Of Mages and Magic
Combat System

1 Introduction

Computers are purely mathematical machines. If we want to tell them how to do something, we have to find a way to explain it to them using numbers. In this document, we're going to explain how we designed the combat system for Of Mages and Magic such that the computer can use it.

The combat system is the part of our code which decides things like whether or not a spell gets a critical hit, how much damage a mage takes when they are hit with a spell, whether or not a mage successfully avoids an attack and other things.

Most of the code which implements this logic is in the `app/models/magic.py` file. There's a lot of code in there, so we'll try and put all the important bits in this document, but it would be a really good idea to check out the actual source file.

For attack moves, we've borrowed a lot from the original Pokémon games (Red, Blue and Yellow). From a computer programming perspective, these games were really interesting because they had to provide deep, complex gameplay while running on very limited computer hardware (your smart-phone is about 1000 times faster than the original Gameboy). This meant that their battle logic had to be complicated enough to require some amount of strategic thinking, but simple enough that it could be run quickly on the constrained machine.

Our program follows roughly the same series of operations as Pokémon for computing the result of your AI's decisions. After your AI picks what spell it wants to use and who it wants to target, our code runs the following sequence of steps:

1. Check if target dodges the attack
2. Check if attack is a critical hit
3. Compute damage
4. Check for element effects (super effective/not very effective)
5. Apply damage

We'll run through each of these in detail below.

2 How Do We Know If The Mage Dodges

In the original Pokémon algorithm, a character had two additional traits – evasion and accuracy. These weren't actual stats like attack, defense and speed. They were modifiers with values between -6 and 6.

The game would subtract the target's evasion modifier from the caster's accuracy modifier and translate the result into a fraction. This fraction would be multiplied by the accuracy of an attack to determine whether or not it would hit an enemy.

A table showing the fractions associated with each of these modifiers is given below.

Stat stage	**Fraction Multiplier**	**Decimal Multiplier**
-6	2/8	0.25
-5	2/7	0.28
-4	2/6	0.33
-3	2/5	0.40
-2	2/4	0.50
-1	2/3	0.66
0	2/2	1
1	3/2	1.5
2	4/2	2
3	5/2	2.5
4	6/2	3
5	7/2	3.5
6	8/2	4

Table 1: Modifiers for evasion and accuracy

We don't really need the table though, because mathematically the whole thing can be written as:

$$\frac{max(2, 2 + (accuracy - evasion))}{min(2, 2 - (accuracy - evasion))}$$

Which is much simpler.

When we were building Of Mages and Magic, we originally had these two modifiers included. Figuring out a suitably transparent way to incorporate them into your Mage's design was a little difficult though. We thought it would make more sense to try and fold these modifiers into an existing stat. It also made more intuitive sense to us that slow mages should find it harder to hit faster mages and vice-versa. So, we removed the evasion and accuracy modifiers and instead compute the probability of evasion using your speed stats.

When your mage makes an attack, the target's speed is subtracted from your speed and the total is divided by the sum of the two speeds, like this:

$$\frac{speed_{caster} - speed_{target}}{speed_{caster} + speed_{target}}$$

This equation gives us a percentage which is between -100% and $+100\%$. We multiply this value by 6, which gives us a modifier that is very similar to the evasion/accuracy stats used in Pokémon. We translate that stat to the fractional values given in our table and multiply the result by the accuracy of the spell.

We then generate a random number between 0 and 100. If the number is greater than the final accuracy of the move, your attack misses. Otherwise the attack hits. In code, this algorithm looks a bit like this:

```
def target_evades(spell, caster, target):
    # Get respective target speeds
    speed_target  = target.get_stat("speed")
    speed_caster = caster.get_stat("speed")

    # Compute accuracy modifier based on difference in speeds
    modifier =
        float(speed_caster - speed_target)/max(1,float(speed_caster+speed_target)↩
            )
    modifier = int(modifier*6)
    modifier = float(max(2, 2 + modifier))/max(2, 2 - modifier)

    # Computer overall accuracy and test for hit using random variable
    accuracy = min(100, spell.accuracy * modifier)
    return random.randint(0, 100) > accuracy
```

3 How Do We Check For A Critical Hit

Critical hits are when your mage gets a lucky hit that does extra damage to your opponent. They ignore any boosts applied to your enemy's defense and any reductions applied to your own attack. They also double the amount of damage that your attack does after all other damage computations have been applied.

If you check the document which describes your spells, you'll notice that they all have a *Critical Hit Probability* value. That number represents a percentage, so it's somewhere between 0 and 100. The bigger that value is, the greater the chance that the spell will do critical damage.

In the code, after you cast a spell, we generate a random number between 0 and 100. If the resulting generated number is less than the *Critical Hit Probability* of the attack, then you've managed to do critical damage.

To use an example, let's take the *Fireball* spell which has a *Critical Hit Probability* of 8. Our code generates a random number and the result is 7. Because 7 is less than 8, your spell is a critical hit. The piece of code which does that test looks like this:

```
def is_critical_hit(spell):
    return random.randint(0, 100) < spell.critical_hit_prob
```

If you do get a critical hit, then before we start computing damage, we may need to make some changes to your attack or your target's defense.

```
# Critical hits ignore positive defense modifiers and negative attack modifiers
if critical_hit:
```

```
attack = max(caster.get_stat("attack"), caster.get_base_stat("attack"))
defense = min(target.get_stat("defense"), target.get_base_stat("defense"))
```

4 How Is Damage Computed

Finally we reach the point where we can calculate how much damage the attack does. Again, we're working heavily from the implementation of the damage formula from the original Pokémon games. Specific details of what the damage formula looked like has changed from game to game, but in general it's written a bit like this:

$$\left(\frac{\left(\frac{2 \times level_{caster}}{5} + 2 \right) \times attack_{caster} \times power_{spell}}{50 \times defense_{target}} \right) + 2$$

All remainders are rounded down to the nearest whole number. Notice that the inclusion of the +2 component at the end ensures that even the weakest attack will at least do some small amount of damage.

Overall this is a pretty nice formula. It scales well as attack, defense and spell power changes and ensures some minimum amount of damage. We don't *have* to use this formula and if we wanted to change how our battles played out, this would be a good place to start.

Our mages don't have levels, so we chose to replace that portion of the equation with a constant value of 4. This is pretty low, but if we set it much bigger, we'd need to allocate a lot more stat points to our mages before we'd see appreciable differences in ability. For now, it's best to keep this number small. After some simplification, the end result looks like this:

$$\frac{2}{25} \times \frac{attack_{caster} \times power_{spell}}{defense_{target}} + 2$$

If the game needs to be balanced, we'll tweak the values in this formula to keep everything fair. If we want to increase the effect of your attack stat, then we'll increase the numerator of the constant fraction. If we want to increase the effect of your defense stat, then we'll increase the denominator. This type of flexibility is extremely important in the design of a good algorithm. This is why we chose to use the Pokémon formula. It is a well tested, proven approach to building battle mechanics for this kind of game. In code, it looks like this:

```
damage = 2 * attack * spell.power//max(1,defense)
damage = damage//25
damage += 2
```

Note the line `max(1,defense)` which protects against division by zero if you set your defence to 0. The `//` symbol in the division makes sure that there is no decimal place in the result.

5 Final Stage

The final stage of attacking an opponent is to include the effects of elemental types and critical hits (if applicable) in the damage. We then apply the result to the target.

The use of elements is quite simple. If the caster's element is strong against the target's element, then the damage is multiplied by 2. If the caster's element is weak against the target's element, then the damage is divided by 2.

Finally if the attack was a critical hit, we multiply the damage by 2. This number is (at last) subtracted from your target's health.

The entire body of code that we've discussed in this document can be seen on the following page.

```
def target_evades(self, caster, target):
    # Get respective target speeds
    evasion  = target.get_stat('speed')
    accuracy = caster.get_stat('speed')

    # Compute accuracy modifier based on difference in speeds
    modifier = float(accuracy - evasion)/max(1,float(accuracy+evasion))
    modifier = int(modifier*caster.modifier_minmax)
    modifier = float(max(2, 2 + modifier))/max(2, 2 - modifier)

    # Computer overall accuracy and test for hit using random variable
    accuracy = min(100, self.accuracy * modifier)
    return random.randint(0, 100) > accuracy

# Simple computation of critical hit depending on critical_hit_prob of spell
def is_critical_hit(self):
    return random.randint(0, 100) < self.critical_hit_prob

def compute_damage(self, caster, target, critical_hit=False):
    if critical_hit:
        # Critical hits ignore +ive defense mods and -ive attack mods
        attack = max(caster.get_stat('attack'),
                     caster.get_base_stat('attack')
                )
        defense = min(target.get_stat('defense'),
                     target.get_base_stat('defense')
                )
    else:
        # If not critical hit, include modifiers
        attack = caster.get_stat('attack')
        defense = target.get_stat('defense')

    # Compute damage using formula
    damage = 2 * attack * self.power//max(1,defense)
    damage = damage//25
    damage += 2

    # Check elemental damage and modify accordingly
    if self.element.is_strong_against(target.element):
        damage *= 2
    elif self.element.is_weak_against(target.element):
        damage //= 2

    # Check critical hit damage and modify accordingly
    if critical_hit:
        damage *= 2

    return damage

def apply_effect(self, caster, target):
    # Test for evasion and report if target dodged
    if self.target_evades(caster, target):
        print("{} evades the attack.".format(target.name))
    else:
        # Apply damage
        damage = self.compute_damage(caster, target, self.is_critical_hit())
        target.take_damage(damage)
```