API Design Document: Omega\_Nu

Leevi Gray, Kevin Jian, Chris Murphy, Patrick Schutz, Brooks Mershon, Andy Bradshaw, William Li, Ken McAndrews, Carlos Reyes, Vincent Wang

Genre

Turn based strategy (with music but not music based combat)

Design Goals

One of the biggest goals is to keep our “combat system” as flexible as possible so the user can create any sort of interaction between objects that they want. An example was thrown around that a “farming simulator” could be possible if our interactions end up being as abstract and generic as we are planning. A way to ensure this flexibility can be achieved in a month of development is to create formulas that are driven by the conjunction of “weighted” parameters, so that the addition and removal of new parameters will effect gameplay in a linear way, and be easily changed at runtime and stored in configuration files. A simple example of such a formula would be:

In an effort to offer an aspect of combat that can be easily tuned by game designers, we will include an editor for designing a probability distribution associated with combat. This distribution will be both visually represented to the player, and more significantly, change the mechanics of gameplay as a function of the curve that is designed. For example, a distribution with the following parameters could take on a unique shape as a result of introducing critical hit bonuses and specifying the number of times an n-sided die is rolled:

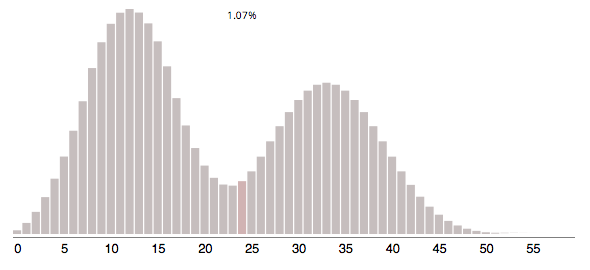


Figure 1 This distribution corresponds to taking the sum of rolling an 8-sided die 3 times, and then with a 15% chance adding to this sum 7 plus the sum of rolling a 4-sided die 7 times. Note the two bimodal distribution created! This would certainly affect gameplay mechanics in a different way than a uniform distribution would. Simply varying 3 to 4 input parameters could create many distributions. Thus gameplay “tuning” is possible after most of the game is configured.

Another design consideration is structure of the levels. Levels must be constructed in a linear order, and will take place on a series of “grids” that the user designs. The purpose of this is to provide an easy way for the user to drive the story of their game through the level progression. In this way, the story of a game and level design are inherently linked.

The game is structured around interactions between units towards an objective for a particular level. These levels progress in pursuit of some overarching goal defined by the user in their story. This is a broad enough structuring to allow for RPG-style turn based games like Final Fantasy Tactics or Fire Emblem, but can also be used to create games as diverse as the farming simulator mentioned before. Since the conditions for beating a stage are user defined, the biggest limit on what can be done is on the users creativity.

Primary Classes & Methods

GameObject class extends JGObject

-getName()

-getGraphic()

-getBehavior()

GameUnit class extends GameObject

-getControllable()

-getItems()

-getStats()

-getAffiliation()

-wait()

-move()

Items class

Equipment class extends Items

-getStatModifiers()

Consumable class extends Items

-getNum()

-getEffect()

Weapons class extends Equipment

-getActions()

-getStatModifiers()

Armor class extends Equipment

-getStatModifiers()

Action class

-getType()

-getPowerLevel()

-getCost()

-getRange()

Stat class

-getStrength

-getDefense

-getAgility

-getIntelligence

-getCritical

-getMaxHP

-getMaxMana

-getCurrentHealth

-getCurrentMana

GameUnits have base stats and properties, the stats are used in determining battle outcomes, the properties (such as health and experience) are affected by the external environment, where changes occur due to factors such as battling. The ways stats are affected are through: equipment, level-ups, and terrain.

Equipment currently falls under two categories, attack (weapons) and defense (armour). Aside from using equipment, increasing stats happens via a level up. Leveling up occurs whenever experience points reach a certain level as defined by the user. Different terrain tiles affect certain stats of the character standing on them (while they are on them) as defined by the user.

Because GameUnits are either controllable or uncontrollable they make up every object in the game that can be interacted with, Game Units can be set as controllable or uncontrollable allowing them to be virtually any interactable object in the game.

Example Code

In the first example of a game in the turn based genre, the units are carried over from stage to stage. This is more RPG like, with the units leveling up and a more story driven basis. The data sheet for this would look like as follows:

class = “unit”

name = “Mario”

str = 10

def = 2

agi = 3

int = 0

movement = 5

items = “plunger, racoon hat”

class = “item”

name = “plunger”

action = “plunge”

type = “physical”

power = 5

cost = 17

range = 1

class = “item”

name = “raccoon hat”

agi = 20

class = “grid”

height = 4

width = 5

start = 3 x 2, 4 x 3

The second example code is a more of a resource based army management turn based combat. The units on each team are generic, with no carry over between stages.

class = “unit”

name = “Infantry”

str = 5

def = 3

movement = 4

items = “gun”

class = “item”

name = “gun”

action = “shoot”

type = “physical”

power = 2

cost = 1

range = 5

class = “object”

name = “factory”

behavior = “make Infantry”

class = “grid”

height = 6

width = 6

3x2 = factory

Alternatives to Design

maps link to other maps

- Before we had the idea of having an overworld, with individual “worlds” within the greater world. Each world has a pointer to a parent world and child worlds, so that characters can traverse different grids at will. An example of such a use would occur when a “house” is entered. The interior of a house would be represented by another grid of the same dimensions as the player used before entering the house.

- For example, a user would step on a town square in the overworld which would bring them to a new town map. They then can exit the town and go back to the overworld

- We decided to implement a concept of “stages” instead of the overworld idea

- Each stage will be a combat map where the user needs to fulfill some sort of goal (e.g. defeating all enemies, surviving for x amount of time, finding certain items, etc.)

- Once a stage goal has been fulfilled, the user progresses to the next stage

- The sequence of stages defines the storyline of the game

- The stages allow the user to better control a sequence of events for the game. If there was an overworld, this would be more of an RPG where the player decides their own storyline

- We wanted the game to be more focused on the turn based aspect of the combat instead of being like an RPG

stages going back and forth - stages only go one way - one direction

- We had an idea where we could let the player go back to stages that they had already beaten in order to level up the character, get more experience, etc.

- We didn’t want to give the player the ability to go back stages since that would affect the storyline of the game

- If the player is only allowed to go forward in the stage sequence, this would retain the storyline that the user had initially defined

Tiles shouldn’t have behavior - utilize inheritance for tiles + map objects, but actually wanted it different. Also object now has passable attribute (can walk on)

- We want to categorize the environment into two parts, one that is capable of engaging in interactions with the players, i.e. a shop, a hospital, a trap, the other that is purely there serving as tiles for the environment and doesn’t have interactions or behaviors

- For example, a shop will have the attribute of providing the character with weapon choices to buy in game whereas grass and water are tiles

- Therefore the first category of non-player objects will be built on top of the second category of basic building tiles, giving the location different interactive values within the game

units have items and each item has an action which defines units’ actions

- We thought about having the actions (e.g. attack range) associated with the units

- This would create problems in that if a player has an initial attack range of 1 because he is equipped with a sword, he would still carry this stat when the character switches its weapon to a bow

- Instead of modifying the player attributes every time, we decide it would make more sense to associate these actions/attributes with items that player carries. i.e. bow will have a range of 5 and dagger will have a range of 1

- Therefore units have items and each item has an action, which defines units’ actions; the units serve merely as the container for the items and are controlled fully by the grid

npc and pc are actually same, just have playable attribute modified

- Originally we had the NPC and player-controlled characters as separate defined classes

- However, we realized that both the NPC as well as the player-controlled characters have all of the same characteristics except for how they are controlled

- Instead we put a variable within the character class that controls whether or not it is controllable by the player

- This allows us to just create many instances of characters and then be able to decide whether or not that the player controls character

- This gives us flexibility and simplicity when creating new characters in a game environment

Team Members’ Roles

Leevi Gray - File Storing, Stage Editor

Chris Murphy - File Storing, Stage Editor

Patrick Schutz - File Storing, Stage Editor

Kevin Jian - Grid

Ken McAndrews - Grid

William Li - Grid

Carlos Reyes - Combat

Andrew Bradshaw - Combat

Vincent Wang - Units

Brooks Mershon – Units/levels