Controlled Chaos: Randomization in Level Design

An introduction to integrating randomness in games using the example of Qake.

Hans-Peter Dietz
hp@jandh.org | http://hp.jandh.org

Keywords: Randomization, Level Design, Game Mapping, Game Design, Difficulty Progression, Balancing

Copyright © 2015 J&H | http://jandh.org

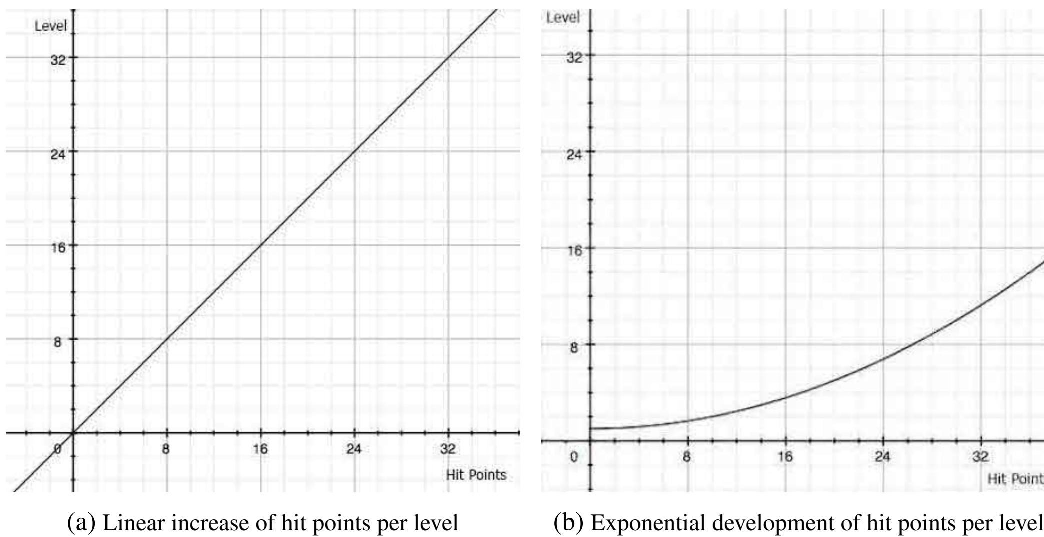
From the first conception of our arcade game QAKE, it was clear to us that we wanted some sort of randomness somewhere within the game. We considered many different approaches, but sometime down the road we decided that this *somewhere* was going to be the levels themselves.

Why would want that? Replayability! - Randomness is a great addition because it enables replayability. If the levels of the game feature random factors, every gaming session can be unique and offer a new, exciting challange to the player. Furthermore, it bears the potential to make players feel *lucky*. Think of roll-playing-games like DIABLO III ®[2], where loot drops randomly. If this wasn't the case but drops would be static, the game would loose a lot of appeal.

Figure 1: "I guarantee it." by Sherlock_Horse on 9gag.com [4]

In the spirit of this 9gag post *(see figure 1)*, we established that our game was going to be like those good old classics - but with a twist: There should be no *hard* finish line *(in terms of a final level)* for the game, but an infinite amount of ever-changing levels. To narrow things down, we established that we wanted randomized enemy configurations for each level, while maintaining comparable difficulty levels.

With this decision the question of *"how does one design and implement randomness in levels?"* arose. To answer this question many factors need to be considered - one of which is progression of difficulty. When you decide how the difficulty will progress within your game, you start off by establishing relevant factors. Maybe your game contains enemies that can be eliminated and therefore have a certain amount of hit points. These hit points then become a factor of difficulty, i.e. an enemy presumably is harder to kill if it has more hit points. Maybe your game is time-based and the player has to finish a certain task within a given timeframe. This timeframe then becomes a factor of difficulty, i.e. a shorter timeframe makes it harder for the player to succeed.



(a) Linear increase of hit points per level (b) Exponential development of hit points per level

Figure 2: Difficulty development example: Hit points

Most major games incorporate several of these factors, often interrelated with each other, forming complex level systems.

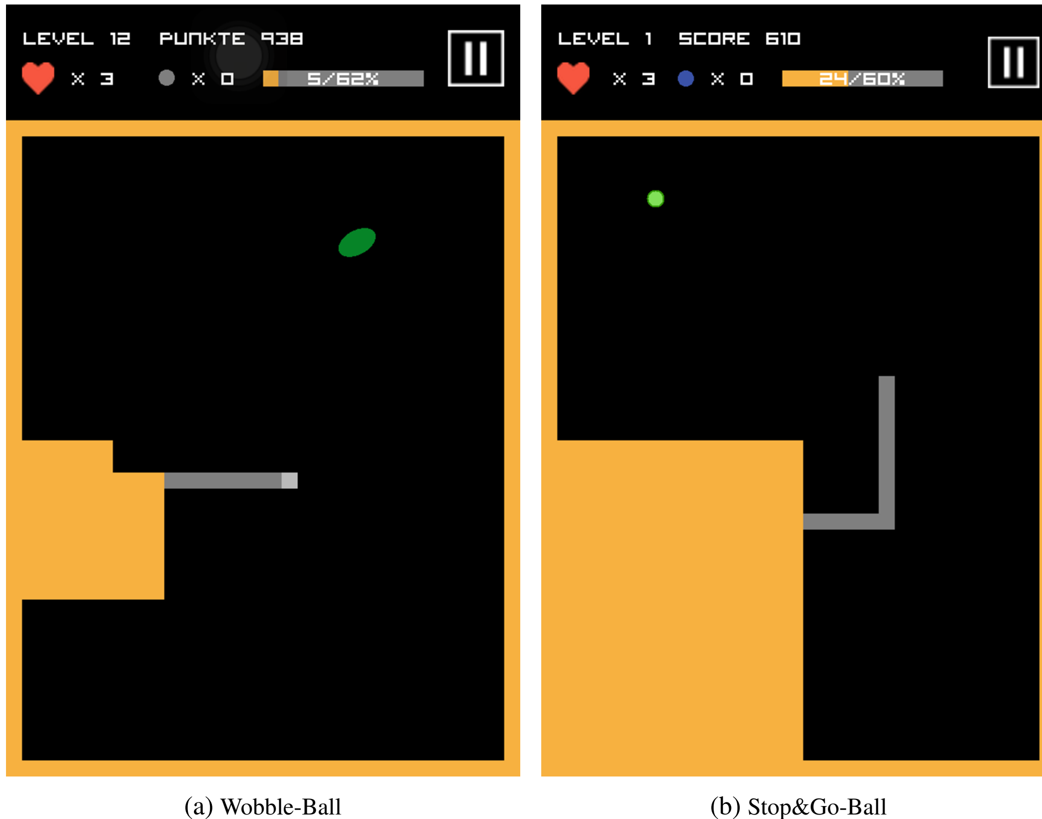
Concerning Qake, we established that there would be three major difficulty factors:

1. The number and type of enemy balls.
2. The speed of enemy balls.
3. The rate at which bombs and power ups spawn.

As a basic first approach you can start out by developing these factors by simple, linear incrementation *(see figure 2a)*.
More sophisticated and more commonly employed are exponential developments *(see figure 2b)*, where factors change slowly in the beginning and increments get bigger with later levels.

Usually not all factors are treated equally but develop at different gradients. In Qake, for example, ball movement speed increases linearly while the number of balls progresses exponentially.

On a side note: An easy way to determine suitable functions for your factors is to use tools like Grapher *(built-in OSX tool)* or gretl[1]. Just enter a function based on an educated guess and tailor it to your needs until you are satisfied.



(a) Wobble-Ball (b) Stop&Go-Ball

Figure 3: Different enemy balls in QAKE

Returning to our random component, we so far have laid down that we wanted the number of balls to increase exponentially but configurations to be random.
Since, however, different ball types also bear different difficulty levels, i.e. a Wobble-Ball *(see figure 3a)* in QAKE is arguably a *harder* enemy than e.g. a Stop&Go-Ball *(see figure 3b)*, we have a complex interrelation, which has to be taken into account when deciding which balls are going to live in each level.

Our approach to solve this problem incorporates a concept that might be familiar from real time strategy games like STARCRAFT ®[?]: Supply. The concept is rather simple and can easily be explained by a real-world metaphor: Entities need to eat, drink, etc. in order to live. In other words: they require supplies. If you do not have the required supplies to spare, you cannot entertain the unit. Simple as that.
Different entities, in our case ball types, are therefore associated with different quantities of required supply they *eat up* in order to live in the game.
A basic SPACEMARINE from Starcraft for example requires one supply, while a TANK - a massive

siege unit - requires three *(see figure 4)*.

Figure 4: Excerpt from the Starcraft 2®Game Guide [3]

Extracting this concept and adapting it to our game allows us to incorporate randomness into our difficulty progression: Instead of increasing the number of balls per level directly, we increase the supply available for balls, which is then allocated by choosing pseudo-randomly until exhausted.
A basic implementation might be nothing more than a few lines of code *(see listing 1)*.

Listing 1: Example: Compiling random enemy types from three tiers based on supply

```
// retrieve supply based on some function
supply = getSupplyForLevel(currentLevel);
enemies = [];

while(supply > 0){
        // retrieve a random enemy based on supply left
        if(supply > 2){
                i = randInt(1, 3);
        } else if(supply > 1){
                i = randInt(1, 2);
        } else {
                i = 1;
        }
        switch(i){
                case 1:
                        enemies.add(new EnemyTier1());
                        break;
                case 2:
                        enemies.add(new EnemyTier2());
                        break;
                case 3:
                        enemies.add(new EnemyTier3());
                        break;
        }
        supply -= i;
}
```

On a side note: A good way to acquire random numbers in Objective-C (or any C-dialect) is to use the `arc4random()` functions. See listing 2 for a few utility functions you are free to use in your projects.

With such a randomized population algorithm it is possible to achieve a lot of different configurations even with only a few different enemy types. Some basic statistics help us out in determining how many are possible:
Given our three enemy tiers *(Tier1 ●, Tier2 ● and Tier3 ● with respective supply-requirements of 1, 2 and 3)* and a supply cap of 6, possible configurations are:

1. ●●●●●●
2. ●●●●●
3. ●●●●
4. ●●●
5. ●●●●
6. ●●●
7. ●●

If we further increase the available supply, the number of possible configurations increases exponentially.
Try listing all possible configurations with a supply cap of 9 for example and you will see that possibilities become vast very fast. To see the result in action have a look at figure 5 where you can see three different variations of QAKE's level 33. Though the configurations differ significantly our approach allows us to maintain comparable difficulty levels, facilitating great replayability and a unique experience at every gaming session.

We hope you enjoyed this article and that it comes in handy to those of you who want to integrate some randomness into their games!

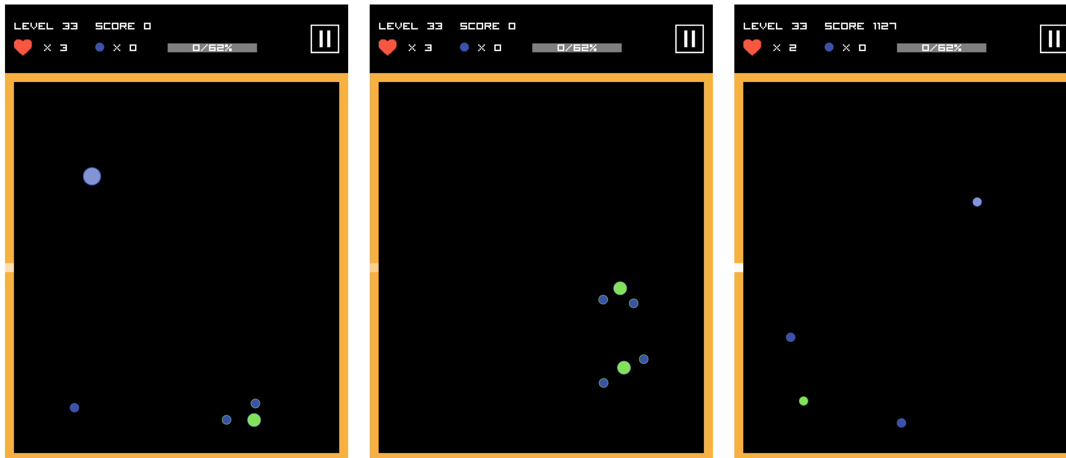


Figure 5: Different ball configurations of level 33 in QAKE

Listing 2: Random number generation functions in Objective-C

```
+(float)randomFloatBetween:(float)min and:(float)max {
    float diff = max - min;
    return (((float) (arc4random() % ((unsigned)RAND_MAX + 1)) /
        RAND_MAX) * diff) + min;
}

+(int)randomIntBetween:(int)min and:(int)max {
    return (int)(min + arc4random_uniform(max + 1 - min));
}

+(NSUInteger)randomNSUIntegerBetween:(NSUInteger)min and:(NSUInteger)
   max{
    return (NSUInteger)(min + arc4random_uniform((unsigned int)max + 1
        - (unsigned int)min));
}

+(CGFloat)randomCGFloatBetween:(CGFloat)min and:(CGFloat)max{
    CGFloat diff = max - min;
    return (((CGFloat)(arc4random() % ((unsigned)RAND_MAX + 1)) /
        RAND_MAX) * diff) + min;
}
```

References

[1] Allin Cottrell, Riccardo Lucchetti. Gnu Regression, Econometrics and Time-series Library. http://gretl.sourceforge.net/. Accessed on Apr. 13, 2015.

[2] Blizzard Entertainment, Inc. Diablo III. http://us.battle.net/d3/. Accessed on Apr. 13, 2015.

[3] Blizzard Entertainment, Inc. Starcraft II Game Guide: Terran Units. http://us.battle.net/sc2/en/game/unit/#terran. Accessed on Apr. 13, 2015.

[4] Sherlock_Horse. I guarantee it. http://9gag.com/gag/a8b5GEY. Accessed on Apr. 13, 2015.