VOOGASalad Design Documentation
Group: VOOGirls Generation

Genre

Describe your game genre and what qualities make it unique that your design will need to support

The genre we selected for our game engine is Turn-Based Strategy. (Ex. Fire Emblem, Final Fantasy Tactics, Advance Wars, Chess). In general, a strategy game is defined as a game that requires decisions/strategies rather than luck. This means that the player's choice in his strategy in games can greatly influence the result of the game. The player is responsible for understanding the rules of the game and devising such strategies to create the most advantages situations for the player. The turn-based genre involves a game in which the player would have to wait until his turn to move (with each player participating in the game taking successive turns), and in each turn, the player can choose among various strategies that he might have come up with. When it is not a given player's turn that player is unable to interact with the game or interrupt the moves being played. Thus the strategy that the player conceives must take into account the actions other players will take on their turn in response to the players moves.The fact that the player takes turns is what makes this genre unique. The turn base strategy game engine is unique in that its design will require the user to choose his pieces and movements allowed for such pieces in order to create his own unique game of the TBS genre. The design will be required to support the ability to implement different types of level environments, different types of playable pieces, and a variety of rules that govern interactions between components of the game to commulatively define the rules that drive the strategy conception process that will be unique for each different implementation of a game.

Design Goals

Describe the goals of your project's design (i.e., what you want to make flexible and what assumptions you may need to make), the structure of how games are represented, and how these work together to support a variety of games in your genre.

The goals of our project's design are to allow a user to build and play a variety of grid-style turn-based strategy games fully within a graphical user interface. Our key assumptions and areas of flexibility are listed as follows:

Assumptions:

- The game needs at least two players (either human or AI)
- The players of the games take turns making plays and cannot play out of turn and cannot play when moves are exhausted in a turn
- The game can be played on a 2-D grid, which is viewed aerially
- The grid will have square patches (until further notice)
- There are clear winning conditions for each level and game
- Pieces and patches do not take up more than one grid space
- Patches will contain only one piece (until further notice)

- When designing through the game authoring environment, interaction between pieces will be defined by changing numerical stat values (i.e. in chess, interaction of a pawn attacking a rook can be described by setting the rook's health stat to zero and removing a piece if its health is zero)

Areas of Flexibility:

- Grid dimensions
- Piece characteristics
    - Movement - Rules governing possible paths of movement/ locations the piece can move to
    - Actions that pieces can perform, as defined by mathematical logic performed on piece stats and predefined action conclusion (i.e. deletion of a piece after a certain action)
    - Numerical stats (i.e. health, attack, number of movements per turn)
    - Image
- Patch characteristics
    - Image
    - Collision characteristics with piece movement
- Different types of winning conditions / rules governing the entire game
    - When designing through the user interface, different types of winning conditions will be predefined for the user to choose from
    - Global rules governing the game can be selected and implemented by the user
    - Design should allow for unlimited types of winning conditions through more Java code

Games will be represented by two main components: the number of players and the characteristics of each level. When a game is first defined, the user must specify the minimum and maximum number of players required/allowed for each game. Then, the user can define the following characteristics for each level:

- The 2D grid dimensions of the playable area
- The shapes of the patches
- The types of patches (as defined by image and interaction with pieces)
- The types of pieces (as defined by image, movement, available actions, and stats)
- The level-controlling rules

In implementing flexibility in the listed characteristics, the game engine should be able to handle games from Chess to Advance Wars to various combinations in between. The grid can be set to any size, with the playable region being any number of patches within the grid. The patch shapes and images can also be defined by the user.

A large portion of the behaviors of a patch or a piece is defined by its interactions with other pieces, which is defined through use of numerical stats. The use of numerical stats allows for ease in defining interactions between pieces and patches and removal of pieces from the grid. For example, in chess, all the pieces may have just a health stat, and when a piece is defeated by another piece, its health is set to zero. The user can then set a removal condition for each piece that states that when a

piece's health reaches zero, it is removed from the board. In the example of Advance Wars, all the pieces may contain health, attack, defense stats. Then, in the case of one piece attacking another, health can be decreased according to mathematical operations based on attack and defense stats.

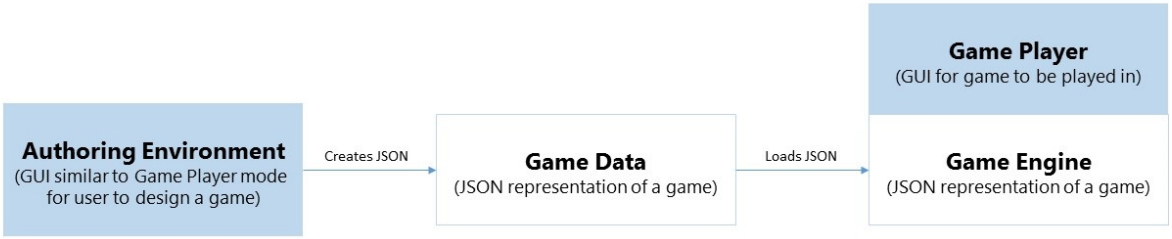
The other main portion of a piece's behavior is its movement, which is defined by the type of movement, place of movement, and number of moves per turn. By type of movement, we refer to whether a piece can jump to its available spots regardless of other pieces or whether a piece must follow a specific path. Any new types of movement need to be added through Java coding. Then, once the movement type is chosen, the user can define which spaces around the piece are available landing spots. During the game, when pieces move, their available landing spots per movement will be determined by what spaces around the piece are available after movement logic has been checked. This allows the GUI to highlight the possible moves for a specific piece dynamically while the game is being run.

The level-controlling rules for the game dictate the logic that ends each player's turn and the entire level. The rules be flexible through coding in Java, as any combination of logic in checking the pieces and patches in play could end the turn/game. In terms of the game loop, the rules should be a class containing a turn-ending rules modules and goals contain a level-ending rules module, which contain methods for checking whether the turn has finished and whether the level has finished respectively. It is we will have to pre-code the different logic in Java and then allow users to pick from those choices in the authoring environment with parameters specified by the author. Examples of turn-ending logic include making a set number of moves or selecting a turn-end button. Examples of level-ending logic include removing a certain piece, capturing of specific territory, or surviving a certain number of turns.

Primary Modules and Extension Points

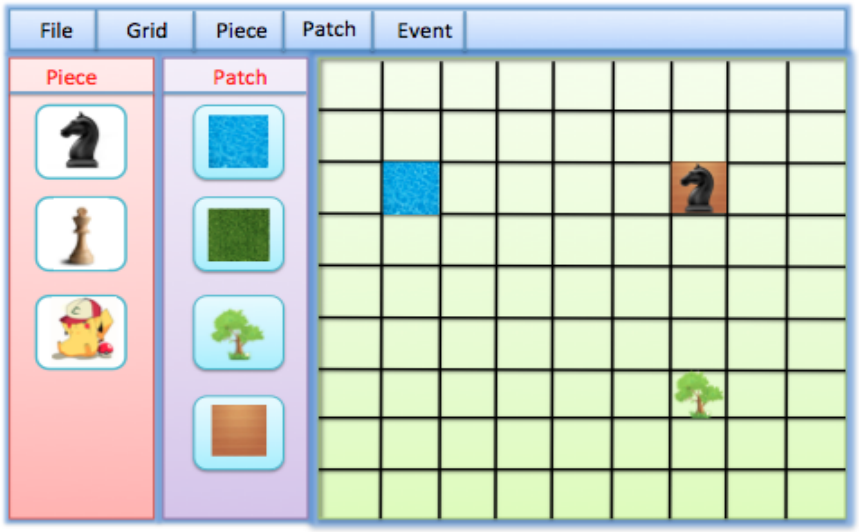
Describe the program's core architecture (focus on behavior not state), including pictures of UML diagrams and "screen shots" of your intended user interface. Each sub-project should have its own API for others in the overall team. Also describe the purpose and format of different data files you will use to save each game's state.

Refer to API Document for API Details

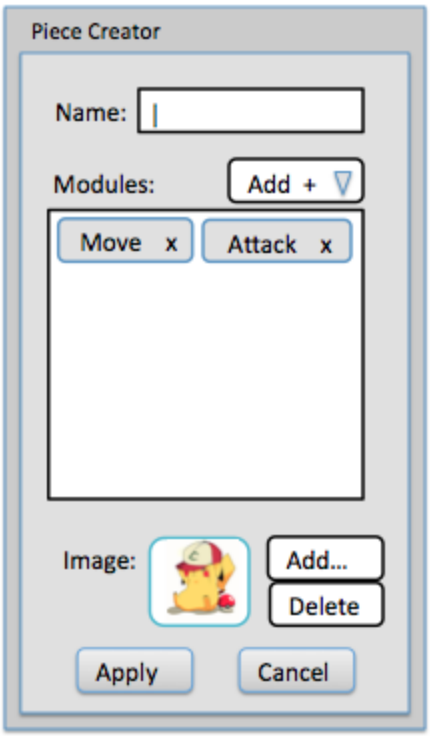


In order to do the extra credit switch between Authoring and Playing mode, we can make it so that once you switch, the current set-up is captured in a JSON and loaded by the Game Engine.

Basic GUI



All the elements in Piece and Patch column can be dragged and dropped on the grid on the right to set up the starting environment and characters.



We open Piece Creator by clicking on “Piece” in top menubar and select “Add...” to open the window above. Then we type in to define the name, add the action the piece can do, and select or upload the image. Once it’s set, it goes to the left column of basic GUI part.

The Graphical Authoring Environment is a GUI in which game designers can build a game by building each level, defining the level environment, and placing every piece, such as obstacles,

monsters, and etc. In addition to placing game elements, this GUI will allow designers to determine the order of advancements, setup graphical elements, assign reactions to different collisions and interactions, and set instructions, splash screen, player setup, etc. Utilizing the modules (defining properties and behaviors) the game designer can construct a fully functional turn-based game customized via the users decision of module selection and selective incorporation of game design components.

The Game Data is a collection of packages that contains organized sets of data files (Classes, images, audio, etc) that contains all modules/elements available for selection and incorporation into a game from the Authoring Environment. From the Game Data, the authoring environment is able to load all available modules/element and the game engine is able to load and incorporate the appropriate data files specific to a game that designers want to build as defined by the JSON created by the Authoring Environment.

The Game Player is a GUI that lets users to play a game that has been designed via the Authoring Environment. The Game Player calls on the Game Engine to load all the appropriate files from Game Data. The Game Player will then display the state of the Game Engine (And thus the game being run) and allow user input/interaction with the game. It will also let users continuously replay a game, switch between games being simultaneously run, see which games have been authored and are available for playing, saving their progress in a game to restart later, keeping track of games' high scores through successive runs of the program until the user clears it, as well as other such game control functionalities.

The Game Engine contains the game loop that is generating and updating the state of a game as defined by the JSON file for that game. The Game Engine loads the appropriate files from the Game Data to construct and prepares the initial game state using the Game Data as specified by the JSON from the Authoring Environment. The Game Engine is then responsible for processing player input, updating the state by executing behavior appropriately, and enforcing rules/goals.

Game— Modules

GAME

- Players — define Controls. (let the developer to bind keys to events)
    - Number of players (min to max)
    - p1, p2,… (Human players)
    - AI
    - Allows selection of whether a player is human or AI
- Linking of Grids (Levels) with transitions(level to level) between them

GRID (Level)

- Level Rules
    - End Turn conditions e.g. user chooses end turn / user runs out of moves
- Goal-e.g. win conditions
    - definition of rules and goals

    - Goals: predefine types of goals to allow user to choose from
- Dimensions
    - defining number of rows and columns
    - adding new rows and columns
- Grid Shape — defines the shape of the board (square/ hexagon/…)
    - Alt. Same shaped grid, only patch shape is defined to be different
- Patch Shape - defines the shape of the patches in the Grid
- Design new levels by designing new Grids

PATCHES
- Shape defined by Grid
- Patch Rules/Properties
- Patch Type
- Patch Collision Interactions (with pieces occupying patch)

PIECES
- Movement
    - Number of Moves
    - Select if Path limited -choose path in Authoring Environment
- Collision detection (Interactions)
    - With patches (check each one in path)
        - Rules and behavior between piece and patch defined
    - With pieces (check for each one in path)
        - Rules and behavior between piece and patch defined
- Interactions with the piece itself
    - Inventory

MOVEMENT (Action)
- Restrictions
    - Path — e.g. L-shape
    - Number of moves
- Collision checking in path

ACTIONS
- Defines interactions between pieces
- Provides behavior for the execution of a specific action between pieces
- General action type/structure coded
    - Specifics of Action determined by user in the Authoring Environment

STATS
- interaction defined entirely by stats: change of stats = interaction vs. everything modular
- stats: numerical representation of state of the piece
- UI: let the user to make variables and logic (stored as map/stats in program)

- every piece has the same “stat”; defined to be zero until user defined;
- Logic implemented by altering stats: Interaction defined by what to do with the numbers
    - e.g. removal logic
- Data packages: each package stores the stats for goal/rule/starting game/attacks…

Example code

Rather than providing specific Java code, unit tests, or actual data files, describe three example games from your genre *in detail* that differ significantly. Clearly identify how the functional differences in these games is supported by your design and enabled by your authoring environment. Use these examples to help make concrete the abstractions you have identified in your design.

Chess

Interesting features:

- Overall square grid
- Square patch shape
- Allows for only two players
- Goal is to attack/defeat a single piece (the king)
- Different pieces have different characteristics
    - Movement scope (i.e. bishops move diagonally, rooks move horizontally/vertically)
    - Movement obstacles (i.e. rooks are blocked by other pieces, knights can jump over other pieces)
    - Attack scope (i.e. pawns move forward one but attack diagonally)
- Piece mutations
    - Modification of pawn to king when the piece arrives at specific patches (extreme row of the board)

Fire Emblem

Interesting features:

- Overall square grid
- Multiple stages/levels
    - Players keep their data(pieces and piece stats) throughout all the stages
    - Pieces develop as they complete each stage (leveling up and changing stats)
    - Complex inventory system
        - Equipped equipment modifies piece stats and has it’s own rules/behavior
        - Tradable items between pieces
    - Multiple winning conditions (unique winning condition for each stage)
        - Having lived for x number of stages
        - Having killed all other players
        - Having killed “boss” piece
        - Having reached specific patch with piece

The design should allow for different grid and patch shapes, as well as any number of players. The software will also have multiple pre-defined goals for the user to choose from, from defeating

certain enemies to taking over a specific region of the board. If the user were to want more unique goals for the game, he/she will need to code those in Java. In chinese checkers, all pieces have the same movement, but the rule defining their movement is somewhat unique. Pieces hop over each other, and thus the logic for their movement is very dependent on the location of other pieces. Thus, that logic will likely need to be coded in Java as well.

Alternate Designs

Each sub-project should describe at least one alternative to your design that the team discussed and explain why the team choose the one it did.

Game Authoring Environment

For the authoring environment, we want the user to be able to design different types of games which are defined by their varying rule sets. After all, it is the rules that define a game - not the position of units or the terrain layout. We discussed several methods for enabling flexible implementation and modification of a game's rules. Providing a flexible platform for rule implementation would allow creation of a huge variety of games.

One approach that was discussed would use a modular RuleSet class that would contain objects of the Rule class. Rule objects would be coded beforehand and swapped in and out of RuleSet to create custom game rules. This is analogous to the Piece-Module implementation and relies on the composition design pattern. This design is certainly extensible, but does require that a wide set of Rules be written beforehand to be slotted into RuleSet. We ultimately decided on a scheme with even greater modularity that would allow the user to design complex rules from basic building blocks.

Our new design divides rules into two groups: global rules and event rules. Global rules are always active and apply to the game as a whole. These define such things as whether or not players can end turn before moving all units. Event rules are trigger-action pairs that activate when a certain condition is met. For example, victories can be easily divided into trigger and action - the trigger being the victory condition (e.g., the enemy king is captured) and the action being the game ending and a congratulations screen opening. This design allows users to link triggers to strings of actions, and thus define very complex behaviors. The authoring environment will contain a set of fundamental triggers and actions - just by combining those the developer can create thousands of unique rules. If they so desire, they can extend the program with new fundamental triggers and actions.

Game Data

In discussing the design for the JSON file, we had to decide how to store all of the information on the game. The organization of this data is essential because it determines how the game is created. We considered creating two separate JSON files, one to construct the game and one to make the setup for the game that would actually be played. In the end the difference is who would be using this information. The first would be used by the Game Engine and the second would be used by the Game Player to initialize the game. In the end we decided to keep them separate but store them as two different objects in the same JSON Object. It doesn't matter how we store it since they are only getting the output information and therefore it'd be more efficient to keep it in one object.

In addition to this, we also considered the nesting of information in our JSON file. Because the game is very complex, it supports deep nesting. For example, in Chess, there are pieces and each piece

has a type, such as pawn. There are different pawn objects that can be represented by a string name or ID. Then this specific item might have different movement modules that determine its path construction ability and the number of moves it can make. However, in the end because the final result of this would be a kind of map, that flattens any nesting we would do. So we decided instead to keep it fairly shallow, only 2-3 deep. The levels represent Class Name > Map of parameters used to construct an instance of that class. The organization of this is designed to make it easy for the Game Engine to use reflection to construct instances of this class.

Finally, we also discussed how much work the Game Data should do. Whether we should write the data to a JSON file and to what extent we should parse it. We decided that since our role is to organize the data, we would parse the JSON to extract the information in the form of strings, integers, and various other data structures. Then the Game Engine and Game Player can get this information for us as needed and use it to create the classes and other things they need.

Game Engine

For the game engine, our group discussed multiple designs for implementing various interactions between pieces. Initially, with interaction between pieces differing so greatly among games, we thought that we would possibly have to hardcode different types of interactions (i.e. attacking, healing, etc.); however, we quickly realized that hardcoding would severely limit our the flexibility in game design, especially with use of the game authoring environment. We then thought that we could use numerical stat values to somehow manage interactions through defining values for health, attack, defense, etc. However, that idea also required hardcoding possible stats as their own individual classes (i.e. a health stat class or an attack stat class). That again would limit flexibility with game design within the authoring environment.

Then, we came up with the idea for storing stats as variables in a map, where the stat names correspond to their respective values unique to each type of piece. The map would be contained by all pieces in the game and contain the types of stats defined by the user. This way, the user would not be limited to just health stats or attack stats; rather, the user could define a value for a piece's happiness or magic. This setup would also improves flexibility in designing through the authoring environment. If interactions between pieces are set fully through changes in numerical stats, the authoring environment could allow for the user to type SLogo-esque math operations using the numerical stats, which then execute during an piece-to-piece interaction. Interactions would then have to be defined for every piece-piece pair. Still, this alternate design would greatly increase the flexibility that the user has in defining the rules of the game without having to write Java code or use hardcoded predefined classes.

Game Player

Our game player consists of a supervising controller and the view components. The supervising controller is responsible for handling all the user input and all the front-end algorithms are placed in the controller for easy testing. The controller uses APIs provided by the game engine to make changes to the game model according to user input. The view is then synchronized with the game model after each action using the observer pattern.

The alternative design we talked about was to have the game Engine driving all the action and have the game player provide the public methods necessary to display the updates of the game. It will call the public methods of the game engine to update the state of the game. More specifically, we thought about providing the methods of changing the game scenes, such as get next turn, and the different levels of the game, for the Game Engine. Upon user interaction, it would call the methods such as setting current turn of the game engine. However, this design creates too many unnecessary dependencies between the view and the model. It is also difficult for the game player to handle all the user input without a controller.

Team Roles

List of each team member's role in the project and a breakdown of what each person is expected to work on.

- **Front-end**
  - **game authoring environment (4)**: program of visual tools for placing, specifying, editing, and combining general game elements together to make a particular game
    - Mike Zhu - Video Game Guru. Provide knowledge of general game mechanics and details. Ensuring front end design decisions adequately support Turn Based Strategy Game mechanics.
    - Yoon Choi - Development of Authoring Environment GUI components to allow for loading of available modules and user input driven graphical game development.
    - Martin Tamayo - Development of Authoring Environment GUI components, specifically the initialization of pieces and patches as well as their placement on the grid.
    - Meng'en Huang -Development of Authoring Environment GUI components to allow for loading of available modules and user input driven graphical game development.
  - **game player (2)**: program that loads the game data and uses the game engine to run a particular game
    - Yiran Jo Zhu - Development of Game Player GUI components to allow for instantiation of various Game Engines running given authored games and mediating user input.
    - Eric Chen - Development of Game Player GUI components to allow for instantiation of various Game Engines running given authored games and mediating user input.
- **Back-end**
  - **game engine (4)**: framework of general Java classes to support any kind of game within a specific game genre
    - Jesse Ling - Provide knowledge of general game mechanics and details. Ensuring back end design decisions adequately support Turn Based Strategy Game mechanics.

        - Minkwon Lee - Design and development of the Game Engine containing the Game Loop which loads and initiates games accordingly depending on JSON inputs and updates the game state accordingly given specific user inputs.
        - Sandy Lee - Design and development of the Game Engine containing the Game Loop which loads and initiates games accordingly depending on JSON inputs and updates the game state accordingly given specific user inputs.
        - Jennie Ju - Design and development of the Game Engine containing the Game Loop which loads and initiates games accordingly depending on JSON inputs and updates the game state accordingly given specific user inputs.
    - **game data (2)**: files, assets, preferences, and code that represent a particular game
        - Rica Zhang - Game Data implementation. JSON development and parsing. Working with Game Engine and Authoring Environment to ensure compatibility between VOOGA Components.
        - Anna Miyajima - Game Data implementation. JSON development and parsing. Working with Game Engine and Authoring Environment to ensure compatibility between VOOGA Components.