Advanced Computer Graphics Practical Session

Final Project

Task – Implement other ‘noise’ generators and compare

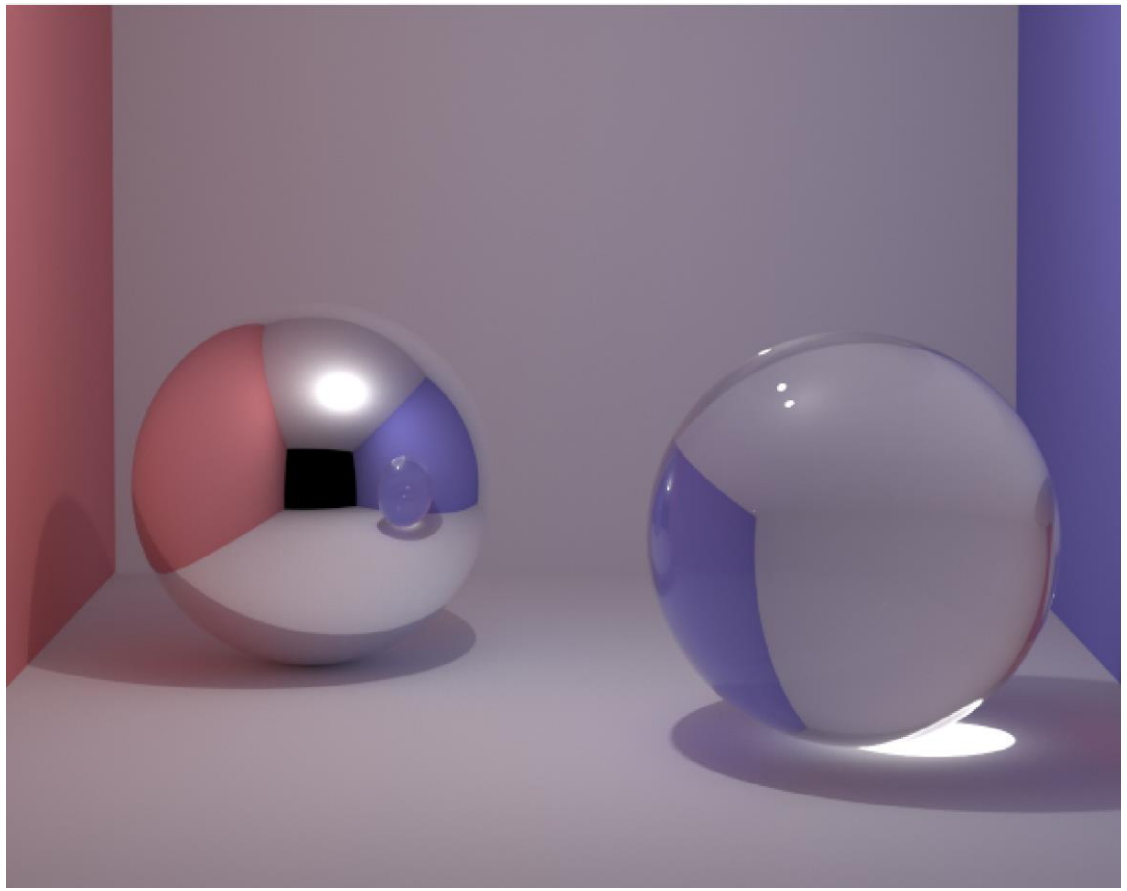
- Add a window region for rendering
- Add other ‘random’ number generators
  - Halton23
  - Stratified
  - Blue Noise
- Use instead of `drand48()`
- Compute a high-quality image for comparison (very large number of samples)
- Evaluate the other generators using lower sample counts with respect to the high-quality image (e.g. visual, difference, snr, run-time)

Task – Implement other ‘noise’ generators and compare

- Add a window region for rendering
- Add other ‘random’ number generators
  - Halton23
  - Stratified
  - Blue Noise
- Use instead of `drand48()`
- Compute a high-quality image for comparison (very large number of samples)
- Evaluate the other generators using lower sample counts with respect to the high-quality image (e.g. visual, difference, snr, run-time)

Task – Implement other ‘noise’ generators and compare

- *Add a window region for rendering*

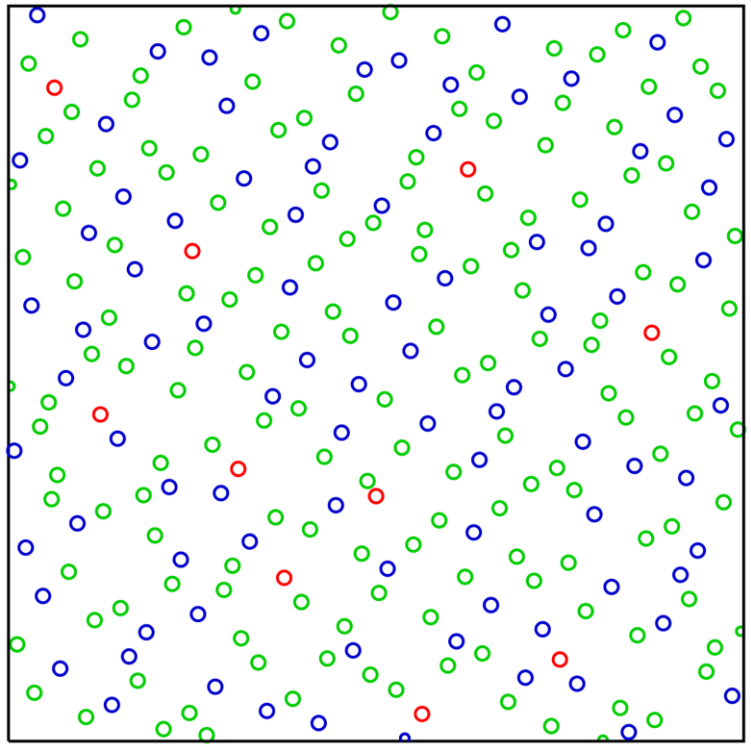


Task – Implement other ‘noise’ generators and compare

- Add a window region for rendering
- Add other ‘random’ number generators
  - Halton23
  - Stratified
  - Blue Noise
- Use instead of `drand48()`
- Compute a high-quality image for comparison (very large number of samples)
- Evaluate the other generators using lower sample counts with respect to the high-quality image (e.g. visual, difference, snr, run-time)

Halton Sampling

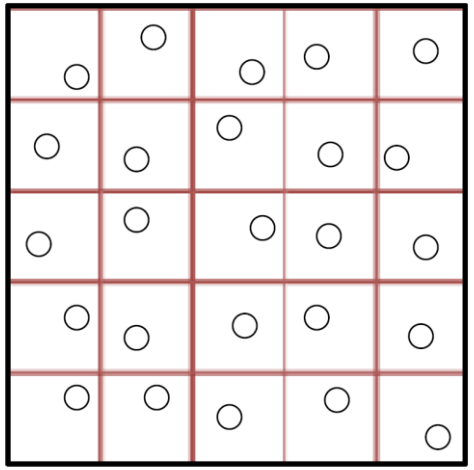
```
double halton(int index, int base) {
    double fraction = 1.0;
    double result = 0.0;

    while (index > 0) {
        fraction /= base;
        result += fraction * (index % base);
        index = floor(index / base);
    }
    return result;
}
```



Halton 2-3 Sampling

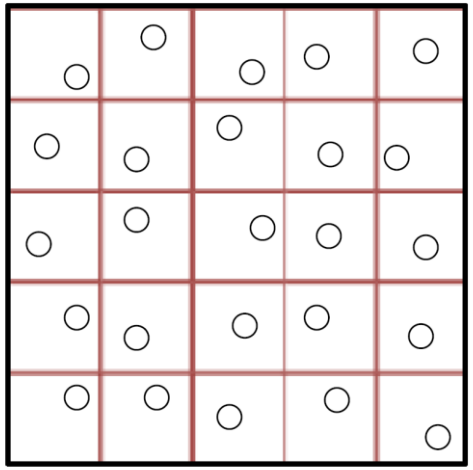
Stratified Sampling

```
const int num_intervals_stratified = 100;
double stratified_sampling_arr[num_intervals_stratified];
int used_samples = num_intervals_stratified;

void populate_arr() {
    for (int i = 0; i < num_intervals_stratified; i++) {
        double lower_bound = ((double) i) / ((double) num_intervals_stratified);
        double upper_bound = ((double) (i + 1)) / ((double) num_intervals_stratified);

        stratified_sampling_arr[i] = lower_bound + drand48() * (upper_bound - lower_bound);
    }
}
```



Stratified Sampling

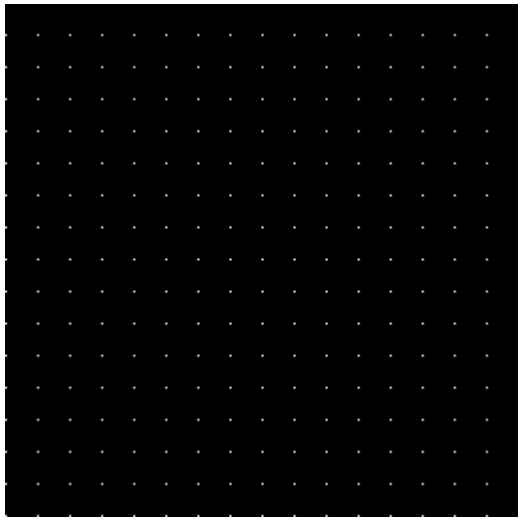
```
double stratified() {
    if (used_samples == num_intervals_stratified) {
        populate_arr();
        used_samples = 0;
    }

    int index;
    do {
        index = rand() % num_intervals_stratified;
    } while (stratified_sampling_arr[index] < 0.0);

    used_samples++;
    double rand_value = stratified_sampling_arr[index];
    stratified_sampling_arr[index] = -1.0;
    return rand_value;
}
```



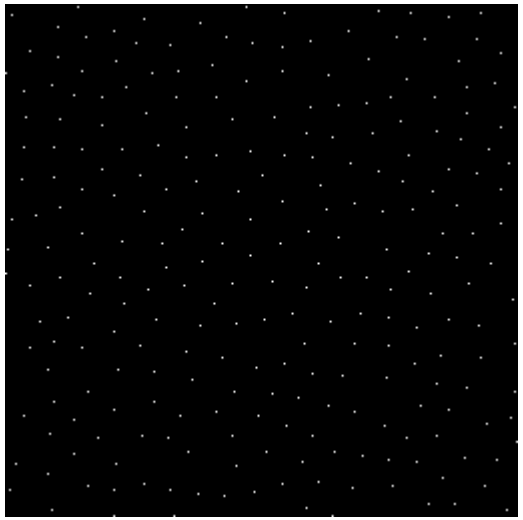
Blue Noise Sampling

- Is between regular sampling and white noise sampling
- Randomly placed points like white noise
- But approximately evenly spaced (more like regular sampling)
- Name since it contains higher amounts of higher frequencies and lower amounts of lower frequencies
  - Similar to blue light
    - Contains higher frequency (bluer) light

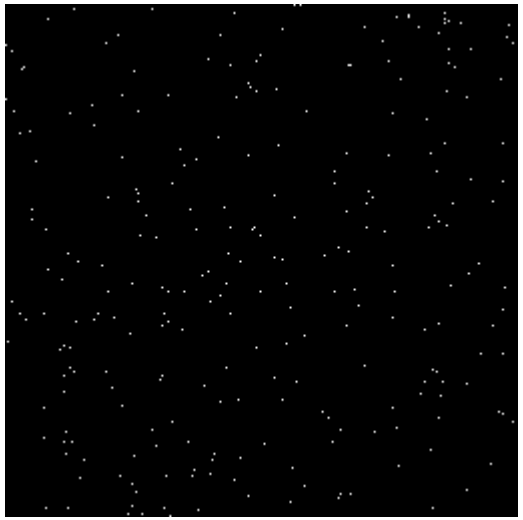
Blue Noise Sampling



Regular Sampling



Blue Noise Sampling



White Noise Sampling

Blue Noise Sampling

```
#include "lutLDBN.h"

std::vector<Point> blue_noise_samples;
const int num_blue_noise_samples = 1000000;

Tuple blue_noise() {
    int position = rand() % num_blue_noise_samples;
    Tuple rand_tuple = Tuple((blue_noise_samples.at(position))[0], (blue_noise_samples.at(position))[1]);

    return rand_tuple;
}

int main(int argc, char *argv[])
{
    // ...

    initSamplers();
    ldbnBNOT(num_blue_noise_samples, blue_noise_samples);

    // ...
}
```

Task – Implement other ‘noise’ generators and compare

- Add a window region for rendering
- Add other ‘random’ number generators
  - Halton23
  - Stratified
  - Blue Noise
- Use instead of `drand48()`
- Compute a high-quality image for comparison (very large number of samples)
- Evaluate the other generators using lower sample counts with respect to the high-quality image (e.g. visual, difference, snr, run-time)

Task – Implement other ‘noise’ generators and compare

- *Use instead of* `drand48()`

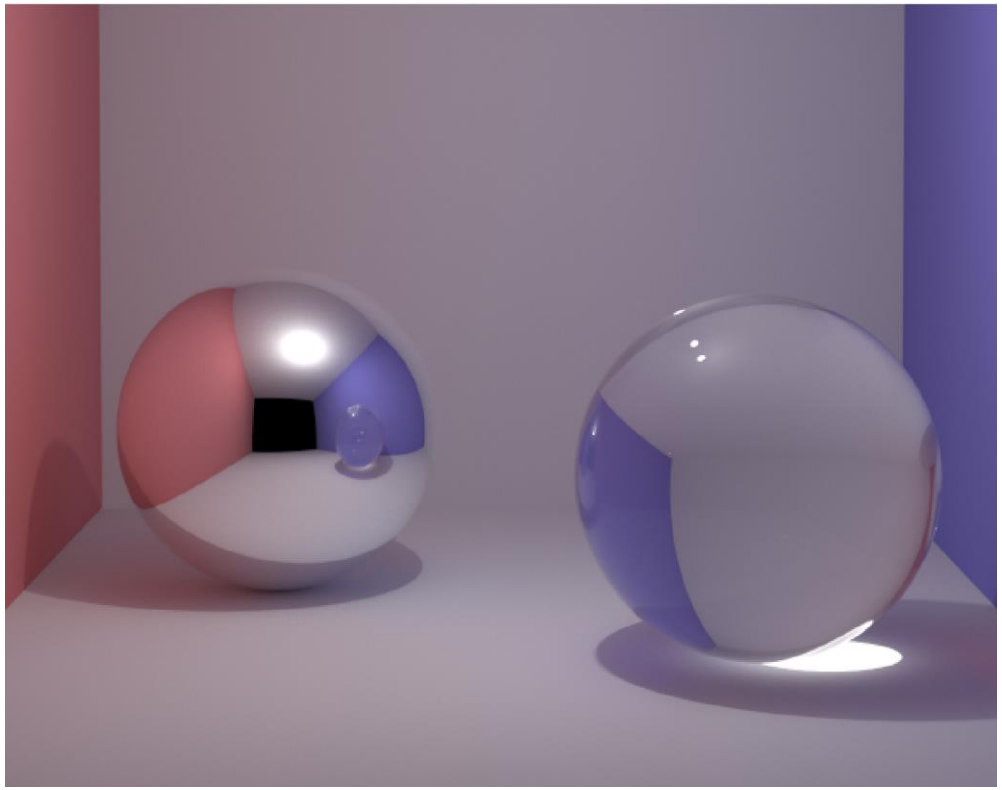
```
enum Rand_Gen_t { DRAND48, HALTON_17_19, STRATIFIED, BLUE_NOISE };
const Rand_Gen_t rand_generator_type = BLUE_NOISE;

int halton_seed = 0;

Tuple random_generator() {
    if (rand_generator_type == DRAND48) {
        return Tuple(drand48(), drand48());
    } else if (rand_generator_type == HALTON_17_19) {
        halton_seed++;
        return Tuple(halton(halton_seed, 17), halton(halton_seed, 19));
    } else if (rand_generator_type == STRATIFIED) {
        return Tuple(stratified(), stratified());
    } else if (rand_generator_type == BLUE_NOISE) {
        return blue_noise();
    }
}
```

Task – Implement other ‘noise’ generators and compare

- Add a window region for rendering
- Add other ‘random’ number generators
  - Halton23
  - Stratified
  - Blue Noise
- Use instead of `drand48()`
- Compute a high-quality image for comparison (very large number of samples)
- Evaluate the other generators using lower sample counts with respect to the high-quality image (e.g. visual, difference, snr, run-time)

Task – Implement other ‘noise’ generators and compare

- *Compute a high-quality image for comparison (very large number of samples)*
- 138k samples
- 3.93 days of rendering



Task – Implement other ‘noise’ generators and compare

- Add a window region for rendering
- Add other ‘random’ number generators
  - Halton23
  - Stratified
  - Blue Noise
- Use instead of `drand48()`
- Compute a high-quality image for comparison (very large number of samples)
- Evaluate the other generators using lower sample counts with respect to the high-quality image (e.g. visual, difference, snr, run-time)

Task – Implement other ‘noise’ generators and compare

- *Evaluate the other generators using lower sample counts with respect to the high-quality image (e.g. visual, difference, snr, run-time)*
- Visual difference:

```
Image diff(original.width, original.height);
for (int y = 0; y < original.height; y++){
    for (int x = 0; x < original.width; x++){
        Color a = original.getColor(x, y);
        Color b = compare.getColor(x, y);
        diff.setColor(x, y, sub_abs(a, b));
    }
}
```

Task – Implement other ‘noise’ generators and compare

- *Evaluate the other generators using lower sample counts with respect to the high-quality image (e.g. visual, difference, snr, run-time)*
- Peak-Signal-to-Noise Ratio (PSNR):
  - Ratio of the peak signal value to the mean squared error of image

$$20 \log_{10} \frac{\max(f)^2}{\sqrt{MSE}}$$

```
double computePSNR(const Image& original, const Image& compare){
    int numPixels = original.height * original.width;
    double mse = 0.;

    for (int i = 0; i < numPixels; i++){
        Color orig = original.pixels[i];
        Color comp = compare.pixels[i];
        double distSquare = pow((orig.x - comp.x), 2)
                          + pow((orig.y - comp.y), 2)
                          + pow((orig.z - comp.z), 2);
        mse += distSquare;
    }
    mse /= numPixels;

    return 20 * log10(pow(255.0, 2) / mse);
}
```

Task – Implement other ‘noise’ generators and compare

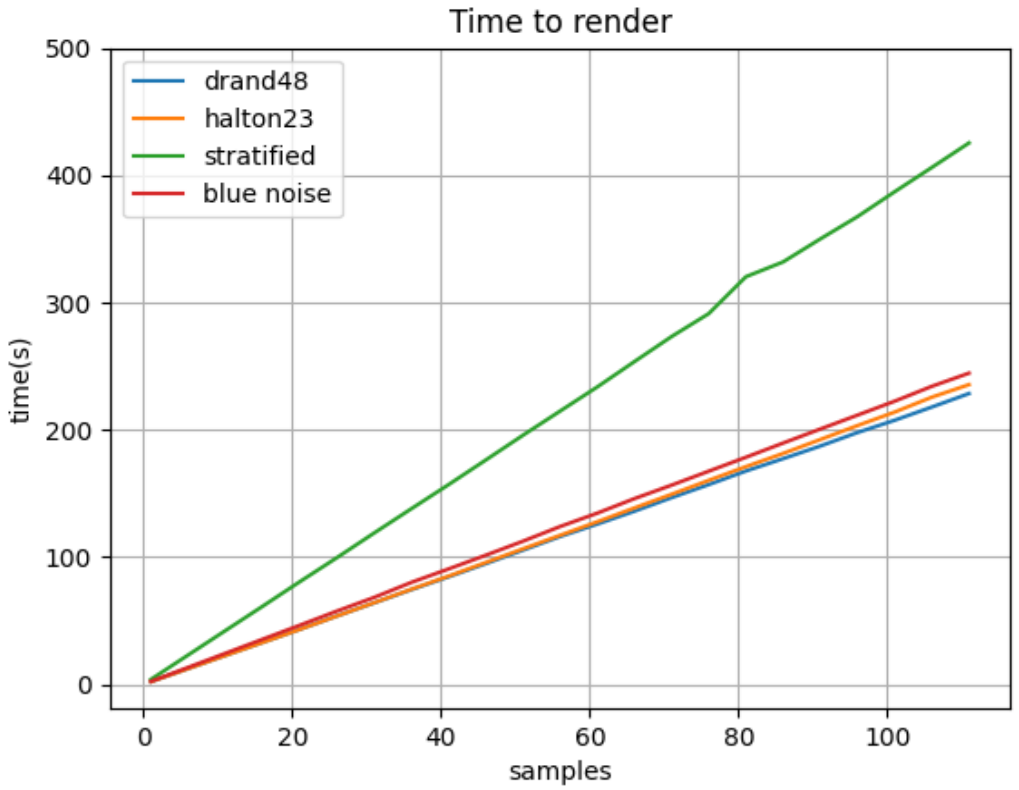
- *Evaluate the other generators using lower sample counts with respect to the high-quality image (e.g. visual, difference, snr, run-time)*
- Runtime:
  - /usr/bin/time to measure time of execution (in seconds)

Python script to collect data

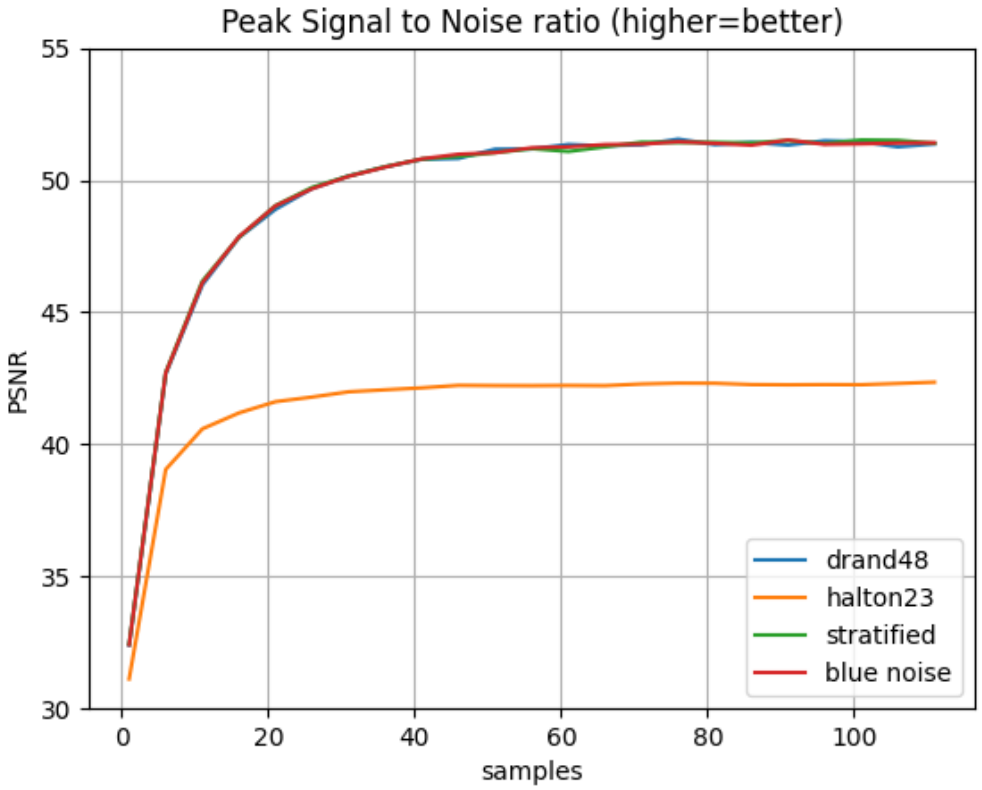
```python
def render_image(samples : int, randomizer : int):
    time = .0
    for i in range(n):
        output = subprocess.run(
            f'/usr/bin/time -f "{{\\"wall\\": %e}}" ./bin/PathTracing {samples} {randomizer}',
            shell=True,
            stderr=subprocess.PIPE,
            stdout=subprocess.PIPE
        ).stderr.decode()
        time += json.loads(output)["wall"]
    time /= n
    print(time)

    return time

def calculate_psnr(samples :int, randomizer :int):
    ratio = subprocess.run(
        f'./bin/Image_diff {samples} {randomizer}',
        shell=True,
        stderr=subprocess.PIPE,
        stdout=subprocess.PIPE
    )

    return json.loads(ratio.stderr.decode())["psnr"]
```

Results

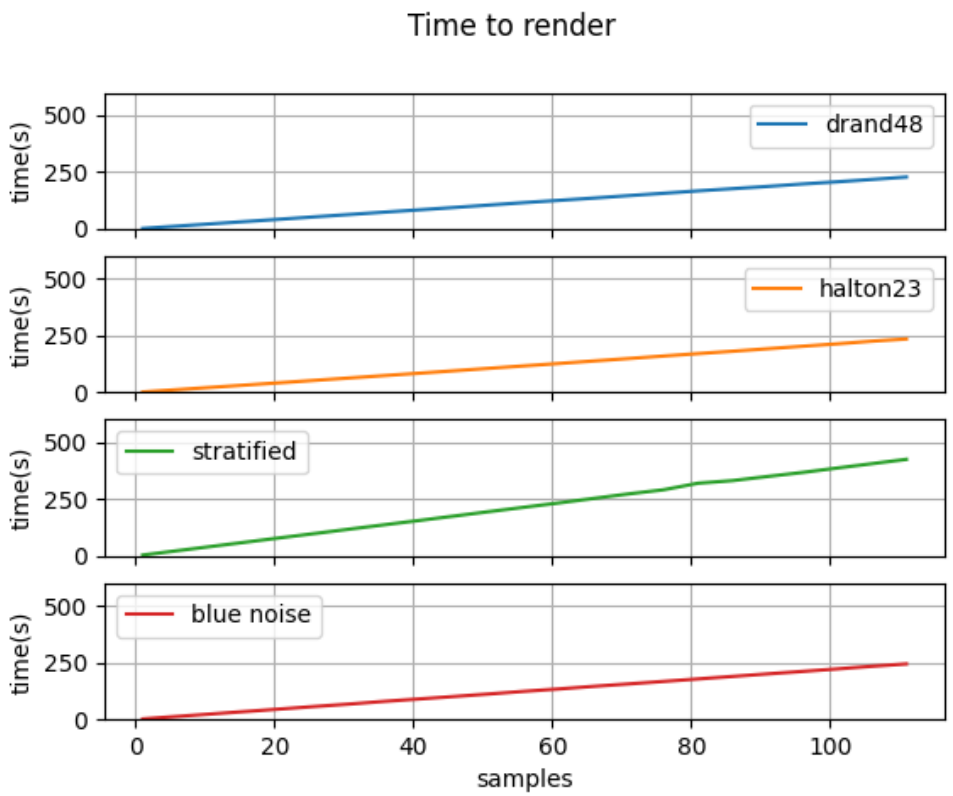
Time


Time to render
drand48
halton23
stratified
blue noise
time(s)
samples
0
100
200
300
400
500
0
20
40
60
80
100

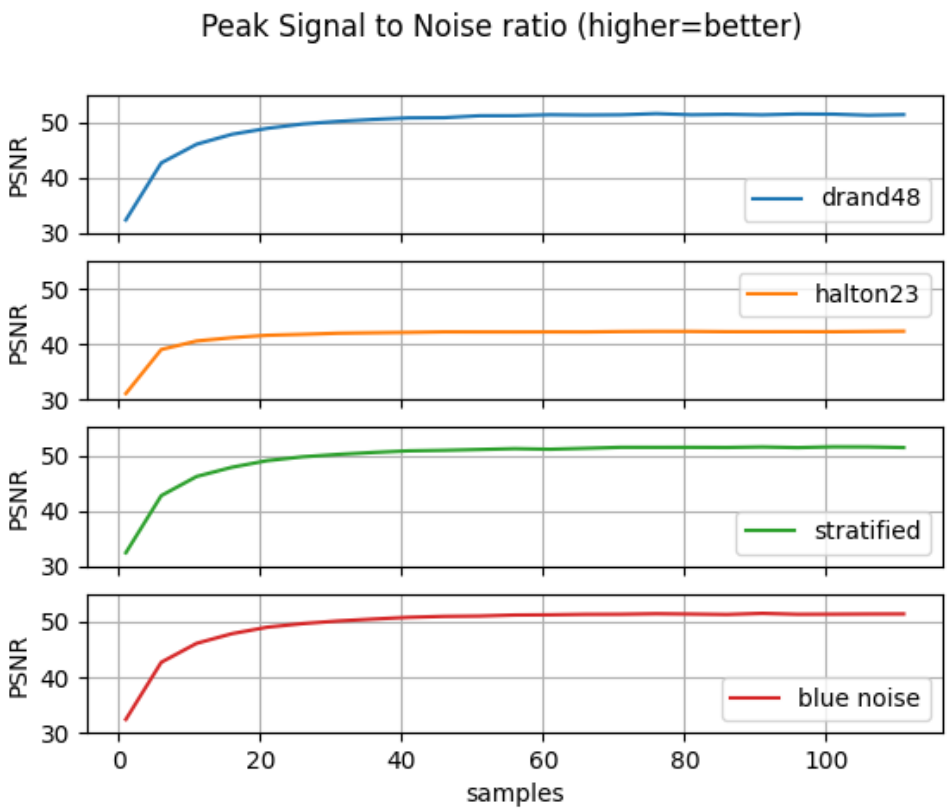
PSNR

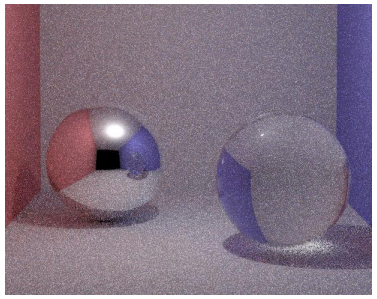
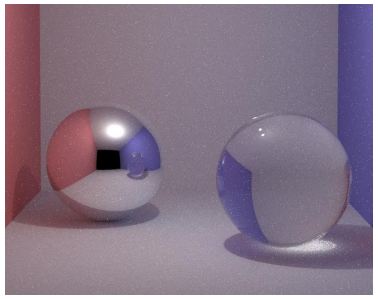
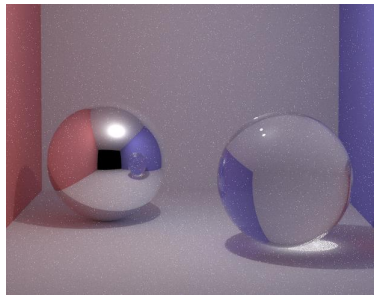
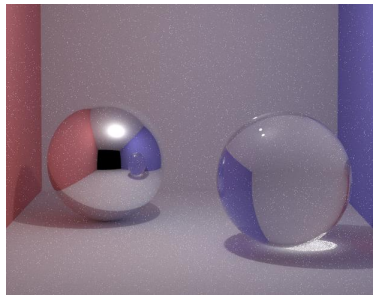
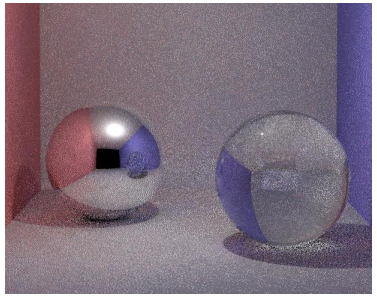
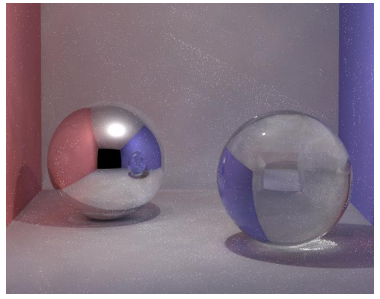
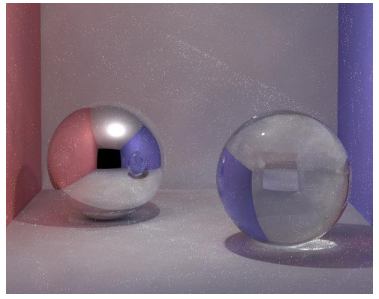
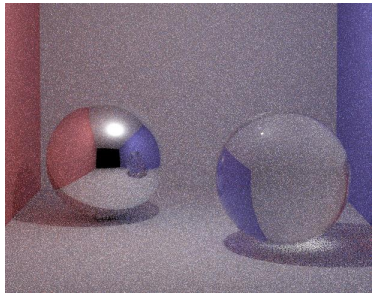
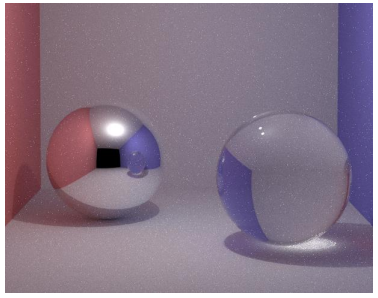
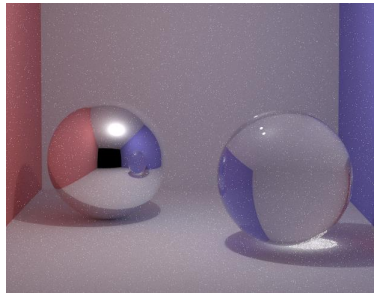
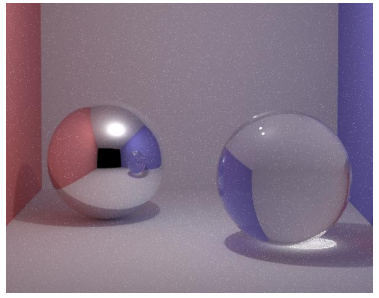
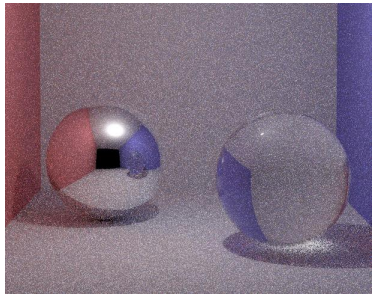
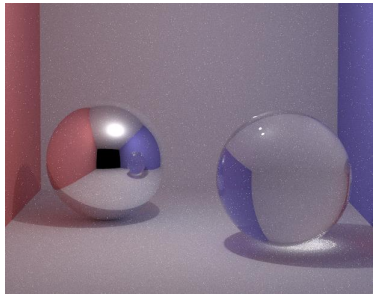
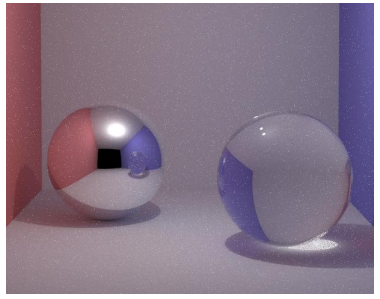
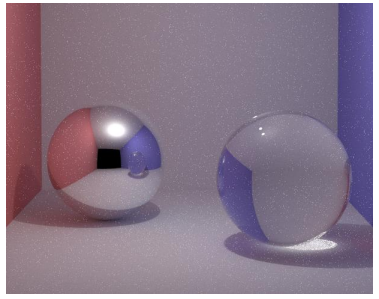

Peak Signal to Noise ratio (higher=better)
PSNR
samples
drand48
halton23
stratified
blue noise
30
35
40
45
50
55
0
20
40
60
80
100

Time


Time to render
time(s)
500
250
0
drand48
halton23
stratified
blue noise
0
20
40
60
80
100
samples

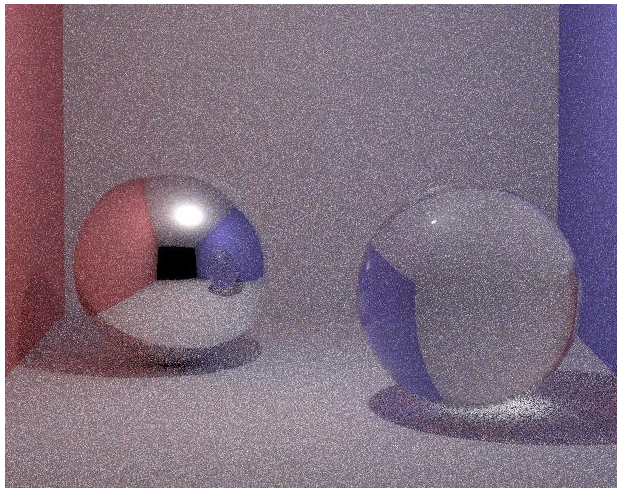
PSNR


Peak Signal to Noise ratio (higher=better)
PSNR
50
40
30
drand48
halton23
stratified
blue noise
0
20
40
60
80
100
samples

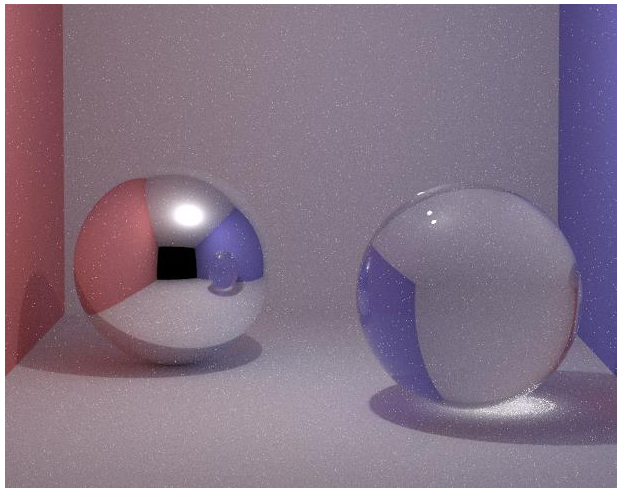
	4 spp	144 spp	304 spp	444 sp
drand48				
Halton 17/19				
Stratified				
Blue Noise				

drand48

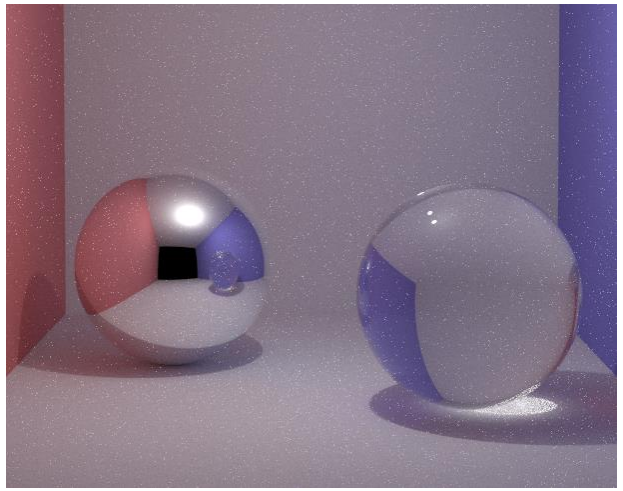
4 spp



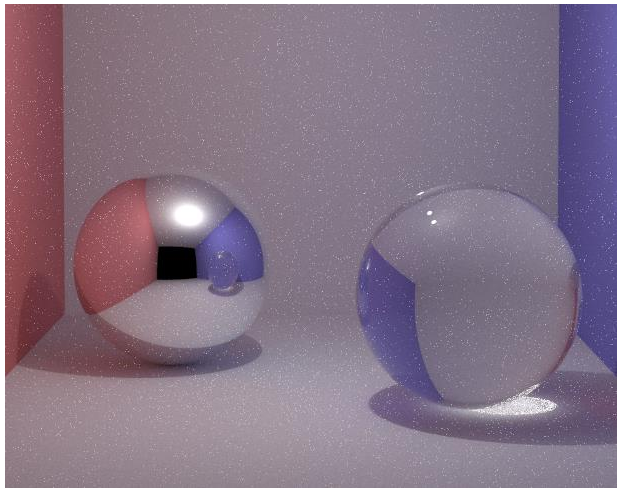
144 spp



304 spp

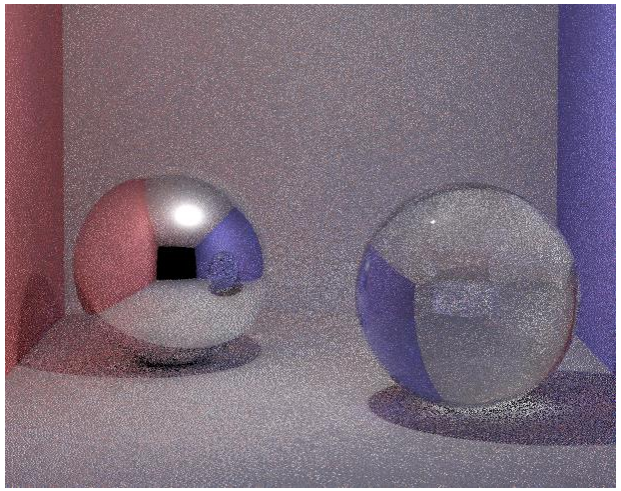


444 sp



Halton 17/19

4 spp



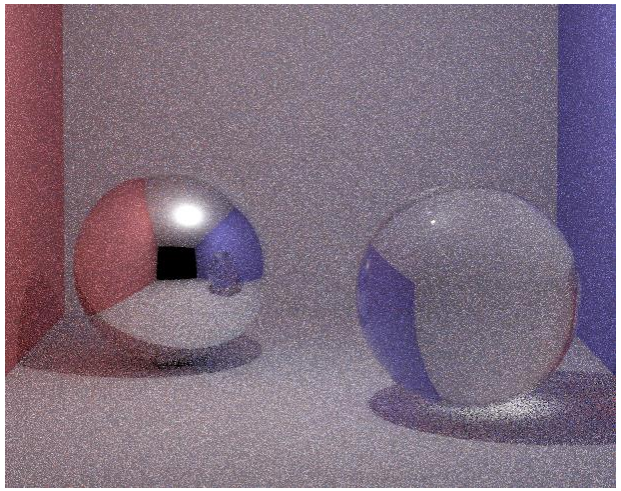
144 spp

304 spp

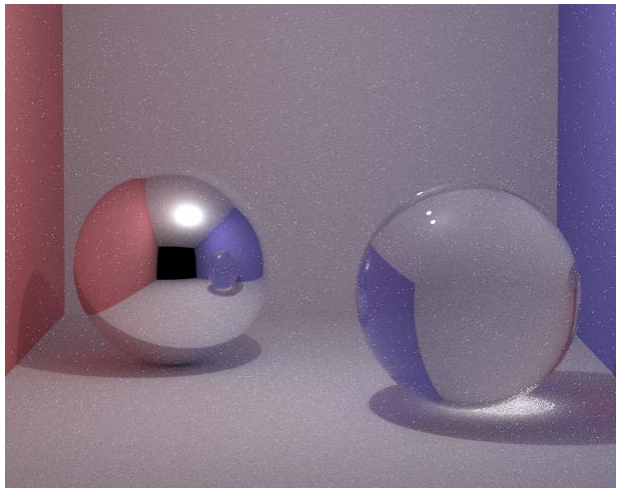
444 sp

Stratified

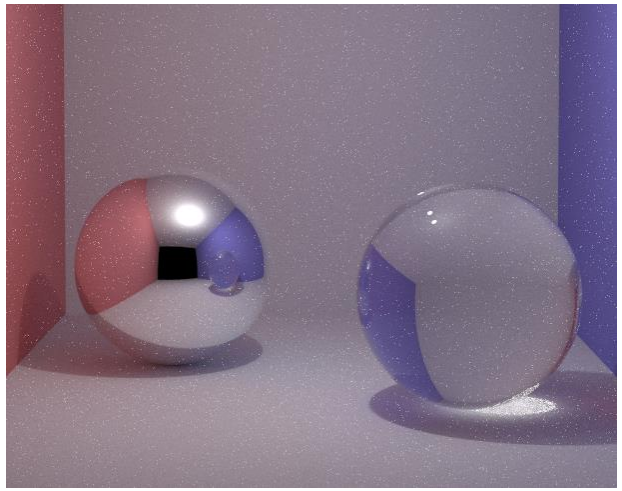
4 spp



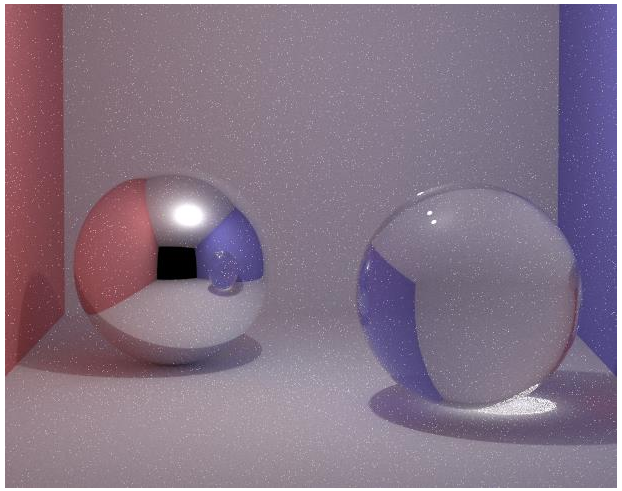
144 spp



304 spp

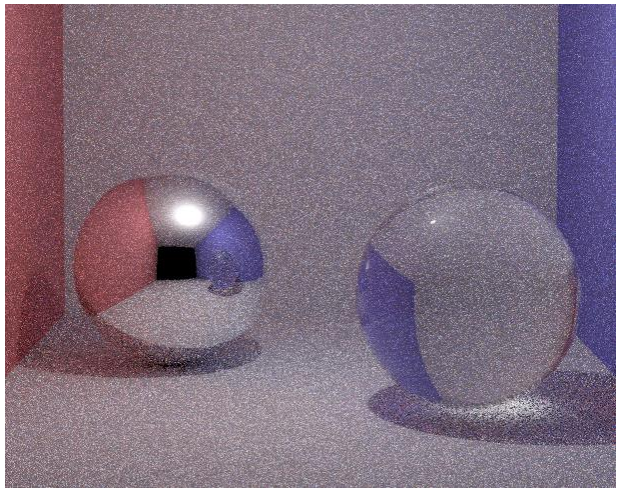


444 sp

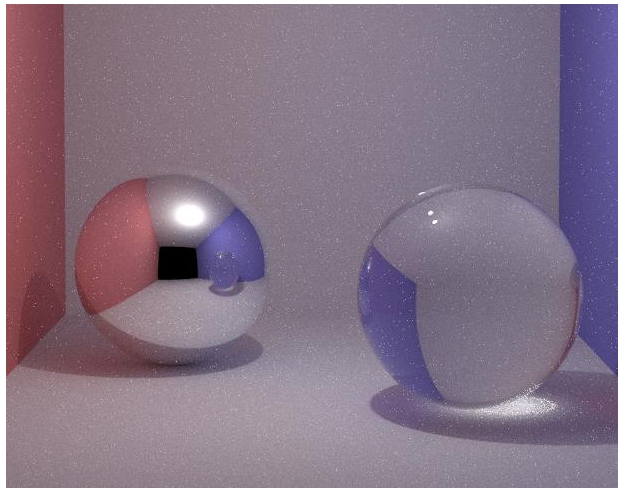


Blue Noise

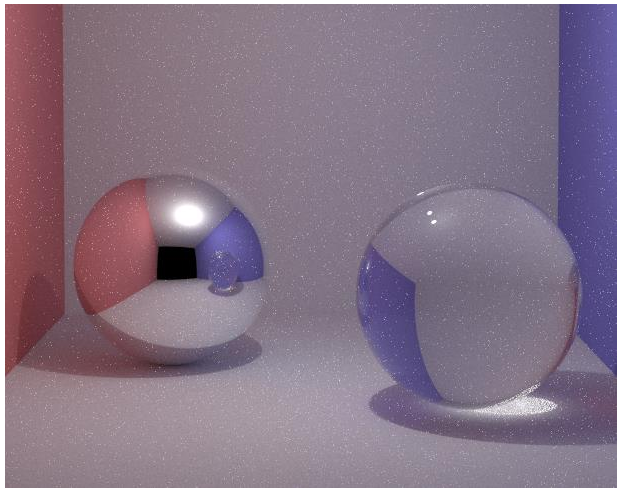
4 spp



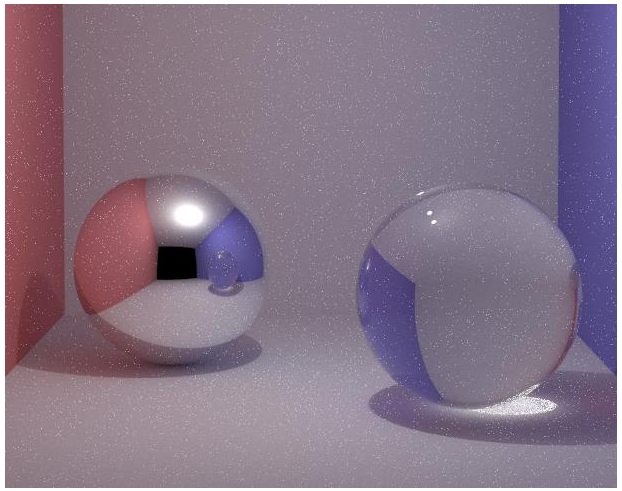
144 spp



304 spp

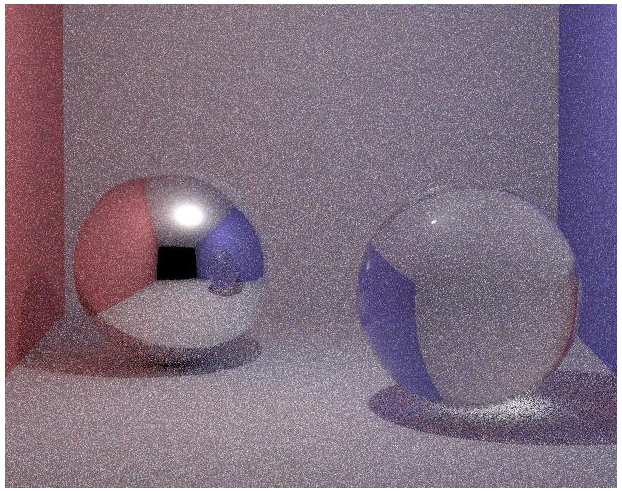


444 sp



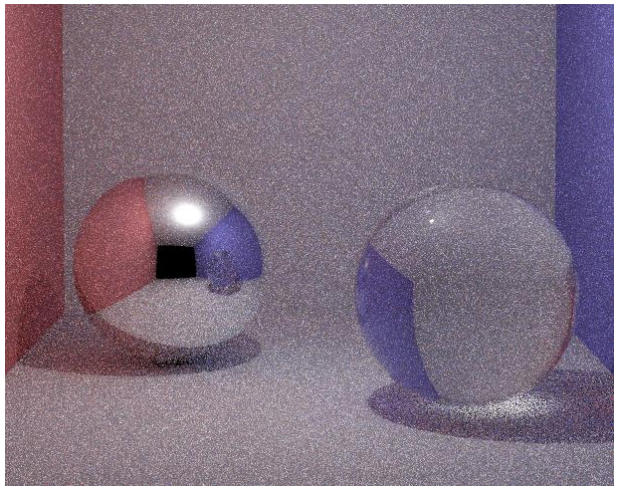
4 spp

drand48

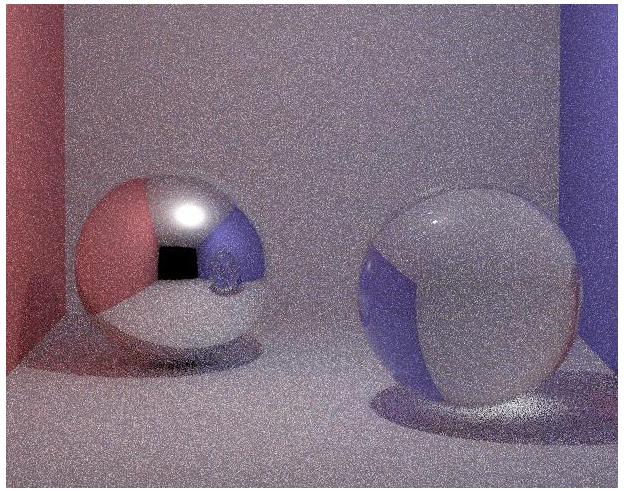


Halton

Stratified

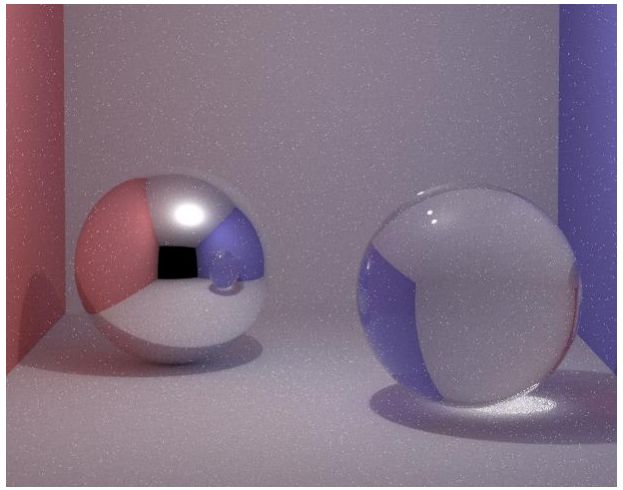


Blue Noise



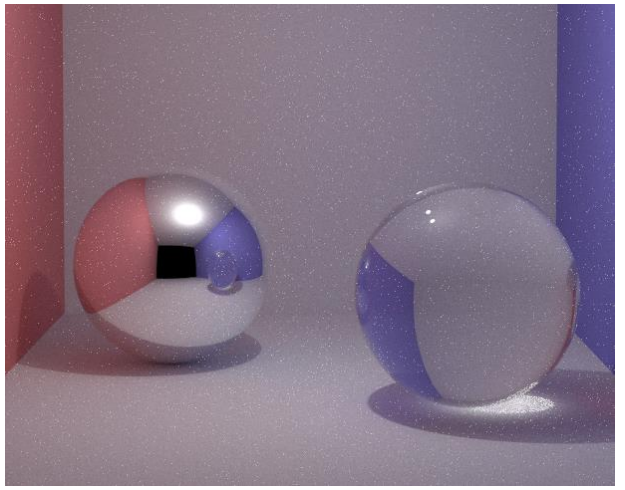
244 spp

drand48

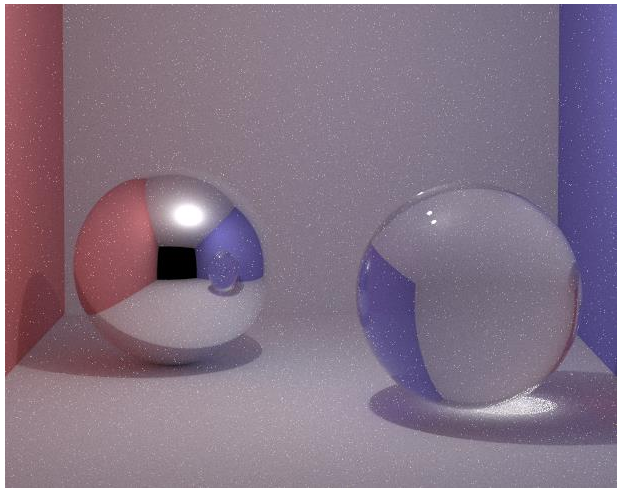


Halton

Stratified

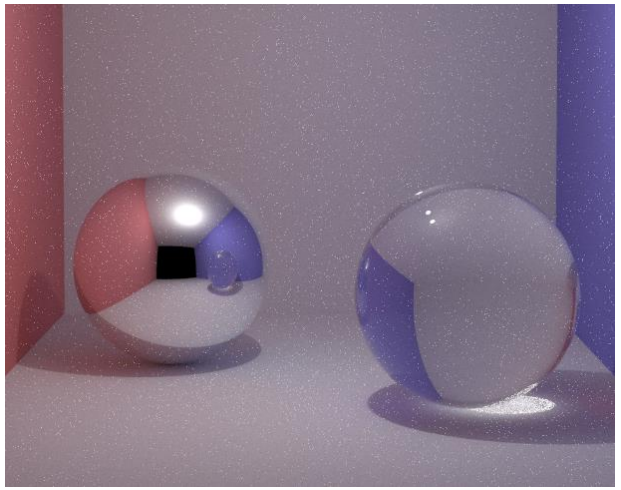


Blue Noise



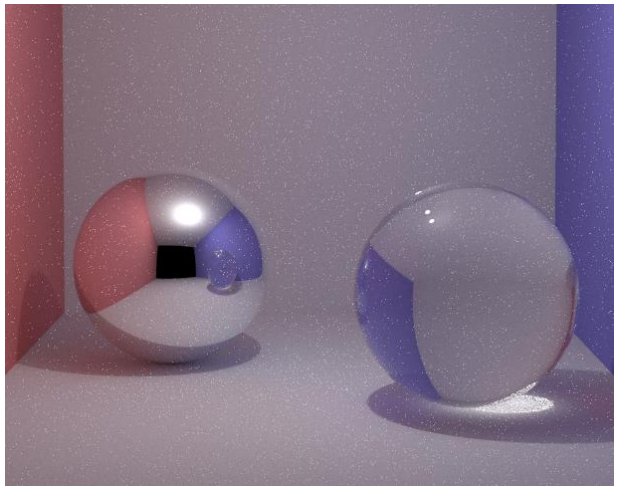
444 spp

drand48

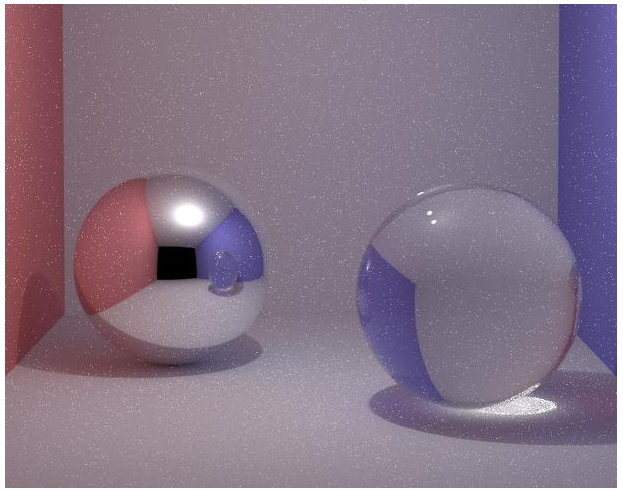


Halton

Stratified



Blue Noise



The beauty of wrong code