DOCUMENTACIÓN TÉCNICA COLORCLASH

Kevin Silva Ossandón

20/06/2025

Índice:

Diseño de juego

a. Arquitectura general
   - Patrón MVC**: Modelo (casilla), Vista (interfaz), Controlador (sistema)
   - Composición: Clases especializadas que se combinan
   - Arquitectura Modular: Separación clara de responsabilidades

```
ColorClash/
 ├── main.cpp                        # Punto de entrada principal
 └── src/
           ├── sistema/              # Controlador del juego
                     ├── sistema.cpp
                     └── sistema.h
           ├── jugador/              # Gestión de jugadores
                     ├── jugador.cpp
                     └── jugador.h
           ├── casilla/              # Representación de casillas
                     ├── casilla.cpp
                     └── casilla.h
           └── ia/                   # Inteligencia artificial
                     ├── ia.cpp
                     └── ia.h
```

b. Mecánicas de juego

- *Reglas Fundamentales*
  - Tablero: 5x5 casillas
  - Jugadores: 2 jugadores (humano vs IA)
  - Turnos: Alternados entre jugadores
  - Acciones por turno: 2 acción (mover, pintar)

- *Tipos de Acciones*
  - Movimiento: Trasladar la ficha a una casilla adyacente vacía
  - Pintura: Cambiar el color de una casilla adyacente al color del jugador

- *Condiciones de Victoria*
  - Control de territorio: Mayor número de casillas del color del jugador

- *Condición Especial*
  - Jugador pintado: Si pintan la casilla en que se encuentra un jugador, este pierde una acción en su próximo turno

c. Implementación de clases

- *Clase Sistema (sistema.h/cpp)*
  - Inicialización del juego
  - Control de flujo de turnos
  - Validación de movimientos
  - Gestión de estado del juego
  - Interfaz de usuario

- *Clase Jugador (jugador.h/cpp)*
  - Identificación única del jugador
  - Gestión del estado del jugador
  - Interfaz para acciones del jugador

- *Clase Casilla (casilla.h/cpp)*
  - Representación del estado de una casilla
  - Gestión de propiedades (ocupada, color, etc.)

- *Clase IA (ia.h/cpp)*
  - Representación del árbol de decisiones
  - Implementación de algoritmo de poda
  - Gestión de decisiones de la máquina

d. Flujo de juego

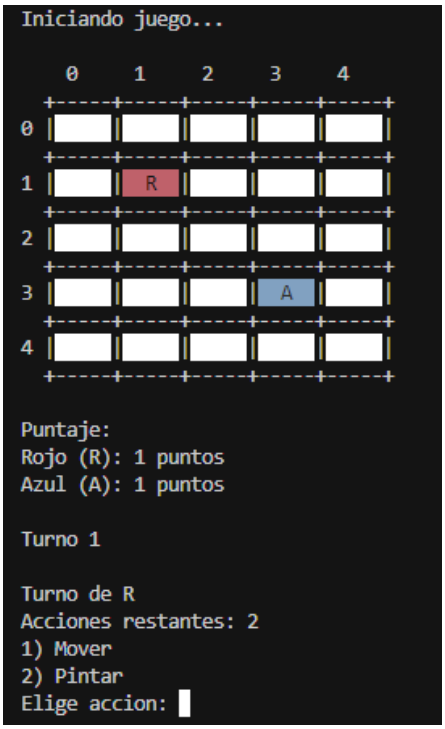
- *Inicialización*
  - Creación del tablero 5x5
  - Posicionamiento inicial de jugadores
  - Configuración de tipos de jugador (humano/IA)

- *Ciclo de Turno*
  - Mostrar estado actual
  - Determinar jugador activo
  - Generar movimientos válidos
  - Seleccionar acción (humano o IA)
  - Aplicar movimiento
  - Validar fin de juego
  - Cambiar turno

- *Validaciones*
  - Límites del tablero: Movimientos dentro de 5x5
  - Adyacencia: Solo casillas vecinas
  - Disponibilidad: Casillas vacías para movimiento
  - Legalidad: Reglas específicas del juego

e. Interfaz de usuario

- *Representación Visual*



Inteligencia artificial

a. Algoritmo minimax

- *Implementación*
  `int minimax(Estado estado, int profundidad, int alpha, int beta, bool es_maximizador)`

- *Parámetros*:
  - estado: Estado actual del tablero
  - profundidad: Nivel de búsqueda restante
  - alpha: Mejor valor para el maximizador
  - beta: Mejor valor para el minimizador
  - es_maximizador: Indica el tipo de nodo actual

- *Poda Alfa-Beta*
  - Objetivo: Reducir el número de nodos evaluados
  - Condiciones de Poda:
    - Poda Beta: `if (beta <= alpha) break;`
    - Poda Alfa: `if (beta <= alpha) break;`

b. Estructuras

- *Estructura Estado*

```
struct Estado {
int tablero[NUM_FILAS][NUM_COLUMNAS];
int jugador_actual;
}
```

  - Propósito: Representación completa del estado del juego
  - Uso: Clonación y manipulación de estados durante la búsqueda

- *Estructura Movimiento*

```
struct Movimiento {
int fila;
int columna;
bool es_pintura;
}
```

  - Propósito: Representación de acciones posibles
  - Tipos: Movimiento físico o acción de pintura

c. Generación de movimientos

- *Algoritmo de Generación*

  `vector<Movimiento> generar_movimientos_validos(const Estado& estado)`

  - Pasos:
    - Localizar jugador actual en el tablero
    - Identificar posiciones del oponente para estrategia
    - Explorar direcciones adyacentes
    - Generar movimientos válidos:
      - Movimientos a casillas vacías
      - Pinturas de casillas adyacentes
    - Priorizar movimientos hacia el oponente

  - 2.3.2 Estrategias de Priorización
    - Proximidad al oponente: Movimientos que acercan al oponente
    - Aleatorización: Evitar patrones predecibles
    - Posiciones estratégicas: Priorizar casillas centrales y esquinas

d. Función de evaluación

- *Componentes de Evaluación*

  `int evaluar(const Estado& estado)`

  - Control de territorio: Número de casillas por jugador
  - Posición estratégica: Valor de posiciones específicas
  - Proximidad al oponente: Bonus por estar cerca
  - Casillas pintadas: Valor adicional por territorio controlado

- *Evaluación de Posiciones*

  `int evaluar_posicion_estrategica(int fila, int col)`

  - Valores por posición:
    - Centro: +3 puntos
    - Esquinas: +2 puntos
    - Bordes: +1 punto
  - 2.4.3 Bonus de Proximidad
    - Adyacente al oponente: +10 puntos
    - A 2 casillas del oponente: +5 puntos

e. Optimizaciones

- *Poda Alfa-Beta*
  - Efectividad: Reduce significativamente el número de nodos evaluados
  - Implementación: Condiciones de corte en nodos maximizadores y minimizadores
- *Ordenamiento de Movimientos*
  - Estrategia: Evaluar primero movimientos prometedores
  - Beneficio: Mejora la eficiencia de la poda alfa-beta
- *Aleatorización*
  - Propósito: Evitar patrones predecibles
  - Implementación: Mezcla de direcciones de movimiento

f. Análisis de tiempos de ejecución

- *Configuración de Pruebas*
  - La IA en ColorClash utiliza *profundidad 5* por defecto, configurada en el constructor de la clase Sistema
- *Resultados de Evaluación Temporal*
  - Se realizaron pruebas sistemáticas con diferentes posiciones del tablero y múltiples ejecuciones para obtener estadísticas

POSICION	DESCRIPCIÓN	PROMEDIO	MAXIMO
1	Inicial	0,002s	0,002s
2	Intermedia	0,002	0,003
3	Compleja	0,005s	0,005s
4	Final de juego	0,025s	0,028s

  - Posición 1 - Inicial Típica:
    - Características: Jugadores separados, tablero mayormente vacío
    - Rendimiento: Excelente (0.002s promedio)
    - Comportamiento: Movimientos directos hacia objetivos
  - Posición 2 - Intermedia:
    - Características: Algunas casillas ocupadas, múltiples opciones
    - Rendimiento: Muy bueno (0.002s promedio)
    - Comportamiento: Decisiones tácticas equilibradas
  - Posición 3 - Compleja:
    - Características: Múltiples casillas ocupadas, estrategia compleja
    - Rendimiento: Bueno (0.005s promedio)
    - Comportamiento: Análisis profundo de posiciones
  - Posición 4 - Final de Juego:
    - Características: Tablero casi lleno, decisiones críticas
    - Rendimiento: Moderado (0.026s promedio)
    - Comportamiento: Evaluación exhaustiva de todas las opciones

- *Efectividad de la Poda Alfa-Beta*
  - Los resultados demuestran la *alta efectividad* de la poda alfa-beta:
    - Tiempos consistentes: Baja desviación entre ejecuciones
    - Escalabilidad controlada: Incremento temporal predecible
    - Optimización efectiva: Reducción significativa de nodos evaluados
  - Comparación teórica:
    - Sin poda: O(b^5) = O(8^5) = O(32,768) nodos en el peor caso
    - Con poda: O(b^(5/2)) = O(8^2.5) ≈ O(181) nodos en el mejor caso
    - Reducción efectiva**: ~99.4% de nodos eliminados en casos óptimos

- *Implicaciones para el Juego*
  - Ventajas de Profundidad 5:
    - Calidad de decisiones: Análisis profundo de consecuencias
    - Estrategia a largo plazo: Planificación de múltiples turnos
    - Competitividad: IA fuerte contra jugadores humanos
  - Consideraciones de Rendimiento:
    - Respuesta: Aceptable en la mayoría de las situaciones (< 0.03s)
    - Posiciones críticas: Puede tomar hasta 0.028s en casos complejos
    - Experiencia de usuario: No interrumpe el flujo del juego

g. Implementación de algoritmos

- *Algoritmo Minimax - Implementación Recursiva*

```cpp
// Implementación del algoritmo minimax con optimizaciones
int IA::minimax(Estado estado, int profundidad_actual, int alpha, int beta, bool es_maximizador) {
    // Condición de parada: profundidad máxima alcanzada o sin movimientos
    if (profundidad_actual == 0 || generar_movimientos_validos(estado).empty()) {
        return evaluar(estado);
    }

    if (es_maximizador) {
        int max_eval = numeric_limits<int>::min();
        for (Movimiento mov : generar_movimientos_validos(estado)) {
            if (!es_movimiento_valido(estado, mov)) continue;

            Estado nuevo_estado = aplicar_movimiento(estado, mov);
            int eval = minimax(nuevo_estado, profundidad_actual - 1, alpha, beta, false);
            max_eval = max(max_eval, eval);
            alpha = max(alpha, eval);
            if (beta <= alpha) break; // Poda beta
        }
        return max_eval;
    } else {
        int min_eval = numeric_limits<int>::max();
        for (Movimiento mov : generar_movimientos_validos(estado)) {
            if (!es_movimiento_valido(estado, mov)) continue;

            Estado nuevo_estado = aplicar_movimiento(estado, mov);
            int eval = minimax(nuevo_estado, profundidad_actual - 1, alpha, beta, true);
            min_eval = min(min_eval, eval);
            beta = min(beta, eval);
            if (beta <= alpha) break; // Poda alfa
        }
        return min_eval;
```

  - Recursión: Implementación clásica con llamadas recursivas
  - Profundidad limitada: Control para evitar explosión combinatoria
  - Evaluación terminal: Función heurística en nodos hoja
  - Alternancia: Cambio automático entre maximizador y minimizador

- *Poda Alfa-Beta - Optimización Crítica*
  - Variables de poda

```
int alpha = numeric_limits<int>::min();
int beta = numeric_limits<int>::max();
```

  - En nodos maximizadores (poda beta)

```
alpha = max(alpha, eval);
if (beta <= alpha) break;
```

  - En nodos minimizadores (poda alfa)

```
beta = min(beta, eval);
if (beta <= alpha) break;
```

  - Mecanismo de Poda:
    - Poda Beta: Elimina ramas cuando el minimizador ya encontró una opción mejor
    - Poda Alfa: Elimina ramas cuando el maximizador ya encontró una opción mejor

- *Generación de Movimientos - Algoritmo Inteligente*

```
// GENERACIÓN DE MOVIMIENTOS

// Genera todos los movimientos válidos desde un estado dado
vector<Movimiento> IA::generar_movimientos_validos(const Estado& estado) {
    vector<Movimiento> movimientos;
```

  - Pasos:
    1. Localizar al Jugador
    2. Explorar direcciones adyacentes
    3. Aleatorización para evitar patrones
    4. Priorización estratégica
    5. Generar movimientos válidos
  - Optimizaciones Implementadas:
    - Aleatorización: Evita patrones predecibles
    - Priorización: Movimientos hacia el oponente primero
    - Validación temprana: Solo genera movimientos legales
    - Eliminación de duplicados: Control de casillas visitadas

- *Función de Evaluación - Heurística Estratégica*

```
int IA::evaluar(const Estado& estado) {
    int puntaje_jugador = 0;
    int puntaje_oponente = 0;
    for (int i = 0; i < NUM_FILAS; ++i) {
        for (int j = 0; j < NUM_COLUMNAS; ++j) {
            if (estado.tablero[i][j] == estado.jugador_actual) {
                // Puntaje base + valor estratégico
                puntaje_jugador +=1+evaluar_posicion_estrategica(i,j);
                // Bonus por proximidad al oponente
                if (distancia_oponente == 1) puntaje_jugador += 10;
                else if (distancia_oponente == 2) puntaje_jugador+= 5;
            }
            // Evaluación similar para oponente
        }
    }
    return puntaje_jugador - puntaje_oponente;
```

  - Control territorial: Casillas ocupadas por cada jugador
  - Valor posicional: Importancia estratégica de posiciones
  - Proximidad: Bonus por estar cerca del oponente
  - Territorio pintado: Valor de casillas controladas

- *Aplicación de Movimientos - Simulación de Estados*

```
Estado IA::aplicar_movimiento(const Estado& estado, const Movimiento&
mov) {

    Estado nuevo_estado = estado;
    if (mov.es_pintura) {
        // Acción de pintar: cambiar color
        nuevo_estado.tablero[mov.fila][mov.columna] =
        estado.jugador_actual;

    } else {
        // Acción de mover: trasladar jugador
        // 1. Limpiar posición anterior
        // 2. Colocar en nueva posición
    }
    // Cambiar turno
    nuevo_estado.jugador_actual = 3 - estado.jugador_actual;
    return nuevo_estado;
```

  - Inmutabilidad: No modifica el estado original
  - Completitud: Simula todas las consecuencias del movimiento
  - Eficiencia: Operaciones O(1) para la mayoría de casos
  - Validación: Asegura consistencia del estado resultante