Technical Documentation

Nine Man's Morris

Stjepan Knežević

My solution

To create this game, I used cocos2d-x game engine. It's a game engine made in C++ in which scripting is done in C++ programming language. To create all the art for this game I've used Aseprite. Aseprite is an animated sprite editor and pixel art tool. I have not used any other libraries outside of cocos2d-x game engine.

To create the game, I have defined some concepts: Piece, Spot, Board, Player, Turn, Stage, Input and Checker.

Classes

Piece

Represents a soldier/piece on the board. It inherits from the cocos2d::Node class. It remembers if it's been placed or selected, has movement logic and contains the sprites for the piece and selected piece.

"isPlaced" method indicates whether the piece has been placed on the board, and setPlaced sets this state.

"moveToPosition" method moves the piece to the specified position on the game board.

"select" method toggles the selection state of the piece.

Spot

Spot represents the spot on the board where a piece can either be placed or moved to. It inherits from the cocos2d::Node class. Spot contains the reference to the occupying piece, logic for adding or removing the piece to and from the spot. It also contains the list of spots that it's connected to and logic for connecting the spots together. It also has the sprite for the spot and sprite for if the spot is available.

Methods 'placePiece', 'isOccupied', 'getOccupyingPiece', and 'removePiece' manage the presence and movement of pieces on the spot.

'connectSpot' and 'connectSpots' methods connect spots.

'showAvailable 'and 'showAvailableConnectedSpots' visually indicate if the spot is available or connected to other spots.

Board

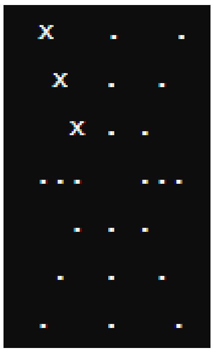
Board contains the references to all the spots on the board. It inherits from the cocos2d::Node class. When a board is created it creates the Spots that are on the board and parents them to itself. The Board sets up itself and connects the spots in a Nine Man's Morris way. Board class contains the sprite for the board.

'init' method initializes the Board object with the provided board sprite and sets up the spots for the game.

'addSpot' method adds a spot to the board at the specified position.

‘getBoundingBoxOfSprite’ method returns the bounding box of the board sprite.

Spawning pattern



The way that spots are spawned is in lines that originate at the center. First the spots are spawned where the x’s are and then going clockwise. So the next spots where the Spots are going to be spawned is the line going straight up from the middle. This is done by distance. Distance used is the distance between center and closest spot, then the rest are positioned accordingly.

Player

Player class represents the player. It inherits from the cocos2d::Node class. It is responsible for players name, color, pieces and TextFields.

'setPlayerName’, ‘getName’ and ‘getColor’ manage the player's name and color.

‘setupNameTextFields’ method sets up TextFields and positions them around the board.

‘setupPieces' method spawns the pieces and positions them around the board. ‘addPieceToPlayer’ and ‘removePieceToPlayer ‘manage the player's pieces. The isPlayerPiece method checks if a given piece belongs to the player.

'isAllPiecesPlaced’ method checks if all of the player's pieces have been placed on the board and ‘isPlayerPiece’ method checks if a given piece belongs to the player.

PlayerInput

PlayerInput class listenes to input events. It inherits from the cocos2d::Node class. It listenes to the Touch events and Keyboard event. When a touch event happens it calls the StageController to handle input. When a keyboard event happens it checks if the key pressed is “TAB” and if so it raises a “restart_game” event. It also keeps track which player is active. This might be a little leftover code as this class was refactored with the interduction of Stage class.

‘onTouchBegan’ method is invoked when a touch event is detected.

‘setActivePlayer’ method sets the active player.

TurnController

Turn class manages who’s turn it is currently. It inherits from the cocos2d::Node class. It listenes to “player_no_mill” and “player_played_mill” events on which it calls ‘nextTurn’ function. It also creates and manages the label to show who’s turn it is currently.

‘nextTurn’ method throws an “end_turn” event, switches the current player and updates the StageController and PlayerInput objects with the new player. Also updates the turn label.

When switching the player it feels like there should be event data with it to decouple it but I am not sure how to do this properly.

StageController

Stage controller manages stages. It inherits from the cocos2d::Node class. It receives the touch input from PlayerInput object and determines which stage objects it will use depending on conditions.

StageController has the stage label that writes out what the player is supposed to do. Changes colours to indicate which player. It listens

‘handleInput’ processes the input and calls ‘handleInput’ on one of the stages.

‘indicateStage’ changes the text on the stage indicator based on the stage.

‘setActivePlayer’ method sets the active player.

Stages

There are multiple stage classes which StageController holds. Place, Move, Fly and Mill Stages. Each of these classes raycasts and does what they need to do in that stage upon input.

Checker

Checker class checks mills and win conditions. It inherits from the cocos2d::Node class. It listens to “player_placed_piece” event on which it checks for a mill and “end_turn” event on which it checks for a win.

‘handlePiecePlaced' method is called when a piece is placed on the board. It calls ‘checkForMill’.

‘handleOnEndTurn’ method is invoked at the end of a player's turn. It calls ‘checkWin’ for each player.

‘checkForMill’ method checks if a mill has been formed by the placement of a piece
‘checkMillFromSpot’ method recursively checks for mills starting from a specific spot.
‘checkMiddlePlacing’ method checks if a piece placed in the middle phase results in a mill formation.
‘checkWin’ method determines if a player has achieved a win condition, such as capturing all opponent's pieces or forming a mill.

‘throwGameOver’ method signals the end of the game, indicating the winning player.

VictoryScreen

VictoryScreen contains a label to show which player won, background sprite and a restart button on which “restart_game” event is thrown.

Scene

Main Menu

Main menu scene has a label and two menu Items. One for starting the game, the other for quitting the game.

GameScene

Game Scene is responsible for setting up the game. In it’s init method most of the nodes are initialised and setup. Board, players, player pieces, turn controller, stage controller, player input, checker and UI elements.

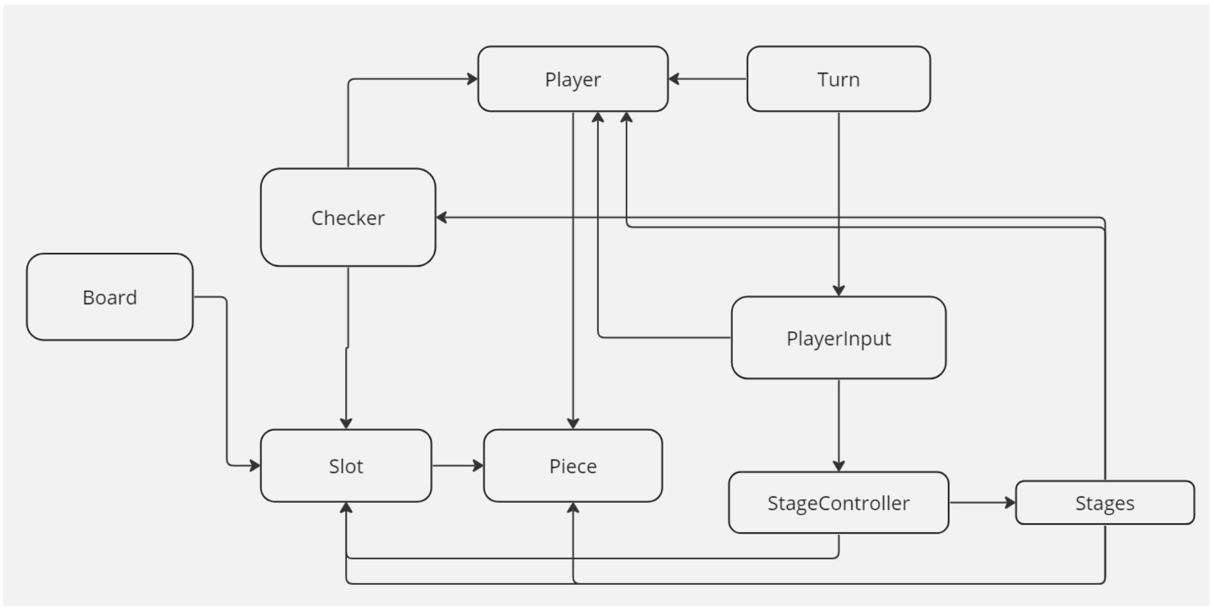
GameScene subscribes to “game_over” and “restart_game” events. On “game_over” the GameScene calls ‘onVictory’ and on “restart_game” calls ‘restartGame’.

‘onVictory’ method creates a VictoryScreen.

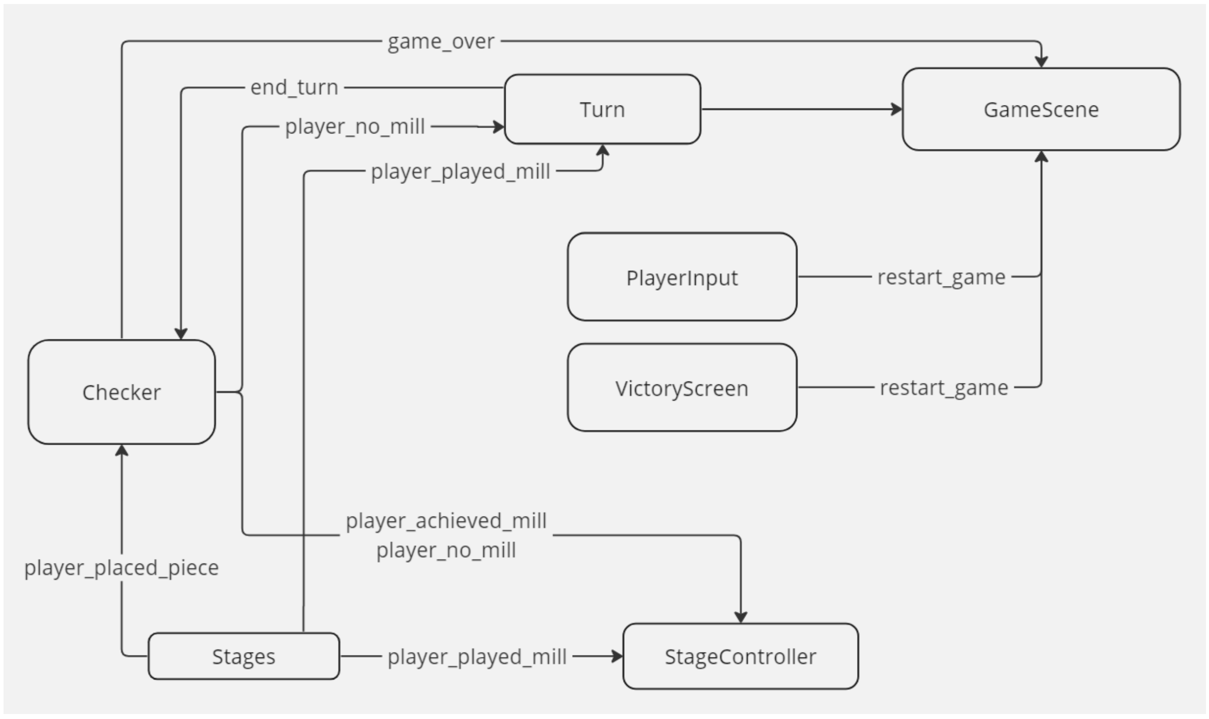
‘restartGame’ method reloads the GameScene.

Architecture Graph

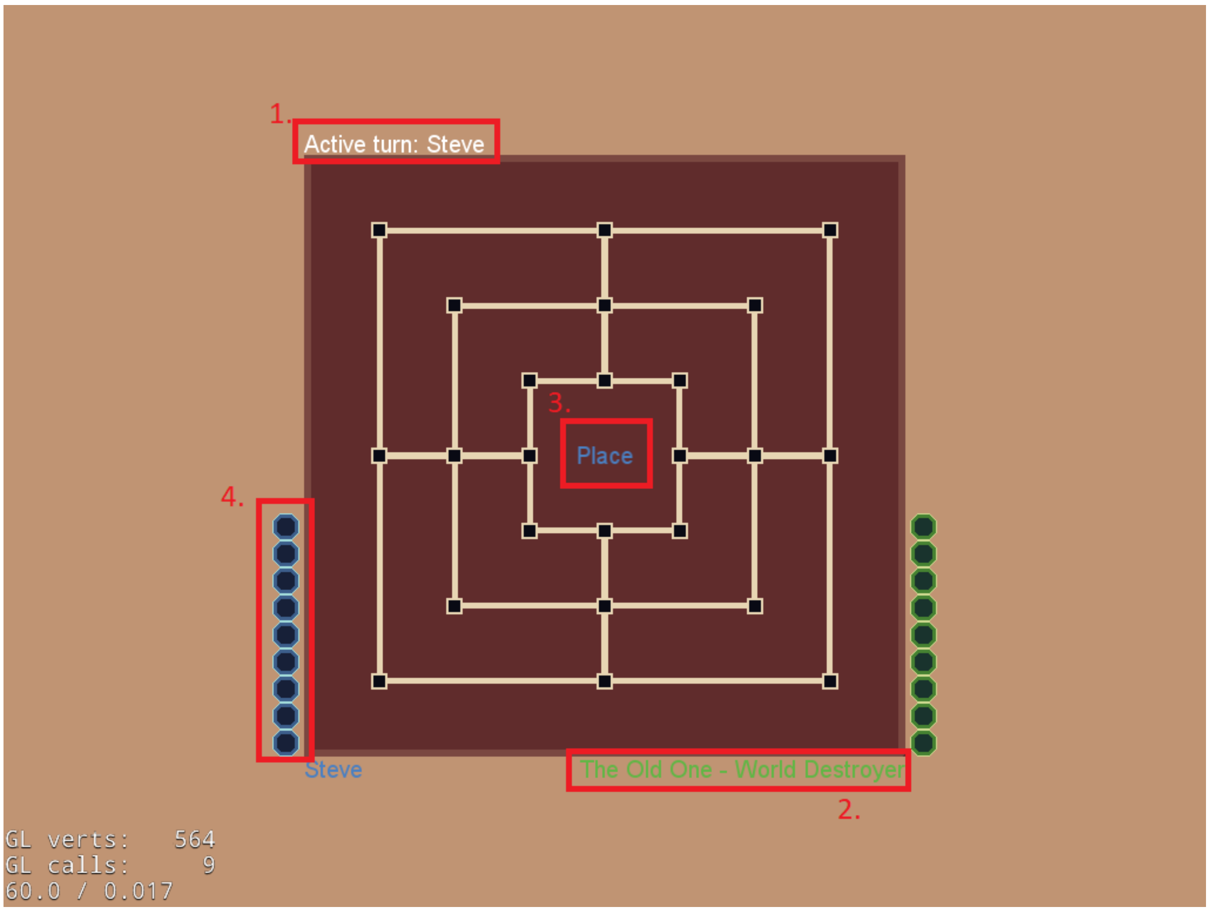
Dependancy Graph



Event Flow Graph



Game screen



```
GL verts:   564
GL calls:     9
60.0 / 0.017
```

1. TurnController's Label for showing who's turn it is.
2. Player's TextField for 2nd player to show players name and where the player can change the name
3. StageController's Label for indicating what to do and indicating by color who should do it.
4. Player pieces that are not placed. Indicating the amount visually.

Input

Tab - Resets the game

Clicking/touching covers everything else.

Improvements to be done

This part of the documentation I want to speak about some things I know are wrong, but I didn't have the time/don't know how to do it better.

To start with the Stages classes, have some repeating code, plus I am using an IF, basically a switch, in StageController to decide which stage we are in. Preferably I would have the stages inheriting an interface or a class which has the repeating code. Also, preferably there would be a concept of a current stage which just gets changed based on some events or input.

There are some instances of code, like in TurnController where I am giving the active player to PlayerInput and StageController, where I would invoke an event that has the player as data but after seeing how unsafe I did it in Checker class I decided against doing this approach until I learn a better way, if it exists.

Board setup seems weird, in Unity I would spawn Spot prefabs in a scene, position them in the editor mode and fill up a list with them in the inspector. Guess I did the same thing here, but it seems weird to have to position everything using some sort of screen positions. I tried to do it in as clean way as I could think of, but it still doesn't sit with me well.

The setup of the player pieces and TextField is a bit cumbersome with the arguments, but I can't find a better way now, it was in the GameScene before, but I've extracted it in the player class.

In general, the state of coupling UI/Front with logic is bad, I am aware of this but due to the limited time frame and my unfamiliarity with the engine I couldn't research too much into it. In Unity I would simply have a class for the logic which throws events with data that the UI/Front end component reads and updates the state of UI components. Here it's all a bit together, makes the code look messier and harder to edit but I've tried to keep it in their own classes so at least it's simpler to make the refactor once it would come to that stage.

One more thing, this is not necessarily wrong. When mill is being checked, the way to check if another Spot is in a row/column is by position. Starting Spots x and y are remembered and then others are compared to that. It works great as it's all full int values and positions are set up programmatically. This way I avoided implementing cardinal directions to each spot. This can be done but I wanted the flexibility of connected spots. Not sure what way is better.

I am sure there are more things, but these are the ones that I know/feel that are bad.