

# 3

## Advanced Randomness Techniques for Game AI *Gaussian Randomness, Filtered Randomness, and Perlin Noise*

*Steve Rabin, Jay Goldblatt, and Fernando Silva*

3.1	Introduction	3.4	Technique 3: Perlin Noise for Game AI
3.2	Technique 1: Gaussian Randomness	3.5	Conclusion
3.3	Technique 2: Filtered Randomness		

### 3.1 Introduction

Game programmers have a special relationship with the `rand()` function. We depend on it for variation in our games, using it to keep our gameplay fresh and our NPCs from becoming predictable. Whether it's decision making, gameplay events, or animation selection, the last thing we want are repetitive characters or predictable gameplay; therefore randomness has become an essential tool.

However, randomness is a fickle beast, and humans are particularly bad at accessing or reasoning about it. This makes it easy to misuse or misunderstand what randomness actually provides. This chapter will introduce three advanced techniques that our trusty old friend `rand()` simply isn't capable of delivering.

The first technique involves discarding uniform randomness and embracing *Gaussian randomness* for variation in agent characteristics and behavior. Whether it's the speed of a unit, the reaction time of an enemy, or the aim of a gun, in real life these biological and

---

physical phenomena display *normal* (Gaussian) distributions, not uniform distributions. Once the difference is understood, you will find dozens of uses for Gaussian randomness in the games you make.

The second technique is to manipulate randomness to appear more random to players over short time frames. Randomness notoriously *does not look random* when looking at small isolated runs, so we will aim to fix this with *filtered randomness*.

The final technique is to use a special type of randomness that isn't uniform or Gaussian, but rather generates a wandering characteristic, where consecutive random numbers are related to each other. Often used in graphics, *Perlin noise* can be leveraged for situations where behavior varies randomly over time in a smooth manner. Whether it is movement, accuracy, anger, attention, or just being in the groove, there are dozens of behavior characteristics that could be varied over time using *one-dimensional* Perlin noise.

With each of the three techniques, there are demos and C++ libraries available on the book's website (<http://www.gameapro.com>) that you can drop right into your game.

### 3.2 Technique 1: Gaussian Randomness

Normal distributions (also known as *Gaussian distributions* or *bell curves*) are all around us, hiding in the statistics of everyday life. We see these distributions in the height of trees, the height of buildings, and the height of people. We see these distributions in the speed of shoppers strolling in a mall, the speed of runners in a marathon, and the speed of cars on the highway. Anywhere we have a large population of creatures or things, we have characteristics about that population that display a normal distribution.

There is randomness in these distributions, but they are not *uniformly* random. For example, the chance of a man growing to be 6 feet tall is not the same as the chance of him growing to a final height of 5 feet tall or 7 feet tall. If the chance were the same, then the distribution would be uniformly random. Instead, we see a normal distribution with the height of men centered around 5 feet 10 inches and dropping off progressively in the shape of a bell curve in either direction. In fact, almost every physical and mental characteristic has some kind of average, with individuals varying from the average with a normal distribution. Whether it's height, reaction time, visual acuity, or mental ability, these characteristics will follow a normal distribution among a given population.

Some random things in life do show a uniform distribution, such as the chance of giving birth to a boy or a girl. However, the large majority of distributions in life are closer to a normal distribution than a uniform distribution. But why?

The answer is quite simple and is explained by the *central limit theorem*. Basically, when many random variables are added together, the resulting sum will follow a normal distribution. This can be seen when you roll three 6-sided dice. While there is a uniform chance of a single die landing on any face, the chance of rolling three dice and their sum equaling the maximum of 18 is not uniform with regard to other outcomes. For example, the odds of three dice adding up to 18 is 0.5% while the odds of three dice adding up to 10 is 12.5%. Figure 3.1 shows that the sum of rolling three dice actually follows a normal distribution.

So now that we've shown that the addition of random variables results in a normal distribution, the question still exists: Why do most distributions in life follow a normal distribution? The answer is that almost everything in the universe has more than one contributing factor, and those contributing factors have random aspects to them.

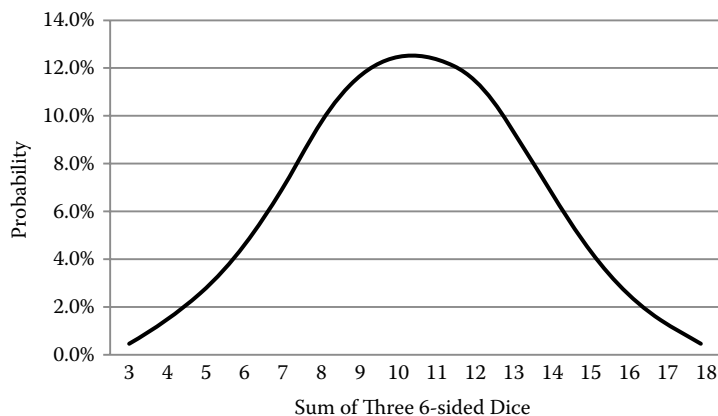


Figure 3.1

The probability of the sum of rolling three six-sided dice will follow a normal distribution, even though the outcomes of any single die have a uniform distribution. This is due to the central limit theorem.

For example, let's take the distribution of mature tree heights in a forest. What determines how tall a tree will grow? Mature tree height is influenced by its genes, precipitation, soil quality, air quality, amount of sunlight, temperature, and exposure to insects or fungi. For an entire forest, each tree experiences varying aspects of each quality, depending on where the tree is located (on the side of a hill versus in a valley, for example). Even two trees right next to each other will experience subtle differences in contributing factors. The final height of the tree is, in essence, the sum of the effects of each individual factor. In other words, the effect of the qualities that influence the tree's height are additive, so they result in a normal distribution of height among all trees. As you might imagine, it is possible to construct a similar argument for almost any other property of a biological system or physical system. Normal distributions are all around us.

### 3.2.1 Generating Gaussian Randomness

Now that we've shown how common normal distributions are in real life, it makes sense to include them in our games. With that in mind, you might be wondering how Gaussian randomness is generated. The previous dice example gives us a clue. If we take three uniform random numbers (generated from `rand()`, for example) and add them together, we can generate random numbers with a normal distribution.

To be more precise, the central limit theorem states that the addition of uniform random numbers in the range of  $[-1, 1]$  will approach a normal distribution with a mean of zero and a standard deviation of  $\sqrt{K/3}$ , where  $K$  is how many numbers are summed. If we choose  $K$  to be 3, then the standard deviation is equal to 1, and we will get nearly a *standard normal distribution*. The code in Listing 3.1 shows how easy it is to create Gaussian randomness.

The code in Listing 3.1 is sufficient for generating Gaussian randomness in games. However, there are some hidden nuances that need to be explained. A true normal distribution will produce values far into the tails, beyond the range of  $[-3, 3]$  provided by Listing 3.1. While certain financial or medical simulations might want an accurate normal

---

**Listing 3.1** Gaussian randomness can be generated by adding three uniformly random numbers. In this example, the uniform random numbers are created with an XOR-shift pseudo-random number generator. The function returns numbers in the range of  $[-3.0, 3.0]$  with 66.7% of them falling within one standard deviation, 95.8% falling within two standard deviations, and 100% falling within three standard deviations.

```
unsigned long seed = 61829450;
double GaussianRand()
{
    double sum = 0;
    for (int i = 0; i < 3; i++)
    {
        unsigned long holdseed = seed;
        seed ^= seed << 13;
        seed ^= seed >> 17;
        seed ^= seed << 5;
        long r = (Int64)(holdseed + seed);
        sum += (double)r * (1.0/0x7FFFFFFFFFFFFFFF);
    }
    return sum; //returns [-3.0, 3.0] at (66.7%, 95.8%, 100%)
}
```

distribution (with a 1 in a million chance of a  $-4$  value), games do not, so it is good to have a guarantee that the generated values will lie within the  $[-3, 3]$  range.

### 3.2.2 Applications of Gaussian Randomness

There are many uses for Gaussian randomness in AI [Mark 09], but one that is very visible within games is the aiming of projectiles [Rabin 08]. While many games probably don't perturb projectiles at all or perturb them using uniform randomness, the correct method is to apply Gaussian randomness.

Figure 3.2 shows both uniform and Gaussian bullet spreads on a target (generated from the sample demo on the book's website). It should be evident that the Gaussian one on the right looks much more realistic, but how is it generated? The trick is to use polar coordinates and a mix of both uniform and Gaussian randomness. First, a random uniform angle is generated from 0 to 360°. This value is used as a polar coordinate to determine an angle around the center of the target. It is important for this value to be uniform because there should be an equal chance of any angle. Second, a random Gaussian number is generated to determine the distance from the center of the target. By combining the random uniform angle and the random Gaussian distance from the center, you can recreate a very realistic bullet spread.

Other applications for Gaussian randomness in games include any aspect of an NPC that should vary within a population. These might include:

- Average or max speed
- Average or max acceleration
- Size, width, height, or mass

- Visual or physical reaction time
- Fire or reload rate for firing
- Refresh rate or cool-down rate for healing or special abilities
- Chance of missing or striking a critical hit

One thing to think about is whether you want to vary a characteristic between members of a population, vary that characteristic each time it is used, or both. For example, an individual soldier might have a slower innate rate of fire than average for the population (determined at unit creation time using Gaussian randomness), but any one instance of that unit firing can then also vary within a normal distribution around the unit's innate rate (sometimes the unit is a little faster at firing and sometimes a little slower).

Now imagine a group of 30 units all firing at the enemy. If each unit has an innate fire rate (normally distributed) and each unit varies around that rate (also normally distributed), then the emergent behavior of the group should be extremely natural, with no units firing in lock-step with each other.

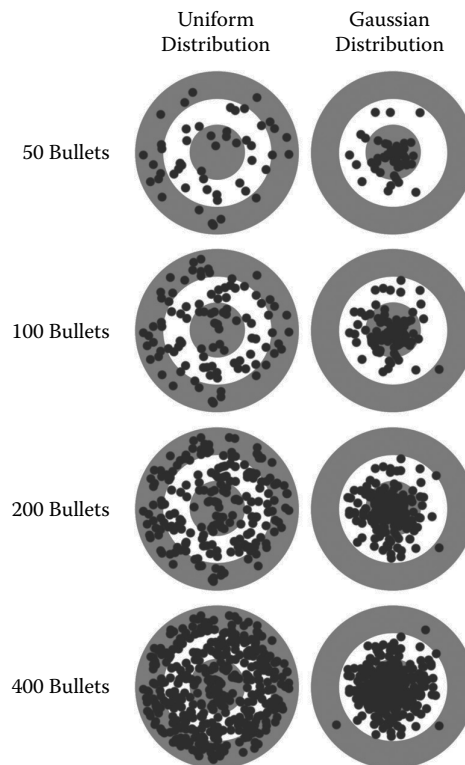


Figure 3.2

Uniform and Gaussian distributions for bullet spreads on a target. Note that each ring in the target represents a standard deviation, with 66.7% of the Gaussian bullets hitting within the innermost ring and 95.8% of the bullets hitting within the two innermost rings. This demo can be found on the book's website.

---

### 3.3 Technique 2: Filtered Randomness

Let's just say it: Randomness is too random (for many uses in games). At first, this seems like a ridiculous statement, but there is a mountain of evidence that humans don't see small runs of randomness as being random [Diener et al. 85, Wagenaar et al. 91]. However, this brings up an interesting follow-up question: If humans don't think small runs of randomness look random, then what do they actually think? Perhaps they think that the thing creating the sequence is either broken, rigged, or cheating—all of which are terrible qualities to attribute to a game or an AI.

#### 3.3.1 Small Runs of Randomness Don't Look Random

Now that we have the outrageous statement out there, let's back it up. First, let's establish that small runs of randomness don't look random. Grab a piece of scrap paper and start writing down 0's and 1's in a random sequence with a 50% chance of each—do it until you have a list of 100 numbers. Go ahead and *really* try this. No really, it'll make you a better person. We'll wait ....

Now to make this truly fair, take out a coin and start flipping it, recording the sequence of heads and tails as 0s and 1s. Flip it 100 times to make a comparable list to the one you created with your mind. Again, we'll wait. You must actually do this if you want the big payoff....

Now we can also compare the two lists you made to a list created by a pseudo-random number generator (PRNG) such as `rand()`, with the same 50% chance of either a 0 or a 1. The following is a random generated sequence of 100 coin flips.

```
011011100001100001010000001001011110011100111000110  
10101011011111101001011110011111010111110100011
```

Notice anything different between your hand-generated list, the coin flip list, and the PRNG generated one? It's very likely that the coin flip and PRNG list are a lot more "clumpy" containing many more long runs of 0's or 1's compared to your hand-generated list. What most people don't realize is that real randomness almost always contains these long runs and that these are very typical and to be expected. However, most people simply don't believe a fair coin or real randomness will produce those long runs of heads or tails. In fact, until you've actually flipped a coin yourself and seen it happen, it is extremely difficult to internalize this lesson (which is why we wanted you to actually do it).

So how does this apply to games? Many games include situations where a uniformly distributed random number determines something that will affect the player, either positively or negatively. If the player has the expectation of certain odds and the game appears to negatively defy those odds in the short term (especially with an outcome that harms the player), then the player thinks the game is broken or cheating [Meier 10]. Remember that we've now entered the realm of psychology, and we have temporarily left mathematics [Mark 09]. If the player thinks the game is cheating, then the game effectively *is* cheating despite what is really happening; perception is far more important than reality when it comes to the player's enjoyment of the game.

For example, imagine that the game designer determined that enemies should strike with a critical hit 10% of the time. Unfortunately, after many battles during the course of a 30 hour game, the player has a very high probability of being hit three times in a row with a critical hit! Did the game designer ever want this to happen? No! But this is the reality

---

of random numbers. In many situations, *randomness is too random* for what the game designer actually wanted.

### 3.3.2 Randomness Doesn't Look Random: The Fix

How can random numbers be forced to look more random to humans? The truthful, but nonhelpful, answer is to make the numbers slightly less random in a very special way. What we will do is trade a very tiny bit of randomness integrity to achieve a sequence that looks more random to humans.

The strategy is very simple: When generating a random sequence of numbers, if the next number will hurt the appearance of randomness, pretend that you never saw it and generate a new number [Rabin 04]. It really is that simple. With that said, since we're dealing with psychology here, the actual implementation of this strategy is subjective. Also, if the elimination of numbers is too overzealous, it can hurt the integrity of the randomness, so the actual implementation must be carefully considered.

### 3.3.3 Identifying Anomalies

Keeping the core strategy in mind, the first task is to identify the kinds of things that make a sequence look less random. As it turns out, there are really only two main causes:

1. The sequence has a pattern that stands out, like in the coin flip sequence 11001100 or 111000.
2. The sequence has a long run of the same number, as in the coin flip sequence 0101111110.

We can classify both of these causes as “anomalies” that look nonrandom (weird or unusual) to humans. The goal is then to write some kind of rules that will identify these anomalies. Once we have the rules, we can throw out the last number that triggers a rule (completes the offending pattern). As an implementation detail, our code will have to track the last 10 to 20 generated numbers, *per decision*, for our rules to examine. We will explore this more in a later section.

So what does a rule look like? That depends on the type of randomness being generated. Rules for a coin flip will be different from rules for a random range or Gaussian randomness. Unfortunately, there is no simple way around this since we are dealing with subjective human feelings.

#### 3.3.3.1 Filtering Binary Randomness

If a 50% chance is desired, like a coin flip, then the following rules will filter randomness in a way that will look more random to humans. Note that this is an ordered list of rules to check every time a new random number is generated. Additionally, only one rule should be allowed to trigger per new generated number.

1. If the newest value will produce a run of 4 or more, then there is a 75% chance to flip the newest value. This doesn't make runs of 4 or more impossible, but progressively much less likely (the probability of a run of 4 occurring goes from 1/8 to 1/128). Runs of a particular length can be prohibited altogether, but this will more negatively affect the integrity of the randomness.

- 
2. If the newest value causes a repeating pattern of four values, like 11001100, then flip the last value (so that the sequence becomes 11001101).
  3. If the newest value causes a repeating pattern of 111000 or 000111, then flip the last value.

Taking the binary generated random sequence from earlier in the article, here it is before and after filtering:

Before filtering:

```
011011000011000010100000010010111100111001110001110
101010110111111010010111100111110101111101000011
```

After filtering (underlined numbers were toggled from the original sequence):

```
011011000110001010100010010010111100111001110011110
10101110011101101001011100111001101011101101000110
```

### 3.3.3.2 Filtering Integer Ranges

Similar to binary randomness, rules can be constructed to filter out anomalies that occur with ranges of numbers. The following is a list of fairly aggressive rules that could be implemented. For this set of rules, any violation of a rule results in rerolling the value and then validating against the rules again:

1. Repeating numbers, like “7, 7” or “3, 3”
2. Repeating numbers separated by one digit, like “8, 3, 8” or “6, 2, 6”
3. A counting sequence of 4 that ascends or descends, like “3, 4, 5, 6”
4. Too many values at the top or bottom of a range within the last *N* values, like “6, 8, 7, 9, 8, 6, 9”
5. Patterns of two numbers that appear in the last 10 values, like “5, 7, 3, 1, 5, 7”
6. Too many of a particular number in the last 10 values, like “9, 4, 5, 9, 7, 8, 9, 0, 2, 9”.

Before filtering:

```
2231255222577750677564061448482102435500989388459
59607889964957780753281574605482138446235103745368
```

After filtering (highlighted numbers are thrown out since they violated a rule):

```
2231255222577750677564061448482102435500989388459
59607889964957780753281574605482138446235103745368
```

### 3.3.3.3 Filtering Floating-Point Ranges

To filter floating-point in the range of [0, 1], we’ll have to design rules that tend to avoid clumps of similar numbers and avoid increasing or decreasing runs. If any of these rules are violated, we’ll simply throw out the value and ask for a new random number that must pass all of the rules.



- 
1. Reroll if two consecutive numbers differ by less than 0.02, like 0.875 and 0.856.
  2. Reroll if three consecutive numbers differ by less than 0.1, like 0.345, 0.421, and 0.387.
  3. Reroll if there is an increasing or decreasing run of 5 values, such as 0.342, 0.572, 0.619, 0.783, and 0.868.
  4. Reroll if too many numbers at the top or bottom of the range within the last  $N$  values, such as 0.325, 0.198, 0.056, 0.432, and 0.216.

#### 3.3.3.4 *Filtering Gaussian Ranges*

Since Gaussian numbers are very similar to floating-point numbers, the same rules would apply. However, you might introduce the following rules to avoid particular anomalies that are unique to Gaussian numbers.

5. Reroll if there are four consecutive numbers that are all above or below zero.
6. Reroll if there are four consecutive numbers that lie within the second or third deviations.
7. Reroll if there are two consecutive numbers that lie within the third deviation.

#### 3.3.3.5 *Randomness Integrity*

The rules outlined in the last four sections are arbitrary and can be changed to be more or less strict. However, the stricter the rules, the less random the resulting values become. In the extreme case, very strict rules will overconstrain the sequence to the point where it might be possible to predict the next number, which would defeat the purpose of using random numbers in the first place.

At this point you should be asking yourself if *any* number of rules for filtering random numbers might significantly hurt the mathematical integrity of the randomness. The only way to categorically answer this is to run benchmarks on the filtered random numbers to measure the quality of the randomness. The open source program ENT will run a variety of metrics to evaluate the randomness, so it would be advisable to run these benchmarks if you design your own rules [Walker 08]. In general, as long as the rules don't overconstrain or predetermine the next number, as in the examples given, then the randomness will be suitable for almost all uses in game AI [Rabin 04].

### 3.3.4 Implementation Details for Filtered Randomness

When implementing filtered randomness, care must be taken to apply the algorithm to each particular *use* of randomness. Each unique random decision using filtered randomness needs to keep its own history from which to filter subsequent random numbers. Otherwise, since the sequence of numbers for a particular use is not consecutive in the overall sequence, pure randomness can creep back in despite your filtering. For example, if you need a random chance for a critical hit from the player, that sequence must be filtered separately from the random chance of a critical hit from an enemy. The two uses are independent, so a critical hit from one should not influence the subsequent chance of a critical hit from the other. One advantage of this is that you can (if you desire) alter your filter characteristics for different uses. For example, you might want to allow more sequences to appear when your character plays poker, or does some smithing, where

---

the sensation of being “on a lucky streak” could benefit the player’s experience, but still strongly constrain sequences of critical hits.

### 3.4 Technique 3: Perlin Noise for Game AI

If you are familiar with computer graphics, you have probably heard of *Perlin noise* [Perlin 85, Perlin 02]. This computer-generated visual effect was developed by Ken Perlin in 1983, who incidentally won an Academy Award for the technique due to its widespread use in digital effects for movies. Perlin noise is typically used as a component to generate organic textures and geometry. Figure 3.3 shows typical examples of Perlin noise textures.

While Perlin noise can be used to help provide an organic feel to visual effects (for example, procedural generation of smoke, fire, or clouds), it’s not immediately obvious how you could use such a technique for game AI. The key realization is that Perlin noise generates a form of randomness that is not uniform or normal, but rather can be described as coherent randomness, where consecutive random numbers are related to each other. This “smooth” nature of randomness means that we don’t get wild jumps from one random number to another, which can be a very desirable trait. But how could this be useful for game AI?

The first step is to visualize Perlin noise in one-dimension, as shown in Figure 3.4. This can be thought of as a random wandering signal (a series of related random numbers). We can use this signal to control the movement or variation of particular behavior traits of our AI characters over time. The following is a list of possibilities for game AI.

- Movement (direction, speed, acceleration)
- Layered onto animation (adding noise to facial movement or gaze [Perlin 97])
- Accuracy (winning or losing streaks, being in the groove, luck, or success)
- Attention (guard alertness, response time)
- Play style (defensive, offensive)
- Mood (calm, angry, happy, sad, depressed, manic, bored, engaged)

So while uniform or Gaussian randomness can be used to vary an individual’s physical or behavioral characteristics within a population, Perlin noise can be used to vary those characteristics *over time*. When you have a large population of characters, this can make

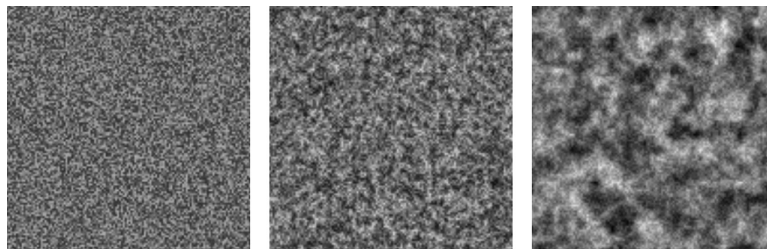


Figure 3.3

Three examples of Perlin noise generated textures with different levels of detail.

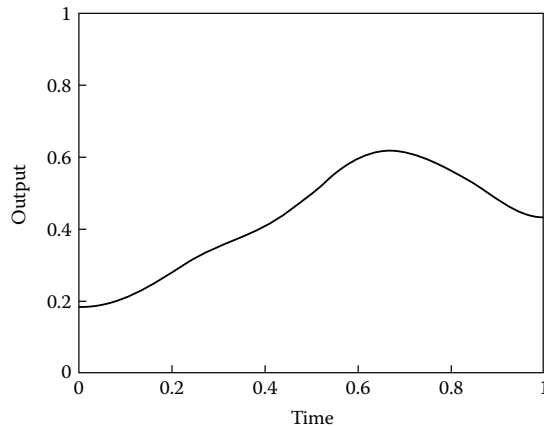


Figure 3.4

Perlin noise in one-dimension. This is a series of random numbers that smoothly wander over time (coherent randomness). We can control the look and feel by manipulating the algorithm that produces the numbers.

the simulation much more interesting, since each particular character can vary their behavior characteristics over durations spanning seconds, minutes, or hours.

Let's explore some of the examples from the previous list of possible game AI uses in more depth. Simple wandering can be achieved by varying the steering direction and speed over time. For arbitrary wandering, Perlin noise is a better alternative to the ad hoc coherent randomness for steering behaviors proposed by Craig Reynolds, because Perlin noise is far more configurable than Reynold's solution [Reynolds 99].

Hot and cold streaks can be purposely simulated, instead of accidentally occurring in retrospect. This is particularly useful since you can anticipate the streak and optionally make the player aware of it midway through, perhaps with an utterance such as, "Boys, I'm feeling lucky tonight!" While uniform randomness will have natural streaks, Perlin noise can be configured to control the behavior of the streaks and predict when you'll be moving into one.

Another possible use is to vary a character's mood in some way, for example, along the calm versus angry scale. While you could come up with an underlying simulation to generate any of these characteristics or mood shifts, it might be overkill and much simpler (development time-wise and computationally) to have them randomly generated using Perlin noise, especially in a large population when no one individual is being scrutinized by the player. However, the players might direct their focus on a character, in which case this apparent lack of rationale behind the deviations in behavior can pose a problem. One solution to this problem is to shift the simulation from Perlin noise to something more robust when it is determined that the character is being noticed. In this sense, Perlin noise can be an LOD level that can be supplanted when it is deemed that it might cause a break in reality [Sunshine-Hill 13a]. Additionally, if the character is now being watched by the player, it might be time to create an *alibi*, or backstory, for the AI, as described in another article in this book [Sunshine-Hill 13b].

---

### 3.4.1 Generating One-Dimensional Perlin Noise

Now that we know why we need one-dimensional Perlin noise, let's look at how we can generate it and craft the output to fit our needs. As we describe the algorithm, take note of the knobs and controls that will allow you to customize the randomness to your own preferences. These will be crucial to getting the most out of the algorithm. As you explore the generation part, you might want to get the demo on this book's website to follow along and try different settings.

While Perlin noise generation is difficult to explain due to the nuances of the math, we'll focus on giving you a more intuitive visual explanation through the use of figures. Note that the exact details can be seen in the example code within the demo.

In one-dimension, Perlin noise is constructed by first deciding how many *octaves* to use. Each octave contributes to the signal detail at a particular scale, with higher octaves adding more fine-grained detail. Each octave is computed individually and then they are added to each other to produce the final signal. Figure 3.5 shows one-dimensional Perlin noise, constructed with four octaves.

In order to explain how each octave signal is produced, let's start with the first one. The first octave is computed by starting and ending the interval with two different uniform random numbers, in the range  $[0, 1]$ . The signal in the middle is computed by applying a mathematical function that interpolates between the two. The ideal function to use is the S-curve function  $6t^5 - 15t^4 + 10t^3$ , because it has many nice mathematical properties, such as being smooth in the first and second derivatives [Perlin 02]. This is desirable so that the signal contained within higher octaves is smooth.

For the second octave, we choose three uniform random numbers, place them equidistant from each other, and then interpolate between them using our sigmoid function. Similarly, for the third octave, we choose five uniform random numbers, place them equidistant from each other, and then interpolate between them. The number of uniform random numbers for a given octave is equal to  $2^{n-1} + 1$ . Figure 3.5 shows four octaves with randomly chosen numbers within each octave.

Once we have the octaves, the next step is to scale each octave with an *amplitude*. This will cause the higher octaves to progressively contribute to the fine-grained variance in the final signal. Starting with the first octave, we multiply the signal by an amplitude of 0.5, as shown in Figure 3.5. The second octave is multiplied by an amplitude of 0.25, and the third octave is multiplied by an amplitude of 0.125, and so on. The formula for the amplitude at a given octave is  $p^i$ , where  $p$  is the *persistence* value and  $i$  is the octave (our example used a persistence value of 0.5). The persistence value will control how much influence higher octaves have, with high values of persistence giving more weight to higher octaves (producing more high-frequency noise in the final signal).

Now that the octaves have been appropriately scaled, we can add them together to get our final one-dimensional Perlin noise signal, as shown at the bottom right of Figure 3.5. While this is all fine and good, it is important to realize that for the purposes of game AI, you are not going to compute and store the entire final signal, since there is no need to have the whole thing at once. Instead, given a particular *time* along the signal, in the range  $[0, 1]$  along the x-axis, you'll just compute that particular point as needed for your simulation. So if you want the point in the middle of the final signal, you would compute the individual signal in each octave at time 0.5, scale each octave value with their correct

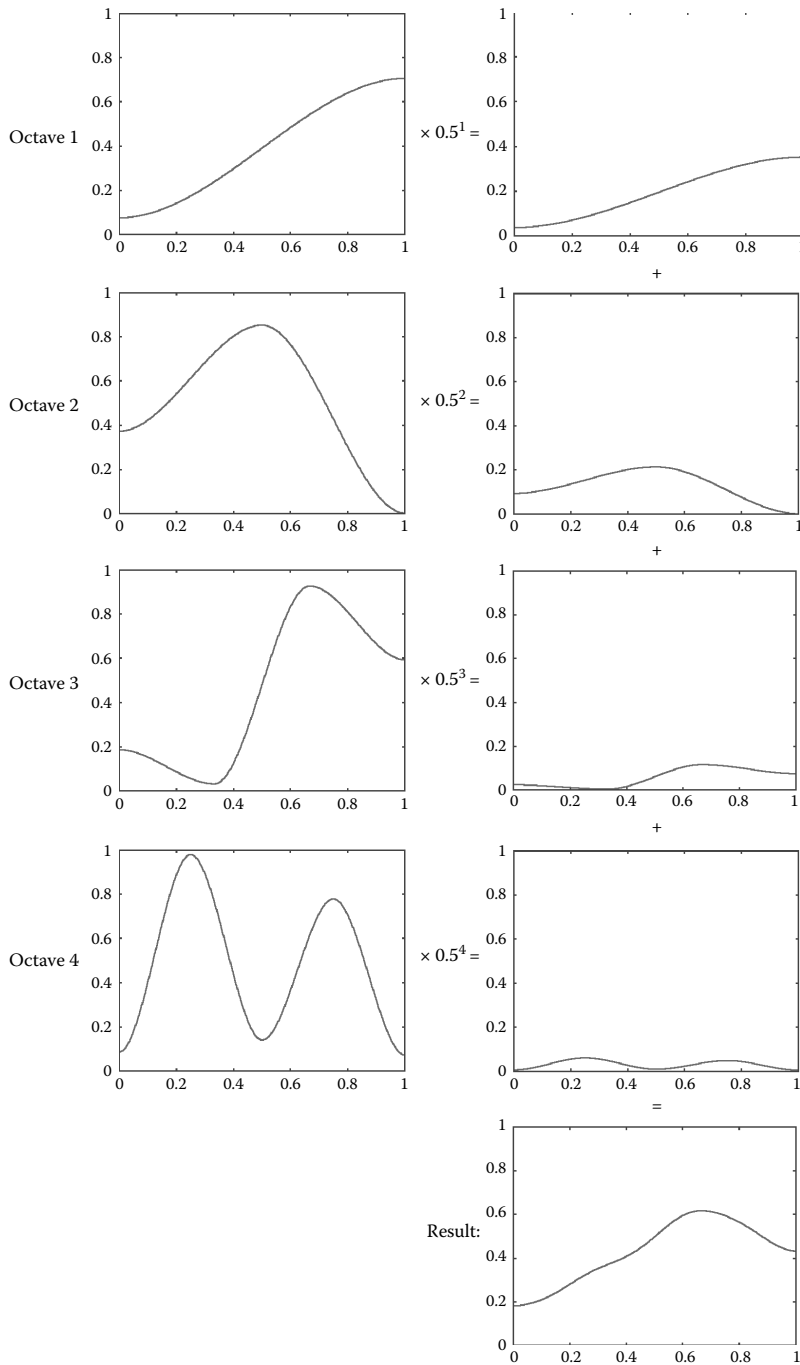


Figure 3.5  
Generation of one-dimensional Perlin noise with four octaves.

---

amplitude, and add them together to get a single value. You can then run your simulation at any rate by requesting the next point at 0.500001, 0.51, or 0.6, for example.

#### 3.4.1.1 Controlling Perlin Noise

As alluded to in the previous section, there are several controls that will allow you to customize the randomness of the noise. The following list is a summary.

- *Number of octaves:* Lower octaves offer larger swings in the signal while higher octaves offer more fine-grained noise. This can be randomized within a population as well, so that some individuals have more octaves than others when generating a particular behavior trait.
- *Range of octaves:* You can have any range, for example octaves 4 through 8. You do not have to start with octave 1. Again, the ranges can be randomized within a population.
- *Amplitude at each octave:* The choice of amplitude at each octave can be used to control the final signal. The higher the amplitude, the more that octave will influence the final signal. Simply ensure that the sum of amplitudes across all octaves does not exceed 1.0 if you don't want the final signal to exceed 1.0.
- *Choice of interpolation:* The S-curve function is commonly used in Perlin noise, with original Perlin noise using  $3t^2 - 2t^3$  [Perlin 85] and improved Perlin noise using  $6t^5 - 15t^4 + 10t^3$  (smooth in the second derivative) [Perlin 02]. However, you might be able to get other interesting effects by choosing a different formula [Komppa 10].

#### 3.4.1.2 Sampling Perlin Noise Beyond the Interval

When looking at Figure 3.5, it might have occurred to you that there is a problem once we get to the end of the interval (the very right edge of the signal). Once we have sampled at time 1.0, there is no place to go since the signal abruptly stops. One solution is to just start over again at time 0.0, but that would cause a huge discontinuity and repetitive behavior. Fortunately, there is a more elegant solution.

When we need to sample beyond time 1.0, we can generate a completely new Perlin noise signal that attaches to the end of our current signal. If we simply copy all of the uniform random numbers at the right edge of each octave into the far left edge of each new octave, then our signal will seamlessly migrate into a new Perlin noise signal (with the smoothness of the transition dependent on the particular interpolation function used). Of course, we'll need newly generated uniform random numbers for the remaining slots within each new octave, but this is desirable so that we begin generating a completely new signal.

#### 3.4.2 Game AI Perlin Noise Demo

On the book's website (<http://www.gameaipro.com>), you'll find the Perlin noise demo that accompanies this section. In the demo you can play with the various knobs to control the noise generation. Perlin noise is used to vary individual agent wandering, speed, and aggressiveness. In addition, look at the source code to discover the exact details of how Perlin noise is generated.

---

## 3.5 Conclusion

This article presented three advanced randomness techniques that help augment our old friend the `rand()` function. While uniform randomness is the backbone of variation in games, techniques such as Gaussian randomness, filtered randomness, and Perlin noise can offer cool tricks that plain old `rand()` just can't deliver on its own.

In the quest for realism, Gaussian randomness gives us normal distributions that help mimic the actual variation that surrounds us in real life, whether it's natural variations in physical traits, cognitive ability, reaction time, or bullet spread. In order to keep the player content, we can leverage filtered randomness to ensure that influential random decisions that impact the player either positively or negatively appear fair and unbiased. Finally, Perlin noise isn't just for graphics anymore. The one-dimensional coherent randomness of Perlin noise can be used to smoothly vary movement, animation, and dozens of other behavioral characteristics over time.

As a final note, all of the three techniques have accompanying code and demos that can be found on the book's website (<http://www.gameaipro.com>).

## References

- [Diener et al. 85] D. Diener and W. Burt Thompson. "Recognizing randomness." *American Journal of Psychology*. 98: 433–447, 1985.
- [Kompaa 10] J. Kompaa. "Interpolation Tricks." <http://sol.gfxile.net/interpolation/>, 2010.
- [Mark 09] D. Mark. *Behavioral Mathematics for Game AI*. Boston, MA: Course Technology, 2009.
- [Meier 10] S. Meier. "GDC 2010 Keynote address: Sid Meier." *Game Developers Conference, 2010*. Available online (<http://www.gamespot.com/sid-meiers-civilization-v/videos/gdc-2010-keynote-address-sid-meier-6253529/>).
- [Perlin 85] K. Perlin. "An image synthesizer." *Computer Graphics* 19(3). 1985.
- [Perlin 97] K. Perlin. "Layered compositing of facial expression." *SIGGRAPH 97*, Technical Sketch, 1997. Demo available at (<http://mrl.nyu.edu/~perlin/experiments/facedemo/>).
- [Perlin 02] K. Perlin. "Improving noise." *Computer Graphics* 35(3). 2002.
- [Rabin 04] S. Rabin. "Filtered randomness for AI decisions and logic." In *AI Game Programming Wisdom 2*, edited by Steve Rabin. Hingham, MA: Charles River Media, 2004, pp. 71–82.
- [Rabin 08] S. Rabin. "Using Gaussian randomness to realistically vary projectile paths." In *Game Programming Gems 7*, edited by Scott Jacobs. Hingham, MA: Charles River Media, 2008, pp. 199–204.
- [Reynolds 99] C. Reynolds. "Steering behaviors for autonomous characters." *Game Developers Conference, 1999*. Available online (<http://www.red3d.com/cwr/steer/>).
- [Sunshine-Hill 13a] B. Sunshine-Hill. "Phenomenal AI level-of-detail control with the LOD trader." In *Game AI Pro*, edited by Steve Rabin. Boca Raton, FL: CRC Press, 2013.
- [Sunshine-Hill 13b] B. Sunshine-Hill. "Alibi generation: fooling all of the players all of the time." In *Game AI Pro*, edited by Steve Rabin. Boca Raton, FL: CRC Press, 2013.
- [Wagenaar et al. 91] W. A. Wagenaar and M. Bar-Hille. "The perception of randomness." *Advances in Applied Mathematics*. 12: 428–454, 1991.
- [Walker 08] J. Walker. ENT: A Pseudorandom Number Sequence Test Program. <https://www.fourmilab.ch/random/>, 2008.