method

The `addCoin()` method creates the visual representation of the coin, the text displaying the coin balance, and sets up event listeners for interaction.

Code:

```
addCoin() {
    this.coin = new Sprite(this.scene, Config.width - 678, Config.height - 50, 'bgButtons', 'btn-coin.png');
    this.txtCoin = this.scene.add.dynamicBitmapText(Config.width - 720, Config.height - 70, 'txt_bitmap', Options.txtCoin,
    this.txtCoin.setDisplayCallback(this.scene.textCallback);
    this.txtCountCoin = this.scene.add.text(Config.width - 700, Config.height - 140, Options.coin, {
        fontSize : '35px',
        color : '#fff',
        fontFamily : 'PT Serif'
    });
    //pointer down
    this.coin.on('pointerdown', this.onCoin, this);
    //pointer up
    this.coin.on('pointerup', () => this.coin.setScale(1));
}
```

Explanation:

1. **Coin Sprite:**
    - A new `Sprite` object is created using the `Sprite` class.

- It is positioned at `Config.width - 678` and `Config.height - 50`.
- The `'bgButtons'` key is used to access the `'btn-coin.png'` image from the game's assets.

2. **Coin Balance Text:**
   - A dynamic `BitmapText` object is created using `this.scene.add.dynamicBitmapText()`.
   - It is positioned at `Config.width - 720` and `Config.height - 70`.
   - The `'txt_bitmap'` key is used to access the bitmap font from the assets.
   - `Options.txtCoin` is the initial text displayed on the coin.
   - `38` specifies the font size.
   - `this.scene.textCallback` is a callback function that handles the display of text.
3. **Coin Count Text:**
   - A `Text` object is created using `this.scene.add.text()`.
   - It is positioned at `Config.width - 700` and `Config.height - 140`.
   - `Options.coin` is the initial coin count displayed.
   - The `fontSize`, `color`, and `fontFamily` properties are set to style the text.
4. **Event Listeners:**
   - A `pointerdown` event listener is added to the `this.coin` sprite, which calls the `this.onCoin()` method when the coin is clicked.
   - A `pointerup` event listener is added to the `this.coin` sprite, which resets the scale of the coin to `1` when the click is released.

4. onCoin() Method

The `onCoin()` method handles the logic when the coin is clicked. It increases the player's coin balance by 10, updates the displayed coin count, and resets the balance if it reaches 50.

Code:

```
onCoin() {
    if (!Options.checkClick && Options.txtAutoSpin === 'AUTO') {
        this.coin.setScale(0.9);
        //play audio button
        this.scene.audioPlayButton();
        if (Options.coin < 50) {
            Options.coin += 10;
            this.txtCountCoin.setText(Options.coin);
            this.scene.maxBet.txtCountMaxBet.setText('BET: ' + Options.coin * Options.line);
        } else {
            Options.coin = 10;
            this.txtCountCoin.setText(Options.coin);
            this.scene.maxBet.txtCountMaxBet.setText('BET: ' + Options.coin * Options.line);
        }
    }
}
```

Explanation:

1. **Check for Click and Auto Spin:**
   - The function first checks if `Options.checkClick` is false and `Options.txtAutoSpin` is equal to 'AUTO'. This ensures that the coin can only be clicked when clicks are enabled and the game is not in auto-spin mode.
2. **Coin Scaling:**
   - If the conditions in step 1 are met, the scale of the coin sprite is reduced to 0.9, providing visual feedback to the player.
3. **Play Audio:**
   - The `this.scene.audioPlayButton()` method is called to play a sound effect when the coin is clicked.
4. **Increase Coin Balance:**
   - If the current `Options.coin` is less than 50, it is increased by 10.
   - The text of `this.txtCountCoin` is updated to display the new coin count.
   - The text of `this.scene.maxBet.txtCountMaxBet` is updated to display the current bet amount, which is calculated as `Options.coin * Options.line`.
5. **Reset Coin Balance:**
   - If the current `Options.coin` is 50 or more, it is reset to 10.
   - The text of `this.txtCountCoin` and `this.scene.maxBet.txtCountMaxBet` is updated to reflect the new balance and bet amount.

src/base_classes/Container.js.md

Container Class Documentation

Table of Contents

Container Class

The `Container` class extends the Phaser `GameObjects.Container` class and represents a single column of symbols in the game. It contains five symbols, each randomly chosen from a set of ten possible symbols.

Constructor

The constructor initializes the container with its position, adds it to the scene, and creates the five symbols.

```
constructor(scene, x, y) {
    super(scene, x, y);
    scene.add.existing(this);

    //symbols column
    const symbols1 = scene.add.sprite(0, 0, 'symbols', 'symbols_' + this.randomBetween(0, 9) + '.png');
    const symbols2 = scene.add.sprite(0, - Options.symbolHeight, 'symbols', 'symbols_' + this.randomBetween(0, 9) + '.png')
    const symbols3 = scene.add.sprite(0, - Options.symbolHeight * 2, 'symbols', 'symbols_' + this.randomBetween(0, 9) + '.p
    const symbols4 = scene.add.sprite(0, - Options.symbolHeight * 3, 'symbols', 'symbols_' + this.randomBetween(0, 9) + '.p
    const symbols5 = scene.add.sprite(0, - Options.symbolHeight * 4, 'symbols', 'symbols_' + this.randomBetween(0, 9) + '.p

    this.add([symbols1, symbols2, symbols3, symbols4, symbols5]);
}
```

The constructor:

1. **Calls the parent constructor:** `super(scene, x, y)`
2. **Adds the container to the scene:** `scene.add.existing(this)`
3. **Creates five symbol sprites:**
   - Each sprite is created using `scene.add.sprite()`, using the 'symbols' key for the texture and a randomly generated file name from the set 'symbols_0.png' to 'symbols_9.png'.
   - Each symbol is positioned vertically, with the first symbol at `(0, 0)` and each subsequent symbol offset by `Options.symbolHeight`.
4. **Adds the symbols to the container:** `this.add([symbols1, symbols2, symbols3, symbols4, symbols5])`

randomBetween Method

The `randomBetween` method is a helper method used to generate a random integer between a minimum and maximum value.

```
randomBetween(min, max) {
    return Phaser.Math.Between(min, max);
}
```

It simply utilizes the `Phaser.Math.Between` function to achieve this functionality.

src/base_classes/Credit.js.md

Credit Class Documentation

Table of Contents

Credit Class Overview

The `Credit` class is responsible for handling the display and interaction of the game's credit screen. It uses Phaser's Sprite class to create the necessary visual elements.

Constructor

The constructor initializes the `Credit` object with a reference to the scene it belongs to. It then calls the `addCredit` function to create the visual elements.

```
constructor(scene) {
```

```
    this.scene = scene;
    this.addCredit();
}
```

addCredit Function

The `addCredit` function is responsible for creating the visual elements of the credit screen:

1. **Credits Button:** It creates a `Sprite` object representing the "Credits" button, positioned at the bottom right of the screen.
    - The sprite is initialized with the appropriate image and scaling.
    - An event listener is attached to the button that triggers the `audioPlayButton` function in the scene.
2. **Paylines Display:** When the "Credits" button is clicked, it creates a `Sprite` object representing the paylines display, centered on the screen.
    - The sprite is initialized with the appropriate image and set to a higher depth to ensure it is visible on top of other elements.
3. **Exit Button:** It creates a `Sprite` object representing the "Exit" button, positioned at the bottom right of the screen.
    - The sprite is initialized with the appropriate image, scaling, and depth.
    - An event listener is attached to the button that triggers the `deleteCredit` function when clicked.

```
addCredit() {
    this.credits = new Sprite(this.scene, Config.width - 235, Config.height - 680,
        'about', 'btn-credits.png').setScale(0.7);
    this.credits.on('pointerdown', () => {
        //play audio button
        this.scene.audioPlayButton();
        this.paylines = new Sprite(this.scene, Config.width / 2, Config.height / 2,
            'about', 'palines.png').setDepth(1);
        this.btnExit = new Sprite(this.scene, Config.width - 30,
                Config.height - 635, 'bgButtons', 'btn_exit.png').
                setScale(0.9).setDepth(1);
        this.btnExit.on('pointerdown', this.deleteCredit, this);
    });
}
```

deleteCredit Function

The `deleteCredit` function is responsible for destroying the visual elements created by the `addCredit` function:

1. **Play Audio:** It triggers the `audioPlayButton` function in the scene.
2. **Destroy Elements:** It destroys the "Exit" button and the paylines display using their respective `destroy` methods.

```
deleteCredit() {
    //play audio button
    this.scene.audioPlayButton();
    this.btnExit.destroy();
    this.paylines.destroy();
}
```

src/base_classes/Info.js.md

Info Class Documentation

Table of Contents

Class Info

The `Info` class is responsible for managing the "Info" button and its associated paytable functionality.

Constructor

```
constructor(scene) {
    this.scene = scene;
    this.addInfo();
    this.click = false;
}
```

The constructor initializes the `Info` object with a reference to the scene it belongs to. It also calls the `addInfo()` method to create the "Info" button and sets the `click` flag to `false`.

addInfo()

```
addInfo() {
    this.info = new Sprite(this.scene, Config.width - 1020, Config.height - 50, 'bgButtons', 'btn-info.png');
    //add bitmap text
    const txtInfo = this.scene.add.dynamicBitmapText(Config.width - 1060, Config.height - 70, 'txt_bitmap', Options.txtInfo
    txtInfo.setDisplayCallback(this.scene.textCallback);
    this.info.on('pointerdown', this.showPayTable, this);
}
```

This method creates the "Info" button using the `Sprite` class, positions it on the screen, and adds a text label using a `dynamicBitmapText` object. The text is fetched from the `Options.txtInfo` constant and formatted using the `textCallback` function defined in the scene. The `pointerdown` event listener is attached to the button, calling the `showPayTable()` method when the button is clicked.

showPayTable()

```
showPayTable() {
    if (!this.click) {
        //set click = true
        this.click = true;
        //play audio button
        this.scene.audioPlayButton();
        //function show table
        this.showTable();
        this.btnExit = new Sprite(this.scene, Config.width - 30,
            Config.height - 635, 'bgButtons', 'btn_exit.png').
            setScale(0.9).setDepth(1);
        this.btnExit.on('pointerdown', this.deleteTable, this);
    }
}
```

The `showPayTable()` method is called when the "Info" button is clicked. It checks if the `click` flag is false (indicating the paytable is not already shown). If it is, the `click` flag is set to `true`, an audio button is played, and the `showTable()` method is called to display the paytable. An "Exit" button is also created and positioned on the screen, with its `pointerdown` event listener set to call the `deleteTable()` method.

showTable()

```
showTable() {
    this.payValues = [];

    this.paytable = new Sprite(this.scene, Config.width / 2, Config.height / 2,
        'about', 'paytable.png').setDepth(1);

    var width = 190, width2 = width, height = 25, height2 = 245;

    for (let i = 0; i < Options.payvalues.length; i++) {
        if (i >= 5) {
            for (let j = 0; j < Options.payvalues[i].length; j++) {
                height2 -= 30;
                this.payValues.push(this.scene.add.text(width2, Config.height / 2 + height2, Options.payvalues[i][j], {
                    fontSize: '30px',
                    color: '#630066',
                    fontFamily: 'PT Serif'
                }).setDepth(1));
            }
            width2 += 225;
            height2 = 245;
        } else {
            for (let j = 0; j < Options.payvalues[i].length; j++) {
                height += 30;
                this.payValues.push(this.scene.add.text(width, Config.height / 2 - height, Options.payvalues[i][j], {
                    fontSize: '30px',
                    color: '#630066',
                    fontFamily: 'PT Serif'
                }).setDepth(1));
            }
            width += 225;
            height = 25;
        }
    }
}
```

The `showTable()` method displays the paytable. It first creates an empty array `payValues` to store the paytable text objects. It then creates a `Sprite` object for the paytable image and positions it in the center of the screen. The code then iterates through the `Options.payvalues` array, which contains the paytable data. For each entry in the array, it creates a text object using `scene.add.text` and adds it to the `payValues` array. The position of each text object is calculated based on its index in the `Options.payvalues` array, ensuring that the paytable is displayed in a readable format.

deleteTable()

```
deleteTable() {
    //set click = false
    this.click = false;
```

```
    //play audio button
    this.scene.audioPlayButton();
    this.paytable.destroy();
    this.btnExit.destroy();
    if (this.payValues.length > 0) {
        for (let i = 0; i < this.payValues.length; i++) {
            this.payValues[i].destroy();
        }
    }
}
```

The `deleteTable()` method is called when the "Exit" button is clicked. It sets the `click` flag to `false`, plays an audio button, and then destroys the paytable image, the "Exit" button, and all the paytable text objects in the `payValues` array. This effectively hides the paytable from the screen.

src/base_classes/Line.js.md

Line Class Documentation

Table of Contents

1. Introduction

The `Line` class is responsible for creating and managing the line bet button and its associated text elements. It utilizes the `Sprite` class to create the button image and `dynamicBitmapText` and `text` classes to display the line count and bet amount.

2. Constructor

The `Line` constructor initializes the class with a reference to the scene it belongs to. It then calls the `addLine()` method to create and configure the line bet button and text elements.

```
//Class Line
export default class Line {
    constructor(scene) {
        this.scene = scene;
        this.addLine();
    }
}
```

3. addLine Method

The `addLine` method creates the following elements:

- **btnLine:** A `Sprite` object representing the line bet button. It is positioned at `Config.width - 865, Config.height - 50` with the image `'btn-line.png'` from the `'bgButtons'` spritesheet.
- **txtLine:** A `dynamicBitmapText` object displaying the text `Options.txtLine` in the `'txt_bitmap'` font style. It is positioned at `Config.width - 915, Config.height - 70` with a font size of 38.
- **txtCountLine:** A `text` object displaying the current line count (`Options.line`). It is positioned at `Config.width - 880, Config.height - 140` with a font size of 35px, white color, and the `'PT Serif'` font family.

The method then adds event listeners to the `btnLine` for `pointerdown` and `pointerup` events.

```
addLine() {
    this.btnLine = new Sprite(this.scene, Config.width - 865, Config.height - 50, 'bgButtons', 'btn-line.png');
    this.txtLine = this.scene.add.dynamicBitmapText(Config.width - 915, Config.height - 70, 'txt_bitmap', Options.txtLine,
    this.txtLine.setDisplayCallback(this.scene.textCallback);
    this.txtCountLine = this.scene.add.text(Config.width - 880, Config.height - 140, Options.line, {
        fontSize : '35px',
        color : '#fff',
        fontFamily : 'PT Serif'
    });

    //pointer down
    this.btnLine.on('pointerdown', () => {
        if (!Options.checkClick && Options.txtAutoSpin === 'AUTO') {
            this.btnLine.setScale(0.9);
            //play audio button
            this.scene.audioPlayButton();

            if (Options.line < 20) {
                Options.line ++;
                this.txtCountLine.setText(Options.line);
                this.scene.maxBet.txtCountMaxBet.setText('BET: ' + Options.line * Options.coin);
            } else {
```

```
                Options.line = 1;
                this.txtCountLine.setText(Options.line);
                this.scene.maxBet.txtCountMaxBet.setText('BET: ' + Options.line * Options.coin);
            }
        }
    });

    //pointer up
    this.btnLine.on('pointerup', () => this.btnLine.setScale(1));
}
```

Pointer Down Event:

- The `pointerdown` event listener checks if the `Options.checkClick` flag is false and the `Options.txtAutoSpin` value is 'AUTO'. This ensures that the button is only clickable when the game is not currently processing a click or is in auto-spin mode.
- If the conditions are met, the button scale is reduced to 0.9, the `audioPlayButton` method of the scene is called to play a sound effect, and the line count is incremented or reset depending on its current value.
- The `txtCountLine` text is updated to display the new line count, and the bet amount is updated in the `txtCountMaxBet` text object of the `maxBet` instance.

Pointer Up Event:

- The `pointerup` event listener sets the button scale back to 1 when the pointer is released.

The `addLine` method ensures that the line bet button and text elements are properly created, positioned, and interactive, allowing the player to adjust the line bet amount during gameplay.

src/base_classes/Maxbet.js.md

Maxbet Class Documentation

Table of Contents

Introduction

The `Maxbet` class is responsible for managing the "Max Bet" button in the game. It handles the display, interaction, and functionality associated with this button.

Constructor

```
constructor(scene) {
    this.scene = scene;
    this.addMaxBet();
}
```

The constructor initializes the `Maxbet` object by storing a reference to the current scene and calling the `addMaxBet()` method to create and configure the button.

addMaxBet() Method

```
addMaxBet() {
    this.maxBet = new Sprite(this.scene, Config.width - 477, Config.height - 50, 'bgButtons', 'btn-maxbet.png');
    this.txtMaxBet = this.scene.add.dynamicBitmapText(Config.width - 550, Config.height - 70, 'txt_bitmap', Options.txtMaxB
    this.txtMaxBet.setDisplayCallback(this.scene.textCallback);
    this.txtCountMaxBet = this.scene.add.text(Config.width - 555, Config.height - 140, 'BET: ' + Options.coin * Options.lin
        fontSize: '35px',
        color: '#fff',
        fontFamily: 'PT Serif'
    });
    //pointer down
    this.maxBet.on('pointerdown', this.onMaxbet, this);
    //pointer up
    this.maxBet.on('pointerup', () => this.maxBet.setScale(1));
}
```

The `addMaxBet()` method performs the following actions:

1. **Creates the Max Bet Button:**
    - Initializes a `Sprite` object representing the button using the provided image and position.
    - Stores this `Sprite` object in the `this.maxBet` property.
2. **Creates Text Display for Max Bet:**

- Initializes a `dynamicBitmapText` object to display the "Max Bet" label.
- Stores this `dynamicBitmapText` object in the `this.txtMaxBet` property.
- Sets the display callback to use the scene's `textCallback` function for text rendering.

3. **Creates Text Display for Current Bet:**

   - Initializes a `text` object to display the current bet amount calculated as `Options.coin * Options.line`.
   - Stores this `text` object in the `this.txtCountMaxBet` property.
   - Configures the text style using the specified font size, color, and font family.

4. **Adds Event Listeners for Interactions:**

   - Attaches a `pointerdown` event listener to the `maxBet` Sprite, calling the `this.onMaxbet` method when the button is pressed.
   - Attaches a `pointerup` event listener to the `maxBet` Sprite, resetting its scale to 1 when the button is released.

onMaxBet() Method

```
onMaxBet() {
    if (!Options.checkClick && Options.line * Options.coin < 1000 && Options.txtAutoSpin === 'AUTO') {
        this.maxBet.setScale(0.9);
        //play audio button
        this.scene.audioPlayButton();
        Options.line = 20;
        this.scene.btnLine.txtCountLine.setText(Options.line);
        Options.coin = 50;
        this.scene.coin.txtCountCoin.setText(Options.coin);
        this.txtCountMaxBet.setText('BET: ' + Options.line * Options.coin);
    }
}
```

The `onMaxBet()` method handles the logic executed when the Max Bet button is pressed. It performs the following steps:

1. **Check Conditions:**

   - Verifies that clicking is allowed (`!Options.checkClick`).
   - Ensures the current bet amount is less than 1000 (`Options.line * Options.coin < 1000`).
   - Confirms that the auto-spin feature is enabled (`Options.txtAutoSpin === 'AUTO'`).

2. **Update Button Appearance:**

   - Scales the `maxBet` button down slightly (`this.maxBet.setScale(0.9)`), providing visual feedback.

3. **Play Button Sound:**

   - Triggers the scene's `audioPlayButton()` method to play a button click sound.

4. **Set Max Bet Values:**

   - Sets the number of lines to 20 (`Options.line = 20`).
   - Updates the line count display on the line button (`this.scene.btnLine.txtCountLine.setText(Options.line)`).
   - Sets the coin value to 50 (`Options.coin = 50`).
   - Updates the coin count display on the coin button (`this.scene.coin.txtCountCoin.setText(Options.coin)`).
   - Updates the bet amount display on the Max Bet button (`this.txtCountMaxBet.setText('BET: ' + Options.line * Options.coin)`).

Note: This method currently implements a simple "max bet" functionality by setting specific line and coin values. It could be further extended to dynamically calculate a maximum bet based on the player's current balance or other game-specific parameters.

src/base_classes/Spin.js.md

Spin Class Documentation

Table of Contents

Spin Class

The `Spin` class handles the logic for processing the results of a spin in the slot machine game. This includes determining winning lines, calculating winnings, and updating the visual display accordingly.

Constructor

The constructor initializes the `Spin` object with a reference to the game scene. It calls the `printResult` and `clearColor` methods to process the spin results and clear any visual effects from the previous spin.

```
constructor(scene) {
    this.scene = scene;
    this.printResult();
    this.clearColor();
}
```

clearColor Method

This method clears the tints from various visual elements on the screen, including the base spin background, the auto spin button, the max bet button, the coin button, the line button, and the music and sound buttons.

```
clearColor() {
    this.scene.baseSpin.bgSpin.clearTint();
    this.scene.autoSpin.buttonAuto.clearTint();
    this.scene.maxBet.maxBet.clearTint();
    this.scene.coin.coin.clearTint();
    this.scene.btnLine.btnLine.clearTint();
    this.scene.btnMusic.clearTint();
    this.scene.btnSound.clearTint();
}
```

printResult Method

This method retrieves the symbols from the result of the spin and stores them in the `Options.result` array. The symbols are retrieved from the `targets` array of the `columnTween` objects, which are part of either the `autoSpin.tweens` or `baseSpin.tweens` objects. The `Options.result` array stores the symbol names for each reel in a nested array format. After storing the result, it calls the `getWinningLines` method to determine any winning lines.

```
printResult() {
    let s1, s2, s3, s4, s5, autoSpin = this.scene.autoSpin.tweens,
    baseSpin = this.scene.baseSpin.tweens;
    if(autoSpin) {
        s1 = autoSpin.columnTween1.targets[0];
        s2 = autoSpin.columnTween2.targets[0];
        s3 = autoSpin.columnTween3.targets[0];
        s4 = autoSpin.columnTween4.targets[0];
        s5 = autoSpin.columnTween5.targets[0];
    } else {
        s1 = baseSpin.columnTween1.targets[0];
        s2 = baseSpin.columnTween2.targets[0];
        s3 = baseSpin.columnTween3.targets[0];
        s4 = baseSpin.columnTween4.targets[0];
        s5 = baseSpin.columnTween5.targets[0];
    }
    //push symbols name
    Options.result.push([s1.list[3].frame.name, s1.list[2].frame.name,
    s1.list[1].frame.name],[s2.list[3].frame.name, s2.list[2].frame.name,
    s2.list[1].frame.name],[s3.list[3].frame.name, s3.list[2].frame.name,
    s3.list[1].frame.name],[s4.list[3].frame.name, s4.list[2].frame.name,
    s4.list[1].frame.name],[s5.list[3].frame.name, s5.list[2].frame.name,
    s5.list[1].frame.name]);
    //function winning lines
    this.getWinningLines();
}
```

getWinningLines Method

This method iterates through each payline defined in the `Options.payLines` array and checks for winning combinations. It uses a nested loop to traverse the coordinates of each payline and compares the symbols at those coordinates to determine if they form a winning streak. If a streak of three or more matching symbols is found, the payline index is added to the `Options.winningLines` array. The method then plays a winning sound effect, calculates the winnings based on the symbol and streak length using the `mathMoney` method, and displays the winning lines using the `getLineArray` method.

```
getWinningLines() {
    for(let lineIndx = 0; lineIndx < Options.line;
        lineIndx ++) {
```

```
        let streak = 0;
        let currentkind = null;
        for(let coordIndx = 0; coordIndx < Options.payLines[lineIndx].
            length; coordIndx ++) {
            let coords = Options.payLines[lineIndx][coordIndx];
            let symbolAtCoords = Options.result[coords[0]][coords[1]];
            if(coordIndx === 0) {
                currentkind = symbolAtCoords;
                streak = 1;
            } else {
                if(symbolAtCoords != currentkind) {
                    break;
                }
                streak ++;
            }
        }
        //check streak >= 3
        if(streak >= 3) {
            lineIndx ++;
            Options.winningLines.push(lineIndx);
            //audio win
            this.audioPlayWin();
            //function math money
            this.mathMoney(currentkind, streak);
        }
        //audio lose
        this.audioPlayLose();
    }
    //get line array
    this.getLineArray(Options.winningLines);
    //reset Options
    this.resetOptions();
}
```

getLineArray Method

This method creates and adds `Sprite` objects representing the winning lines to the `Options.lineArray` array. It iterates through the `lineArr` (array of winning line indices) and creates a `Sprite` object with the appropriate payline image for each index.

```
getLineArray(lineArr) {
    if(!lineArr.length) {
        return;
    }
    for(let i = 0; i < lineArr.length; i++) {
        let lineName = 'payline_' + lineArr[i] + '.png';
        Options.lineArray.push(new Sprite(this.scene, Config.width / 2,
            Config.height / 2, 'line', lineName));
    }
}
```

mathMoney Method

This method calculates the winnings based on the winning symbol and the length of the winning streak. It uses the `symbolValue` method to retrieve the payout value for the symbol and streak length, and then multiplies it by the total bet amount to calculate the total winnings.

```
mathMoney(symbolName, streak) {
    let index = streak - 3;
    if(streak === 3)
        this.symbolValue(symbolName, index);
    else if(streak === 4)
        this.symbolValue(symbolName, index);
    else
        this.symbolValue(symbolName, index);
}
```

resetOptions Method

This method resets the `Options` object to its initial state, clearing the winnings, results, and winning lines.

```
resetOptions() {
    //reset win && result
    Options.win = 0;
    Options.moneyWin = 0;
    Options.result = [];
    Options.winningLines = [];
}
```

symbolValue Method

This method retrieves the payout value for a specific symbol based on the streak length. It uses a `switch` statement to match the symbol name to the corresponding payout value in the `Options.payvalues` array.

```
symbolValue(symbolName, index) {
    switch(symbolName) {
        case 'symbols_0.png':
            this.getMoney(Options.payvalues[0][index]);
            break;
        case 'symbols_1.png':
            this.getMoney(Options.payvalues[1][index]);
            break;
        case 'symbols_2.png':
            this.getMoney(Options.payvalues[2][index]);
            break;
        case 'symbols_3.png':
            this.getMoney(Options.payvalues[3][index]);
            break;
        case 'symbols_4.png':
            this.getMoney(Options.payvalues[4][index]);
            break;
        case 'symbols_5.png':
            this.getMoney(Options.payvalues[5][index]);
            break;
        case 'symbols_6.png':
            this.getMoney(Options.payvalues[6][index]);
            break;
        case 'symbols_7.png':
            this.getMoney(Options.payvalues[7][index]);
            break;
        case 'symbols_8.png':
            this.getMoney(Options.payvalues[8][index]);
            break;
        default:
            this.getMoney(Options.payvalues[9][index]);
            break;
    }
}
```

audioPlayWin Method

This method plays the winning sound effect if the music is enabled.

```
audioPlayWin() {
    if (this.scene.audioMusicName === 'btn_music.png') {
        //play audio win
        this.scene.audioObject.audioWin.play();
    }
}
```

audioPlayLose Method

This method plays the losing sound effect if the music is enabled.

```
audioPlayLose() {
    if (this.scene.audioMusicName === 'btn_music.png') {
        //play audio lose
        this.scene.audioObject.audioLose.play();
    }
}
```

getMoney Method

This method calculates the total winnings based on the payout value and the total bet amount. It updates the `Options.win` variable with the calculated winnings and calls the `setTextureWin` method to display the updated winnings on the screen.

```
getMoney(money) {
    let maxBet = Options.line * Options.coin;
    let payValue = money / Options.line;
    Options.win += (payValue * maxBet);
    this.setTextureWin(Options.win);
}
```

setTextureWin Method

This method updates the display of the winnings on the screen. It sets the `Options.moneyWin` variable to the current total winnings, updates the `scene.valueMoney` variable, calculates the width of the text based on the winnings, and creates or updates the `scene.txtWin` text object to display the winnings. It also saves the updated winnings to local storage using the `scene.baseSpin.saveLocalStorage` method.

```
setTextureWin(value) {
    Options.moneyWin = value;
    this.scene.valueMoney += Options.moneyWin;
    //function set width text win
    let width = this.setTextWidthWin();
    //check empty text win
    if (!this.scene.txtWin) {
```

```
        this.scene.txtWin = this.scene.add.text(width, Config.height - 130, 'WIN: ' + Options.moneyWin + ' $ ', {
            fontSize : '20px',
            color : '#25a028',
            fontFamily : 'PT Serif'
        });
    } else {
        this.scene.txtWin.destroy();
        this.scene.txtWin = this.scene.add.text(width, Config.height - 130, 'WIN: ' + Options.moneyWin + ' $ ', {
            fontSize : '20px',
            color : '#25a028',
            fontFamily : 'PT Serif'
        });
    }
    //save localStorage
    this.scene.baseSpin.saveLocalStorage();
}
```

setTextWidthWin Method

This method calculates the width of the text displaying the winnings based on the value of the winnings. This ensures that the text is properly aligned on the screen, regardless of the amount of the winnings.

```
setTextWidthWin() {
    let width;
    if(Options.moneyWin >= 100000)
        width = Config.width - 340;
    else if(Options.moneyWin >= 10000)
        width = Config.width - 335;
    else if(Options.moneyWin >= 1000)
        width = Config.width - 330;
    else if(Options.moneyWin >= 100)
        width = Config.width - 322;
    else
        width = Config.width - 340;
    return width;
}
```