JDCS
디지털콘텐츠학회논문지
Journal of Digital Contents Society
Vol. 26, No. 1, pp. 195-202, Jan. 2025

Check for updates

실시간 게임 응용에 적합한 의사난수 생성 알고리즘 성능 분석 및 제안

도 성 진[1] · 김 동 회[2*]
[1]강원대학교 IT대학 전기전자공학과 학사과정
[2]강원대학교 IT대학 전기전자공학과 교수

Performance Analysis and Proposal of Unity Random Libraries for Real-Time Game Application

SeongJin Do[1] · Dong-Hoi Kim[2*]

[1]Undergraduate, Electrical and Electronic Engineering, IT College, Kangwon National University, Chuncheon 24341, Korea
[2]Professor, Electrical and Electronic Engineering, IT College, Kangwon National University, Chuncheon 24341, Korea

[요 약]

본 연구는 Unity Engine에서 자주 사용되는 랜덤 라이브러리와 알고리즘인 CryptoRandom, SystemRandom, UnityRandom, Xorshift의 성능을 비교하는 것을 목표로 한다. 연구의 주요 목적은 특히 실시간 게임 응용에 적합한 가장 빠른 라이브러리를 찾는 것이다. PC 플랫폼에서 생성 시간을 중심으로 실험을 진행한 결과, Xorshift가 가장 우수한 성능을 보였다. CryptoRandom은 Xorshift에 비해 약 8~16배 더 많은 시간이 소요되었으며, SystemRandom과 UnityRandom도 Xorshift보다 평균적으로 1.5~2배정도 더 느렸다. Xorshift는 난수 생성 속도가 가장 빨라 실시간 반응이 중요한 게임에 적합하다. 이에 반해 CryptoRandom은 속도가 느려, 대량의 난수를 생성해야하는 실시간 게임에는 적합하지 않다. 따라서 실시간 게임과 같이 성능이 중요한 게임에서는 Xorshift의 사용을 제안한다. 실험 결과, CryptoRandom은 XorShift에 비해 속도가 최대16배 느리고, System.Random과 UnityEngine.Random은 XorShift에 비해 최대 3배 느리게 동작할 수 있다는 것을 확인하였다.

[Abstract]

This study compares the performance of commonly used random libraries and algorithms in Unity Engine: CryptoRandom, SystemRandom, UnityRandom, and Xorshift. The primary objective is to identify the fastest library suitable for real-time game applications. Experiments conducted on a PC platform demonstrate that Xorshift provides the best performance in terms of generation time. Specifically, CryptoRandom takes approximately 8 to 16 times longer than Xorshift, while SystemRandom and UnityRandom are 1.5 to 2 times slower than Xorshift. Given its superior speed, Xorshift is particularly well-suited for real-time games, whereas CryptoRandom's slower performance makes it less ideal for scenarios requiring the rapid generation of random numbers. Therefore, this study recommends using Xorshift for real-time applications and aims to assist developers achieve efficient random number generation for game development.

색인어 : 알고리즘, 게임, 라이브러리, 성능 비교, 유니티 엔진
Keyword : Algorithm, Game, Library, Performance Comparison, Unity Engine

http://dx.doi.org/10.9728/dcs.2025.26.1.195

Received 13 November 2024; **Revised** 13 December 2024
Accepted 30 December 2024

***Corresponding Author; Dong-Hoi Kim**
Tel: +82-33-250-6349
E-mail: donghk@kangwon.ac.kr

Ⅰ. 서 론

게임 개발에서 랜덤성은 다양한 요소에 큰 영향을 미치는 중요한 요소 중 하나이다. 랜덤성은 캐릭터의 움직임, 아이템 생성, 이벤트 발생 등 게임의 재미와 예측 불가능성을 제공하기 위해 필수적이다. 특히, Unity Engine은 전 세계적으로 널리 사용되는 게임 개발 플랫폼으로, 다양한 랜덤 함수 라이브러리를 제공하여 개발자들이 게임의 다양한 랜덤 요소를 쉽게 구현할 수 있도록 돕는다. 이러한 랜덤 함수는 게임의 성능에 직접적인 영향을 미치며, 잘못된 선택은 게임의 프레임 저하와 같은 부정적인 결과를 초래할 수 있다. 따라서 랜덤 함수의 적절한 선택은 게임의 전반적인 성능을 좌우하는 중요한 요소이다.

게임 산업은 꾸준한 기술 발전을 바탕으로 빠르게 성장하고 있으며, 사용자들은 점점 더 현실적이고 몰입감 있는 경험을 기대하고 있다. 이로 인해 게임 개발자는 방대한 데이터를 처리하고, 더 복잡한 시스템을 구축하는 상황에 직면하게 되었다. 랜덤성은 이러한 복잡한 게임 시스템에서 중요한 역할을 하며, 캐릭터의 행동, 아이템 생성, 전투 시스템 등 다양한 게임 요소에 필수적이다. 게임에서 랜덤성은 게임의 재미와 흥미를 높이고, 예측 불가능한 요소를 추가하여 사용자들에게 독특한 경험을 제공한다[1].

게임 개발의 진입 장벽은 과거에 비해 크게 낮아졌으며, 이는 상용 게임 엔진의 발전 덕분이다. 과거에는 고비용의 게임 엔진을 직접 설계해야 했지만, 현재는 상용 엔진을 저렴하거나 무료로 이용할 수 있게 되면서 게임 개발 환경이 크게 개선되었다. 그중에서도 Unity Engine과 Unreal Engine은 대표적인 상용 엔진으로, 각각 다른 특성과 장점을 가지고 다양한 게임 개발에 널리 활용되고 있다[2],[3].

현재 Unity Engine은 다양한 랜덤 함수 라이브러리를 제공하고 있지만, 이들 각각의 성능에 대한 구체적인 비교 연구는 매우 부족하다. 많은 게임 개발자들은 랜덤 함수 선택 시 성능이나 효율성보다 사용 편의성에 의존해 익숙하거나 손쉽게 사용할 수 있는 라이브러리를 선택하는 경우가 많다. 이는 게임의 성능 최적화에 부정적인 영향을 미칠 수 있으며, 게임의 데이터에 복잡한 랜덤성을 적용해야 할 때 치명적인 성능 저하를 일으킬 수 있다.

본 연구의 목적은 Unity Engine에서 사용 가능한 네 가지 주요 랜덤 라이브러리 및 알고리즘인 CryptoRandom.Next, System.Random.Next, UnityEngine.Random.Range, XorshiftRandom의 성능을 비교 분석하는 것이다. 연구에서는 생성 시간을 주요 지표로 설정하여 각 랜덤 함수의 성능을 평가하고, 다양한 픽셀 수 처리 작업을 통해 이들 함수가 제공하는 효율성을 측정한다. 이를 통해 각 라이브러리와 알고리즘이 다양한 게임 환경에서 어떤 특성과 성능을 가지는지를 명확히 파악하고자 한다.

실시간으로 작동하는 게임에서는 특히 빠르고 효율적인 랜덤 함수가 필수적이다. 게임은 사용자의 입력과 시스템의 다양한 이벤트가 밀리초 단위로 이루어지기 때문에, 랜덤 함수의 선택이 게임의 실시간 성능에 중대한 영향을 미칠 수 있다. 실시간 랜덤 함수의 성능이 좋지 않다면, 캐릭터 동작이나 전투 시스템 등의 반응 속도가 늦어지며 사용자 경험이 저하될 수 있다. 그러한 이유로 본 연구를 통해 각 라이브러리의 성능을 비교 하는 것은 큰 의미가 있다.

본 논문의 II장에서는 실시간 게임에서 응용 가능한 의사난수 생성 알고리즘 및 라이브러리를 설명하고, III장에서는 각 알고리즘의 성능을 비교하기 위한 방법을 기술한다. IV장에서는 성능을 비교하여 실시간 게임에서 사용하기 적절한 의사난수 생성 알고리즘을 결정하고, 다른 알고리즘과의 성능 차이를 분석한다. 최종적으로 가장 적절한 알고리즘을 선정하여, V장에서 실험 결과에 대해 고찰하며 결론을 맺는다.

Ⅱ. 관련 연구

2-1 연구 배경

실시간으로 사용자끼리 상호작용하는 실시간 게임이 대세가 된 요즘, 실시간 게임에서 랜덤 요소를 통해 사용자 경험을 풍부하게 하려는 시도가 다수 존재한다. 이러한 경우, 난수 생성에 소요되는 시간이 매우 중요하므로 빠른 시간내에 난수를 효율적으로 생성하는 Xorshift 알고리즘의 사용을 제안한다. 기존 관련 연구들에 대해 살펴본다.

2-2 난수란 무엇인가

난수란 수학적으로 예측할 수 없는 수를 말한다. 하지만 컴퓨터에서 생성되는 난수는 의사 난수로, 실제 무작위가 아닌 수학적 알고리즘을 통해 생성된다. 이를 의사 난수 생성기(PRNG; Pseudo-Random Number Generator) 라고 하며, 컴퓨터가 생성하는 난수는 복잡한 패턴을 가지기 때문에 일반적으로는 예측이 어렵다[4].

2-3 Random 라이브러리

본 연구에서는 Unity Engine에서 사용 가능한 CryptoRandom.Next, System.Random.Next, UnityEngine.Random.Range, XorshiftRandom의 네 가지 대표적인 랜덤 함수와 알고리즘의 성능을 비교하였다.

1) CryptoRandom.Next

.NET에서 제공하는 보안성이 높은 의사 난수 생성 라이브러리로, 높은 수준의 보안성을 요구하는 상황에서 사용된다. 하지만 속도가 상대적으로 느리다[5].

2) System.Random.Next

.NET에서 기본적으로 제공하는 라이브러리 함수로,.NET을 활용하는 일반적인 목적의 난수 생성기로 사용하기 쉽고, 게임 개발에서 다양한 용도로 사용된다[6].

3) UnityEngine.Random.Range

Unity에서 기본적으로 제공되는 난수 생성 함수로, 사용이 간편하여 게임 개발자들에게 많이 사용된다[7].

4) XorshiftRandom

Xorshift는 George Marsaglia에 의해 개발된 의사난수 생성 알고리즘으로, 매우 효율적으로 작동하는 PRNG 중 하나이다. Xorshift는 비트 시프트 연산과 배타적 논리합(XOR)을 반복하여 난수를 생성하며, 이러한 방식은 소프트웨어에서 매우 빠르게 실행된다. Xorshift의 성능은 매우 우수하지만, 일부 통계적 테스트를 통과하지 못하는 약점이 있다. 그러나, 난수의 품질보다, 생성속도에서 오는 이득이 매우 크기 때문에, 게임 개발에서 간간히 사용된다[8].

Xorshift 알고리즘은 주어진 초기 상태 값(seed)에서 시작하여, 비트 시프트와 XOR 연산을 반복하여 난수를 생성하는 방식으로 동작한다. 먼저 상태 값을 왼쪽 또는 오른쪽으로 시프트한 뒤, 시프트된 값과 원래 값에 대해 XOR 연산을 수행한다. 이 과정을 여러 번 반복하여 비트들을 섞은 후 최종 결과 값을 난수로 반환한다. 이때 생성된 값은 다음 난수를 계산하기 위한 새로운 상태 값으로 갱신되며, 이 연속적인 계산을 통해 빠르고 효율적으로 난수 시퀀스를 생성한다.

이는 유사 난수를 훌륭히 생성하지만, 결국은 시드값에 따라 결과가 정해지므로 적절한 시드값을 선정하는 것이 중요하다. 따라서, 신뢰할 수 있는 난수 생성기를 위해서는 초기 시드값의 선택이 성능과 결과의 품질에 중대한 영향을 미친다.

2-4 게임 제작에서 Random 활용

게임 개발에서는 난수가 여러 용도로 사용된다. 캐릭터 능력치 설정, 아이템 드랍 확률 설정, 전투 시스템의 무작위성 등 다양한 상황에서 활용된다. 예를 들어, 특정 아이템의 드랍 확률을 랜덤하게 결정하거나, AI의 행동 패턴을 다양하게 만드는 데 난수를 사용할 수 있다. 또한, 노이즈 생성 등 그래픽 요소에서도 랜덤 값을 사용해 화면의 자연스러움을 향상시킬 수 있다[1].

2-5 Unity Engine

Unity Engine은 원래 게임 개발을 위해 설계되었지만, 현재는 VR/AR 콘텐츠, 애니메이션 제작, 시뮬레이션 등 다양한 분야에서 활용되고 있는 통합 콘텐츠 제작 도구이다. Unity의 주요 장점은 사용 용이성, 다양한 플랫폼 지원, 그리고 풍부한 커뮤니티와 자원이다. 이러한 특성 덕분에 Unity는 캐주얼 게임부터 고난도의 VR 게임까지 폭넓은 게임 개발에 사용되고 있다. 또한, 유니티는 C#을 스크립트 언어로 사용하여 생산성과 개발 속도가 뛰어나다[9].

Ⅲ. Random 라이브러리 성능 비교

3-1 실험 환경

실험 환경은 PC 성능에 따라 High, Mid, Low로 각 3종씩 구성하였다. 같은 플랫폼 환경 설정을 위하여 Windows 10 Pro의 C# 런타임 환경에서 CPU와 Memory에 초점을 맞춰 성능 차이를 두었다. 유니티는 기본적으로 C#을 스크립트 언어로 사용하기 때문에, 실험 프로그램의 모든 코드는 C#을 사용한 유니티 애플리케이션으로 진행한다.

PC1은 Quad-Core, PC2는 Hexa-Core, PC3은 Octa-Core로 코어 수에 차이를 눈에 띄는 차이를 두었다.

표 1. PC 실험 환경
Table 1. PC experimental environment

Device	Item	Specification
PC1 (Low)	CPU	Intel Core i5-6500 3.2GHZ
	Number of Core	Quad-Core
	Memory	DDR4 16GB
	OS	Windows 10 Pro
PC2 (Mid)	CPU	Intel Core i5-12400 2.5GHZ
	Number of Core	Hexa-Core
	Memory	DDR4 32GB
	OS	Windows 10 Pro
PC3 (High)	CPU	AMD Ryzen 7 7800X3D 4.2GHz
	Number of Core	Octa-Core
	Memory	DDR5 32GB
	OS	Windows 10 Pro

3-2 실험 내용

실험은 각 랜덤 함수의 생성 시간을 측정하는 방식으로 진행되었으며, 다양한 픽셀 수 처리 작업을 통해 생성 시간을 측정하여 이를 통해 성능을 평가하였다. 난수 생성에서는 보안도 중요한 고려사항이지만, 보안이 크게 중요하지 않은 실시간 게임들에서는 난수의 보안성보다는 생성 시간이 사용자 경험에 더 큰 영향을 미친다. 그러한 이유로 보안성을 고려하지 않은 생성 시간으로만 성능을 평가한다.

실험은 다음과 같은 단계로 구성된다. 지정된 크기의 이차원 배열을 생성하고, 해당 배열의 모든 요소의 흑백 여부를 랜덤하게 결정하여 노이즈 이미지를 생성한다. 흑백 여부는 0부터 지정된 범위까지의 난수를 생성하여, 생성된 수가 절반보다 크면 검은색으로, 작으면 하얀색으로 픽셀을 지정하는

방법을 사용한다. 생성 함수가 작동하기 시작하면 C#의 Stopwatch 기능을 활용하여 생성 구간을 측정하고, 이를 생성 방법, 생성 픽셀 수 등과 함께 저장하여 json 형식으로 저장한다. 실험에서는 각 픽셀 수 마다 10번씩 진행하였다.

위와 같은 방법으로 측정한 시간을 json 데이터로 뽑아내어, 이를 excel의 json 변환 기능을 통해 표, 그래프로 나타낸다.

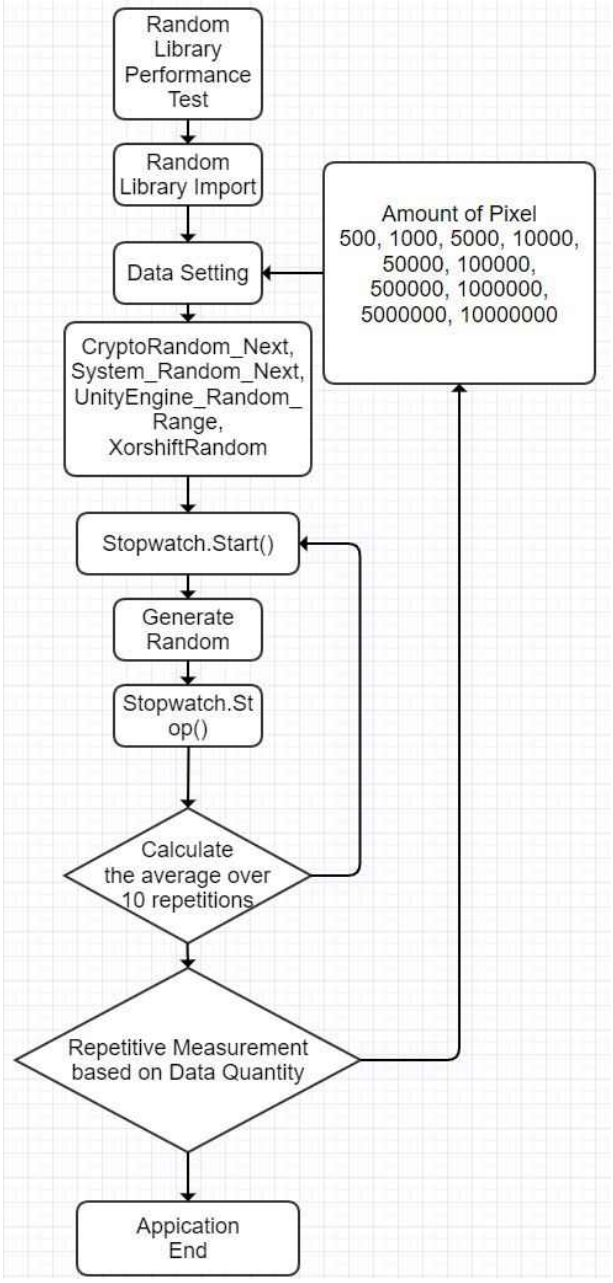

그림 1. 실험용 애플리케이션 알고리즘 흐름도
Fig. 1. Test application algorithm flowchart

그림 1는 실험 애플리케이션의 알고리즘 흐름도다. 프로그램 실행 시, 각 라이브러리를 불러오고, 사용자의 입력에 따라 생성 픽셀 수, 생성 알고리즘 등을 설정한다. 연구에서는 사전에 설정된 10가지 값을 사용한다. 데이터 설정 후, 난수 생성이 진행된다. 난수 생성 진행 직전에 Stopwatch가 시작되어 시간을 측정한다. 지정된 픽셀 수에 해당하는 수의 난수가 생성된 후, Stopwatch가 종료된다. 이 후, 지정된 반복 수 만큼 이를 반복한다. 연구에서는 평균을 내기 위하여 10번씩 진행하였다. 반복이 모두 종료되면 다른 모든 방법으로도 차례로 진행한 후 프로그램을 종료한다.

```
public class ReportData
{
    public string type;
    public int pixel;
    public int range;
    public long time;

    public ReportData(string type, int pixel, int range, long time)
    {
        this.type = type;
        this.pixel = pixel;
        this.range = range;
        this.time = time;
    }
}

public class ReportManager : Singleton<ReportManager>
{
    public List<ReportData> report = new List<ReportData>();

    public void ConvertReportToJson()
    {
        string json = JsonConvert.SerializeObject(report, Formatting.Indented);

        string path = Path.Combine(UIManager.Instance.path, UIManager.Instance.reportName);

        File.WriteAllText(path, json);

        UIManager.Instance.DebugMessage("Report saved to " + path);
    }
    public void ResetReport()
    {
        report.Clear();
    }
}
```

그림 2. 생성 정보 구조체 및 json 변환 함수
Fig. 2. Structure for generated information and function for JSON conversion

그림 2는 생성 정보를 저장하는 구조체와 이를 List에 저장한 후, json 형식으로 저장하여 출력하는 코드이다. ReportData란 이름의 구조체이며, 생성 방법을 나타내는 type, 생성 픽셀 수를 나타내는 pixel, 난수 생성 범위를 나타내는 range, 생성 시간을 microsecond 단위로 나타내는 time로 구성되어 있다. 하단의 ReportManager 클래스는 랜덤 이미지 생성시 저장되는 ReportData 인스턴스를 모아두는 리스트와, 이 리스트를 저장하여 json 파일로 추출하는 메소드로 구성되어 있다. 이 클래스를 통하여, 생성 시간에 대한 정보를 얻을 수 있다.

```
// 1. UnityEngine.Random.Range
public bool[,] UnityEngine_Random_Range()
{
    lastMethod = "UnityEngine_Random_Range";
    bool[,] map = new bool[y, x];
    int halfRange = range / 2;

    Stopwatch stopwatch = new Stopwatch();
    stopwatch.Start();
    for (int i = 0; i < y; i++)
    {
        for (int j = 0; j < x; j++)
        {
            int value = UnityEngine.Random.Range(0, range);
            if (value >= halfRange)
                map[i, j] = true;
            else
                map[i, j] = false;
        }
    }
    stopwatch.Stop();
    //UIManager.Instance.DebugMessage($"UnityEngine_Random_Range - {x}_{y} : {stopwatch
    long elapsedTicks = stopwatch.ElapsedTicks;
    long microseconds = (long)((double)elapsedTicks / Stopwatch.Frequency * 1_000_000);
    UnityEngine.Debug.Log($"UnityEngine_Random_Range - {x}_{y} : {microseconds} µs");
    time = microseconds;
    return map;
}
```

그림 3. UnityEngine.Random.Range 테스트 함수
Fig. 3. UnityEngine.Random.Range test function

```
// 2. System.Random.Next
public bool[,] System_Random_Next()
{
    lastMethod = "System_Random_Next";
    bool[,] map = new bool[y, x];
    int halfRange = range / 2;
    System.Random random = new System.Random();

    Stopwatch stopwatch = new Stopwatch();
    stopwatch.Start();
    for (int i = 0; i < y; i++)
    {
        for (int j = 0; j < x; j++)
        {
            int value = random.Next(0, range);
            if (value >= halfRange)
                map[i, j] = true;
            else
                map[i, j] = false;
        }
    }
    stopwatch.Stop();
    //UIManager.Instance.DebugMessage($"System_Random_Next - {x}_{y} : {stopwatch.Elaps
    long elapsedTicks = stopwatch.ElapsedTicks;
    long microseconds = (long)((double)elapsedTicks / Stopwatch.Frequency * 1_000_000);
    UnityEngine.Debug.Log($"UnityEngine_Random_Range - {x}_{y} : {microseconds} µs");
    time = microseconds;
    return map;
}
```

그림 4. System.Random.Next 함수
Fig. 4. System.Random.Next test function

그림 3~6은 각 라이브러리 및 알고리즘에 해당하는 테스트 코드이다. 함수의 시작과 Stopwatch 기능을 통해, 생성 시간을 microsecond 단위로 측정하였다. 각 난수 생성에 해당하는 부분만 다르게 구성하였으며, 그 외의 부분은 동일하게 구성하여 생성 시간 측정에 영향이 가지 않게 하였다.

```
// 3. System.Security.Cryptography.RandomNumberGenerator
public bool[,] CryptoRandom_Next()
{
    lastMethod = "CryptoRandom_Next";
    bool[,] map = new bool[y, x];
    int halfRange = range / 2;
    byte[] randomNumber = new byte[4];
    using (RandomNumberGenerator rng = RandomNumberGenerator.Create())
    {

        Stopwatch stopwatch = new Stopwatch();
        stopwatch.Start();
        for (int i = 0; i < y; i++)
        {
            for (int j = 0; j < x; j++)
            {
                rng.GetBytes(randomNumber);
                int value = Math.Abs(BitConverter.ToInt32(randomNumber, 0) % range);
                if (value >= halfRange)
                    map[i, j] = true;
                else
                    map[i, j] = false;
            }
        }
        stopwatch.Stop();
        //UIManager.Instance.DebugMessage($"CryptoRandom_Next - {x}_{y} : {stopwatch.Elapsed
        long elapsedTicks = stopwatch.ElapsedTicks;
        long microseconds = (long)((double)elapsedTicks / Stopwatch.Frequency * 1_000_000);
        UnityEngine.Debug.Log($"UnityEngine_Random_Range - {x}_{y} : {microseconds} µs");
        time = microseconds;
    }
    return map;
}
```

그림 5. CryptoRandom.Next 테스트 함수
Fig. 5. CryptoRandom.Next test function

```
// 4. Xorshift Algorithm
public bool[,] XorshiftRandom()
{
    lastMethod = "XorshiftRandom";
    bool[,] map = new bool[y, x];
    int halfRange = range / 2;
    uint state = (uint)DateTime.Now.Ticks;

    Stopwatch stopwatch = new Stopwatch();
    stopwatch.Start();
    for (int i = 0; i < y; i++)
    {
        for (int j = 0; j < x; j++)
        {
            state ^= state << 13;
            state ^= state >> 17;
            state ^= state << 5;
            int value = Math.Abs((int)(state % range));
            if (value >= halfRange)
                map[i, j] = true;
            else
                map[i, j] = false;
        }
    }
    stopwatch.Stop();
    //UIManager.Instance.DebugMessage($"XorshiftRandom - {x}_{y} : {stopwatch.ElapsedMi
    long elapsedTicks = stopwatch.ElapsedTicks;
    long microseconds = (long)((double)elapsedTicks / Stopwatch.Frequency * 1_000_000);
    UnityEngine.Debug.Log($"UnityEngine_Random_Range - {x}_{y} : {microseconds} µs");
    time = microseconds;
    return map;
}
```

그림 6. XorshiftRandom 테스트 함수
Fig. 6. XorshiftRandom test function

그림 7. 성능 테스트
Fig. 7. Performance test

그림 7의 UI를 통해, 이미지의 가로, 세로 길이와 생성 함수의 종류 등을 선택할 수 있다. 하단에는 진행 상황을 파악할 수 있는 디버그 메시지가 출력되게 하였다. 좌측 상단의 입력란에서 생성 픽셀 수와 생성 알고리즘 등을 선택하고, 우측의 버튼을 통해 난수 생성 및 기록 리포트를 생성할 수 있다.

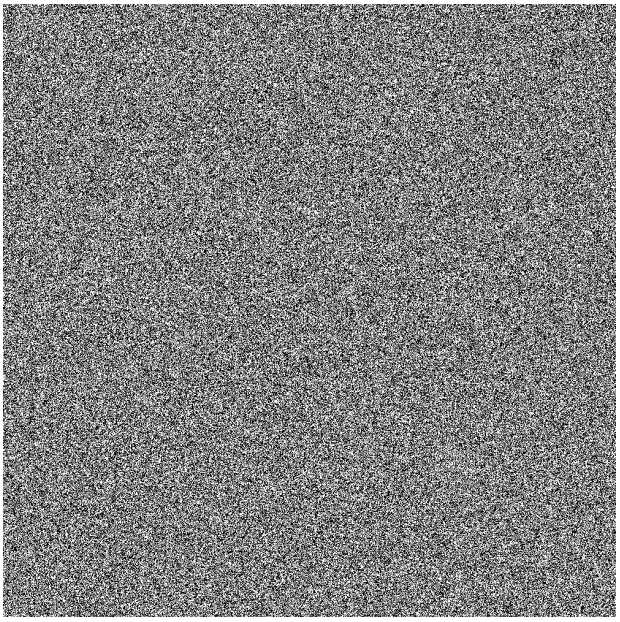

그림 8. 랜덤 이미지 예시
Fig. 8. Random image example

그림 8은 생성 난수를 시각화시킨 이미지다. 각 난수의 값에 따라 픽셀의 흑백 여부를 결정하여, 이미지로 출력하였다.

Ⅳ. 성능 테스트 결과

표 2는 PC 3종에 대해서 각 픽셀 수 마다 10번 반복 후 생성 시간의 평균을 측정한 내용이다. '3-1'에 기재된대로, PC1은 Quad-Core, PC2는 Hexa-Core, PC3은 Octa-Core의 CPU를 가진 PC로 성능을 나누었다.

UnityEngine.Random.Range는 'A'로 표기하였으며, 각각 System.Random.Next는 'B'로, CryptoRandom.Next는 'C'로, XorshiftRandom은 'D'로 표기하였다. 기기 성능에 따라서 각 라이브러리의 성능도 대체적으로 비례하게 측정되었다. 실험한 Random 라이브러리에서 A는 B와 C에 비해 평균적으로 6~8배의 시간이 소요되었으며, D는 상급PC에서는 B와 C 비해 절반정도의 시간이 소요되었지만, 하급PC에서는 유사한

표 2. PC 성능에 따른 Random 라이브러리 생성 결과
Table 2. Generation results of Random libraries based on PC performance

Device	Amount of pixel	A	B	C	D
PC 1 (Low)	500	139	28	38	26
	1000	203	29	36	28
	5000	847	102	116	100
	10000	1790	204	203	199
	50000	9170	922	965	981
	100000	17653	2011	1931	1908
	500000	90125	9508	9491	9152
	1000000	192751	18420	18201	19727
	5000000	868639	94944	96032	92078
	10000000	1722292	184501	183534	183542
Device	Amount of pixel	A	B	C	D
PC 2 (Mid)	500	74	15	14	2
	1000	107	14	12	4
	5000	533	70	58	36
	10000	1066	138	116	79
	50000	5106	691	577	416
	100000	10338	1317	1100	833
	500000	52238	6613	5494	4156
	1000000	106245	13126	11248	8702
	5000000	521576	66818	57433	41286
	10000000	1034520	137909	109832	83312
Device	Amount of pixel	A	B	C	D
PC 3 (High)	500	58	12	13	2
	1000	76	11	10	5
	5000	356	66	65	19
	10000	712	115	101	32
	50000	3641	561	505	259
	100000	7062	1175	991	559
	500000	37323	5701	4949	2889
	1000000	72755	11343	10015	6066
	5000000	362014	58832	49124	28556
	10000000	731336	117192	99696	58047

시간이 소요되며, PC 성능이 올라갈수록 B, C와 유사한 성능을 가지게되었다. 결과적으로, D는 네가지 라이브러리 및 알고리즘 중에서 가장 짧은 시간을 소요한다고 볼 수 있다. 생성 픽셀 수에 따라 생성 시간의 차이가 근소하게 존재하지만, 대체적으로 D가 가장 짧은 시간을 소요하였다. 이는 실시간 반응성이 중요한 게임에서 Xorshift 알고리즘이 기존 라이브러리의 대체제가 될 수 있음을 의미한다고 할 수 있다.

그림 9, 10, 11은 표의 데이터를 기반으로 각각의 PC환경에서 각 Random 라이브러리의 생성 속도를 비교한 그래프이다. 세로축은 측정 속도의 단위를 microsecond를 나타내며, 로그 눈금으로 표현하였다. 로그 눈금으로 표현하는 이유는 생성 픽셀수를 곱 단위로 증가시키기 때문이다. 이로 인해, 데이터간의 차이를 정확히 비교하기 어렵지만, 라이브러리 간의 성능 비교에는 문제가 없다. 가로축은 생성 픽셀 수를 나타낸다.

UnityEngine.Random.Range는 'A'로 표기하였으며, 각각 System.Random.Next는 'B'로, CryptoRandom.Next는 'C'로, XorshiftRandom은 'D'로 설명한다.

생성 픽셀수가 적은 500, 1000 픽셀 구간을 제외한 대부분의 구간에서 유사한 비율의 생성 시간 차이를 가진다.

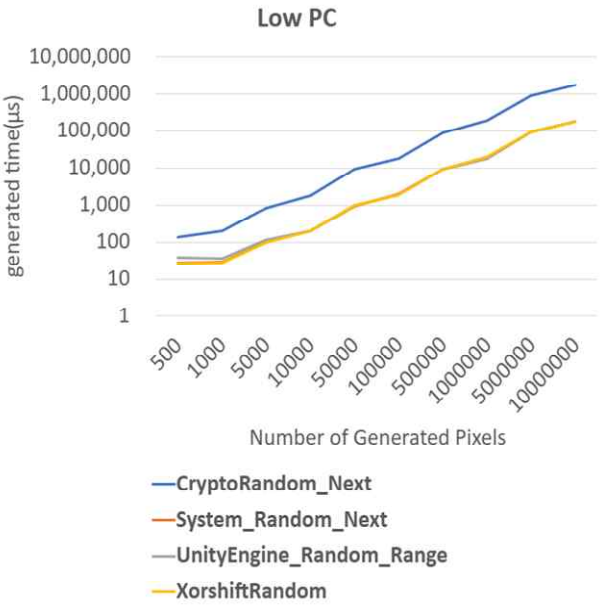

그림 9. Low PC Random 라이브러리 생성 시간 비교
Fig. 9. Low PC Random library generation time comparison

그림 9는 Low PC에 해당하는 PC1의 생성 결과 그래프다. C를 제외한 모든 알고리즘이 유사한 생성 시간을 가진다. C는 나머지 알고리즘에 비해 약 10배의 생성 시간을 가진다.

그림 10은 Mid PC에 해당하는 PC2의 생성 결과 그래프다. 500, 1000의 생성 픽셀 수에서 D가 매우 낮은 시간을 가지지만, 5000 픽셀 부터는 모든 구간에서 유사한 지표를 보인다. C는 D의 약 10~13배에 해당하는 생성 시간을 가지며, A와 B는 D의 약 1~1.5배에 해당하는 생성 시간을 가진다.

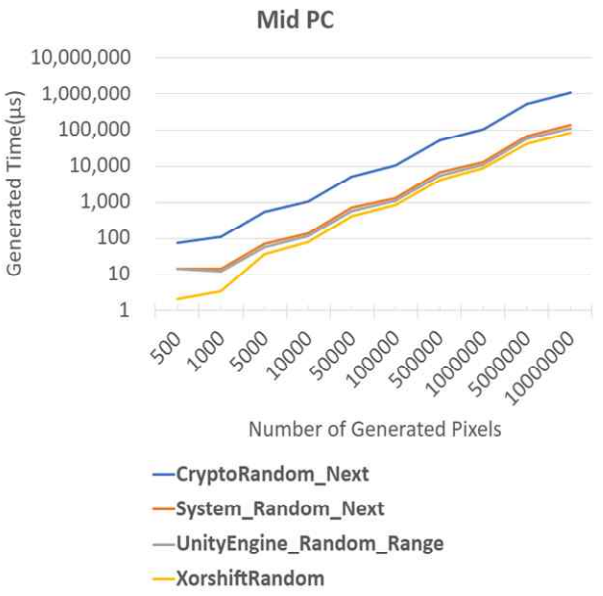

그림 10. Mid PC Random 라이브러리 생성 시간 비교
Fig. 10. Mid PC Random library generation time comparison

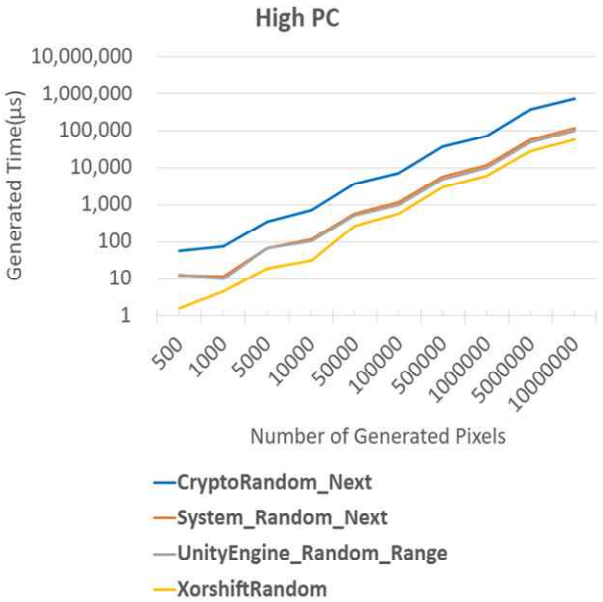

그림 11. High PC Random 라이브러리 생성 시간 비교
Fig. 11. High PC Random library generation time comparison

그림 11은 High PC에 해당하는 PC3의 생성 결과 그래프다. 모든 구간에서 C는 D의 10~13배에 해당하는 생성 시간을 가지며, A와 B는 D의 2~3배에 해당하는 생성시간을 가진다.

PC 성능에 따라 차이가 존재하지만, C가 가장 많은 시간을 소요하며, A와 B가 그 다음으로 시간을 많이 소요하며, D가 가장 적은 시간을 소모한다. 즉, XorshiftRandom이 모든

구간에서 가장 생성 시간을 적게 소모한다고 할 수 있다.

실험 결과를 보면, CryptoRandom은 XorShift에 비해 속도가 최대 16배 느리고, System.Random과 UnityEngine.Random은 XorShift에 비해 최대 3배까지도 느리게 동작할 수 있다는 것을 확인하였다. 오랜 시간이 소요된다는 것은 실시간 반응성이 중요한 게임에서 많은 양의 랜덤 데이터를 생성할 때 불리하게 작용할 수 있다. 따라서, XorShift 알고리즘을 사용하여, 랜덤 함수를 구성하는 것이 성능 면에서 유리하다.

각 PC 환경에 따라, XorShift와 다른 함수의 성능 차이가 일정하지 않아보인다. 이는 PC환경에 따른 차이로 보인다. 그러나, XorShift가 가장 빠른 생성 시간을 가진다는 경향성은 일정하다. XorShift를 제외한 다른 라이브러리들은 프로그래머가 바로 사용할 수 있는 편리한 기능들을 다수 포함하고 있다. 하지만, 성능 면에서는 XorShift에 비해 떨어지기 때문에, 게임 최적화를 중요시하는 실시간 게임 개발에서는 XorShift를 사용하는 것이 바람직할 것이다.

V. 결 론

게임 산업은 지속적인 기술 발전과 사용자 요구 증가에 힘입어 빠르게 성장하고 있으며, 이 과정에서 랜덤성 구현은 게임 개발의 핵심 요소로 자리 잡고 있다. 랜덤성은 게임의 몰입도를 높이고 플레이어에게 신선한 경험을 제공하는 데 중요한 역할을 한다. 특히 적의 행동 패턴, 환경 변화 등의 요소에 랜덤성을 적용하면 매번 새로운 상황이 연출되어 게임의 이 높아지고, 플레이어에게 예측할 수 없는 긴장감을 선사할 수 있다. 또한 실시간 게임이 현대의 인기 장르로 자리 잡으면서, 빠르고 효율적인 난수 생성의 중요성은 더욱 커지고 있다. 난수 생성 알고리즘의 성능은 게임의 반응 속도와 프레임 유지에 직접적인 영향을 미쳐, 실시간으로 처리되는 다양한 게임 요소의 품질을 좌우한다. 그러한 이유로 본 연구는 실시간 게임의 성능 향상에 중요한 의미가 있다.

향후에는 다양한 플랫폼(예: 모바일, 콘솔)에서의 성능을 비교하거나, 다른 난수 생성 알고리즘을 추가로 비교 분석하여 더욱 폭넓은 가이드라인을 제공할 필요가 있다. 이러한 연구는 개발자들이 사용자 경험을 최적화하는 데 실질적인 도움을 줄 것으로 기대된다.

참고문헌

[1] Unity. Unpredictably Fun: The Value of Randomization in Game Design [Internet]. Available: https://unity.com/kr/blog/games/unpredictably-fun-the-value-of-randomization-in-game-design.

[2] Korea Creative Content Agency. Global Game Industry Trend (No. 2022 May+June) [Internet]. Available: https://welcon.kocca.kr/ko/info/trend/1952054

[3] Epic Games. Unreal Engine [Internet]. Available: https://www.unrealengine.com/ko.

[4] Telecommunications Technology Association. Pseudo-Random Number [Internet]. Available: https://terms.tta.or.kr/dictionary/dictionaryView.do?subject=%EC%9D%98%EC%82%AC+%EB%82%9C%EC%88%98.

[5] Microsoft Learn. System.Security.Cryptography Namespace [Internet]. Available: https://learn.microsoft.com/ko-kr/dotnet/api/system.security.cryptography?view=net-8.0.

[6] Microsoft Learn. Random Class (System) [Internet]. Available: https://learn.microsoft.com/ko-kr/dotnet/api/system.random?view=net-8.0.

[7] Unity Documentation. Random.Range [Internet]. Available: https://docs.unity3d.com/ScriptReference/Random.Range.html.

[8] Sebastiano Vigna. Xoshiro / Xoroshiro Generators and the PRNG Shootout [Internet]. Available: https://vigna.di.unimi.it/xorshift/.

[9] Unity Technologies. Unity Real-Time Development Platform [Internet]. Available: https://unity.com/.

도성진(SeongJin Do)

2019년～현 재: 강원대학교 IT대학 전기전자공학과 재학

※관심분야 : 게임, 알고리즘, 유니티 엔진 등

김동회(Dong-Hoi Kim)

2005년 : 고려대학교 전파공학과 (공학박사)

1989년 1월～1997년 1월: 삼성전자 전임연구원

2000년 8월～2005년 8월: 한국전자통신연구원 선임연구원

2006년 3월～현 재: 강원대학교 IT대학 전기전자공학과 교수

2020년 6월～2022년 8월: 강원대학교 정보화본부장 등

※관심분야 : 인공지능(AI), 무선 네트워크 및 사물인터넷(IoT) 등