DinoBattleGame Smart Contract Overview

The `DinoBattleGame` contract is a Solidity smart contract deployed on the Ethereum blockchain that enables a betting-based game where players compete with characters having various attributes. The contract manages player registration, game play, and fund management.

Key Components

1. Structs
    - `Character`: Defines the attributes of a character.
        - `name`: The name of the character.
        - `attack`: The attack power of the character.
        - `defense`: The defense power of the character.
        - `specialAttack`: The special attack power of the character.
        - `specialDefense`: The special defense power of the character.
        - `speed`: The speed attribute of the character.
    - `Bet`: Represents a player's bet in the game.
        - `player`: The address of the player making the bet.
        - `character`: The `Character` object associated with the player's bet.
        - `amount`: The amount of ether bet by the player.
2. State Variables
    - `winnings`: Maps player addresses to their winnings.
    - `players`: An array of addresses of players who have placed bets.
    - `registeredPlayers`: Maps addresses to a boolean indicating whether they are registered players.
    - `admin`: The address of the contract's admin.
    - `minBetAmount`: Minimum amount of ether required to place a bet.
    - `maxBetAmount`: Maximum amount of ether allowed for a bet.
3. Events
    - `GameStarted`: Emitted when a game starts with two characters.
    - `GameEnded`: Emitted when a game ends, indicating the winning character.
    - `BetPlaced`: Emitted when a player places a bet.
    - `WinningsWithdrawn`: Emitted when a player withdraws their winnings.
    - `PlayerRegistered`: Emitted when a player registers.
    - `AdminWithdrawal`: Emitted when the admin withdraws funds from the contract.
    - `DepositReceived`: Emitted when funds are deposited into the contract.
4. Modifiers
    - `onlyAdmin`: Restricts function execution to the contract admin only.
    - `onlyRegistered`: Ensures that only registered players can place bets.
5. Functions
    - `constructor`: Sets the deployer of the contract as the admin.
    - `registerPlayer`: Allows a player to register themselves. Emits the `PlayerRegistered` event.
    - `depositFunds`: Allows anyone to deposit ether into the contract. Emits the `DepositReceived` event.

- `playGame`: Allows a registered player to place a bet with their chosen character and the opponent's character. The bet amount must be between `minBetAmount` and `maxBetAmount`. Emits `GameStarted` and `GameEnded` events. Determines the winner and distributes winnings if the player's character wins.
- `determineWinner`: Internal function to calculate the winner between two characters based on their attributes and a randomness factor.
- `calculateScore`: Internal function to calculate the score of a character based on its attributes.
- `adminWithdraw`: Allows the admin to withdraw the entire balance from the contract. Emits the `AdminWithdrawal` event.

Summary

The `DinoBattleGame` contract is a game and betting system where players can register, deposit funds, and place bets on games between characters with various attributes. The contract handles player registration, bet validation, game execution, and fund management, while providing transparency through events and requiring admin oversight for fund withdrawals.

Process of Playing a Game with the `DinoBattleGame` Contract

Here's a step-by-step guide on how to play a game using the `DinoBattleGame` smart contract:

1. Register as a Player

- Action: Call the `registerPlayer` function.
- Purpose: To become a registered player eligible to place bets.
- Requirement: The player must register before placing any bets.

```
function registerPlayer() public;
```

2. Deposit Funds into the Contract

- Action: Call the `depositFunds` function and send ether with the transaction.
- Purpose: To ensure that there are sufficient funds in the contract for betting and potential winnings.
- Requirement: Players need to deposit funds if they want to place bets.

```
function depositFunds() public payable;
```

3. Play the Game

- Action: Call the `playGame` function with details of two characters and the amount of ether to bet.
- Parameters:
  - `_character1`: The player's chosen character with its attributes.
  - `_character2`: The opponent's character with its attributes.
  - `msg.value`: The amount of ether being bet.
- Purpose: To initiate the game with a bet placed by the player.
- Requirement: The bet amount must be between `minBetAmount` and `maxBetAmount`. The player must be registered.

```
function playGame(Character memory _character1, Character memory _character2) public payable;
```

- Process:
  1. The function checks if the bet amount is within the allowed range and that the player is registered.
  2. The bet is recorded and a `BetPlaced` event is emitted.
  3. The game logic determines the winner based on the attributes of both characters and some randomness.
  4. A `GameStarted` event is emitted when the game begins.
  5. A `GameEnded` event is emitted with the name of the winning character.
  6. If the player's character wins, the winnings are calculated and transferred to the player.

4. View Game Results

- Action: Listen for `GameStarted` and `GameEnded` events.
- Purpose: To receive notifications about the game status and outcome.
- Requirement: The frontend or application should handle these events to display the results to the user.

5. Withdraw Winnings

- Action: (For players) Call the `winnings` mapping to check and withdraw winnings.
- Purpose: To retrieve any winnings the player has earned from winning games.

```
function winnings(address player) public view returns (uint256);
```

6. Admin Withdraw Funds

- Action: Call the `adminWithdraw` function.
- Purpose: To withdraw the entire balance from the contract for the admin's use.
- Requirement: Only the admin (deployer of the contract) can perform this action.

```
function adminWithdraw() public;
```

Summary

1. Register as a player using `registerPlayer`.
2. Deposit Funds into the contract with `depositFunds`.
3. Play the Game by calling `playGame` with your character, the opponent's character, and the bet amount.
4. View Results through the emitted events (`GameStarted`, `GameEnded`).
5. Withdraw Winnings if applicable.
6. Admin can withdraw the contract's balance using `adminWithdraw`.

CODE:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.3;

contract DinoBattleGame {
    struct Character {
        string name;
        uint256 attack;
        uint256 defense;
        uint256 specialAttack;
        uint256 specialDefense;
        uint256 speed;
    }

    struct Bet {
        address player;
        Character character;
        uint256 amount;
    }

    mapping(address => uint256) public winnings;
    address[] public players;
    mapping(address => bool) public registeredPlayers;
```

```
    address public admin;

    uint256 public minBetAmount = 0.01 ether;
    uint256 public maxBetAmount = 1 ether;

    event GameStarted(string character1Name, string character2Name);
    event GameEnded(string winningCharacterName);
    event BetPlaced(address player, string characterName, uint256 amount);
    event WinningsWithdrawn(address player, uint256 amount);
    event PlayerRegistered(address player);
    event AdminWithdrawal(uint256 amount);
    event DepositReceived(address sender, uint256 amount);

    modifier onlyAdmin() {
        require(msg.sender == admin, "Only admin can perform this action");
        _;
    }

    modifier onlyRegistered() {
        require(registeredPlayers[msg.sender], "You must be registered to place a bet");
        _;
    }

    function registerPlayer() public {
        registeredPlayers[msg.sender] = true;
        emit PlayerRegistered(msg.sender);
    }

    constructor() {
        admin = msg.sender; // Set the deployer as the admin
    }

    function depositFunds() public payable {
        require(msg.value > 0, "Deposit amount must be greater than zero");
        emit DepositReceived(msg.sender, msg.value);
    }

    function playGame(Character memory _character1, Character memory _character2) public
payable onlyRegistered {
        require(msg.value >= minBetAmount && msg.value <= maxBetAmount, "Bet amount must be
within the allowed range");
        require(msg.value > 0, "Bet amount must be greater than zero");
        require(msg.value <= maxBetAmount, "Bet amount exceeds the maximum limit");

        // Create the player's bet
        Bet memory playerBet = Bet({
            player: msg.sender,
            character: _character1,
            amount: msg.value
        });

        players.push(msg.sender);
```

```
        emit BetPlaced(msg.sender, _character1.name, msg.value);

        // Determine the winner
        string memory winningCharacterName = determineWinner(_character1, _character2);

        // Emit events
        emit GameStarted(_character1.name, _character2.name);
        emit GameEnded(winningCharacterName);

        // Distribute winnings if the player has won
        if (keccak256(abi.encodePacked(playerBet.character.name)) ==
keccak256(abi.encodePacked(winningCharacterName))) {
            uint256 winningsAmount = 2 * playerBet.amount;
            require(address(this).balance >= winningsAmount, "Insufficient contract balance to
pay winnings");

            payable(msg.sender).transfer(winningsAmount);
        }
    }

    function determineWinner(Character memory _character1, Character memory _character2)
internal view returns (string memory) {
        uint256 score1 = calculateScore(_character1);
        uint256 score2 = calculateScore(_character2);

        // Add randomness to the score by hashing several inputs
        uint256 randomFactor = uint256(keccak256(abi.encodePacked(block.timestamp,
block.prevrandao, _character1.name, _character2.name, blockhash(block.number - 1))));
        uint256 randomScore1 = score1 + (randomFactor % 100);
        uint256 randomScore2 = score2 + ((randomFactor / 100) % 100);

        return randomScore1 >= randomScore2 ? _character1.name : _character2.name;
    }

    function calculateScore(Character memory character) internal pure returns (uint256) {
        return character.attack + character.defense + character.specialAttack +
character.specialDefense + character.speed;
    }

    function adminWithdraw() public onlyAdmin {
        uint256 contractBalance = address(this).balance;
        require(contractBalance > 0, "No funds to withdraw");
        payable(admin).transfer(contractBalance);

        emit AdminWithdrawal(contractBalance);
    }
}
```