VOOGASalad
Team *CoolIronicName*

Team Members:
Dimeji Abidoye, Kevin Button, Monica Choe, Tanaka Jimha,
Keng Low, Wenjun Mao, Steven Pierre, Kevin Rhine, Eirika Sawh

Genre

We chose turn-based strategy for our game genre. We felt this game type would allow us to create a dynamic engine that could produce different types of games including both single and multiplayer, and ranges from the traditional SLG (games like Fire Emblem, which we will further discuss in the latter parts) to a simple chess game.

The basic idea is that we allow the game designer to set attributes for the given game objects and through the interaction of these objects achieve different types of game scenarios.

Design Goals

To create an extensible yet functional design with our game engine, we chose to make assumptions about some parts of our design. Specifically, we chose to specify a grid-based layout for the game, where game pieces move around the grid and interact. Another assumption we made is that each player will possess a set of game pieces, and each grid location will be occupied by at most one of these pieces. Pieces interact with each other based on user specified rules or conditions to achieve some objective in the game. We chose to make all of these assumptions to set a realistic scope for this game engine. Without the assumptions, our engine would be extremely general and would be little help to a user who wanted to make a turn-based strategy game.

To keep our design extensible and customizable for future game authors, we left some aspects of our design open-ended. Grid size and shape can be specified, as can the size and shape of the individual grid locations. Additionally, users can specify rules for object interaction or movement and conditions for winning the game. Additionally, users can dictate how many objects are placed on the board, how these objects behave, and how many players are involved in the game. With all of these attributes left open to the user, we can keep the design customizable so that many different games can be created from this single game engine.

Primary Modules and Extension Points

For VOOGA, we created seven primary modules. Two modules are solely dedicated to the Graphical Authoring Environment, two modules are dedicated to the Game Player, and three other modules are shared between the two. The two modules of the Graphical Authoring Environment are a View module, which is in charge of creating a display for the GAE, and an Authoring Controller module, which is a controller for the GAE. The two modules of the Game Player are a View module, which is in charge of displaying the game being played, and an Playing Controller module, which is a controller for the Game Player.

Both GamePlayer and the Graphical Authoring Environment share a module called Models, which are all the classes setting up the properties for the attributes, and a Data module, which keeps track of all the states of both the Game Player and the Game Authoring Environment. The final module is the Toggle module, this module will handle switching between the Game Player and the Graphical Authoring Environment, in the process updating the Data and passing the information to each program's respective controller.

There are extension points in the Models module, and the View Module. Our Models extension will be all the attribute that we will be able to create using our controller. We will be able to add further attributes thanks to the extension capability of our Models hierarchy. The view Module will have extension points for the type of boards, the type of objects, or other attributes of the view that you want to include in the future.

APIs - *Please Refer to Github 'apibranch' for basic API structure*

Game Authoring:

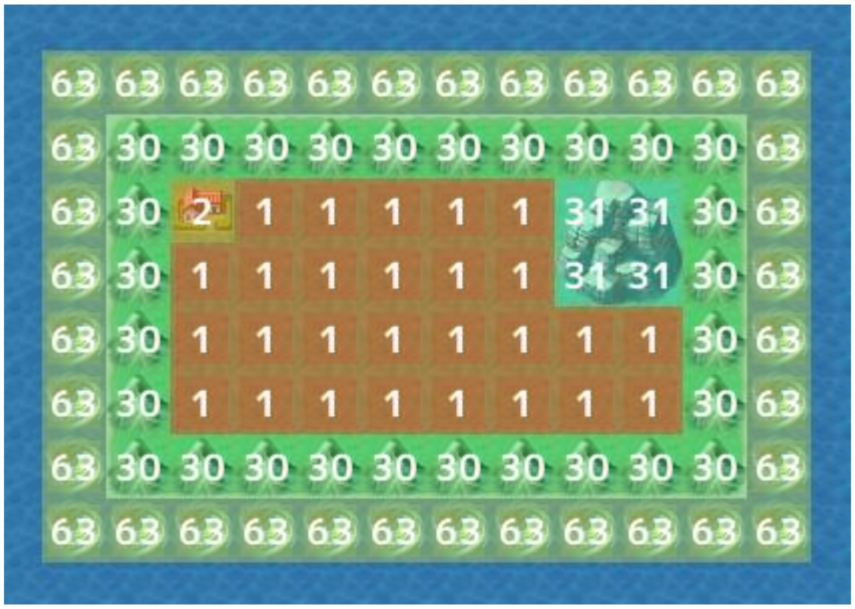
Two modules of our design are exclusive to the game authoring environment, and three modules are shared with the environment. The game authoring environment is composed of the authoring controller and authoring view, and shares a model with the game playing environment. In general, the game authoring environment has saving and opening capabilities as well as game creation functionality. The authoring view handles all user interaction and notifies the controller of changes, which in turn updates the game data based on the shared game model. The authoring environment then communicates with the game playing environment by sending data from its controller to the playing controller.

Since our goal is to let game designers interactively editing the game instead of just type in the binary numbers for enable/disable certain attributes, we will provide a data-authoring environment and let users choose from the attributes and edit stats for the objects of the game.

The core design for a turn based game is a game board which carries the whole game environment and allows players to deploy their heroes on the board and move around. With the experience of building a similar game in the first project, I used to hardcode in binary numbers to set obstacles and boundaries for the board.

Our design this time is when user is generating a board, first step is allowing them to click and toggle a certain landscape (ground, tree, water,etc.) and simply place on the empty board. Then another channel is shown to let the user choose the "height" of each piece of land they put on the first part. The object is only allowed to move between blocks that have a maximum of height difference which is set by the user. If a certain spot is an obstacle, user can just set the "height" to be a large number to make it inaccessible. A height range of 0-63 is provided. And we can also allow an object with "flying" attribute to ignore the height restriction.

We will make further design for our UI during the process of implementing different functionalities. I think this can be a utility to share in different groups, since as we can see, we are able to build both scrolling and tower defense maps with a little modification from this kind of board designer.



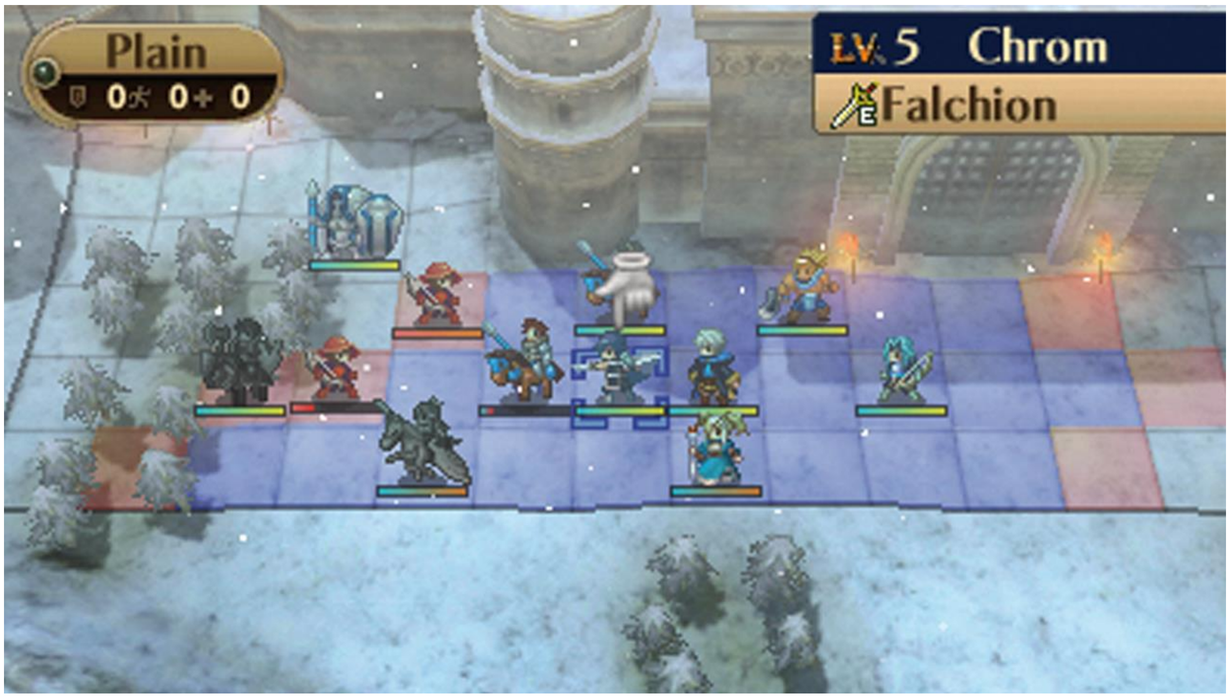
Game Play:

Within the game playing environment, a controller will regulate communication. This controller will be separate from the authoring environment controller, and will initialize game state by reception of data from the authoring environment or through a file received from the view. The playing view will receive all user input and pass it to the controller, as well as output any relevant changes to game data. The view will communicate exclusively with the controller, which in turn will communicate with any other modules. On the other side of the controller, the game data will be updated whenever the view receives input, by passing the input to the model, determining new states, and updating the game data accordingly. When the game data is changed, the view will be updated to display the current state of the game. This process continues until a manager detects that end-of-game conditions are met, prompting a new view to be displayed.

Example Code (Descriptions)

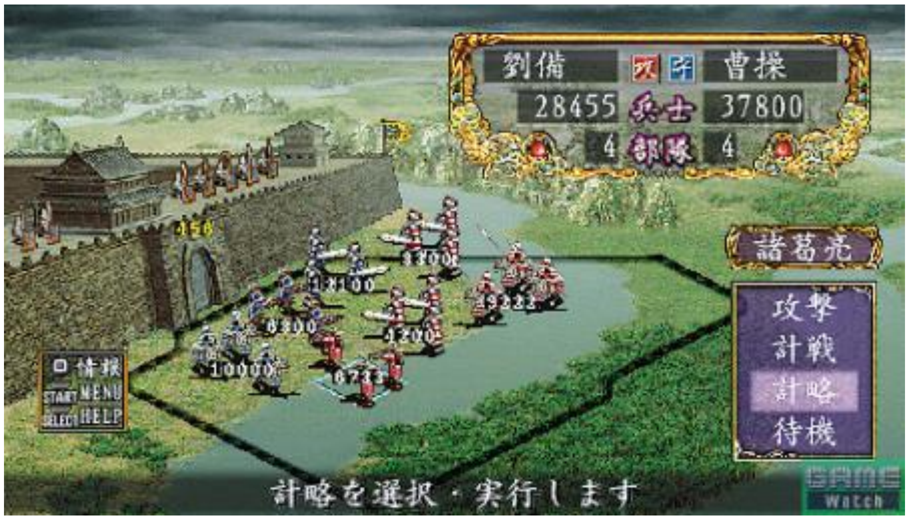
Fire Emblem:

- Fire Emblem is a Turn Based Strategy game in which the player chooses where he will place his players, combining his characters in one box, and choosing what kind of action the player will perform once he is at his position. In this kind of turn based strategy game the player can essentially decide what kind of action he can perform from wherever on the field. On our code, we would first create our board, place a restriction on having enemy AI and player sharing the same grid area, and allowing the player to have multiple attributes at a location. Then we would code for each character the player can control, and give attributes to each characters.



Civilization/Romance of three kingdoms series:

- Civilization is another Turned Based Strategy game just like Fire Emblem with an added twist, the player get to change the board as the game moves forward. In this case, we would create everything on the side of the player in the same manner. However, for the board the player the author would allow for objects to be added during gameplay so that the dynamics of the game would be different. This would be handled by allowing the board to also receive some extra features.
- Use money to construct buildings turn based instead of real time, specific rules of ending a turn is checked (either give each player certain amount of Action Points,or only allow the execution of certain actions for certain times, such as in Fire Emblem, the moving action and attack action is only allowed once for each player unless other skills are enabled)
- buildings create different kinds of units, generate money, etc.



Chess

- Chess is also a Turned Based Strategy game. It is the simplest game of the three above. one player makes his move and the other does the same. You can either win by defeating all of the other players pieces (unlikely), or you can pin the kind and/or queen of the opposing player. To create this game, we would have to first create the board, and limit the amount of character's per location on the board. Then we would have to create each distinct piece, limit the player's movement by one action at a time, and give them their respective attributes. One we set the rules that calculate a loss with the model for the board, we can start playing until someone loses.

Alternate Designs

Initially, simply agreed on implementing the Model-View Controller (MVC) design pattern however, we still had to spend some time deciding the exact details of the implementation. Because we don't have a traditional database, to hold the data, we critically weighed out our implmentation options. One of the our alternate options was to have the data stored in in the controller, or in the model classes. However, we decided to let the Model only hold the application logic(validation logic and data access logic). The data itself would then be stored in a separate package. This separation means that the model will not have to format the data or read the formatted data and thus directly holding/interacting with the the display logic. Instead, the IData module we created separate from the model can hold the data and hence have the display logic separate from the model and conntroller. This way, the model simply acts as an interface to the data store.

Another significant design decision we had to make was using inheritance in our Game Objects vs using Composition. Alternately, we could have had our game characters/objects inherit their attributes/behaviour from classes, for example, an object would inherit/extend/implement the flying class or the running class in order to make give it ability to fly or run or enact any other actions not described in the general game object class. In such a case, it would have meant that to create an object with these multiple attributes, we would have had to create a subclass of a general object, and have it implement the different combinations of all the attributes we would like to have. However, instead of doing so, we decided to use composition in such a way that whenever a game character is supposed to have special attributes, it simply contains instances of those attributes. For example, a character that can run and fly will simply contain instances of the run and fly attributes.

Team Roles

<u>Game Authoring Environment:</u> This group will create the module for authoring games. They'll craft both the view and controller that will be specific to the authoring environment, and they'll design the module to interact with the game playing environment. The team will consist of: Eirika (Frontend), Keng (Backend), and Wenjun (Backend).

<u>Game Engine:</u> This group will be responsible for writing the model of the program. They'll write the classes that will represent the generic components of any game created, as well as subclasses for specific game and feature implementations. The team will consist of: Kevin B., Dimeji.

<u>Game Player:</u> This group will create the module for playing games. They'll code the view and controller specific to the playing environment, and they'll design the module to receive data from the authoring environment as well as set up games from read in files. The team will consist of: Kevin R., Monica, Steven.

<u>Game Data:</u> This group will create classes and modules pertaining to data reading and writing. They will be responsible for the communication between the two environments, as well as reading from and writing to memory for both game types and states. The team will consist of: Tanaka Jimha