[G] Avoid unnecessary storage writes can save gas

https://github.com/latteswap-official/survival-game-contract/blob/4f3104597bb2f9acc5e67043807efc517e71e7d6/contracts/SurvivalGame.sol#L342

```
341 for (uint256 i = 0; i < _remainingPlayerCount; ++i) {
342     bytes memory _data = abi.encodePacked(_entropy, address(this), msg.sender, ++nonce);
343     // eliminated if hash value mod 100 more than the survive percent
344     bool _survived = _survivalBps > (uint256(keccak256(_data)) % 1e4);
345     if (_survived) {
346         ++_survivorCount;
347     }
348 }
```

The gas cost of storage writes (`SSTORE`) is significant.

At L342, the storage variable `nonce` is being write once in each loop, and the for loop be repeated for up to 1000 times.

Create a local variable can save a lot of gas.

Recommendation

Change to:

```
341 for (uint256 i = 0; i < _remainingPlayerCount; ++i) {
342     bytes memory _data = abi.encodePacked(_entropy, address(this), msg.sender, nonce+i+1);
343     // eliminated if hash value mod 100 more than the survive percent
344     bool _survived = _survivalBps > (uint256(keccak256(_data)) % 1e4);
345     if (_survived) {
346         ++_survivorCount;
347     }
348 }
349 nonce += _remainingPlayerCount;
```

[S] Consider adding `unclaimedPrize` and `totalUnclaimedPrize` to give winners more time to claim

In the current design, winners must `claim` before next game starts. Otherwise, the prizes will be returned to the prize pool.

This makes the game must wait a fair amount of time before starting the next game. If the claim period is too short, when the network is congested, many users may lose their prize.

To solve this problem, we suggest:

- Add two variables, `gameInfo.unclaimedPrize` and `totalUnclaimedPrize`;
- Change `prizePoolInLatte()` to:

```
function prizePoolInLatte() public view returns (uint256) {
    uint256 _balance = IERC20Upgradeable(latte).balanceOf(address(this))
    if (totalUnclaimedPrize > _balance) {
        return 0;
    }
    return _balance - totalUnclaimedPrize;
  }
```

- Set `gameInfo.unclaimedPrize` and `totalUnclaimedPrize` in `_complete()`;
- Add parameter `gameId` to `claim()` and update `gameInfo[gameId].unclaimedPrize` and `totalUnclaimedPrize` in `claim()`;

If the claim period is designed to be limited, and the unclaimed prizes are supposed to be returned to the prize pool, a limit can still be added to the `claim()` function and we can add a new function named `returnUnclaimedPrize()` to return the unclaimed prizes.

With these changes, the claim period can unbundle from the interval of games, allowing games to run faster one after another.

[N] Redundant code in `initialize()`

https://github.com/latteswap-official/survival-game-contract/blob/4f3104597bb2f9acc5e67043807efc517e71e7d6/contracts/SurvivalGame.sol#L149-L151

```
149     gameId = 0;
150     nonce = 0;
151     lastUpdatedBlock = 0;
```

Setting `uint` variables to `0` is redundant as they default to `0`.

Recommendation

Consider removing the above code or change to:

```
149     gameId = 1;
150     nonce = 1;
151     lastUpdatedBlock = block.timestamp;
```

This will lower the gas cost for the first user.

[N] Misleading variable names

https://github.com/latteswap-official/survival-game-contract/blob/4f3104597bb2f9acc5e67043807efc517e71e7d6/contracts/SurvivalGame.sol#L159-

L167

```
/// @dev only the one having a OPERATOR_ROLE can continue an execution
  modifier onlyOper() {
    require(hasRole(OPERATOR_ROLE, _msgSender()), "SurvivalGame::onlyOper::only OPERATOR role");
    require(
      uint256(block.timestamp) - lastUpdatedBlock >= operatorCooldown,
      "SurvivalGame::onlyOper::OPERATOR should not proceed the game consecutively"
    );
    _;
  }
```

`lastUpdatedBlock` is actual a timestamp.

Consider renaming to `lastUpdatedTimestamp`.

[N] Misleading comments

https://github.com/latteswap-official/survival-game-contract/blob/4f3104597bb2f9acc5e67043807efc517e71e7d6/contracts/SurvivalGame.sol#L40-L41

```
    // Minimum required blocks before operator can execute function again
    uint256 public operatorCooldown;
```

`operatorCooldown` is in seconds, but the comment says it's in blocks.

https://github.com/latteswap-official/survival-game-contract/blob/4f3104597bb2f9acc5e67043807efc517e71e7d6/contracts/SurvivalGame.sol#L343-L344

```
// eliminated if hash value mod 100 more than the survive percent
bool _survived = _survivalBps > (uint256(keccak256(_data)) % 1e4);
```

Should be `mod 10000`.

[L] `Ownable` is redundant

https://github.com/latteswap-official/survival-game-contract/blob/4f3104597bb2f9acc5e67043807efc517e71e7d6/contracts/SurvivalGame.sol#L25-L25

The `SurvivalGame.sol` contract is already using `AccessControl`, and `onlyOwner` is not being used.

Therefore, `Ownable` can be removed to make the code simpler and save some gas.

[N] Inconsistent use of `_msgSender()`

Direct use of `msg.sender` vs internal call of `_msgSender()`.

https://github.com/latteswap-official/survival-game-contract/blob/4f3104597bb2f9acc5e67043807efc517e71e7d6/contracts/SurvivalGame.sol#L154-L155

```
154 _setupRole(DEFAULT_ADMIN_ROLE, _msgSender());
155 _setupRole(OPERATOR_ROLE, _msgSender());
```

https://github.com/latteswap-official/survival-game-contract/blob/4f3104597bb2f9acc5e67043807efc517e71e7d6/contracts/SurvivalGame.sol#L170-L173

```
170 modifier onlyEntropyGenerator() {
171     require(msg.sender == address(entropyGenerator), "SurvivalGame::onlyEntropyGenerator::only entrc
172     _;
173 }
```

[L] `start()` Redundant code

https://github.com/latteswap-official/survival-game-contract/blob/4f3104597bb2f9acc5e67043807efc517e71e7d6/contracts/SurvivalGame.sol#L266-L271

```
266 function start() external onlyOper onlyOpened {
267     gameInfo[gameId].status = GameStatus.Processing;
268     _requestRandomNumber();
269     lastUpdatedBlock = block.timestamp;
270     emit LogSetGameStatus(gameId, "Processing");
271 }
```

L269 is redundant as `_proceed()` will update `lastUpdatedBlock` at L434.

[G] Cache storage variables in the stack can save gas

For the storage variables that will be accessed multiple times, cache them in the stack can save ~100 gas from each extra read (`SLOAD` after Berlin).

For example:

- `gameId` in `_proceed()`

  https://github.com/latteswap-official/survival-game-contract/blob/4f3104597bb2f9acc5e67043807efc517e71e7d6/contracts/SurvivalGame.sol#L423-L435

```
423 function _proceed(uint256 _entropy) internal {
424     uint8 _nextRoundNumber = gameInfo[gameId].roundNumber.add(1);
425     roundInfo[gameId][_nextRoundNumber].entropy = _entropy;
426     emit LogSetEntropy(gameId, _nextRoundNumber, _entropy);
```

```
427
428         gameInfo[gameId].roundNumber = _nextRoundNumber;
429         emit LogSetRoundNumber(gameId, _nextRoundNumber);
430
431         gameInfo[gameId].status = GameStatus.Started;
432         emit LogSetGameStatus(gameId, "Started");
433
434         lastUpdatedBlock = block.timestamp;
435     }
```

- `gameId` in `check()`
- `gameId` in `buy()`

[S] Consider adding a function to re-request randomness when VRF fulfillment failed

Per the Chainlink VRF document: https://docs.chain.link/docs/vrf-security-considerations/#fulfillrandomness-must-not-revert

> If your fulfillRandomness implementation reverts, the VRF service will not attempt to call it a second time.

Even though it's unlikely for the `consumeRandomNumber()` function to revert, but when it happens, or if VRF fulfillRandomness failed for other reasons, the whole game will be stuck in `Processing` status.

Therefore, we suggest adding a function to re-request randomness or force fulfillRandomness as last resort.