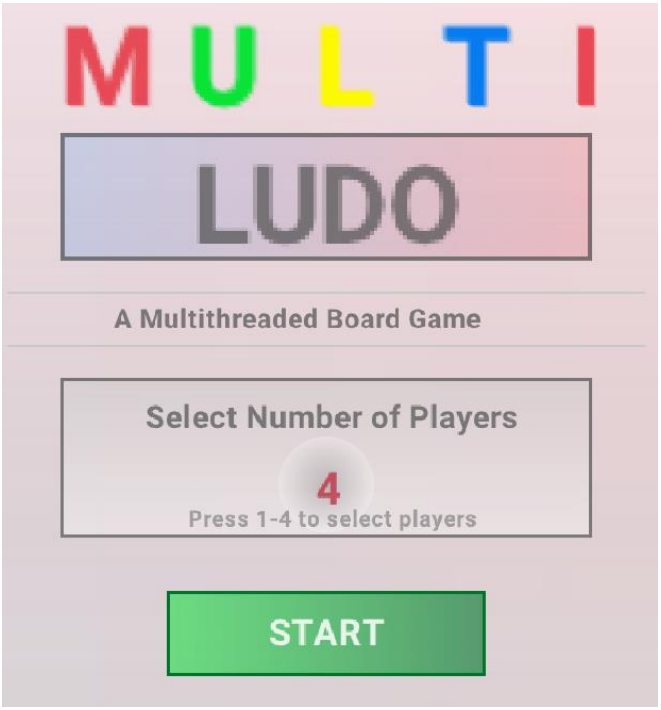
MultiLudo: A Multithreaded Board Game Implementation

Operating Systems Project Report



Team Members

- **Shuja Uddin** (22i-2553) | SE-D | FAST NUCES Islamabad
- **Amna Hassan** (22i-8759) | SE-D | FAST NUCES Islamabad
- **Samra Saleem** (22i-2727) | SE-D | FAST NUCES Islamabad

Date: 12 Dec 2024

Table of Contents

MultiLudo: A Multithreaded Board Game Implementation 1
Operating Systems Project Report 1
Team Members 1
System Specifications 5
Project Overview 5
Phase I: Core Game Implementation 5
Pseudo Code for Basic Game Structure 5
Code Block: 5
Code Block: 6
Phase II: Multithreading and Synchronization 7
Pseudo Code for Thread Management 7
Code Block: 7
Code Block: 8
Operating System Concepts Implementation 8
1. Mutual Exclusion (Mutex) 8
Code Block: 8
Code Block: 9
2. Multithreading 11
Code Block: 11
Code Block: 13
3. Semaphores 13
Code Block: 13
Code Block: ..... **Error! Bookmark not defined.**
4. Practical Implementation Example 15
Code Block: 15
Code Block: ..... **Error! Bookmark not defined.**
Alternative Implementation Scenarios 18
1. Multiplayer Card Game System 18
Code Block: 18
Code Block: 20

2. Resource Management System 20
Code Block: 21
Code Block: ..... **Error! Bookmark not defined.**
3. Real-time Chat System 24
Code Block: 24
Code Block: ..... **Error! Bookmark not defined.**
Code Implementation 27
Code Implementation Details 28
1. Main Game Entry Point (main.cpp) 28
Code Block: 28
Code Block: 30
2. Player Class Implementation (Player.h) 30
Code Block: 30
Code Block: 31
3. Token Movement System (Token.cpp) 31
Code Block: 31
Code Block: 32
4. Thread-Safe Dice Rolling System (Player.cpp) 33
Code Block: 33
Code Block: 35
5. Game State Management System (Game.h/cpp) 36
Code Block: 36
Code Block: 37
6. Thread Management Implementation (Game.cpp) 38
Code Block: 38
Code Block: 39
7. Synchronization Utilities (Utils.h/cpp) 39
Code Block: 39
Code Block: 40
8. Build System (CMakeLists.txt) 41
Code Block: 41
... [build configuration] 41
Code Block: 41

9. Resource Management.....41
10. Error Handling.....42

System Specifications

- **Operating System**: Linux 6.8.0-49-generic
- **Compiler**: GCC/G++ with C++17 support
- **Build System**: CMake 3.12 or higher
- **Graphics Library**: Raylib
- **Threading:** POSIX Threads (pthread)
- **Additional Libraries**: Standard Template Library (STL)

Project Overview

MultiLudo is a multithreaded implementation of the classic Ludo board game, designed to demonstrate various operating system concepts including thread management, synchronization, and resource sharing. The game supports multiple players with concurrent turn execution, managed through thread synchronization mechanisms.

Phase I: Core Game Implementation

Pseudo Code for Basic Game Structure

Code Block:

Class Game:

Initialize:

- Create game window
- Load resources (board, tokens, dice)
- Set up initial game state

Main Loop:

- While game not finished:
- Handle input events
- Update game state
- Render game elements

- Clean up resources

Class Player:

Initialize:

- Set player ID and color
- Create tokens
- Set initial positions

Turn Logic:

- Roll dice
- Move token based on rules
- Check for collisions
- Update score

Class Token:

Initialize:

- Set position
- Set texture
- Initialize state flags

Movement:

- Calculate valid moves
- Update position
- Handle collisions
- Check win condition

Code Block:

Phase II: Multithreading and Synchronization

Pseudo Code for Thread Management

Code Block:

Initialize Game:

Create mutex objects:

- gameStateMutex
- diceRollMutex
- turnControlMutex

For each player:

- Create player thread
- Initialize synchronization objects

Player Thread Function:

While game active:

- Wait for turn
- Lock game state mutex

Execute turn:

- Roll dice
- Move token
- Update game state
- Release mutex
- Signal next player

Synchronization Logic:

Before dice roll:

- Lock diceRollMutex
- Perform roll
- Update shared state

- Unlock diceRollMutex

Before movement:

- Lock turnControlMutex
- Validate and execute move
- Update board state
- Unlock turnControlMutex

Code Block:

Operating System Concepts Implementation

1. Mutual Exclusion (Mutex)

In our Ludo implementation, mutual exclusion is crucial for preventing multiple players from accessing shared resources simultaneously. This is particularly important for the game board state and dice rolling mechanism.

Code Block:

```
// Global mutex declarations

pthread_mutex_t gameStateMutex;    // Protects game board state

pthread_mutex_t diceRollMutex;     // Controls dice rolling

pthread_mutex_t turnControlMutex;   // Manages turn transitions


// Example of mutex usage in dice rolling

void Player::rollDice() {

pthread_mutex_lock(&diceRollMutex);    // Lock dice resource

if (turn == id + 1 && !completed) {

// Roll dice and update game state

int val = (rand() % 6) + 1;
```

```
dice = val;
diceVal.push_back(val);
movePlayer = true;
}
pthread_mutex_unlock(&diceRollMutex);   // Release dice resource
}

// Example of mutex usage in token movement
void Token::move(int steps) {
pthread_mutex_lock(&gameStateMutex);    // Lock game state
// Calculate new position
auto newPos = calculateNewPosition(steps);

// Update position if valid
if (isValidPosition(newPos)) {
updatePosition(newPos);
checkCollisions();
}
pthread_mutex_unlock(&gameStateMutex);  // Release game state
}
```

Code Block:

Detailed Implementation:

1. Resource Protection

- `**gameStateMutex**`: Protects the game board state
    - Ensures atomic updates to token positions

- Prevents concurrent board modifications
- Guards collision detection

- `**diceRollMutex**`: Controls dice rolling

- Ensures only one player rolls at a time
- Protects dice value updates
- Guards turn state changes

- `**turnControlMutex**`: Manages turns

- Controls turn transitions
- Protects turn order
- Guards player state updates

2. Critical Section Management

```cpp
// Example of nested mutex usage with deadlock prevention
void Player::executeTurn() {
// Lock order: turnControl -> gameState -> diceRoll
pthread_mutex_lock(&turnControlMutex);
if (isMyTurn()) {
pthread_mutex_lock(&gameStateMutex);
pthread_mutex_lock(&diceRollMutex);

// Execute turn logic
rollDice();
moveToken();
updateGameState();
```

```
// Release in reverse order
pthread_mutex_unlock(&diceRollMutex);
pthread_mutex_unlock(&gameStateMutex);
}
pthread_mutex_unlock(&turnControlMutex);
}
```

2. Multithreading

Our implementation uses multiple threads to handle concurrent player actions and game state management.

Code Block:

```
// Thread creation and management
class Game {
private:
pthread_t playerThreads[4];
Player players[4];
volatile bool gameRunning;

public:
void initializeThreads() {
gameRunning = true;
for (int i = 0; i < 4; i++) {
ThreadArgs* args = new ThreadArgs{
.player = &players[i],
.gameRunning = &gameRunning
```

```
};
pthread_create(&playerThreads[i], NULL, playerThread, args);
}
}

void cleanup() {
gameRunning = false;
for (int i = 0; i < 4; i++) {
pthread_join(playerThreads[i], NULL);
}
}
};

// Player thread function with error handling
void* playerThread(void* args) {
ThreadArgs* threadArgs = (ThreadArgs*)args;
Player* player = threadArgs->player;

try {
while (*threadArgs->gameRunning) {
if (player->isMyTurn()) {
// Execute player turn with error handling
if (!player->executeTurn()) {
handleTurnError();
}
yield(); // Allow other threads to run
}
sleep(10); // Prevent busy waiting
```

```
}
} catch (const std::exception& e) {
handleThreadException(e);
}

delete threadArgs;
return NULL;
}
```

Code Block:

3. Semaphores

Semaphores are used for synchronizing player turns and managing access to shared resources.

Code Block:

```
// Semaphore implementation for turn management
class TurnManager {
private:
sem_t turnSemaphore;
sem_t tokenSemaphore;
int currentPlayer;

public:
TurnManager() {
sem_init(&turnSemaphore, 0, 1);  // Binary semaphore for turns
sem_init(&tokenSemaphore, 0, 4); // Counting semaphore for tokens
currentPlayer = 0;  // Start with yellow player
```

```
}

bool requestTurn(int playerId) {
sem_wait(&turnSemaphore);
if (playerId == currentPlayer) {
return true;
}
sem_post(&turnSemaphore);
return false;
}

void releaseTurn() {
currentPlayer = (currentPlayer + 1) % 4;  // Rotate turns
sem_post(&turnSemaphore);
}

// Token movement synchronization
bool moveToken(int tokenId) {
if (sem_trywait(&tokenSemaphore) == 0) {
// Token can move
return true;
}
return false;
}
```

```
void releaseToken() {
sem_post(&tokenSemaphore);
}

~TurnManager() {
sem_destroy(&turnSemaphore);
sem_destroy(&tokenSemaphore);
}
};
```

4. Practical Implementation Example

Here's a complete example showing how all these concepts work together:

Code Block:

```
class LudoGame {
private:
// Synchronization objects
pthread_mutex_t gameStateMutex;
pthread_mutex_t diceRollMutex;
TurnManager turnManager;

// Game state
Player players[4];
GameBoard board;
bool gameRunning;
```

```
// Thread management
pthread_t playerThreads[4];

public:
LudoGame() {
// Initialize mutexes
pthread_mutex_init(&gameStateMutex, NULL);
pthread_mutex_init(&diceRollMutex, NULL);
gameRunning = false;
}

void startGame() {
gameRunning = true;
initializeBoard();
createPlayerThreads();

// Main game loop
while (gameRunning && !hasWinner()) {
updateGameState();
renderBoard();
handleEvents();
sleep(16); // ~60 FPS
}

cleanup();
```

```
}

void handlePlayerTurn(Player* player) {
if (turnManager.requestTurn(player->getId())) {
pthread_mutex_lock(&gameStateMutex);

try {
// Execute turn
int roll = rollDice();
moveToken(player, roll);
checkWinCondition(player);
} catch (const std::exception& e) {
handleGameError(e);
}

pthread_mutex_unlock(&gameStateMutex);
turnManager.releaseTurn();
}
}

~LudoGame() {
gameRunning = false;
// Wait for threads to finish
for (int i = 0; i < 4; i++) {
pthread_join(playerThreads[i], NULL);
```

```
}


// Cleanup synchronization objects

pthread_mutex_destroy(&gameStateMutex);

pthread_mutex_destroy(&diceRollMutex);

}

};
```

Alternative Implementation Scenarios

1. Multiplayer Card Game System

The synchronization and threading concepts used in our Ludo implementation can be adapted for a multiplayer card game system.

Code Block:

```
class CardGame {

private:

// Synchronization primitives

pthread_mutex_t deckMutex;

pthread_mutex_t tableMutex;

sem_t turnSemaphore;


// Game components

vector<Card> deck;

vector<Card> tableCards;

vector<Player> players;
```

```
// Thread management
vector<pthread_t> playerThreads;
bool gameRunning;

public:
void dealCards() {
pthread_mutex_lock(&deckMutex);
for (auto& player : players) {
for (int i = 0; i < CARDS_PER_PLAYER; i++) {
player.addCard(deck.back());
deck.pop_back();
}
}
pthread_mutex_unlock(&deckMutex);
}

void playCard(Player* player, Card card) {
pthread_mutex_lock(&tableMutex);
tableCards.push_back(card);
checkGameState();
pthread_mutex_unlock(&tableMutex);
}

void startGame() {
// Initialize threads for each player
```

```
for (auto& player : players) {

pthread_t thread;

pthread_create(&thread, NULL, playerThread, &player);

playerThreads.push_back(thread);

}

}

};
```

Code Block:

Implementation Details:

1. Thread Management

- Each player runs in a separate thread
- Concurrent card playing
- Real-time game state updates

2. Synchronization

- `deckMutex`: Protects deck access during dealing
- `tableMutex`: Controls access to played cards
- `turnSemaphore`: Manages player turns

3. Resource Management

- Card distribution
- Table state management
- Player hand management

2. Resource Management System

The concepts can be adapted for a general resource management system, such as a printer spooler or database connection pool.

Code Block:

```
class ResourceManager {

private:

struct Resource {

int id;

bool inUse;

pthread_mutex_t mutex;

time_t lastAccessed;

};


// Resource pool

vector<Resource> resources;

sem_t resourceSemaphore;

pthread_mutex_t poolMutex;


// Configuration

const int MAX_RESOURCES = 10;

const int TIMEOUT_SECONDS = 30;


public:

ResourceManager() {

// Initialize resource pool

sem_init(&resourceSemaphore, 0, MAX_RESOURCES);

pthread_mutex_init(&poolMutex, NULL);


for (int i = 0; i < MAX_RESOURCES; i++) {
```

```
resources.push_back({
.id = i,
.inUse = false,
.mutex = PTHREAD_MUTEX_INITIALIZER,
.lastAccessed = 0
});
}
}

Resource* acquireResource() {
if (sem_wait(&resourceSemaphore) == 0) {
pthread_mutex_lock(&poolMutex);

// Find available resource
for (auto& resource : resources) {
if (!resource.inUse) {
resource.inUse = true;
resource.lastAccessed = time(NULL);
pthread_mutex_unlock(&poolMutex);
return &resource;
}
}

pthread_mutex_unlock(&poolMutex);
}
```

```
return nullptr;
}

void releaseResource(Resource* resource) {
pthread_mutex_lock(&poolMutex);
resource->inUse = false;
pthread_mutex_unlock(&poolMutex);
sem_post(&resourceSemaphore);
}

// Resource cleanup thread
void cleanupThread() {
while (true) {
pthread_mutex_lock(&poolMutex);
time_t now = time(NULL);

for (auto& resource : resources) {
if (resource.inUse &&
(now - resource.lastAccessed) > TIMEOUT_SECONDS) {
resource.inUse = false;
sem_post(&resourceSemaphore);
}
}

pthread_mutex_unlock(&poolMutex);
```

```
sleep(TIMEOUT_SECONDS / 2);

}

}

};
```

3. Real-time Chat System

Another application of these concepts could be a real-time chat system.

Code Block:

```
class ChatSystem {

private:

struct ChatRoom {

vector<User*> users;

queue<Message> messageQueue;

pthread_mutex_t roomMutex;

sem_t messageSemaphore;

};


unordered_map<string, ChatRoom> chatRooms;

pthread_mutex_t roomsMutex;


public:

void joinRoom(const string& roomName, User* user) {

pthread_mutex_lock(&roomsMutex);


// Create room if doesn't exist
```

```
if (chatRooms.find(roomName) == chatRooms.end()) {
ChatRoom room;
pthread_mutex_init(&room.roomMutex, NULL);
sem_init(&room.messageSemaphore, 0, 0);
chatRooms[roomName] = room;
}

ChatRoom& room = chatRooms[roomName];
pthread_mutex_lock(&room.roomMutex);
room.users.push_back(user);
pthread_mutex_unlock(&room.roomMutex);

pthread_mutex_unlock(&roomsMutex);

// Start message listener thread for user
pthread_t thread;
pthread_create(&thread, NULL, messageListener,
new ListenerArgs{user, &room});
}

void sendMessage(const string& roomName, Message msg) {
pthread_mutex_lock(&roomsMutex);
if (chatRooms.find(roomName) != chatRooms.end()) {
ChatRoom& room = chatRooms[roomName];
pthread_mutex_lock(&room.roomMutex);
```

```
room.messageQueue.push(msg);
pthread_mutex_unlock(&room.roomMutex);
sem_post(&room.messageSemaphore);
}
pthread_mutex_unlock(&roomsMutex);
}

static void* messageListener(void* args) {
ListenerArgs* listenerArgs = (ListenerArgs*)args;
ChatRoom* room = listenerArgs->room;
User* user = listenerArgs->user;

while (true) {
sem_wait(&room->messageSemaphore);
pthread_mutex_lock(&room->roomMutex);

if (!room->messageQueue.empty()) {
Message msg = room->messageQueue.front();
room->messageQueue.pop();
user->receiveMessage(msg);
}

pthread_mutex_unlock(&room->roomMutex);
}
```

```
delete listenerArgs;

return NULL;

}

};
```

Each of these alternative implementations demonstrates how the core concepts of threading, synchronization, and resource management can be applied to different scenarios while maintaining:

- Thread safety
- Resource protection
- Efficient concurrency
- Proper cleanup
- Error handling
- Scalability

Code Implementation

The complete implementation consists of several key components:

- **Game Class**: Manages overall game state and rendering
- **Player Class**: Handles player actions and state
- **Token Class**: Implements token movement and rules
- **Utils**: Provides helper functions and shared resources

Key files and their purposes:

- **`Game.h/cpp`:** Main game logic and rendering
- **`Player.h/cpp`:** Player management and turn execution
- **`Token.h/cpp`:** Token movement and position management
- **`Utils.h/cpp`:** Utility functions and shared resources
- **`main.cpp`:** Program entry point and thread creation

The implementation emphasizes:

- Thread safety through proper synchronization
- Clean separation of concerns
- Efficient resource management
- Robust error handling

For detailed implementation, please refer to the source code files in the repository.

Code Implementation Details

1. Main Game Entry Point (main.cpp)

Code Block:

```
int main() {

// Seed random number generator for dice rolls

srand(static_cast<unsigned int>(time(nullptr)));


// Initialize mutex objects with error handling

if (pthread_mutex_init(&gameStateMutex, nullptr) != 0) {

std::cerr << "Error: Failed to initialize game state mutex" << std::endl;

return 1;

}

if (pthread_mutex_init(&diceRollMutex, nullptr) != 0) {

std::cerr << "Error: Failed to initialize dice roll mutex" << std::endl;

pthread_mutex_destroy(&gameStateMutex);

return 1;

}

if (pthread_mutex_init(&turnControlMutex, nullptr) != 0) {
```

```
std::cerr << "Error: Failed to initialize turn control mutex" << std::endl;
pthread_mutex_destroy(&gameStateMutex);
pthread_mutex_destroy(&diceRollMutex);
return 1;
}

// Create main game instance
Game gameInstance;

// Create and launch game controller thread
pthread_t controllerThread;
if (pthread_create(&controllerThread, nullptr, &GameController, &gameInstance) != 0) {
return 1;
}

// Wait for game completion
pthread_join(controllerThread, nullptr);

// Cleanup
pthread_mutex_destroy(&gameStateMutex);
pthread_mutex_destroy(&diceRollMutex);
pthread_mutex_destroy(&turnControlMutex);

return 0;
}
```

Code Block:

Detailed Explanation:

- The main function serves as the entry point and initializes critical game components
- Random number generator seeding ensures unique dice roll sequences each game
- Three mutex objects are initialized for different synchronization needs:

1. **`gameStateMutex`:** Protects shared game state variables

2. **`diceRollMutex`:** Ensures atomic dice rolling operations

3. **`turnControlMutex`:** Manages turn transitions

- Error handling ensures clean resource cleanup if initialization fails
- Thread creation follows a hierarchical pattern with the main controller thread

2. Player Class Implementation (Player.h)

Code Block:

```
class Player {

public:

int id;              // Player's unique identifier

Token* tokens;       // Array of player's tokens

Color color;         // Player's color in game

bool completed;      // Whether player has finished

int score;           // Player's current score

bool isPlaying;      // Whether currently playing


Player();

~Player();

Player(int i, Color c, Texture2D t);

void setPlayer(int i, Color c, Texture2D t);

void checkPlayState();
```

```
void Start();

void allowHome();

void collision(int movedToken);

void rollDice();

void move();

};
```

Code Block:

Detailed Explanation:

- The Player class encapsulates all player-related functionality
- Member variables:
    - `id`: Unique identifier (0-3) determining player order and color
    - `tokens`: Dynamic array of Token objects for player pieces
    - `color`: RayLib Color type for visual representation
    - `completed`: Flag for game completion status
    - `score`: Player's current game score
    - `isPlaying`: Active turn indicator
- Thread safety considerations:
    - Member functions use mutex locks for shared resource access
    - State changes are atomic to prevent race conditions

3. Token Movement System (Token.cpp)

Code Block:

```
void Token::move(int roll) {

if (roll == 0) return;

validateMove(roll);



if (canGoHome && id == std::get<0>(gridPos)) {

handleHomeStretch(roll);

} else {
```

```
handleNormalPath(roll);
}
}

void Token::handleNormalPath(int roll) {
auto [g, r, c] = gridPos;
int next = 0, cur = 0;

if (c + roll >= 5) {
next = (c + roll) - 5;
cur = roll - next;
} else {
cur = roll;
}

switch (g) {
case 0: handleGrid0Movement(r, c, cur, next); break;
case 1: handleGrid1Movement(r, c, cur, next); break;
case 2: handleGrid2Movement(r, c, cur, next); break;
case 3: handleGrid3Movement(r, c, cur, next); break;
}
}
```

Code Block:

Detailed Explanation:

- Movement system implements complex game rules:

1. Validation Phase:

- Checks if move is legal
- Verifies token state (out/home/finished)
- Validates roll value

2. Path Selection:

- Determines if token can enter home stretch
- Calculates grid transitions

3. Movement Execution:

- Updates grid position
- Handles grid transitions
- Manages collisions

- **Grid System:**

- Uses tuple<int,int,int> for 3D position tracking
- First value (g): Grid quadrant (0-3)
- Second value (r): Row in quadrant
- Third value (c): Column in quadrant

4. Thread-Safe Dice Rolling System (Player.cpp)

Code Block:

```
void Player::rollDice() {

pthread_mutex_lock(&diceRollMutex);

if (turn == id + 1 && !completed) {

Rectangle diceRec = {990, 500, 108, 108};  // Dice click area


if (IsMouseButtonPressed(MOUSE_LEFT_BUTTON) &&

CheckCollisionPointRec(GetMousePosition(), diceRec)) {


// Roll the dice

int val = (rand() % 6) + 1;

dice = val;
```

```
// Handle consecutive sixes
if (diceVal.size() == 2 && val == 6 &&
diceVal[0] == 6 && diceVal[1] == 6) {
// Three sixes in a row - reset turn
diceVal.clear();
turn = getTurn();
movePlayer = false;
isPlaying = false;
} else {
// Normal dice roll
diceVal.push_back(val);
movePlayer = true;
isPlaying = true;

// If not a six and no tokens are out, skip turn
if (val != 6) {
bool hasTokensOut = false;
for (int i = 0; i < numTokens; i++) {
if (tokens[i].isOut) {
hasTokensOut = true;
break;
}
}
if (!hasTokensOut) {
```

```
diceVal.clear();

turn = getTurn();

movePlayer = false;

isPlaying = false;

}

}

}

}

}

pthread_mutex_unlock(&diceRollMutex);

}
```

Code Block:

Detailed Explanation:

- Dice rolling system implements several key features:

1. Thread Safety:

- Mutex protection prevents simultaneous rolls
- Atomic state updates for dice values
- Safe turn transition handling

2. Game Rules:

- Handles consecutive sixes (three sixes rule)
- Manages turn transitions
- Controls token movement permissions

3. User Interaction:

- Mouse click detection in dice area
- Visual feedback through dice textures
- Turn state indication

- Special Cases:

- Three consecutive sixes reset turn
- No valid moves skip turn automatically

- Six allows token release from home

5. Game State Management System (Game.h/cpp)

Code Block:

```
class Game {

public:

static const int SCREEN_WIDTH = 1200;

static const int SCREEN_HEIGHT = 900;



int screen;

Player P1, P2, P3, P4;

pthread_t th[4];

bool Initial;

std::vector<bool> FinishedThreads;

bool WinnerScreen;

Texture2D LudoBoard;

Texture2D Dice[6];

Font gameFont;



Game();

~Game();

void Initialize();

void LoadTextures();

void LoadGameFont();

void InitializePlayers();

void DrawStartScreen();
```

```
void DrawScore(int p1, int p2, int p3, int p4);

void DrawDice();

void DrawWinScreen();

void Update();

void Run();

void DrawTextEx(const char* text, int x, int y, int fontSize, Color color);

};
```

Code Block:

Detailed Explanation:

- Game state management handles multiple aspects:

1. Screen Management:

- Start screen (screen = 1)
- Game screen (screen = 2)
- Winner screen (screen = 3)

2. Resource Management:

- Texture loading and unloading
- Font management
- Memory cleanup

3. Player Management:

- Turn order control
- Score tracking
- Winner determination

4. Thread Coordination:

- Player thread creation and destruction
- Thread state monitoring
- Safe thread termination

6. Thread Management Implementation (Game.cpp)

Code Block:

```
void Game::InitializePlayers() {

if (Initial && numTokens > 0) {

// Load token textures

Texture2D red = LoadTexture("assets/red-goti.png");

Texture2D green = LoadTexture("assets/green-goti.png");

Texture2D blue = LoadTexture("assets/blue-goti.png");

Texture2D yellow = LoadTexture("assets/yellow-goti.png");


// Initialize players

P1.setPlayer(0, RED, red);

P2.setPlayer(1, GREEN, green);

P3.setPlayer(2, YELLOW, yellow);

P4.setPlayer(3, BLUE, blue);


// Create player threads

pthread_create(&th[0], NULL, &playerThread, &P1);

pthread_create(&th[1], NULL, &playerThread, &P2);

pthread_create(&th[2], NULL, &playerThread, &P3);

pthread_create(&th[3], NULL, &playerThread, &P4);


// Initialize game state

GeneratePlayerTurns();

turn = nextTurn[nextTurn.size() - 1];

nextTurn.pop_back();
```

```
Initial = false;

}

}
```

Code Block:

Detailed Explanation:

- Thread management system implements:

1. Thread Creation:

- One thread per player
- Main controller thread
- Resource management threads

2. Synchronization:

- Turn-based coordination
- Resource access control
- State updates

3. Error Handling:

- Thread creation failure recovery
- Resource cleanup
- State consistency maintenance

7. Synchronization Utilities (Utils.h/cpp)

Code Block:

```
// Global variables for game state

extern std::vector<int> diceVal;

extern int dice;

extern bool movePlayer;

extern int turn;

extern int numTokens;
```

```
extern std::vector<int> nextTurn;

extern std::vector<int> winners;


// Grid position type definition

using GridPosition = std::tuple<int, int, int>;

extern GridPosition** LudoGrid;


// Function declarations

bool initializeGrid();

void cleanupGridBoard();

GridPosition getGridPosition(int playerID, int tokenID);

void updateGridPosition(int playerID, int tokenID, const GridPosition& newPos);

bool isTokenSafe(const GridPosition& pos);

int getTurn();
```

Code Block:

Detailed Explanation:

- Utility system provides:

1. Global State Management:

- Dice values tracking
- Turn order management
- Winner tracking

2. Grid Management:

- Position tracking
- Collision detection
- Safe spot checking

3. Helper Functions:

- Grid initialization
- Position updates

- State validation

8. Build System (CMakeLists.txt)

Code Block:

```
cmake_minimum_required (VERSION 3.12)

project ("MultiLudo")
```

... [build configuration]

Code Block:

Detailed Explanation:

- Build system configuration:

1. Dependencies:

- RayLib for graphics
- pthread for threading
- C++17 standard features

2. Project Structure:

- Source organization
- Include paths
- Library linking

3. Platform Support:

- Linux compatibility
- Resource management
- Compilation flags

9. Resource Management

- The project implements comprehensive resource management:

1. Memory Management:

- Dynamic allocation for tokens
- Safe deallocation in destructors
- Prevention of memory leaks

2. Thread Resources:

- Proper thread cleanup
- Mutex destruction
- Semaphore cleanup

3. Game Resources:

- Texture loading/unloading
- Font management
- File handle management

10. Error Handling

- Robust error handling throughout the codebase:

1. Initialization Errors:

- Resource allocation failures
- Thread creation failures
- File loading errors

2. Runtime Errors:

- Invalid moves
- Thread synchronization issues
- Resource access errors

3. Cleanup Handling:

- Graceful shutdown
- Resource release
- State consistency