Game Technical Design Document

https://github.com/ahhlecks/Tactics-Deckbuilder-master

Summary

High Concept: Use your wit to defeat your enemies with deckbuilding and tactics on a 3D hex grid. Move units around the map, and use cards to attack and defeat your opponents.

Game Genre: Deck Building, Strategy, Wargame, Card Game

Game Engine: GODOT

Programming Language: GDScript, C#
https://docs.godotengine.org/en/stable/tutorials/scripting/gdscript/gdscript_basics.html

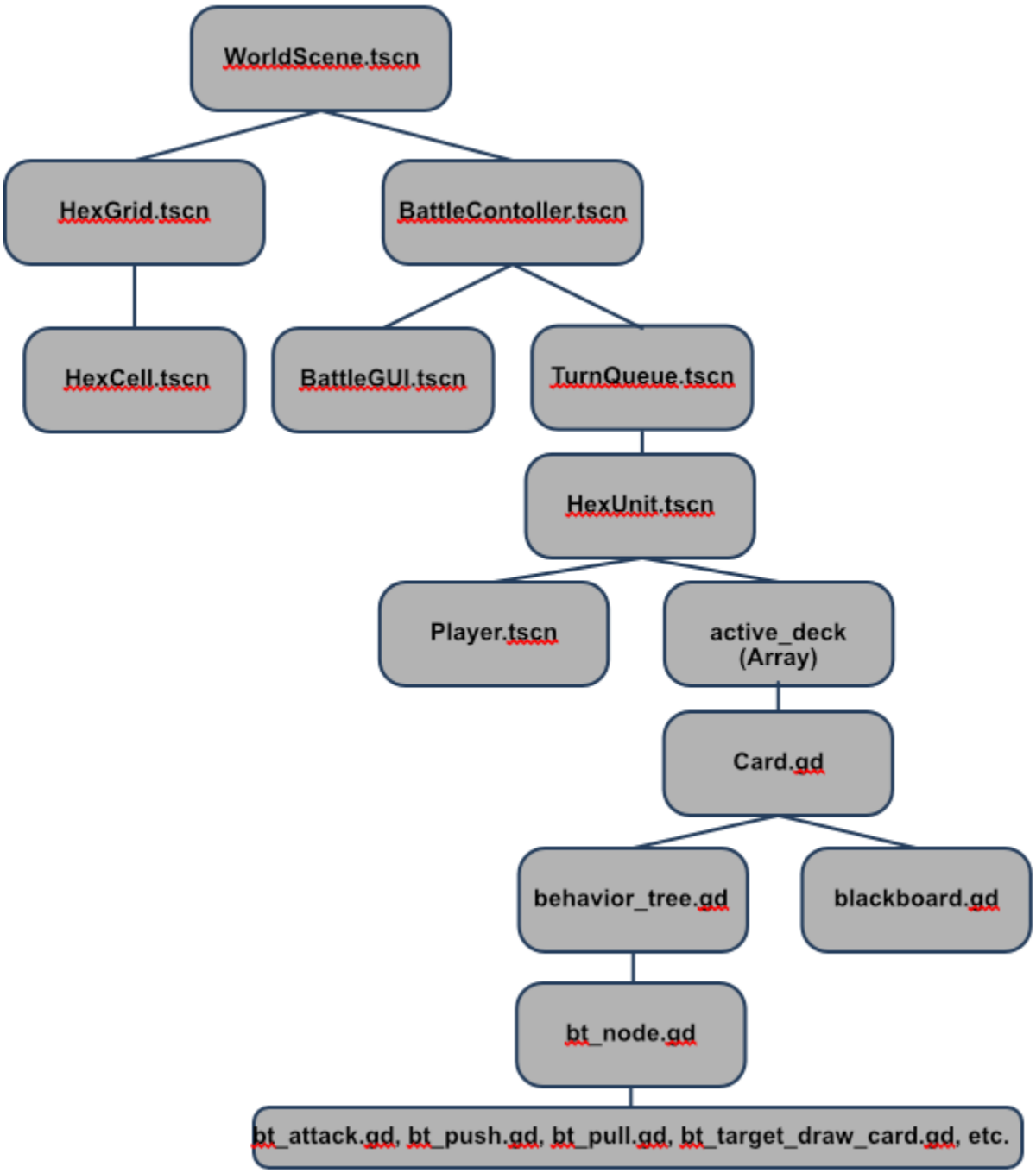
Start Date: Beginning of 2020

End Date: TBD

Features:

- Turn-based mechanics
- Cards
- Hexagonal Grid
- Pathfinding
- AI
- Directory/File manipulation

High-Level Diagram



Turn-Based

The core game starts with a function that loads a .map file that was previously created from a tool called the map editor that was also made for the project. The function “load_map” accepts a string parameter that is the file name of the map in the project folder. This is in the script “HexGrid.gd”

```
func load_map(map:String):
    clear_map()
    battle_controller.clearAll()
    var map_data = MapLoader.loadSingleMapFile(map)
    var unit_data = map_data.get("Units")
    var cell_data = map_data.get("Cells")
    for i in cell_data.size():
        var x = cell_data[i].get("x")
        var z = cell_data[i].get("z")
        var height = cell_data[i].get("height")
        var surface = cell_data[i].get("surface")
        var cell = createCell(x,z,height,surface)
    for i in cells.size():
        setCellNeighbors(cells[i])
        cells[i].updateHeight(cells[i].cell_height)
    for i in unit_data.size():
        loadUnit(unit_data[i])
    clear = false
    emit_signal("map_loaded")
```

The function clears the current map and clears all battle controllers before loading a new map from a MapLoader.

The map data is then loaded and stored in variables called "unit_data" and "cell_data". A for loop is then used to iterate through each cell in the map data and retrieve its x and z coordinates, height, and surface type. A cell is then created with these parameters using the "createCell" function.

Another for loop is used to iterate through each cell in the "cells" list and set its neighbors using the "setCellNeighbors" function, as well as update its height with the "updateHeight" function.

The unit data is then loaded using the "loadUnit" function, and a signal called "map_loaded" is emitted once the map is loaded.

```
func setUnitList(unit_list:Array, player_party:int) -> void:
    battle_gui.unit_gui.visible = true
    self.unit_list = unit_list
    self_player = PlayerPrefab.instance()
    self_player.set_script(player_script)
    add_child(self_player)
    self_player.battle_gui = battle_gui
    self_player.battle_controller = self
    battle_gui.player = self_player
    enemy_player = PlayerPrefab.instance()
    enemy_player.set_script(ai_player_script)
    add_child(enemy_player)
    enemy_player.battle_gui = battle_gui
    for unit in unit_list.size():
        unit_list[unit].grid = grid
        unit_list[unit].battle_controller = self
        unit_list[unit].setTeamColor()
        if unit_list[unit].team == player_party:
            unit_list[unit].unit_owner = self_player
            self_player.units.append(unit_list[unit])
        else:
            unit_list[unit].unit_owner = enemy_player
            enemy_player.units.append(unit_list[unit])

    turn_queue.units = unit_list
    yield(get_tree(), "idle_frame")
    startMatch()
```

The function first sets the visibility of a battle GUI to true, and then sets the "unit_list" variable to the passed-in "unit_list" parameter. It also creates a new instance of the "PlayerPrefab" class and sets its script to "player_script".

A for loop is then used to iterate through each unit in the "unit_list" array. For each unit, its "grid" and "battle_controller" variables are set to the current object, and its team color is set using the "setTeamColor" function. If the unit's team is the same as the passed-in "player_party" parameter, its "unit_owner" variable is set to the current player instance and the unit is added to the current player's "units" list. Otherwise, the unit's "unit_owner" variable is set to the enemy player instance and the unit is added to the enemy player's "units" list.

The "units" variable of the "turn_queue" object is then set to the "unit_list" array. A yield statement is used to wait for the next idle frame of the game engine, and then the "startMatch" function is called to begin the battle.

Cards

Cards are objects that are instantiated when the unit is loaded into the game. Inside a card object are many variables including: action_costs, card_level, card_attack, can_attack, can_defend, delay, etc.

Also, a card contains a **behavior tree** and **blackboard**. A **behavior tree** is a computational model commonly used in artificial intelligence and robotics for decision-making and control of agents or systems. It is represented as a directed acyclic graph (DAG) where nodes represent different behaviors or actions, and edges indicate the relationships between them.

- *This screenshot is from the in-game card maker scene*

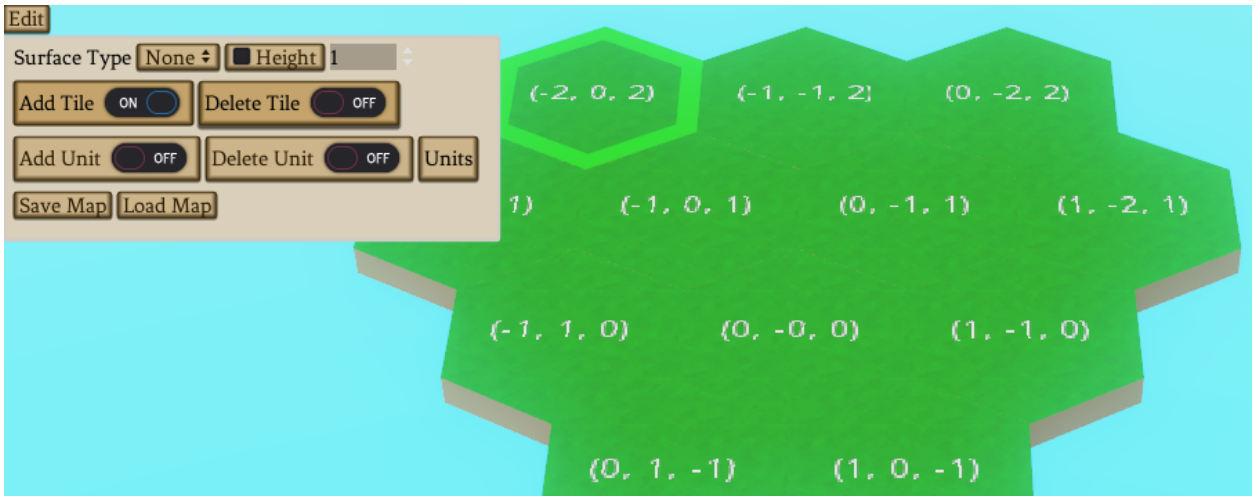
For example, this behavior tree will run in sequence from top to bottom. First, it executes **“SetLeftRightTargets”** a script which will find and set the left and right hex cell from the targeted cell.

Next, it runs **“CasterModifyStat”** which accepts parameters and modifies the stats of the caster of the card.

Finally, it runs **“Attack”** which executes an attack toward the targets.

Hexagonal Grid

The hexagonal grid is a 3-Dimensional grid that contains an array of “HexCells”. The grid is created from a for loop that instantiates HexCells in a grid-like fashion. The cells’ height values and surfaces are able to be manipulated through the in-game map editor.



Pathfinding

```
func frontierSearchNeighbors(frontier, horizontalCheck, verticalCheck, isMove):
	var sourceCell = currentCell
	var current_cell = frontier.front() # currentCell is first cell in frontier
	frontier.pop_front() # remove first cell from frontier
	for i in current_cell.neighbors.size():
		var frontier_cell = current_cell.neighbors[i]
		var ref_cell = weakref(current_cell.neighbors[i])
		if ref_cell.get_ref() != null:
			if isMove: #this is a move range check
				if validMoveCells.find(ref_cell.get_ref()) == -1: #if it's not valid yet
					if current_cell.walkDistance < horizontalCheck: #if it's within walking range
						if abs(ref_cell.get_ref().cell_height - current_cell.cell_height) <= verticalCheck: #if it's within jumping range
							if ref_cell.get_ref().unit == null: #if the cell is not occupied by a unit
								ref_cell.get_ref().walkDistance = current_cell.walkDistance + 1
								ref_cell.get_ref().pathFrom = current_cell
								validMoveCells.append(ref_cell.get_ref())
								frontier.push_back(ref_cell.get_ref()) #add valid neighbor cells to frontier
							elif ref_cell.get_ref().unit.team == team: #if occupied by unit, if the unit is of the same team
								ref_cell.get_ref().walkDistance = current_cell.walkDistance + 1
								ref_cell.get_ref().pathFrom = current_cell
								validMoveCells.append(ref_cell.get_ref())
								frontier.push_back(ref_cell.get_ref()) #add valid neighbor cells to frontier
			else: #this is an attack range check
				if validRangeCells.find(frontier_cell) == -1: #if it's not valid yet
					if current_cell.walkDistance < horizontalCheck: #if it's within walking range
						if current_cell.walkDistance == horizontalCheck - 1 and (!frontier_cell.extended and !current_cell.extended):
							frontier_cell.walkDistance = current_cell.walkDistance + int(frontier_cell.cell_height >= sourceCell.cell_height - 2)
							frontier_cell.extended = true
							#currentCell.extended = true
						else:
							frontier_cell.walkDistance = current_cell.walkDistance + 1
	#				if !frontier_cell.extended or (frontier_cell.extended and abs(frontier_cell.cell_height - sourceCell.cell_height) <= vertical
					frontier_cell.pathFrom = current_cell
					validRangeCells.append(frontier_cell)
					frontier.push_back(frontier_cell) #add valid neighbor cells to frontier
```

The function begins by initializing two variables - "sourceCell" and "current_cell" - with the values of "currentCell" (which is the first cell in the "frontier") and the first cell in the "frontier", respectively. The first cell in the "frontier" is then removed using the "pop_front()" function.

The function then loops through each neighbor of the "current_cell" and initializes two variables - "frontier_cell" and "ref_cell" - with the values of the current neighbor cell and a weak reference to the current neighbor cell, respectively. If the weak reference is not null, the function checks if it is a move range check or an attack range check based on the value of the "isMove" parameter.

If it is a move range check, the function checks if the current neighbor cell is a valid move cell based on certain conditions such as if it is within walking range and jumping range, and if it is not already occupied by a unit. If it is a valid move cell, the function sets its "walkDistance" and "pathFrom" attributes, adds it to the "validMoveCells" list, and adds it to the "frontier" for further processing.

If it is an attack range check, the function checks if the current neighbor cell is a valid attack cell based on certain conditions such as if it is within walking range, and if it is not already marked as a valid range cell. If it is a valid attack cell, the function sets its "walkDistance" and

"pathFrom" attributes, marks it as extended if certain conditions are met, adds it to the "validRangeCells" list, and adds it to the "frontier" for further processing.

Overall, the function performs a search for neighboring cells within a given range based on certain conditions, and adds the valid neighboring cells to the "frontier" and "validMoveCells" or "validRangeCells" lists for further processing.

AI

A function is intended to make an AI-controlled unit perform the best possible move it can make during its turn. The function does this by checking the available cards in the unit's hand, choosing the best card to play, and then executing the card's action (if possible) while moving the unit to an optimal location. If the unit has low health, it will try to retreat. Finally, the unit will turn towards the nearest enemy.

Directory/File Manipulation

```
func loadSingleCardFile(card_name:String, save_dir:bool) -> Dictionary:
    var dir = Directory.new()
#   if !dir.dir_exists(PlayerVars.CARD_LOAD_DIR):
#       dir.make_dir_recursive(PlayerVars.CARD_LOAD_DIR)
#       return {}
    if !save_dir:
        if dir.file_exists(PlayerVars.CARD_LOAD_DIR + card_name + ".crd"):
            var file:File = File.new()
            var error = file.open(PlayerVars.CARD_LOAD_DIR + card_name + ".crd", File.READ)
            if error == OK:
                var card_data:Dictionary = file.get_var()
    #           load_card(card_data)
                file.close()
                return card_data
            else:
                file.close()
        elif dir.file_exists(PlayerVars.CARD_SAVE_DIR + card_name + ".crd"):
            var file:File = File.new()
            var error = file.open(PlayerVars.CARD_SAVE_DIR + card_name + ".crd", File.READ)
            if error == OK:
                var card_data:Dictionary = file.get_var()
    #           load_card(card_data)
                file.close()
                return card_data
            else:
                file.close()
```

The function loadSingleCardFile loads a dictionary object from a file, which contains data for a single card.

The function takes two arguments: card_name is the name of the file (without the ".crd" extension), and save_dir is a boolean indicating whether to load the file from the directory where the game saves card data or where it loads card data.

The first part of the code creates a new Directory object, which is used to check if the file exists in the appropriate directory. If the file exists, a new File object is created and the file is opened for reading. If the file is opened successfully, its contents are read and converted to a dictionary object using the get_var() function, which is a method of the File class. Finally, the file is closed and the dictionary object is returned. If the file does not exist or cannot be opened, the function returns an empty dictionary.