Random Numbers and Mappings

Steve Rotenberg
CSE168: Rendering Algorithms
UCSD, Spring 2014

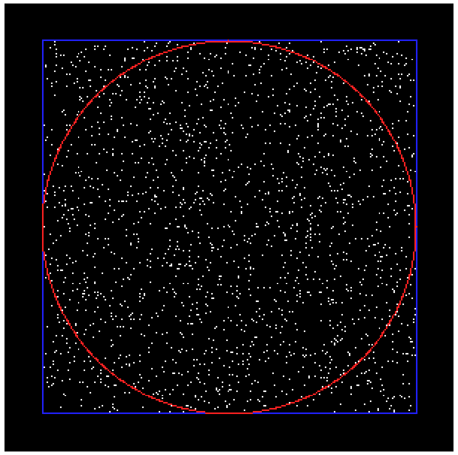
Monte Carlo Integration

- We’ve seen a few examples so far where we use randomness and random sampling to resolve complex issues such as handling area light sources and pixel antialiasing
- These are actually examples of a technique called *Monte Carlo integration*
- As Monte Carlo is a place typically associated with gambling, the term was applied to mathematical integration processes that use a series of random evaluations to estimate a complex function

Estimating π

- Let’s say that we want to use random Monte-Carlo integration to compute the value of π
- We can do so by generating random 2D points in the [-1…1] interval and then counting how many of them are within a distance of 1.0 from the origin
- The ratio of the number points inside the circle to the total number of points will be approximately equal the ratio of the area of the circle (π) to the area of the square (4)

$$\frac{N_{inside}}{N_{total}} \approx \frac{\pi}{4} = 0.78539816\ldots$$



Random Numbers

Random Number Generators

- Digital computer logic is inherently incapable of producing true ‘random’ numbers
- However, we can come up with digital algorithms that generate number sequences that appear to be random
- We call these *pseudo-random number generators*
- It is actually possible to generate true *random* numbers with special purpose hardware. For example, some special cryptography hardware contains radioactive materials that randomly decay and produce detectable particles that can be used to generate random numbers
- This is overkill for our purposes, and we’ll be fine with algorithmic pseudo-random numbers
- Still, we will be generating a lot of random numbers and so we do care about the quality and performance of the random number generator

Random Number Generators

- The standard C library contains some functions for generating random numbers such as rand(), and others
- In general, these are considered unreliable for the purposes of high quality rendering, as they tend to vary in quality from system to system
- Most renderers will use their own random number generator or a trusted 3rd party number generator
- The classic '*ran1*' function from chapter 7 of <u>Numerical Recipes</u> is fast and produces good quality results and can be easily found online. It produces sequences of 32-bit random numbers, and the sequence will eventually repeat after 2^{31} numbers are generated
- Another popular, modern random number generator is the '*Mersenne Twister*' developed in 1997 by Makoto Matsumoto and Takuji Nishimura. It generates either 32 or 64-bit numbers and repeats after 2^{19937} numbers…

Random Number Seeds

- Pseudo-random number generators usually allow the user to set a ‘seed’ value, which sets the position in the sequence
- This allows one to re-generate the same sequence of pseudo-random numbers by resetting the seed back to an earlier value
- Often times, the seed is just initialized to some arbitrary number like 1234567 at the start of the program, or it is set to the current clock value
- One important thing to keep in mind is that when one spawns new threads in a multi-threaded app, the number sequences generated by each thread will be the same, since they start off with the same seed values. It is therefore a good idea to reset the random number seed for each thread when the thread is started up

Pseudo-Random Number Generators

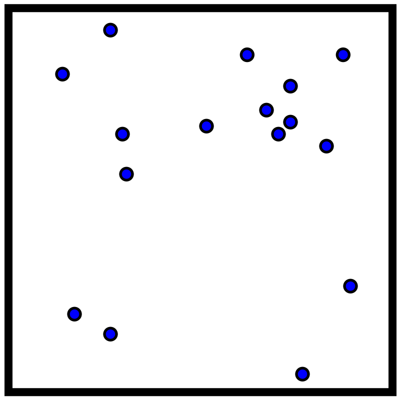
- There are a variety of approaches to generating pseudo-random numbers, differing by quality and performance
- Some of the simplest are *linear congruential generators* of the form:

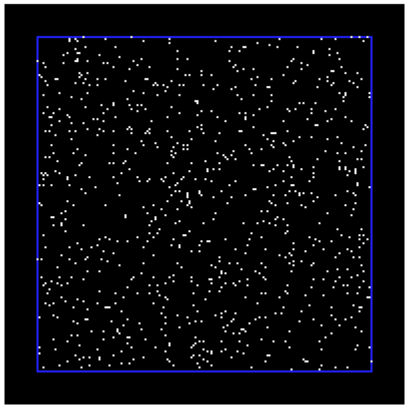
$$n_{i+1} = \mathrm{mod}(an_i + c, m)$$

- Where all variables are 32 bit unsigned ints. The result can be easily converted to a float in the [0…1] range
- *a*, *c*, and *m* are selected carefully to produce good results. For example, *a*=7^5=16807, *c*=0, and *m*=2^{31}-1=2147483647 are popular choices
- Extensions to the linear congruential generators tend to add bit shuffling and combine multiple sequences generated from different constants

Random Numbers

- If we generate a bunch of random 2D points on the [0…1] interval, and plot the results, we expect to see something like this:

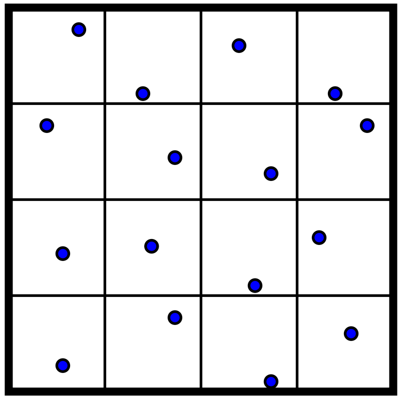


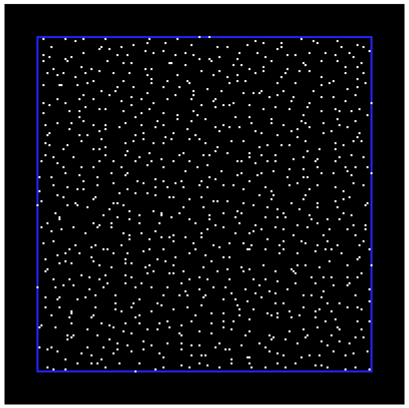


- There are a lot of cases where we need to generate 2D, 3D or n-D random points according to some distribution, and we typically start with random points in the [0…1] interval

Stratified (Jittered) Sampling

- We can generate a more uniform, but still random coverage of the unit square by using *jittered* or *stratified* sampling
- With jittered sampling, we grid up the unit square by some resolution and place a random point in each cell
- The catch is that we need to decide ahead of time how many total samples we want, and we much pick a number that is either the square of an integer or the product of two integers at the very least





Jittered Sampling

- For Monte-Carlo integration applications, jittered sampling is *always* better than purely random sampling
- This holds true for rendering applications such as antialiasing, area light sampling, camera focus, texture EWA sampling, and others
- The main catch is that it requires deciding ahead of time how many samples are required which is usually not a big problem
- Another catch is due to the high dimensionality of the functions we want to integrate. This can lead to potential problems if we are using jittered sampling at multiple points within the pixel rendering process. For example, if we are generating camera rays by jittering the pixel position and jittering the camera lens position, we need to make sure that the two jitter patterns don't sync up in any way

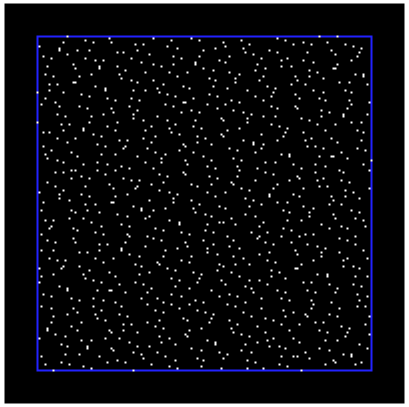
Quasi-Random Numbers

Quasi-Random Numbers

- *Quasi-random* numbers are sequences of numbers specifically designed to fill an interval (typically [0…1] in n-dimensional space) in a relatively uniform way (compared to purely random or pseudo-random)
- Each new number (or vector) in the sequence is spaced as far as possible from the previous ones
- The goal is to be able to generate randomized, but relatively uniformly spaced patterns, similar to jittering, but not requiring knowledge ahead of time about the total number of samples

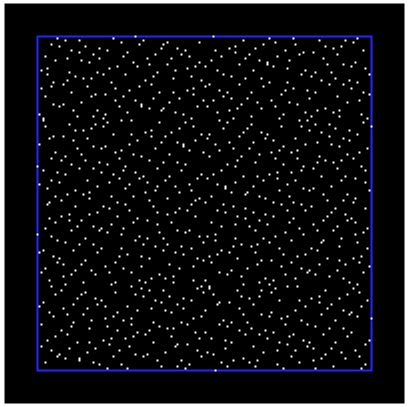
Halton Sequence

- The *Halton sequence* is one example of a quasi- random number generator
- To generate the i^{th} number in the sequence h_i, we write *i* in base *b*, where *b* is a preselected prime number (for example, *i*=17 in base *b*=3 is 122)
- We then reverse the digits and add a decimal point at the beginning (for example 0.221, base 3)
- To generate an n dimensional vector, you use the same value of *i* with a different prime number *b* for each dimension



Sobol Sequence

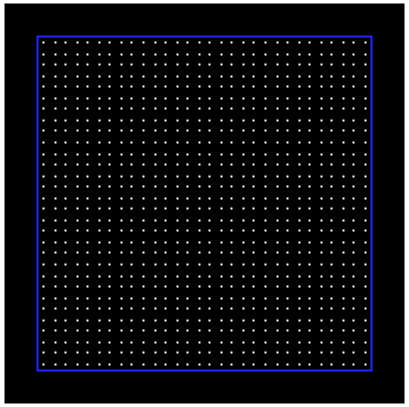
- The *Sobol seqence* is another example of a quasi-random generator
- It uses a more complex algorithm to generate numbers than the Halton sequence, but generates essentially similar results
- It tends to work better in rendering applications, as the Halton sequence creates more visible patterns

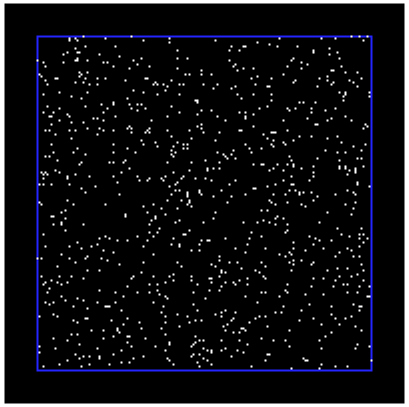


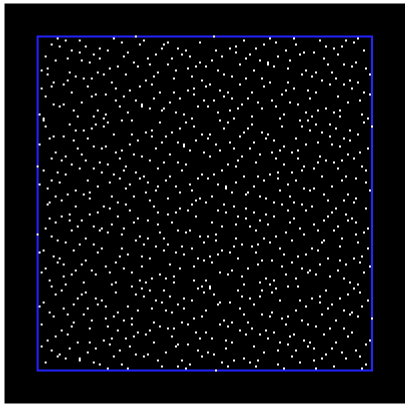
Quasi-Random Numbers

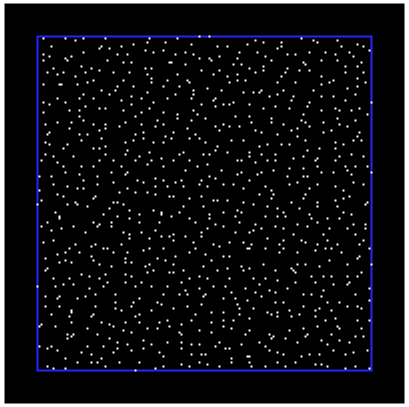
- Quasi-random numbers are a useful tool in ray tracing and random sampling
- They offer the ability to generate well spaced sequences similar to jitter patterns but without requiring knowledge ahead of time of how many samples are needed
- This means that they have some good potential for use in *adaptive sampling* schemes
- However, they can suffer from some strange visual artifacts such as visual patterns in the images
- Like any tool, they need to be understood and experimented with to determine their most effective uses

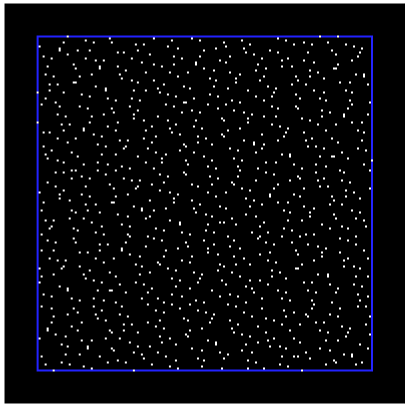
Summary of Number Generators


Uniform


Random


Sobol


Jittered / Stratified


Halton

Number Generators

Type	Pro	Con
Uniform	• Good distribution of samples • Consistent: should not generate any noise	• Need to know total ahead of time • Subject to aliasing and Moiré patterns
Random (pseudo-random)	• Breaks up patterns • Don’t need to know total ahead of time	• Clumping of samples & empty regions leads to inefficient estimation
Jittered	• Good distribution of samples • Faster convergence of estimated functions	• Need to know total ahead of time
Quasi-random	• Good distribution of samples • Faster convergence of estimated functions • Don’t need to know total ahead of time	• Subject to strange patterns in results

Mappings

Mappings

- It’s nice to be able to generate random or jittered patterns within a unit square, but usually, we are interested in generating random points across some other shape
- For example, if we have a triangle area light, we might need to select a random point on the triangle
- Or if we are estimating the total light arriving on a diffuse surface, we might need to select several random directions in a hemisphere
- To achieve these things, we use various *mappings* that take points in the unit square [0…1] and map them to some other shape

Pixel Mappings

- For example, in the antialiasing lecture, we talked about pixel supersampling and using either the Gaussian or Shirley mappings to bias the samples towards the center of the pixel

Pixel Mappings

- Let's say we start with two random numbers *s* and *t* that are uniformly distributed in the [0…1] interval
- We want to generate two new random numbers *s'* and *t'* in the same interval but with some type of weighting towards the center
- Gaussian:

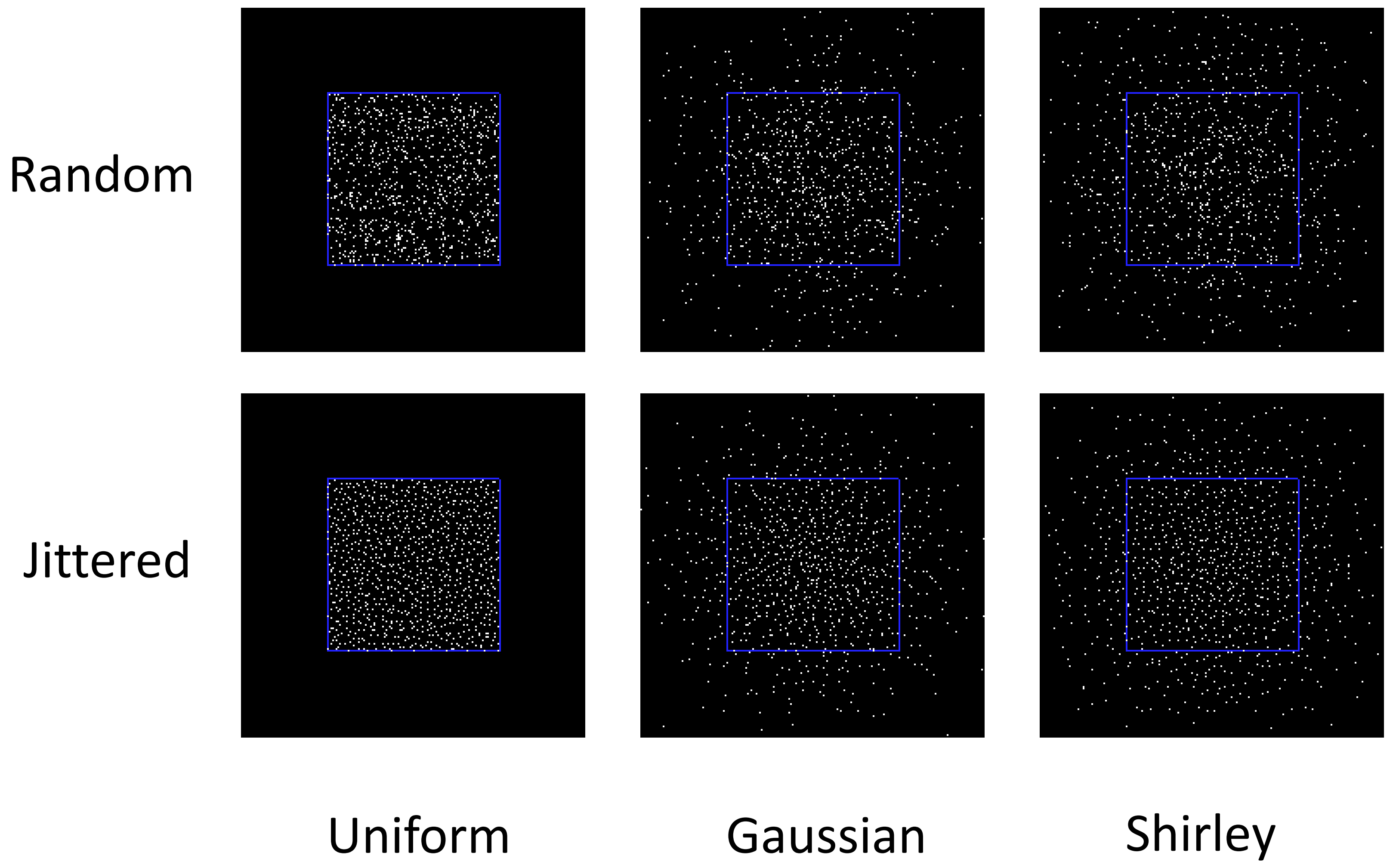
$$a = \mu\sqrt{-2\log s} \qquad (\mu \approx 0.4)$$
$$b = 2\pi t$$
$$s' = 0.5 + a \cdot \sin b$$
$$t' = 0.5 + a \cdot \cos b$$

- Shirley:

$$s' = \begin{cases} -0.5 + \sqrt{2s} & if\ s < 0.5 \\ 1.5 - \sqrt{2 - 2s} & if\ s \geq 0.5 \end{cases}$$

$$t' = \begin{cases} -0.5 + \sqrt{2t} & if\ t < 0.5 \\ 1.5 - \sqrt{2 - 2t} & if\ t \geq 0.5 \end{cases}$$

Pixel Sampling



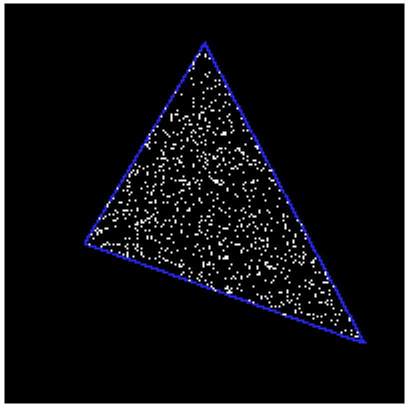
Triangle

- There are a variety of reasons why we might want to choose a random point on a triangle
- In ray tracing, it is used to sample area lights, but there are various other uses within computer graphics and animation
- Given a triangle defined by three vertices **a**, **b**, and **c**, and two random numbers *s* and *t* in the [0…1] interval, we proceed by generating random barycentric coordinates:

$\alpha = \mathrm{sqrt}(s) * t$

$\beta = 1 - \mathrm{sqrt}(s)$

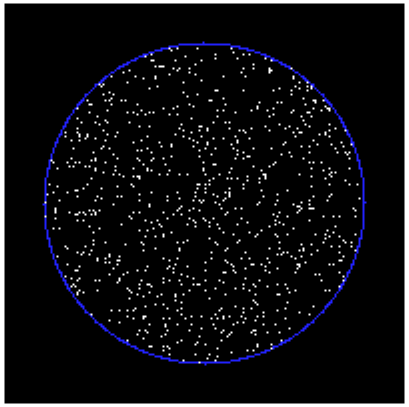
$\mathbf{p} = \mathbf{a} + \alpha(\mathbf{b} - \mathbf{a}) + \beta(\mathbf{c} - \mathbf{a})$



Disk

- In some situations, we need to distribute samples evenly across a disk
- Examples include sampling the sun as an area light source, or sampling a camera lens for focus effects
- To map uniform random numbers *s* and *t* to a disk of radius 1.0 centered at the origin:

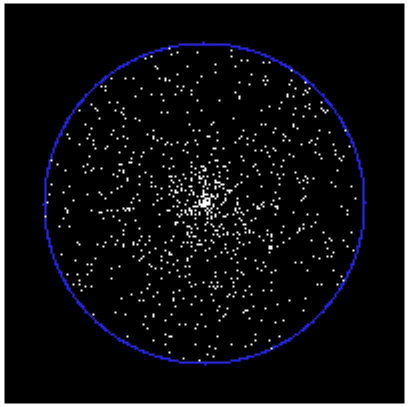
$$p_x = \sqrt{t} \cdot \cos(2\pi s)$$
$$p_y = \sqrt{t} \cdot \sin(2\pi s)$$



Disk

- Why not just use the random variables *s* and *t* directly as the polar coordinates?
- This will lead to a non-uniform distribution

$$\cancel{\begin{aligned} p_x &= t \cdot \cos(2\pi s) \\ p_y &= t \cdot \sin(2\pi s) \end{aligned}}$$

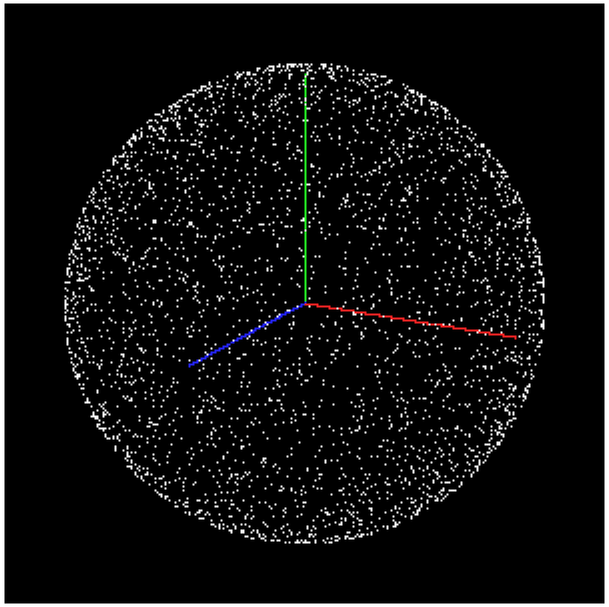


Sphere

- If we are ray tracing fog and other volumetric scattering effects, we may need to generate random directions distributed around a unit sphere

$$u = 2\pi s$$
$$v = \sqrt{t(1-t)}$$

$$p_x = 2v \cdot \cos(u)$$
$$p_y = 1 - 2t$$
$$p_z = 2v \cdot \sin(u)$$

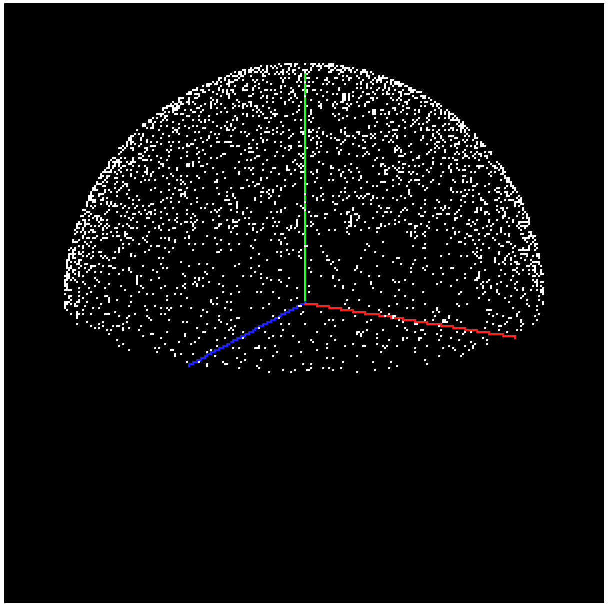


Hemisphere

- For a variety of surface lighting calculations, we need to distribute rays in a hemisphere:

$$u = 2\pi s$$
$$v = \sqrt{1 - t^2}$$

$$p_x = v \cdot \cos(u)$$
$$p_y = t$$
$$p_z = v \cdot \sin(u)$$



Cosine Weighted Hemisphere

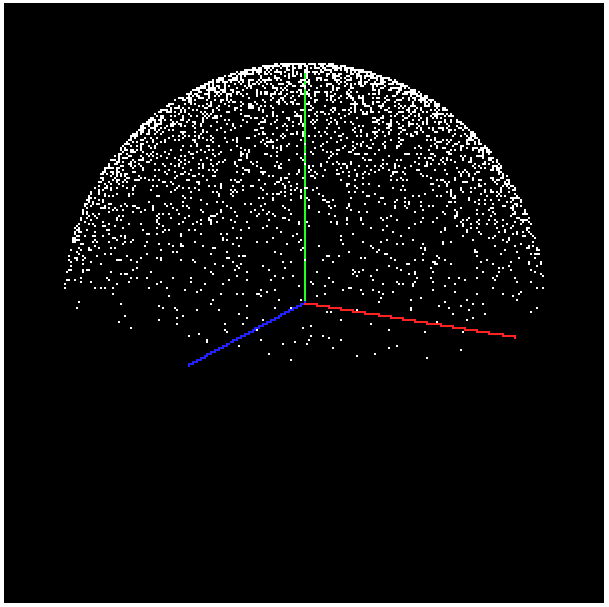
- In some lighting situations, it is actually better to distribute random rays across a cosine weighted hemisphere
- This means that there are fewer rays along the horizon, and increase in density as we go up to the pole
- If we consider diffuse lighting, for example, the incoming light is weighted by the cosine of the angle of the light ray with the surface normal
- If we are trying to sample the sources contributing light to a diffuse surface, we can use the cosine weighting and weight all samples equally, rather than use a uniform weighting and have to scale the rays by the cosine

Cosine Weighted Hemisphere

- To distribute a random point (*s*,*t*) onto a hemisphere with a cosine weighting:

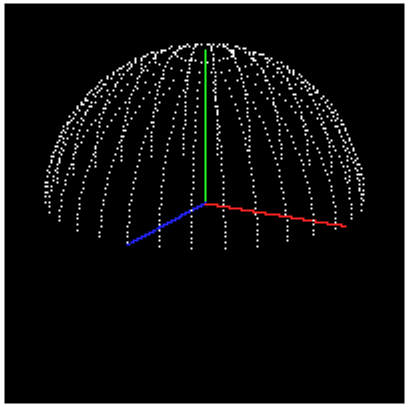
$$u = 2\pi s$$
$$v = \sqrt{1 - t}$$

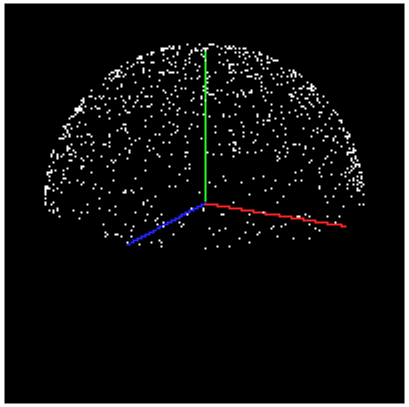
$$p_x = v \cdot \cos(u)$$
$$p_y = \sqrt{t}$$
$$p_z = v \cdot \sin(u)$$

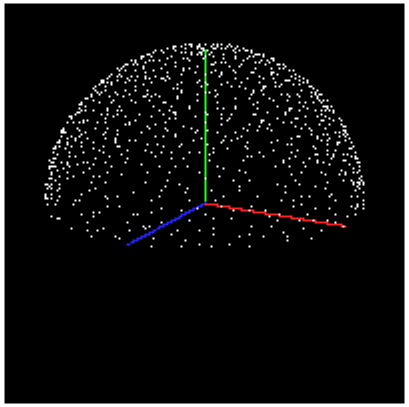


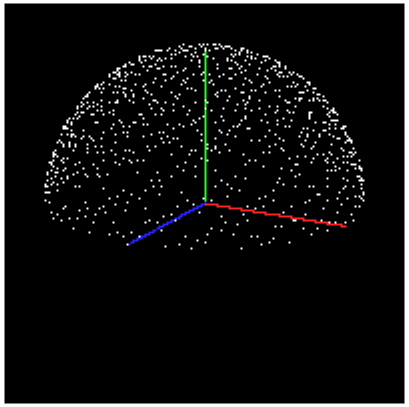
Mappings and Number Generators

- The different mappings will preserve the properties of the underlying number generator to some degree
- For example, if you take a jitter pattern and map it to a hemisphere, you should still see evidence of the jitter pattern in the hemisphere
- Some mappings do better than others at preserving the properties of the underlying number generator


Uniform

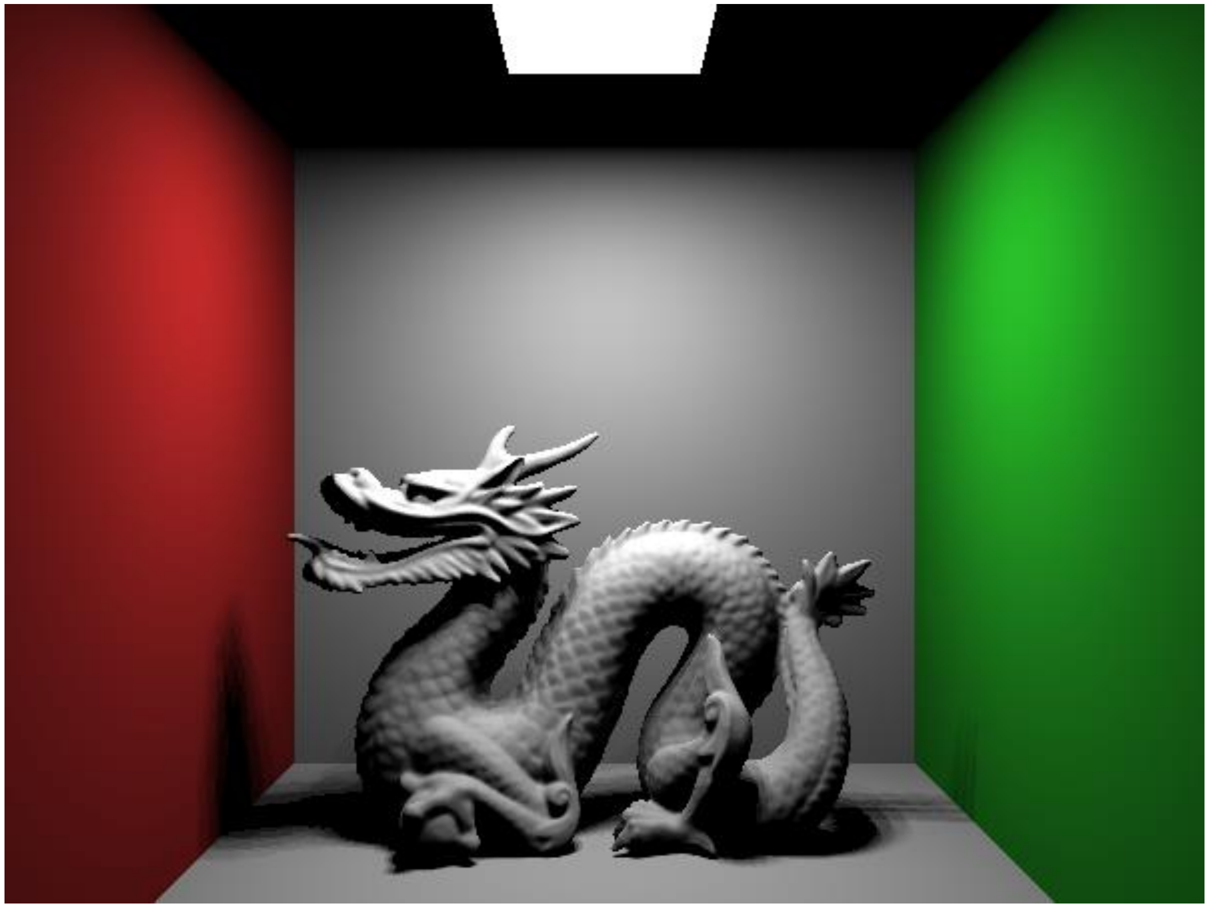

Random


Jittered

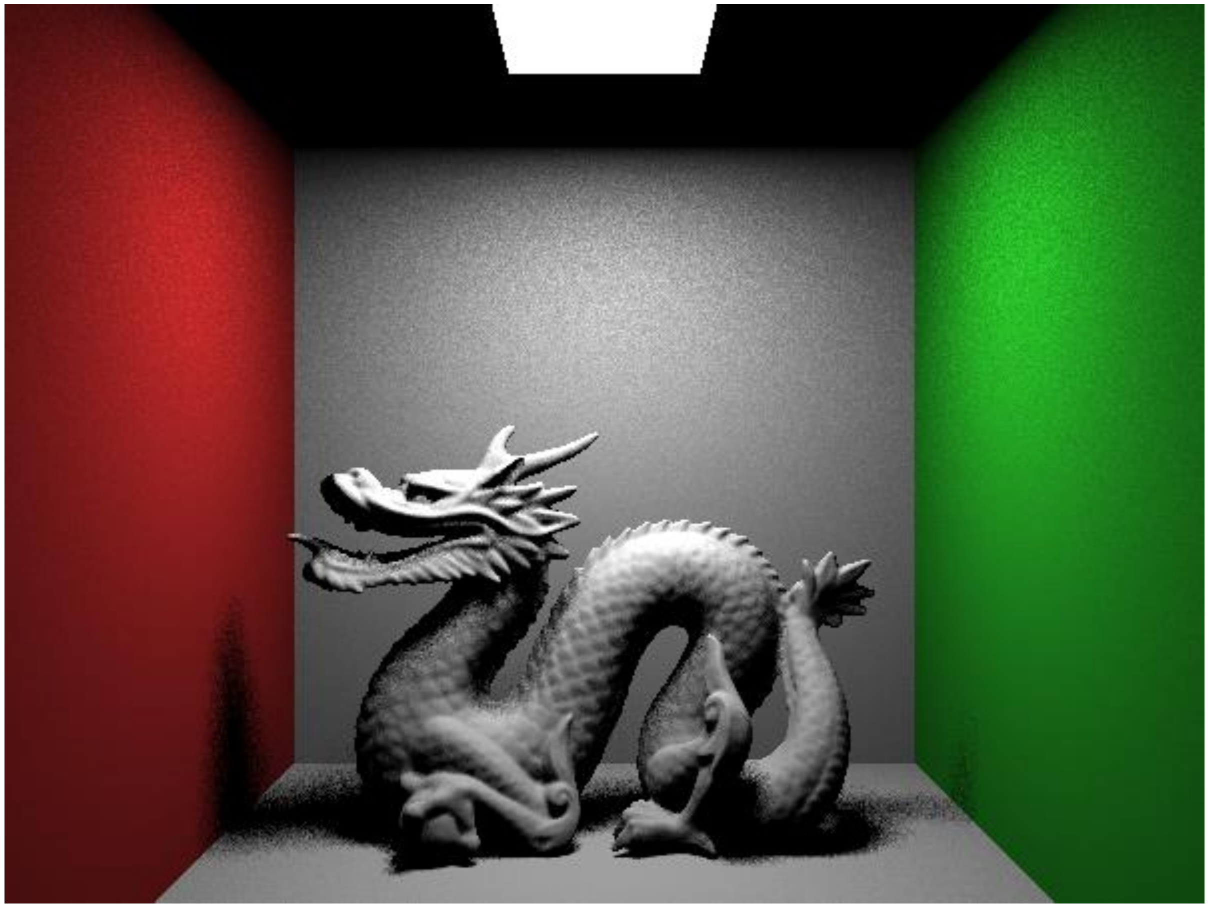

Halton

Area Light Examples

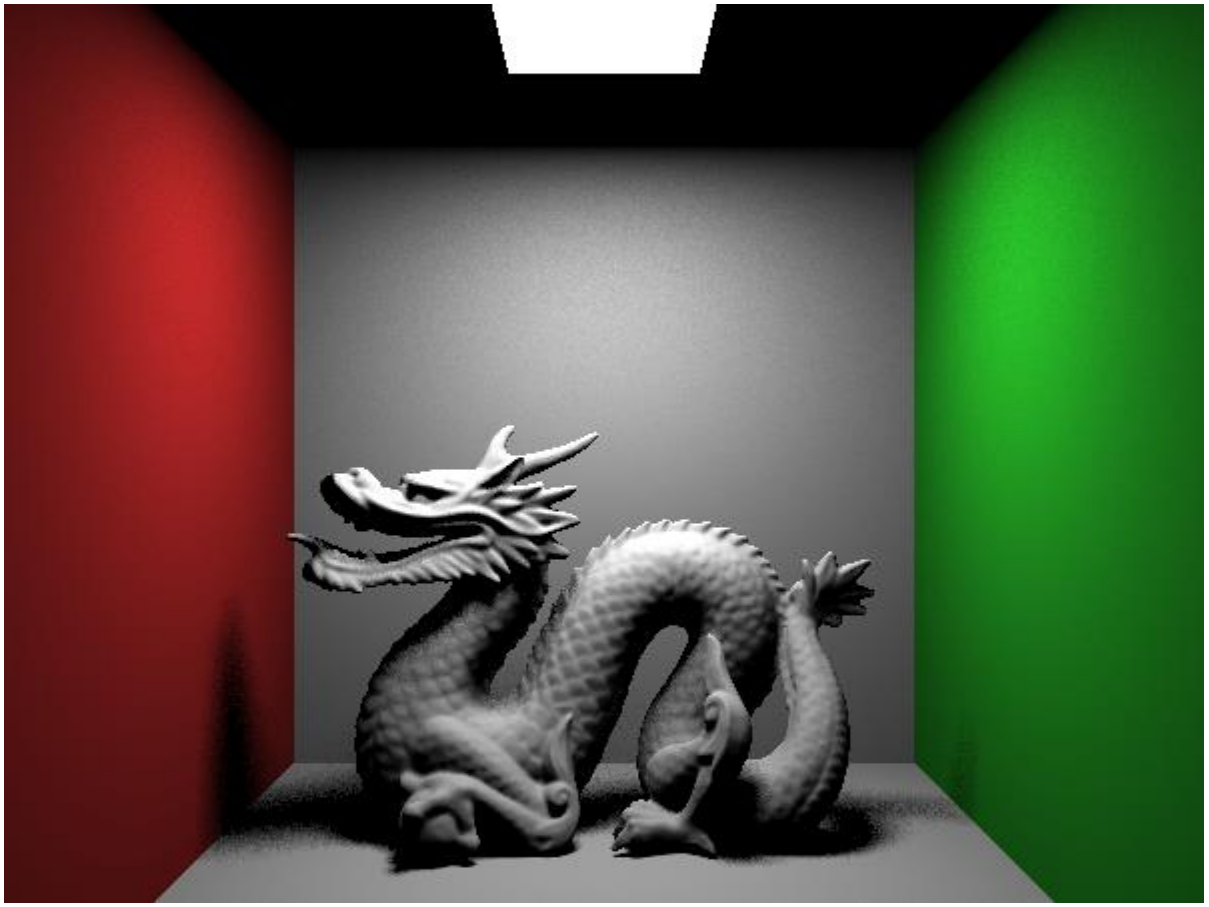
Uniform: 9 Samples



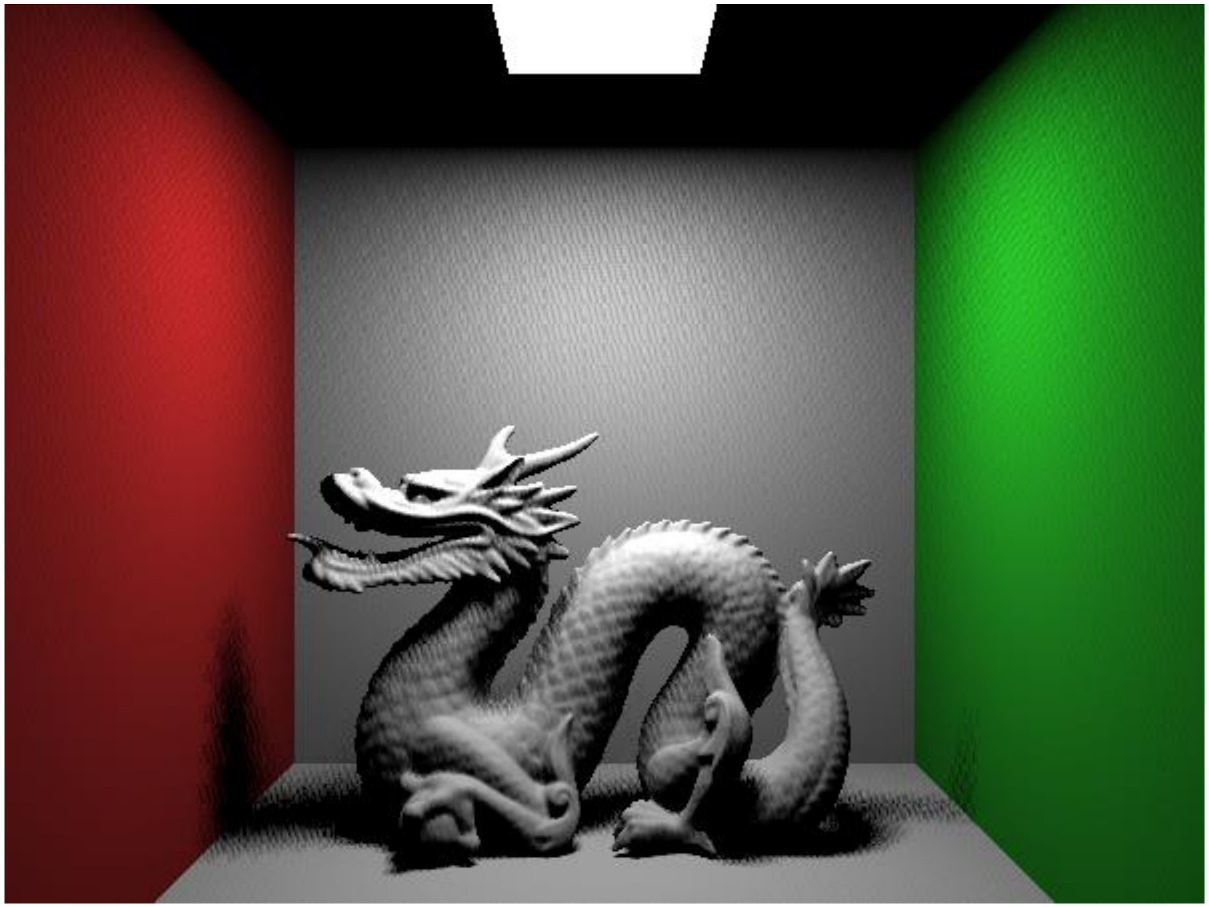
Random: 9 Samples



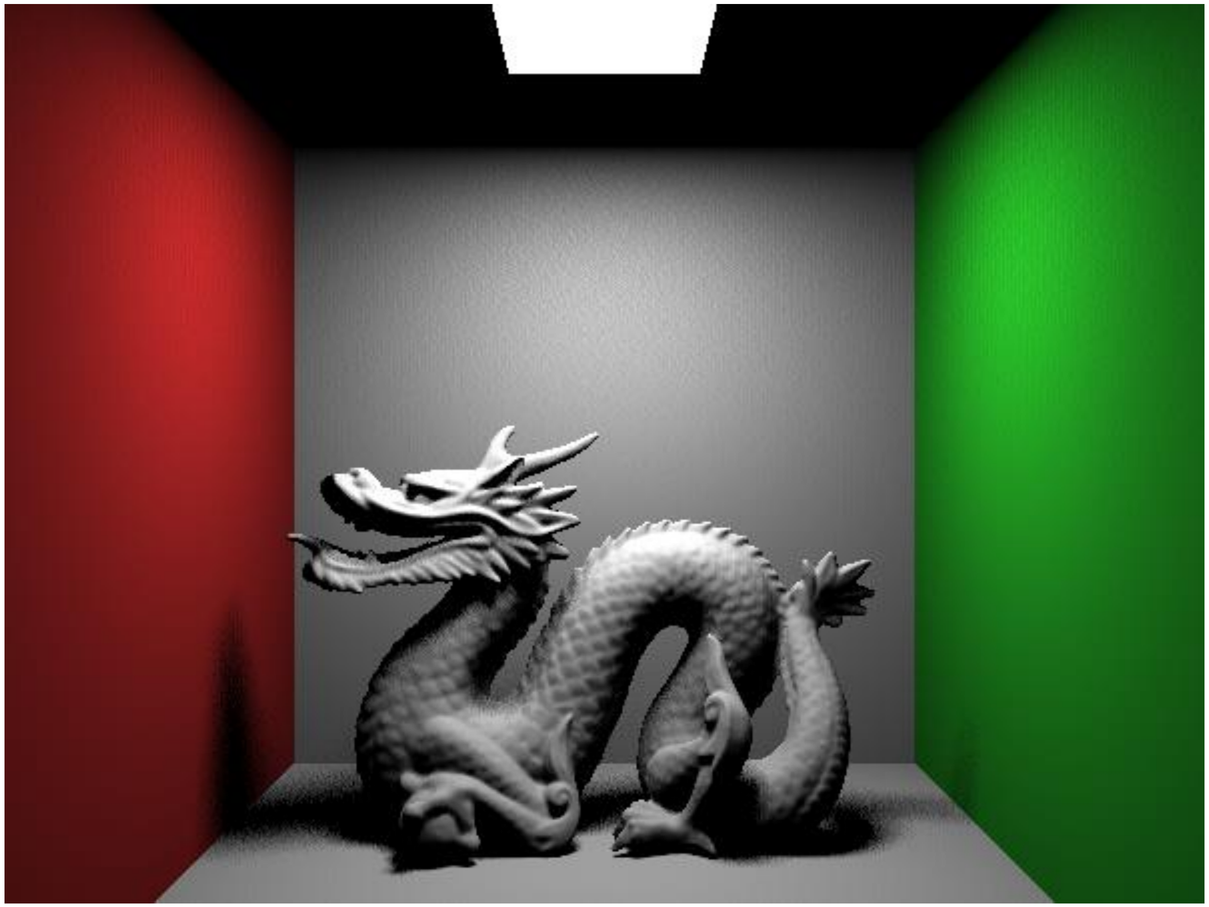
Jittered: 9 Samples



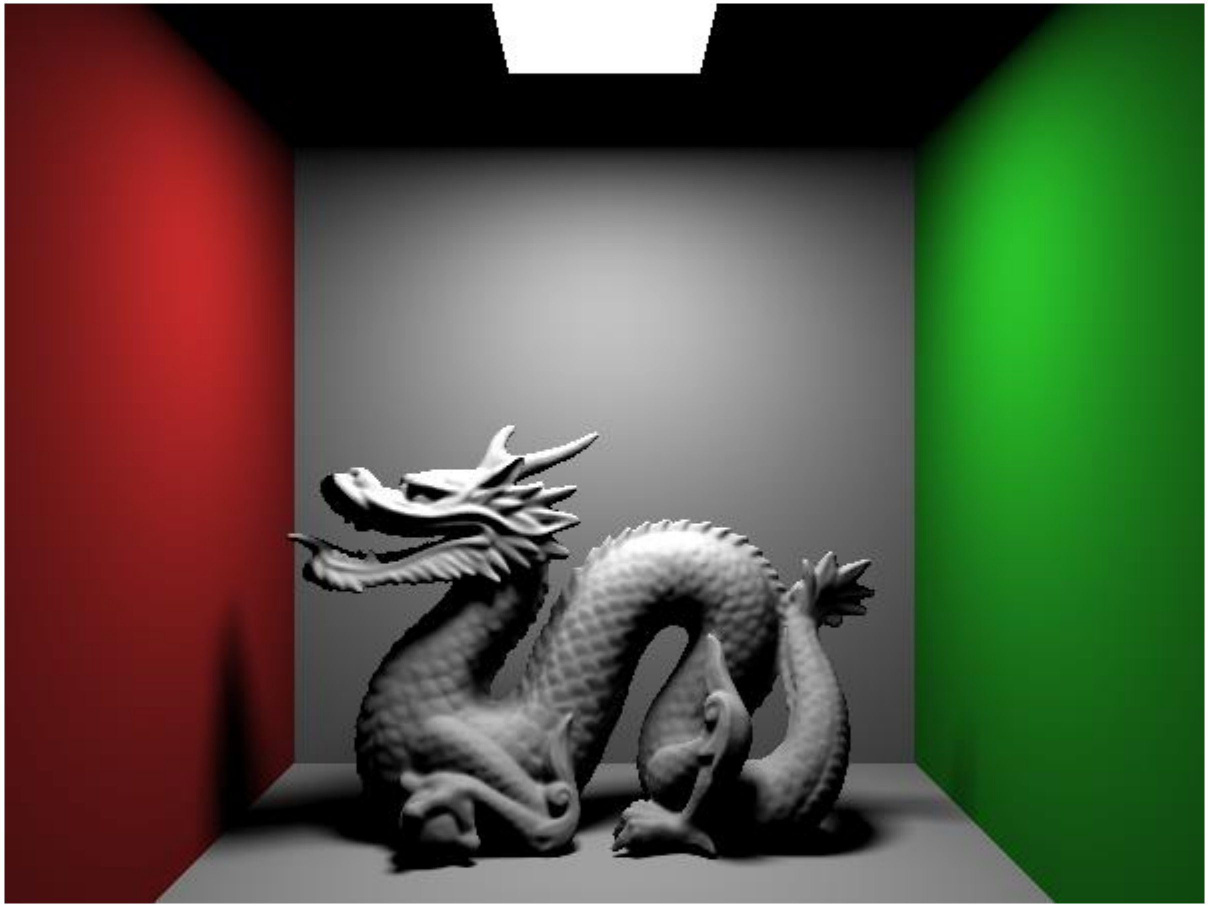
Halton: 9 Samples



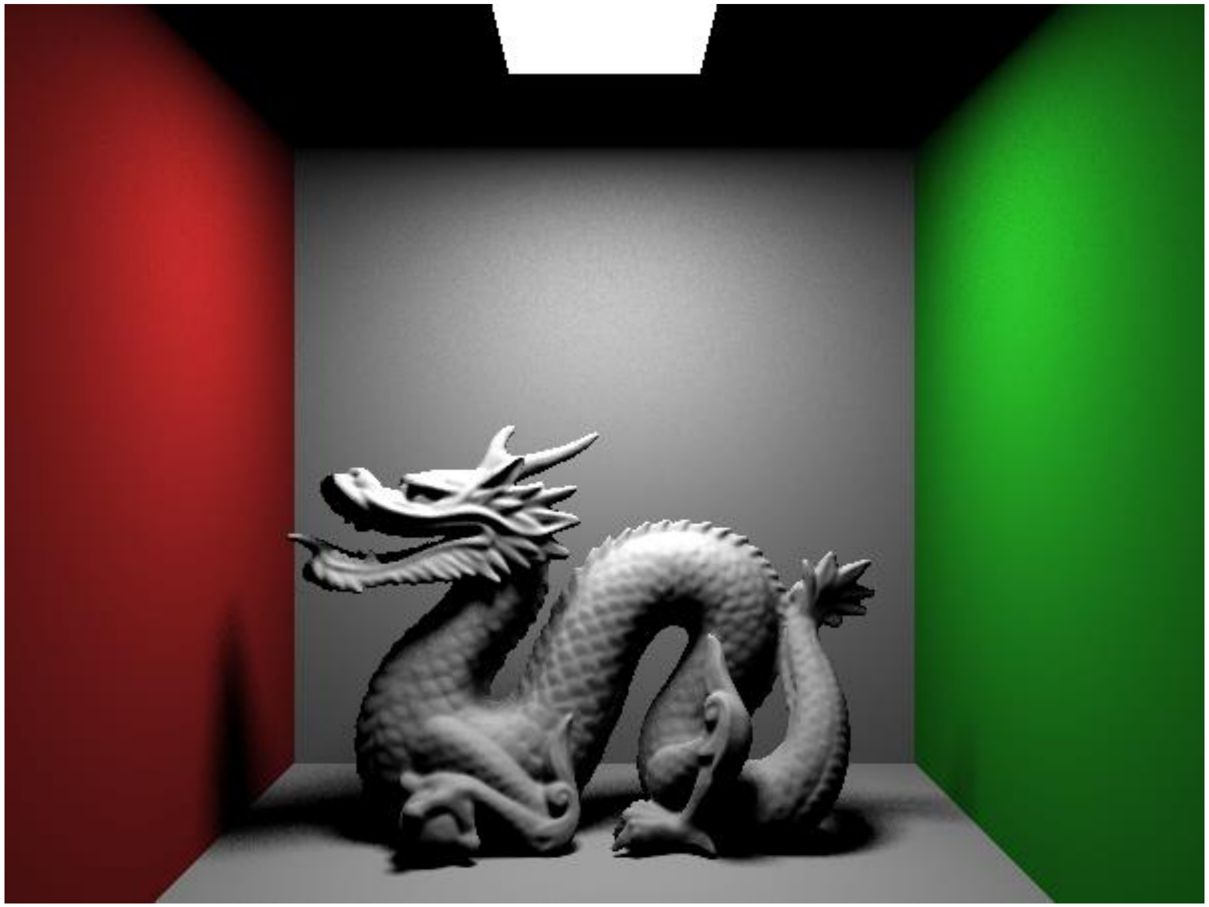
Sobol: 9 Samples



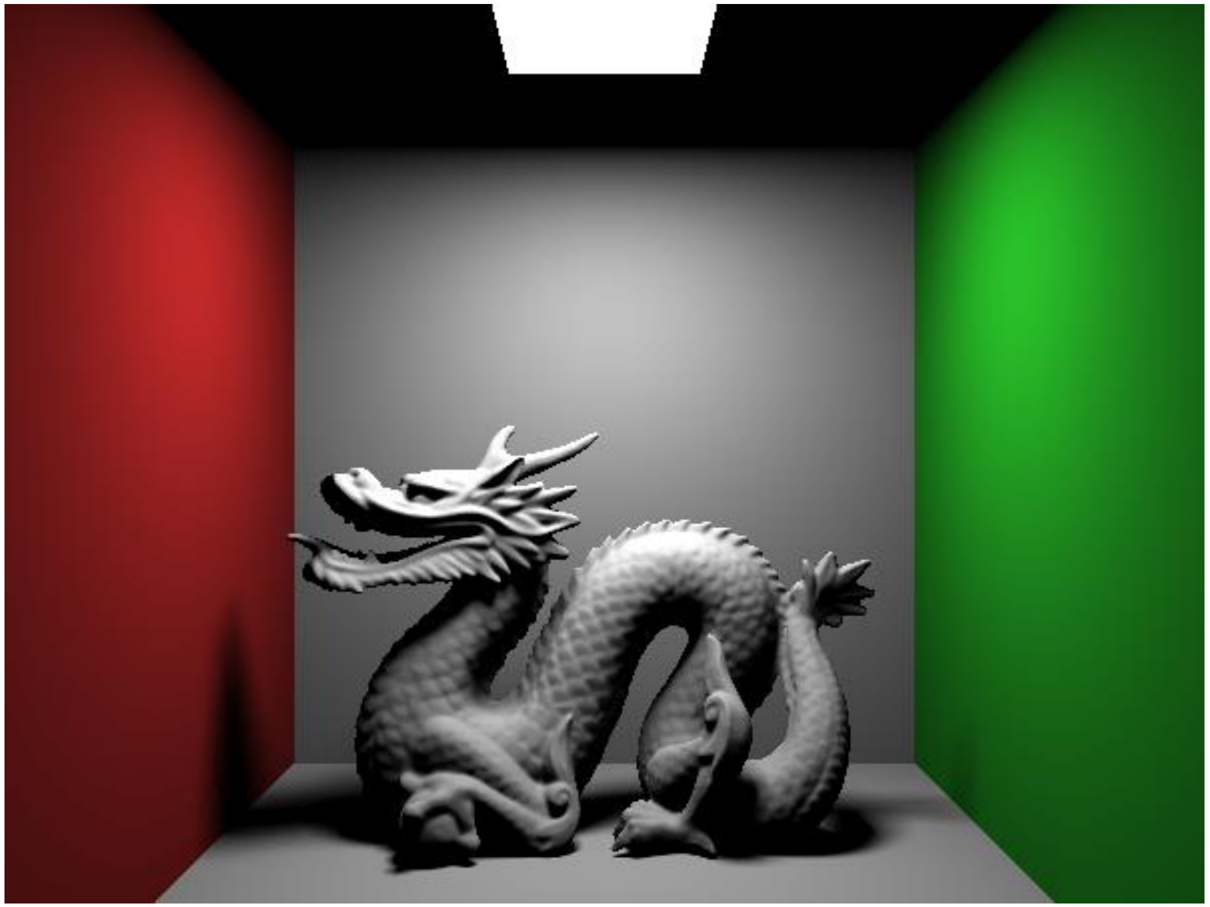
Uniform: 100 Samples



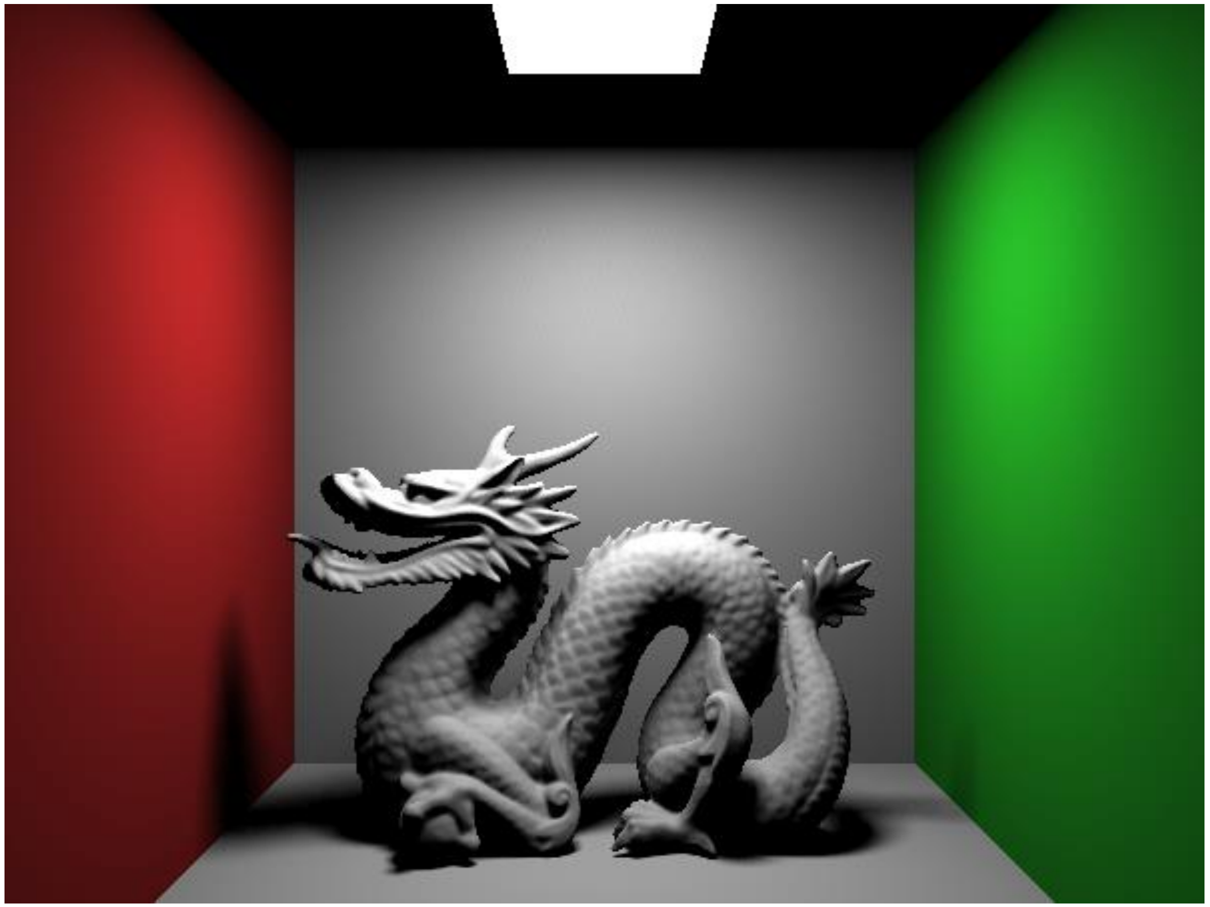
Random: 100 Samples



Jittered: 100 Samples



Halton: 100 Samples



Sobol: 100 Samples