Investigating Generating Random Numbers With Various Methodologies

Henry Oliver

Media Design School
henry983615@gmail.com

Vaughan Webb

Media Design School
vaughanw2012@gmail.com

Abstract* - *Random Number Generation (RNG) is used in a variety of applications from encryption to video games. The purpose of this paper is to investigate various methods to generate random numbers, comparing the quality, performance, and implementation difficulty of each methodology. We will be comparing a Linear Congruential Generator, a Mersenne Twister, Mouse Movement, and a Uranium Decay Simulator as our examples.

1 Introduction

Random Number Generators (RNG) in the context of computer science refers to the ability to generate random numbers during runtime. This is important as many systems rely on random numbers to function correctly. For a random number generator to be considered for use in a project you need to account for three main points.

Quality: *How many iterations it takes to repeat the sequence of numbers generated. The longer the better.*

Performance: *How quickly numbers can be generated by the algorithm. The quicker the better.*

Implementation: *The difficulty of implementing the generator.*

We will be focusing on using different methods to generate random numbers whose purpose is to be implemented into a video game.

Linear Congruential Generator (LCG): *The LCG method of generating pseudo-random numbers may be one of the oldest known but its weaknesses are only truly present when using a period length that is too short, on top of that it is one of the easiest to implement as it uses modular arithmetic.* [9]

Mersenne Twister: *The Mersenne Twister method of generating pseudo-random numbers is one of the most common methods today and is used by many programming languages. It was created by Takuji Nishimura and Makoto Matsumoto in 1998 [15] to solve flaws found in older pseudo-random number generators. It has several advantages one of the main being the very long period length. Its main disadvantage is that it is not cryptographically secure however that is out of our scope for this investigation*

The reason we have chosen these methods to test with as a baseline is because they both are very popular and well-known choices for pseudo-random number generation. We will then use a mix of computer science methods of gathering random data

from the world. The methods that we will be using are as follows.

System Time: *System time is often used in programming to seed RNGs, so we will use it in our investigation.*

Mouse Movement: *The Linux kernel relies on using device noise as sources of entropy, this includes but is not limited to mouse movement [3].*

Radioactive Decay: *Physical devices are capable of generating truly random numbers which are a lot better than what pseudo-random number generators can achieve, because of this we chose radioactive decay since the decay of radioactive material is random and unpredictable. [22]*

2 Question

Is there an improved methodology for generating random numbers over current methods being used inside of video games?

3 Investigating Methods

We started work on implementing random number generators and decided that we would use the Unity Game Engine (2019.3.7f1) for this project. We did this as we both have experience in the engine and making the nodes display would be easy with no restrictions on what we want to show allowing us full control over what methods of random number generation we choose to use.

3.1 LCG PRNG

We created a Linear Congruential Generator (LCG) Pseudo-Random Number Generator (PRNG), inspired by RANDU which is a spectacularly bad choice of values for an LCG with the multiplier being 65539 and the modulus being 2^{31} which started seeing use in the 1960s and was seeing widespread use in the early 1970s [13] that still saw notable use in 2014.

$$((a \times X) + c) mod m \quad (1)$$

a = the multiplayer
X = the seed/start value
c = the increment
m = the modulus

we are using the exact values that RANDU uses which are:
$a = 65539$
$x = 1234$
$c = 0$
$m = 2^{31}$

[19]

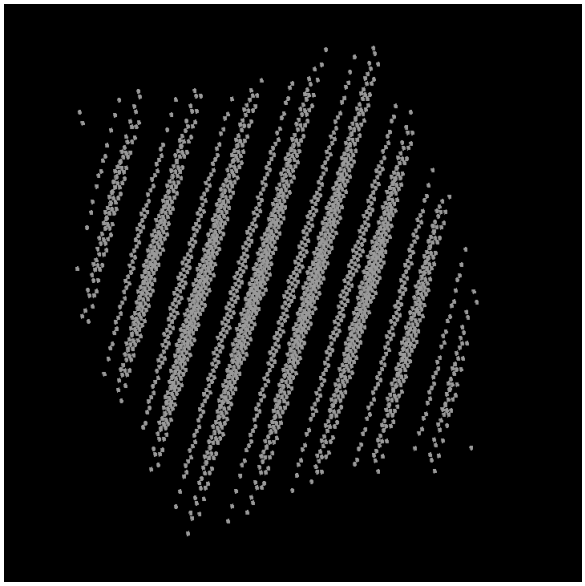

Figure 1: Example of a LCG with a weak period length

From the screenshot, we can observe that our linear congruential generator has a flaw in its random number generation technique. This is what we were hoping to see as this is the same flawed pattern that emerges from the RANDU linear congruential generator algorithm, however, if viewed from a different perspective the pattern blends together and is masked from the observer. From this, we can gather that we have successfully recreated the

RANDU linear congruential generator. The main reason for this obvious pattern emerging is the fact that when RANDU was being created the increment value c was chosen to be zero, which means that it becomes a multiplicative congruential generator, when working with multiplicative congruential generators you must avoid zero and your cycle length will be limited to $m - 1$. The creators of RANDU did not account for this and chose parameters that were computationally fast which caused this flaw in their random number generator. [10] It should also be noted that using an increment value of zero doesn't mean that your algorithm is flawed as C++11 uses a multiplicative congruential pseudo-random number generator with an increment value of zero for their *minstd_rand()* function.[18]

3.2 Using Mouse Movement

We then created an RNG using mouse movement as input. We created a system that used user mouse input to generate random numbers, this was inspired by how the Linux kernel uses mouse movement as a source of entropy[3], as humans cannot move a mouse in a perfectly straight line and because of this can utilize the inconsistencies that a machine cannot produce normally. The cons of this method however is that for numbers to be generated not only does that player have to be moving their mouse but they have to be moving it beyond a "threshold" value as otherwise the numbers would be generated in a predictable range. We achieved this by using Input.GetAxis() (which is from a Unity Game Engine Library) then check that it is greater then 0 and then if it was we would mod the result by 10 (our threshold value), we found that if the velocity of the mouse cursor is below 10 then the generated points would generate in a diagonal line.

Now what we have done here is created what's called an internally-conditioned entropy source which means that we didn't just take and use all the raw data it was spitting out, we put it through a sanity check which in our case is making it past the threshold. By doing this we made a Full Entropy Source. When doing this the goal is to have minimal post-processing as it's said that if complex post-processing is needed to make the number more random that it indicates that it is a weaker true random number generator and that more focus should be put on the entropy collection stage. Even running our sanity check its important to be wary as the more complex check is run the more likely that we could end up having a bias towards certain values and end up making the numbers less. random.[1]

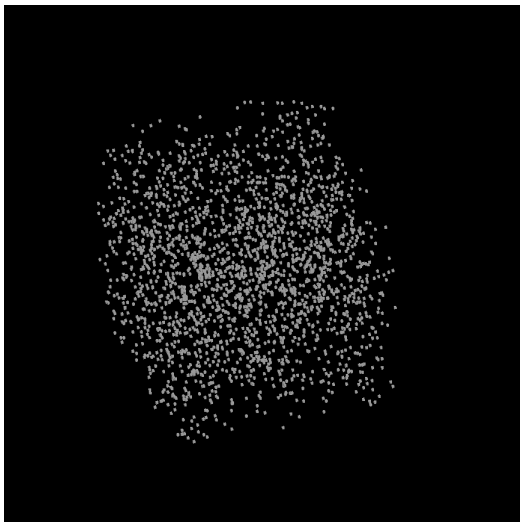

Figure 2: Example of RNG using mouse movement as input

3.2.1 Integrating System Time

We also investigated combining mouse movement with time to add more variance to the input. The formula for to generate points using this method similar to our previous mouse RNG so we start with Input.GetAxis() then if it is greater than 10 which was the threshold value we used in the normal mouse movement, then we multiply it by System.DateTime.Now.Millisecond then after that modulus it again by our threshold value. Doing this keeps it clamped within the same size as everything else while also not adding any bias towards a number in particular as we want to keep it as random as possible.

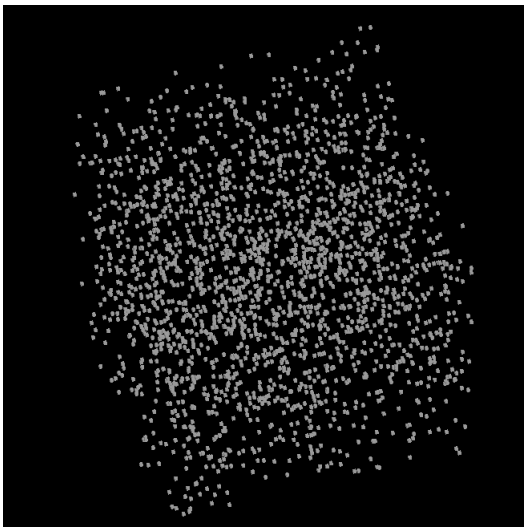

Figure 3: Example of RNG using mouse movement and time as input

As you can see in the picture there practically no difference between this method and the previous mouse method, This was somewhat surprising but with some foresight it makes sense. as stated in the previous section for just mouse movement. We had already struck a good amount of post-processing for our TRNG and in this example, the numbers aren't any more random as they were already random, to begin with. So although there was care taken to not bias any number and risk making it less random by multiplying it by System.DateTime.Now.Millisecond before the modulus of the original threshold value, the extra post-processing would have just harmed the computational speed and provided no real purpose as we already had an internally-conditioned Full Entropy Source. [1]

3.3 Mersenne Twister RNG

We created a Mersenne Twister, this is another PRNG (pseudo-random number generator) like our previous one based on RANDU but it is of much higher complexity, this is because the Mersenne Twister was invented in 1997, as a response to rectify issues of existing pseudo-random number generators such as RANDU. The Mersenne Twister we specifically included is a modified version of the MT19937[14], as it is currently implemented with a lot of notable software that is currently in use today such as: Microsoft Excel [16], Glib[4], MATLAB [24], PHP[6], Python[5] and C++11 onward[2].

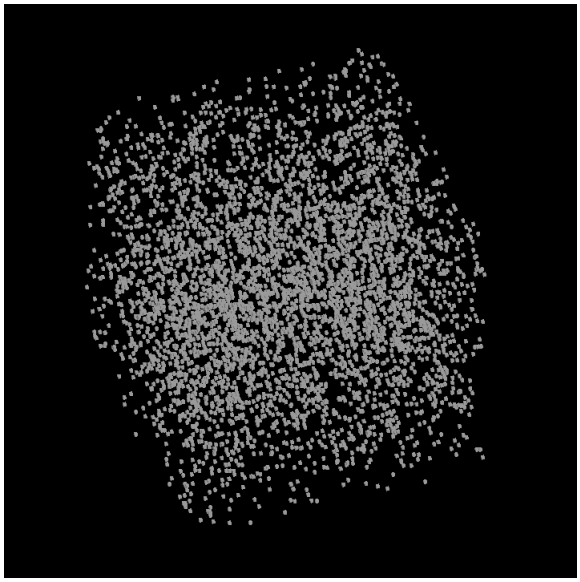

Figure 4: Example Of The Mersenne Twister

The implementation of the Mersenne Twister is much more complicated to understand compared to RANDU but at its core, it is more or less the same as they both start with a seed, do some mathematical operations then return a number, just the mathematical operations on MT19937 is much more complected and includes a large amount of bit-shifting operations.

3.4 Use of radioactive isotopes for RNG

The inherit decay behavior of radioactive isotopes is unpredictable as we cannot determine when particles will emit only that a particle will emit within a time interval [21][22] which makes them a great source of entropy. For this example, we will be using a gram of Uranium as the source of our entropy. Sadly because

of ethical reasons we cannot use real uranium and will instead need to simulate the decay.

3.4.1 Building a simulator

To do this we will need to calculate the DPM (disintegrations per minute) of one gram of uranium, we will then need to estimate how efficient our detector is. Once we have done all this we will be able to simulate the decay we can measure.

We need to first determine the length of a half-life for a gram of uranium (a half-life is the amount of time needed for half of the nuclei to disintegrate, once a half-life is finished the next half-life begins and will reduce the remainder by another half, this repeats until the nuclei are depleted). Luckily the amount of time for a half-life to occur only changes between isotopes but not for the quantity of an isotope, because of this having a ton of uranium and a gram uranium will not make a difference in regard to the period time of their half-lives. [22] Uranium-238 (U-238) is the isotope we will be using in the simulation. U-238 has a half-life of 4.468×10^9 years. [11] Now that we have the amount of time for a half-life to occur we need to calculate the rate of decay which can be done with the following equation.

$$A = -\frac{\Delta N}{\Delta t} \tag{2}$$

[11]
A = the amount of activity
ΔN = change in amount of atoms in the isotope
Δt = change in amount of time

We need to find the number of atoms in the isotope. To do this we can use the unified atomic of U-238 and convert it to grams to get the total weight of one atom then find how many atoms there are in a gram. Once we have this information we can find the change in atoms for our first half-life.

$$1u = 1.6605402(10) \times 10^{-27} kg \tag{3}$$

[23]

By using the above equation we can find the weight of one atom (The unified atomic mass of U-238 is 238.051). We can simply convert the unified atomic mass of U-238 to kilograms. Which tells us that one atom of U-238 weighs $3.95292951 \times 10^{-25}$kg or $3.952929509999678 \times 10^{-22}$g. Since we have one gram of U-238 we can easily find the number of atoms using the following equation.

$$Na = \frac{W}{A} \tag{4}$$

Na = Number of Atoms
W = Object weight in kg
A = Amount of atoms in a kg

We can divide the amount of weight in two to find the number of atoms that will be affected in the first half-life there are $5.06067296438849 \times 10^{21}$ atoms that will disintegrate in the first half-life in a gram of isotope U-238. Now that we have the number of atoms changed and the amount of time passed for the duration of the first half-life we can use the Equation 2, which gives us a result of 2154962.7 DPM.

When estimating the efficiency of our detection there are multiple factors to account for such as particles missing the detector, barriers that may stop particles reaching the detector, the type of detector used and how it is implemented because of this we are going to assume that our detector used in our simulation has an efficiency of 1%. Looking online we can find a cheap detector for $50USD (RadiationD-v1.1) that claims it can read a maximum of 25 CPM, by using the following formula:

$$DPM = \frac{CPM}{Efficiency} \tag{5}$$

[7]
We can find that the detector can detect a max of 2500 DPM or 41.67 DPS. Now that we know how many DPS our detector could detect we can proceed to build a simulator.

```
public float lastSeenDecayDelay = 0.0f;
[Range(1,31)]
public int binaryLength = 31;
[Range(0.00001f, 1.0f)]
public float timeBeforeDecay = 0.024f;

private int currentSeedingValue = 1;
private string seedingString = "";
private bool decayFlag = false;
private float decayTimer = 0.0f;

//Used to read the current seeding value
public int readUranium()
```

```
{
return currentSeedingValue;
}

//Update loop called each frame
void Update(){
//add deltaTime to decayTimer
//flip decayFlag
if (decayTimer >= timeBeforeDecay) {
    //make coin be a random number between 0-10
    if (coin == 6){
        //Convert decayFlag to int, append to
     seedingString
        if (seedingString.Length == binaryLength){
            //set currentSeedingValue to an int using
     seedingString as binary input
            //multiply currentSeedingValue by -1 if
     value is below 0
            //reset seedingString
        }
        //reset decayTimer
    }
}
}
```

Listing 1: Decay Simulator Pseudo Code

Using the pseudo code above we can simulate decay events and provide a seed to an RNG. It's method is once every frame, flip *decayFlag*, increment a timer by *deltaTime* then check if that timer is bigger than or equal to *timeBeforeDecay*. If it is then we generate a number between 0-10. if the value is 6 then we append our boolean as a "1" or "0" onto the end of *seedingString*. We then check if the length of *seedingString* is equal to *binaryLength* if it is then we replace *currentSeedingValue* with a positive integer based on the binary value of *seedingString's* string. By doing this we can generate 31-bit integers periodically that can be then served to an RNG function. Going with our 1% efficient example that would mean that we would generate a bit every 0.024 seconds and a new seed every 0.744 seconds. We can then also multiply the seed with time to ensure that the seed will be different for each frame as well as accounting for race conditions in relation to how quickly the simulator can construct a new value.

3.4.2 Combining LCG with Uranium Decay Simulator

By using the simulator we built we can now seed random number generators. We wanted to see how much of difference our uranium decay simulator made to our weak LCG by taking all the values from RANDU except with the uranium decay simulator being mixed into the increment.

$$((a \times X) + c) mod m \tag{6}$$

$$c = u \times Tm \tag{7}$$

a = the multiplier
X = the seed value
c = the increment
m = the modulus
u = uranium seed value
Tm = system time milliseconds

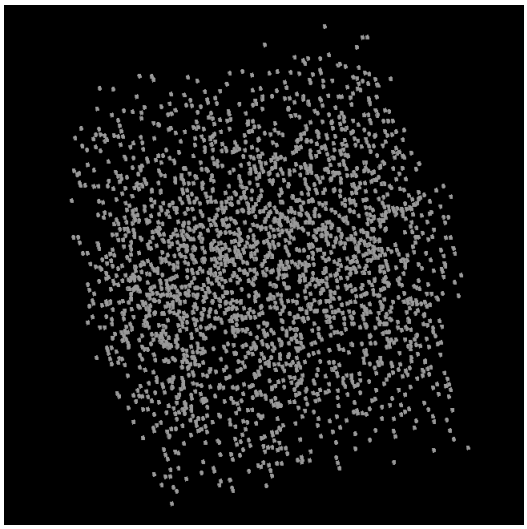

Figure 5: Combining LCG with our Uranium Decay Simulator

After implementing it we saw that this greatly improved the random generator results. This is great that we were able to combine these two methods as the main weakness that the LCG had was its quality and the Uranium Decay Simulator is able to cover that up nicely.

3.4.3 Combining The Mersenne Twister with Uranium Decay Simulator

After observing the huge difference our uranium decay simulator made to our LCG we wanted to see what would happen if we seeded the Mersenne Twister using our decay simulator.

We could not visually see any difference in quality between combining the uranium decay simulator with

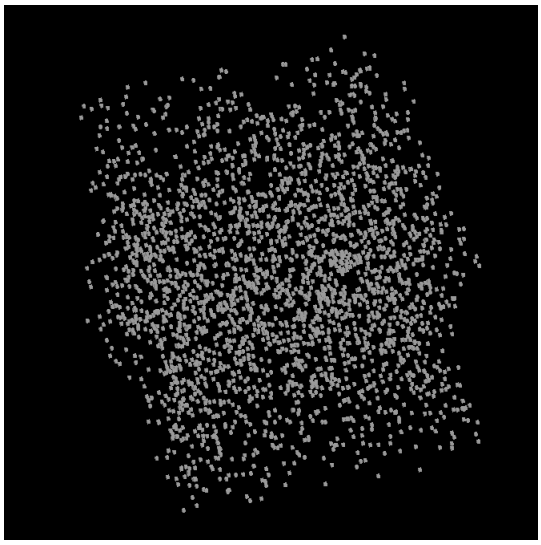

Figure 6: Combining The Mersenne Twister with our Uranium Decay Simulator

either the Mersenne Twister or the LCG. With it being computationally more intensive then the LCG method however this method

3.5 Potential Improved Methodology For Generating Random Numbers

Now that we have gone through and collected all the data necessary from all of the methods, the next step is to compare all of the data we have received to determines which method is the most fit for use in the context of video games.

3.5.1 Hypnosis

We predict that using a combination of our Mersenne Twister and Uranium Decay Simulator will yield the best results in our testing in terms of both performance and quality, closely followed by either our Mersenne Twister or our combination of our Linear Congruential Generator and Uranium Decay Simulator. The reason for this prediction is that we believe that there will be a negligible amount of computational time between most of our generators. From this reasoning we can predict that the worst performing random number generator in our tests will be our Linear Congruential Generator as even though it should have the highest performance score, it will generate at the lowest quality compared to our other generators, while Good performance is highly looked upon in video games quality is also an important factor that will be holding it back.

3.5.2 Data from the methods

Key
LCG = RANDU (Linear congruential generator)
MT = Mersenne Twister
MSE = Mouse Movement
MSE & T = Mouse Movement & Seeding time
LCG & DK = RANDU & Uranium Decay Simulator
MT & DK = Mersenne Twister & Uranium Decay Simulator

Performance Test The performance is the amount of time in nanoseconds that it takes for the algorithm to generate 3 floats, which are then converted into a vector3 (position) at which a node is spawned. The timer measures exactly how long the computer spent on the code, it didn't measure any engine overhead or take into account extra delays such as for the mouse method if the user stops moving the mouse which gives valid input the timer stops being run. We did this to achieve a reading for just the computational process of each node to get the fairest reading possible.

Performance Test						
RNG	T1	T2	T3	T4	T5	Avg.
LCG	1588	1628	1638	1464	1582	1580
MT	2099	1919	1814	1953	1922	1941
MSE	3829	3714	3736	4365	7491	4627
MSE&T	5647	5604	5945	6066	5653	5783
LCG&DK	2381	1902	2223	2607	2024	2227
MT&DK	5493	5825	5836	5422	5592	5634

RNG - Type of RNG used
Test # (T#) - Test Iteration, measured in nanoseconds
Avg. - Average Time, measured in nanoseconds

Period length The period length is the amount of

numbers that can be generated before the generator will repeat the same set of numbers again.

Quality Test	
RNG	Period length
LCG	$2^{29} - 1$ [8]
MT	$2^{19937} - 1$ [20]
MSE	arbitrarily large*
MSE & T	arbitrarily large*
LCG & DK	arbitrarily large*
MT & DK	arbitrarily large*

**arbitrarily large means that the generation of numbers without repetition is, in theory, infinite but due to things like the "uranium" decaying after* 3.21×10^{11} *years, or no valid input from the user for mouse generator may be affected.*

Implementation Difficulty For this, we used the Agile Estimation Technique that is used for creating User Stories where you use Fibonacci numbers (1, 2, 3, 5, 8, 13...) to rate the time estimation for each method based on what the other methods took [17], while it is subjective it is still necessary to get an overall feel of how all of the methods line up to each other. This method is usually used for estimation before the task is done, however, since we are doing this in past tense we are able to take the actual time it would have taken to complete it from scratch and place it on a point on the Fibonacci sequence

Implementation Difficulty		
RNG	Hours	Fibonacci
LCG	1	1
MT	7	3
MSE	2	2
MSE & T	2	2
LCG & DK	21	5
MT & DK	27	5

Final Chart The final chart will be what we weigh everything up against, since we are using a lower number better system we will convert the data from Performance Test and Period length into the same method that we used for the Implementation Difficulty by using Fibonacci numbers[17]. We will also handicap mouse movement as an RNG due to it relying on a user providing enough input to generate numbers which rely on the user mouse settings which could not be sufficient to pass the threshold to generate numbers.

RNG Statistics NON WEIGHTED			
RNG	P	Q	I
LCG	1	8	1
MT	2	5	3
MSE	5	3	2
MSE & T	8	3	2
LCG & DK	2	1	5
MT & DK	8	1	5

RNG - Type of RNG used
Performance (P) - Performance Score
Quality (Q) - Quality Score
Implementation Difficulty (I) - Implementation Difficulty Score

WEIGHTING we will also be attaching weight onto various values since we are trying to find an improved methodology in relation to video games, because of this we need to prioritize quality and performance even more so whereas the implementation difficulty is much less important as its something that needs to be set up once then it can be left alone and will no nothing to affect gameplay. We will be reflecting these changes by multiplying the values:
$Implementation\ Difficulty \times 0.5$
$Quality \times 1.5$
$Performance \times 2$

RNG Statistics WEIGHTED				
RNG	P	Q	I	Total
LCG	2	12	0.5	14.5
MT	4	7.5	1.5	13
MSE	10	4.5	1	15.5
MSE & T	16	4.5	1	21.5
LCG & DK	4	1.5	2.5	8
MT & DK	16	1.5	2.5	20

RNG - Type of RNG used
Performance (P) - Performance Score
Quality (Q) - Quality Score
Implementation Difficulty (I) - Implementation Difficulty Score

3.5.3 Analysis of the data

From our results that we gathered, we can observe that the best random number generator that we utilized in our tests was the combination of our linear congruential generator with our uranium decay simulator being used as the increment value. Our results indicate that using a simple LCG with our uranium decay simulator as the increment value strikes the best balance between performance, quality, and implementation difficulty in the context of video game development. The reason that this RNG did some well is most likely due to the fact our LCG algorithm is the least computational intensive compared to our other RNGs, which provides a good performance score which is then combined with our uranium decay simulator to gain one of the best possible quality scores, this covers the biggest weakness of our RANDU based LCG on its lonesome.

The second-best result is the Mersenne Twister, this is logical as it is the method that is currently in popular use with video games and it provides a decent balance between Performance and Quality while also having a low Implementation difficulty making it appealing for people that are looking for a quick result and is good enough for most scenarios.

The third best result was the LCG, this was unexpected as after seeing the grouping of nodes that formed after a short time we determined that the quality would cause the ranking to be one of if not the lowest, however after being weighted towards performance and quality, it's performance score rose it's ranking into the top three. Along with having the easiest implementation method of all of our random number generators that we tested and because we are in conducting our study in relation to video games we are able to excuse the downside of bad quality over performance.

The fourth best result was the Mouse random number generator, this is somewhat surprising to not see it higher ranked as this also sees use in the industry currently, such as in the Linux kernel as a way of generating entropy for security measures [3], however, the performance for security measures is not as necessary whereas in video games it is of very high consideration and the Mouse has a quite bad performance which is holding it back compared to the other methods. The mouse also has the downside that it requires the end-user to be constantly be moving their mouse above a threshold to be generating numbers which in video games is not always viable.

The second to last option is the Mersenne Twister combined with our uranium decay simulator which sees a large jump in the score, this is because of the simple logical reason that integrating the uranium decay simulator into any system will give it an incredibly high-quality score so integrating the uranium decay simulator with the Mersenne Twister is pointless as the effect this change has on quality remains unchanged when comparing it to integrating our linear congruential generator with our uranium decay simulator, and the effect on performance between these two is significant as our Mersenne Twister combined with our uranium decay simulator take almost 3 times longer to compute. Especially in the context of video games where the performance is the most important aspect to design for, it also lacks having any real benefits over our linear congruential generator with our uranium decay simulator method so it makes sense that it is one of the lowest-ranked options available.

Finally, the worst ranking method is our Mouse combined with Time Method. This runs into the same problem as our Mersenne Twister combined with our uranium decay simulator as it is simply taking true random numbers (TRNG)[12] and running them through a more complex and computational intensive method that doesn't achieve any higher quality as it was already random, to begin with, and instead just causes the performance to worsen. The reason this is the worst is that it still suffers from all the limitations that the regular mouse method provides with the slight added flaw of it being an extra 25% slower based on our recordings.

If a video game company were to attempt to apply this method of generating random numbers in the real

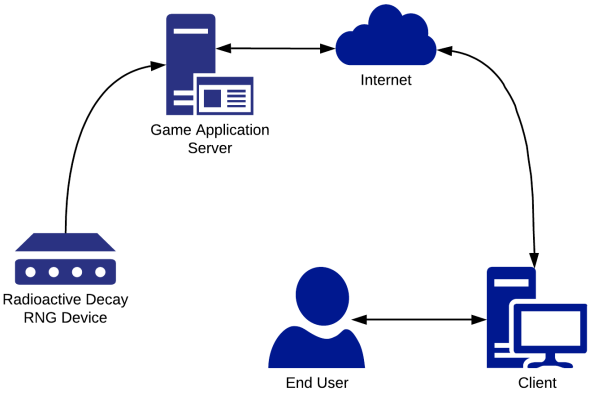

Figure 7: Example of how radioactive decay RNG could be used in games.

world typically end-users will not possess a device to measure radioactive activity that will feed into their video game. Therefore it would be important to supply an end-user with the necessary means to generate random numbers. Our proposed solution to this issue is presented in Figure 7. A company would have a Radioactive Decay RNG Device that could convert detected particles into a large integer using our method. The company would also host a Game Application Server that could read from the Radioactive Decay RNG Device. When an end-user opens a game, the game connects to the Game Application Server and will be served the latest seeding value from the Radioactive Decay RNG Device, It will then use that value to generate random numbers. During run-time, the game will periodically request the Game Application Server for it's latest seeding value to update it's own.

4 Conclusion

In conclusion, our results indicate that there is a potentially improved methodology for generating random numbers over current methods being used inside of video games. Using our linear congruential generator combined with our uranium decay simulator proved to have the best score from our weighted results that scores favoring targeting towards the use of random number generators in the context of video games. Below we have linked our GitHub repository for downloading, viewing code and our license as well as reproducing results.
GitHub Repository: `https://github.com/henry9836/RNG-VENTURE`

Note: results may vary based on different computer specifications.

References

[1] Elaine Barker and John Kelsey. *Recommendation for random bit generator (rbg) constructions*. Tech. rep. National Institute of Standards and Technology, 2016.

[2] Walter E Brown. "Random number generation in C++ 11". In: *ISO/IEC JTC1/SC22/WG21 document N* 3551 (2013).

[3] Adam Everspaugh et al. "Not-so-random numbers in virtualized linux and the whirlwind rng". In: *2014 IEEE Symposium on Security and Privacy*. IEEE. 2014.

[4] GNOME Foundation. *Random Numbers*. 2014. URL: `https://developer.gnome.org/glib/stable/glib-Random-Numbers.html`.

[5] Python Software Foundation. *9.6. random — Generate pseudo-random numbers*. 2017. URL: `https://docs.python.org/3.3/library/random.html`.

[6] The PHP Group. *mt_rand*. 2020. URL: `https://www.php.net/manual/en/function.mt-rand.php`.

[7] Steven D. Kahl, G. Sitta Sittampalam, and Jeffrey Weidner. *Calculations and Instrumentation used for Radioligand Binding Assays*. Oct. 2012. URL: `https://www.ncbi.nlm.nih.gov/books/NBK91997/`.

[8] Dean Karlen. "Pseudo-Random Numbers". In: *Physics 75.502/487* Computational Physics Fall/Winter 1998/99*. Carleton University, 1999, p. 146. URL: http://kfe.fjfi.cvut.cz/~limpouch/numet/karlen-book.pdf.

[9] Pierre L'Ecuyer. "History of uniform random number generation". In: *WSC 2017 - Winter Simulation Conference*. Las Vegas, United States, Dec. 2017. URL: https://hal.inria.fr/hal-01561551.

[10] Peter A Lewis. *Graphical analysis of some pseudo-random number generators*. Tech. rep. NAVAL POSTGRADUATE SCHOOL MONTEREY CA, 1986.

[11] Libretexts. *9.5: Rates of Radioactive Decay*. Nov. 2017. URL: https://chem.libretexts.org/Courses/Valley_City_State_University/Chem_122/Chapter_9:_Nuclear_Chemistry/9.5:_Rates_of_Radioactive_Decay.

[12] Yuan Ma, Jingqiang Lin, and Jiwu Jing. "On the entropy of oscillator-based true random number generators". In: *Cryptographers' Track at the RSA Conference*. Springer. 2017, pp. 165–180.

[13] George Markowsky. "The sad history of random bits". In: *Journal of Cyber Security and Mobility* 3.1 (2014).

[14] Makoto Matsumoto. *Mersenne Twister in C, C++, C*. Oct. 2010. URL: http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/VERSIONS/C-LANG/c-lang.html.

[15] Makoto Matsumoto and Takuji Nishimura. "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator". In: *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 8.1 (1998).

[16] Guy Mélard. "On the accuracy of statistical procedures in Microsoft Excel 2010". In: *Computational statistics* 29.5 (2014).

[17] Mario E Moreira. "Working with story points, velocity, and burndowns". In: *Being Agile*. Springer, 2013, pp. 187–194.

[18] C++ Resources Network. *std::minstd_rand*. 2020. URL: http://www.cplusplus.com/reference/random/minstd_rand/.

[19] William H Press et al. *Numerical recipes in Fortran 77*. 1986.

[20] John Savard. *The Mersenne Twister*. 2012. URL: http://www.quadibloc.com/crypto/co4814.htm.

[21] Patrick Suppes. "Explaining the unpredictable". In: *Epistemology, Methodology, and Philosophy of Science*. Springer, 1985.

[22] Ed Vitz et al. *19.8: The Rate of Radioactive Decay*. Feb. 2020. URL: https://chem.libretexts.org/Bookshelves/General_Chemistry/Book:_ChemPRIME_(Moore_et_al.)/19Nuclear_Chemistry/19.08:_The_Rate_of_Radioactive_Decay.

[23] International Bureau of Weights et al. *The international system of units (SI)*. US Department of Commerce, Technology Administration, National Institute of . . ., 2001.

[24] Math Works. *RandStream.list*. 2020. URL: https://au.mathworks.com/help/matlab/ref/randstream.list.html.