Global Illumination using Photon Maps

Team 32:
Fengshi Zheng
Hongyu He
Kehan Xu
Zijun Hui

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

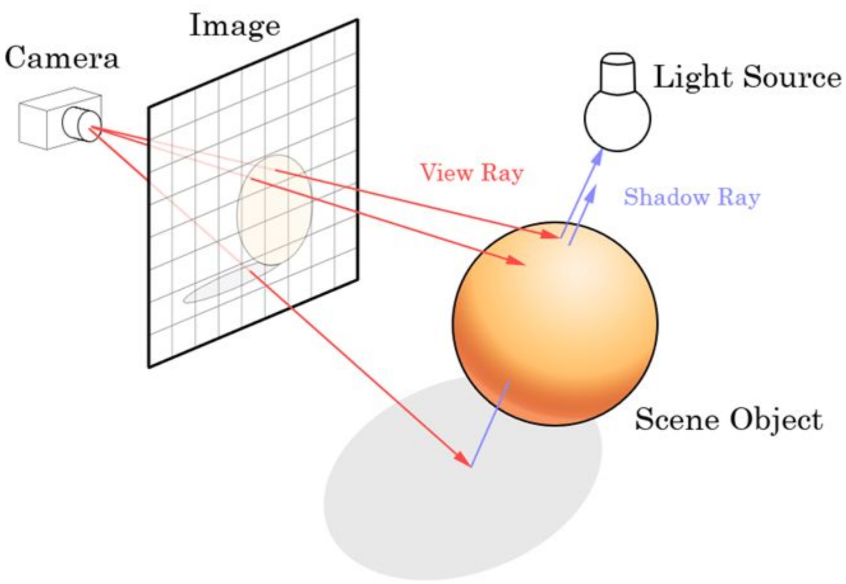
Presentation Overview

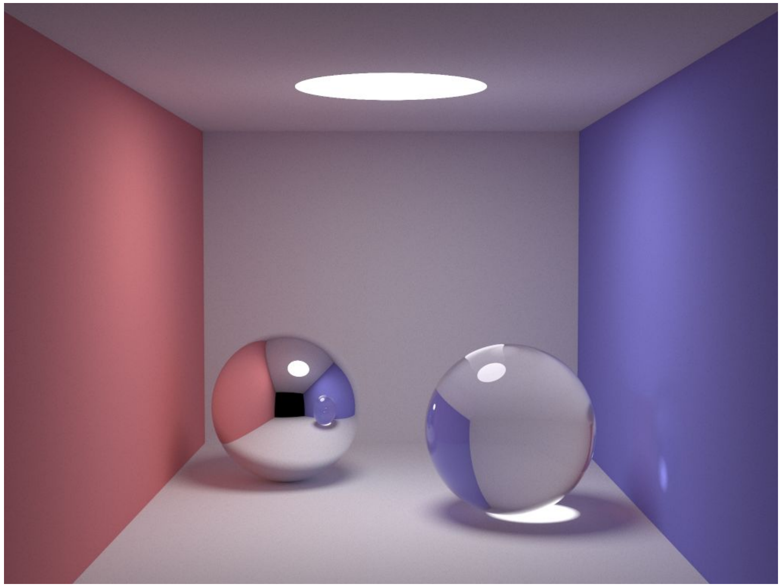
- **Introduction**
- **Challenges and Bottlenecks**
- **Optimizations**
- **Results**
- **Next Steps**

Introduction

Ray Tracing (Path Tracing)

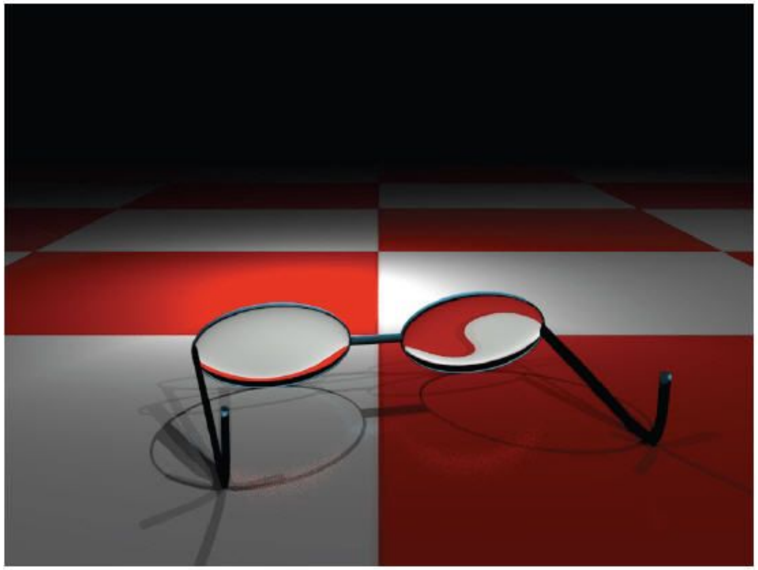
- The *de-facto* rendering algorithm widely used in film industry
- Cast rays from the camera and do multiple intersections with scene described by simple geometries (spheres, triangles)



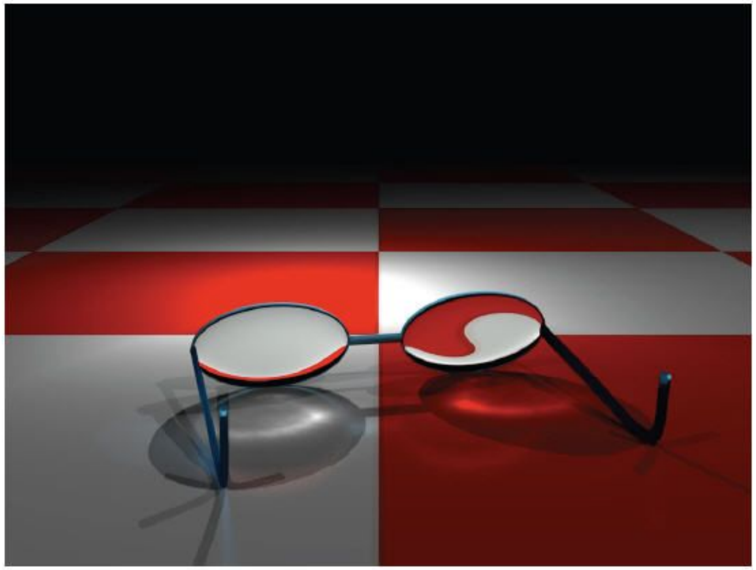


Limitations

- It is hard for path tracing to find light sources under particularly tricky lighting conditions.
- Effects caused by complex specular light paths are difficult to render, e.g., caustic.



Ray Tracing



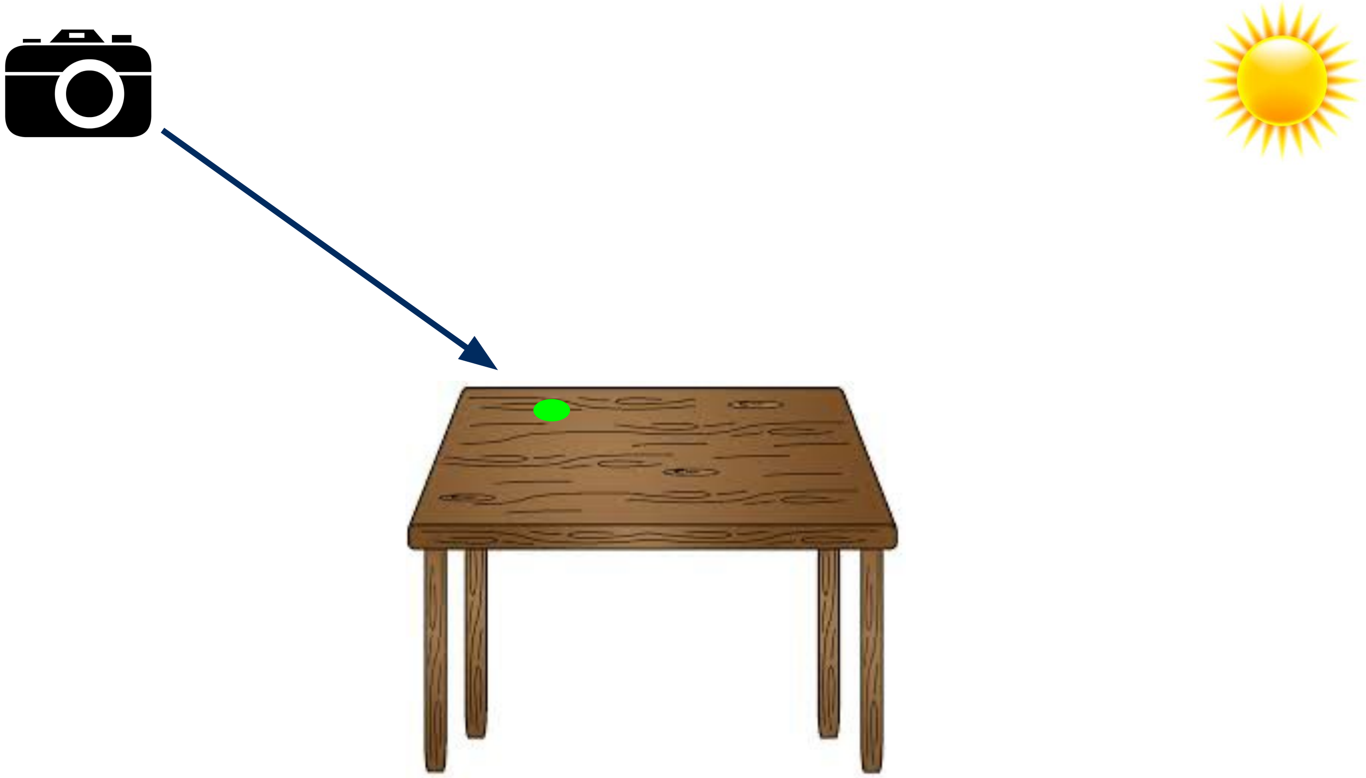
Ray Tracing w/ Photon Map

Improvement-Photon Mapping

General idea: make path tracing **bidirectional** (from camera & light source)

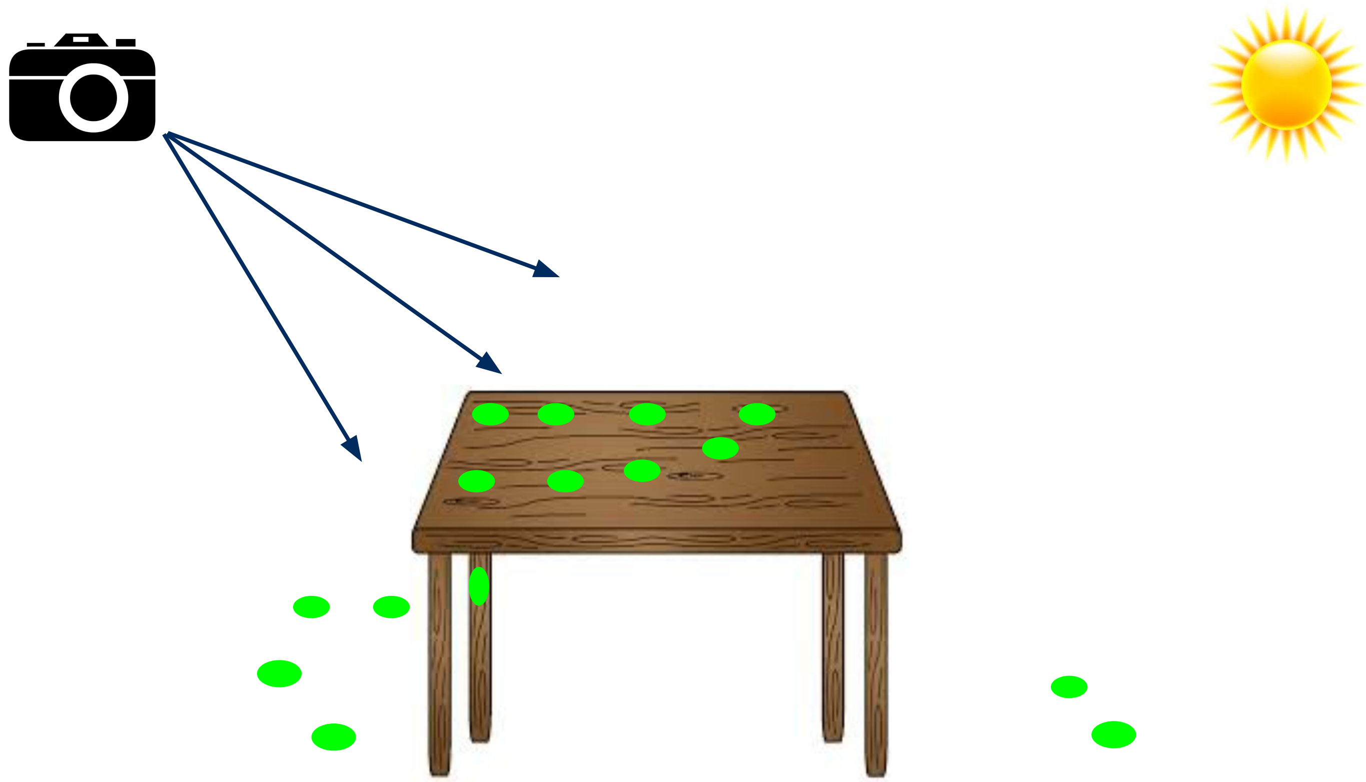
Photon Mapping

Camera pass: cast rays and store the **visible points**



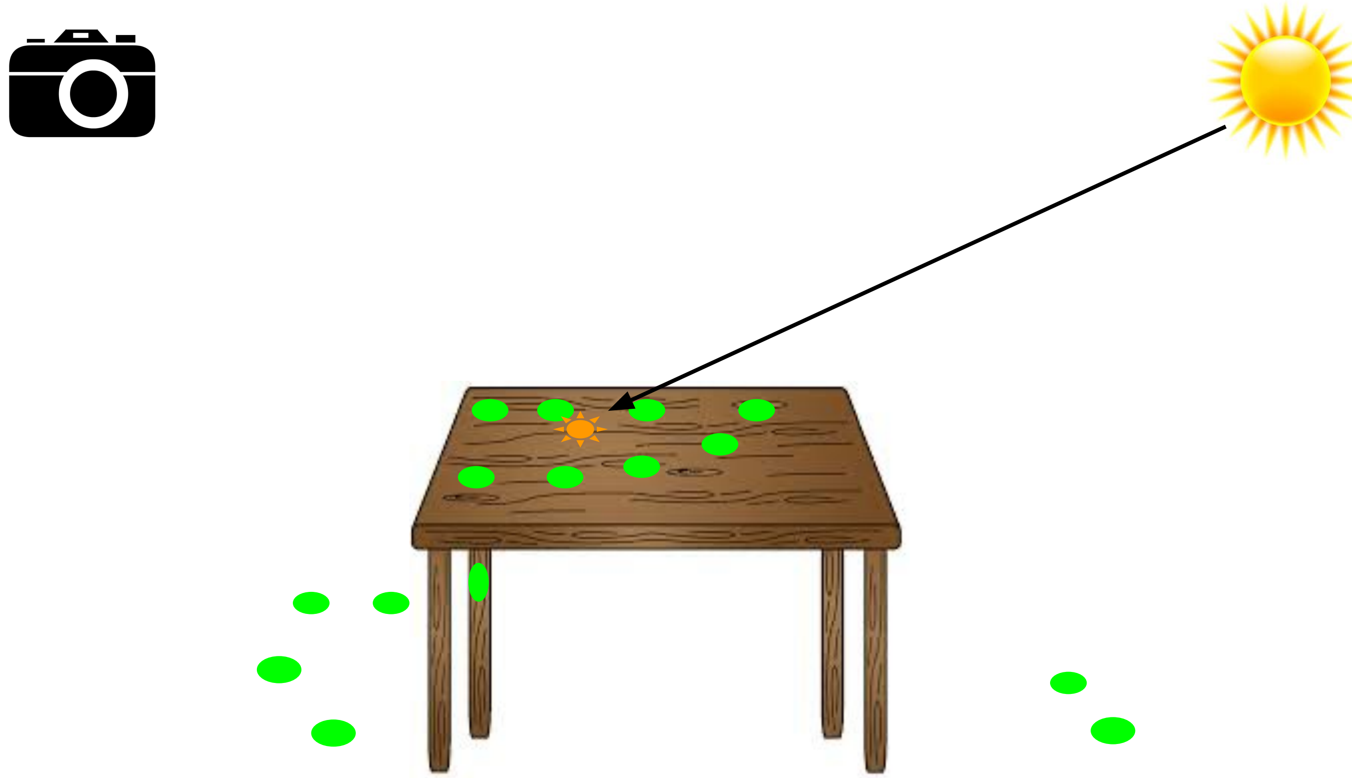
Photon Mapping

Camera pass: cast rays and store the **visible points**



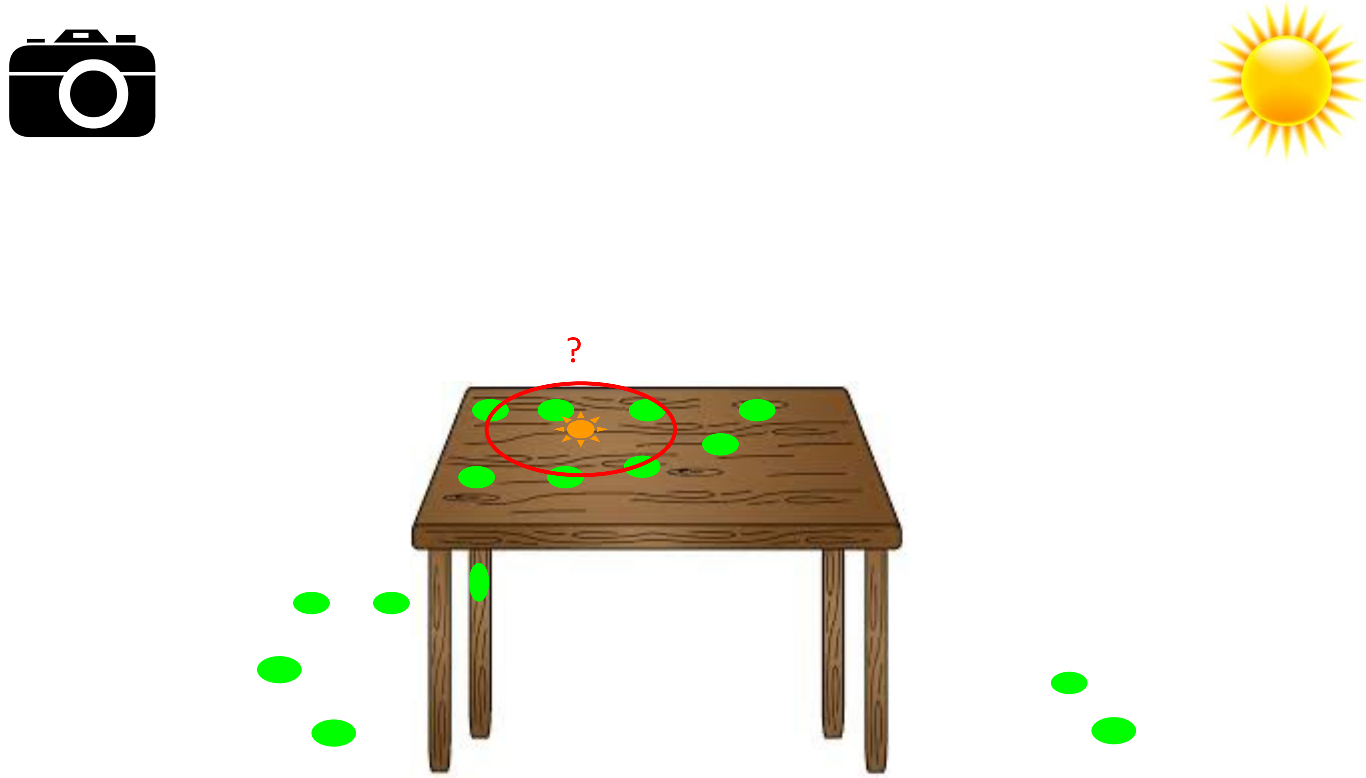
Photon Mapping

Photon pass: shoot photons from the light source



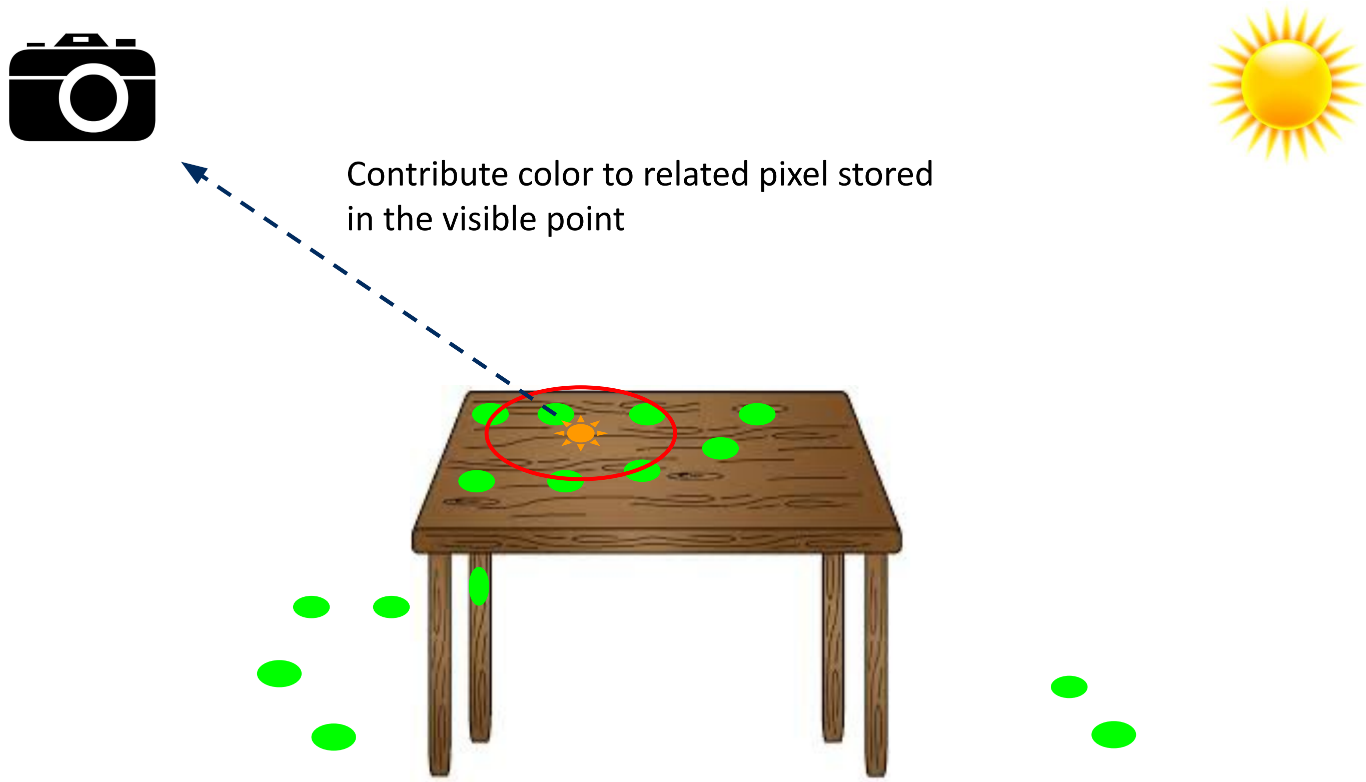
Photon Mapping

Photon pass: shoot photons from the light source and contribute energy to nearby visible points



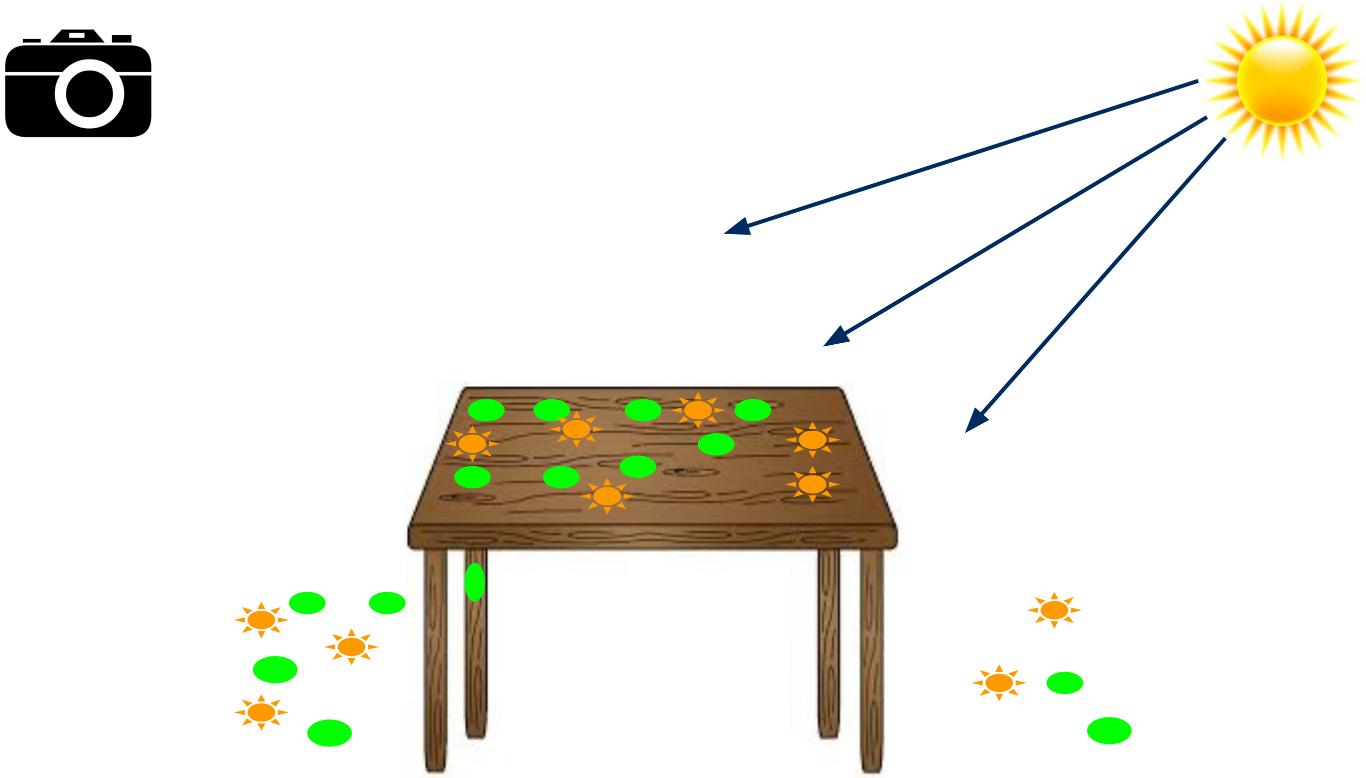
Photon Mapping

Photon pass: shoot photons from the light source and contribute energy to nearby visible points



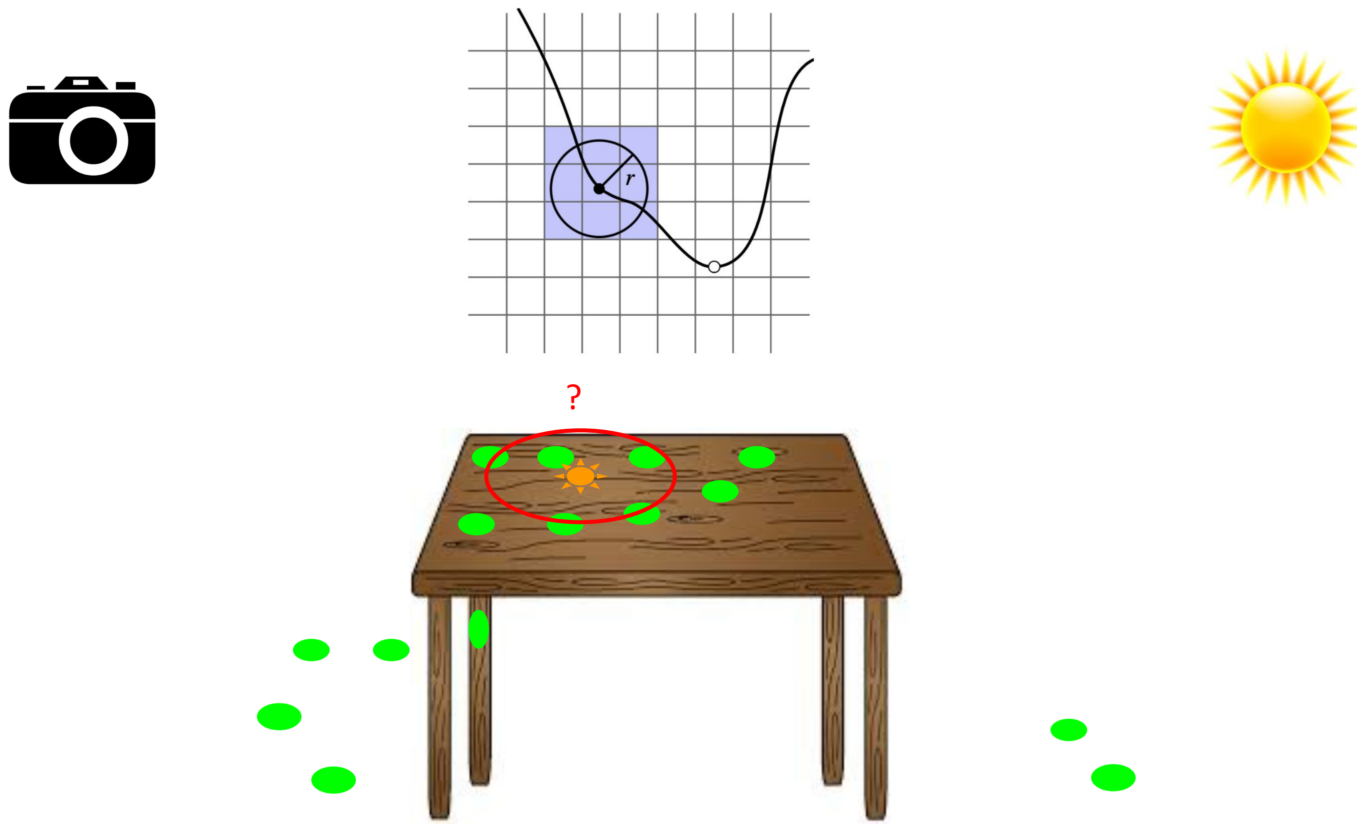
Photon Mapping

Repeat thousands to millions of times



Photon Mapping

Detail: we use a **spatial hashtable** to facilitate the neighboring query process



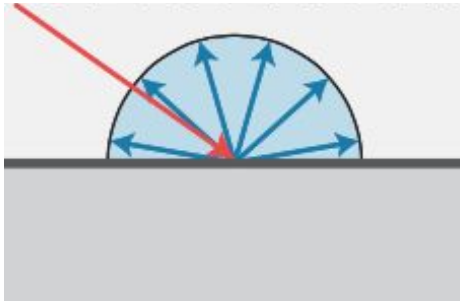
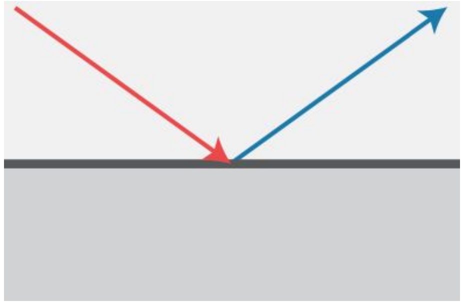
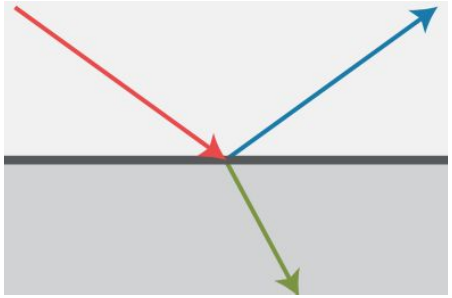
Algorithm

Algorithm 1: Stochastic Progressive Photon Mapping (SPPM)

```
InitializeDataStructure()
for each iteration do
    Function CameraPass() is
        // record the intersection point for each ray's first hit on a diffuse surface
        for each pixel do
            generateRay()
            for bounce time < max ray depth do
                intersectRayWithScene()
                if hit surface = DIFFUSE then
                    computeDirectLighting()
                    record intersection point
                    return
                end
                generateBouncedRay() // based on hit material
            end
        end
    end
    Function BuildLookUpTable() is
        // store all recorded intersection points according to spatial location
        for each pixel do
            findGridIndicesByHashing()
            putIntersectionPointIntoEachGrid()
        end
    end
    Function PhotonPass() is
        // emit photons, find nearby intersection points and add to its light value
        for each photon do
            emitPhotonFromLightSource()
            for bounce time < max ray depth do
                intersectRayWithScene()
                if bounce time ≥ 1 then
                    // only compute indirect lighting
                    findGridIndexByHashing()
                    for each intersection point in grid do
                        if is nearby point then
                            addToIntersectionPointLightValue()
                        end
                    end
                end
            end
        end
    end
    Consolidate()
end
StorePixelValue()
```

Implementation Constraints

- **Object: sphere only**
  - Wall and ground represented by sphere with large radius
- **Lighting: also sphere only**
  - As emission property of object
- **Support 3 basic types of material**
  - Diffuse, specular, dielectric
- **Everything stored in floats and ints**

Challenges

- Implementing a rendering system with ***complex data structures*** from scratch
  - Vector algebra library
  - Spatial hashtable
  - Support data structures for meshes, intersections, etc.
- Complex numerical algorithm with ***stochastic features***
- Inherently an algorithm with ***memory bottleneck***

Challenges

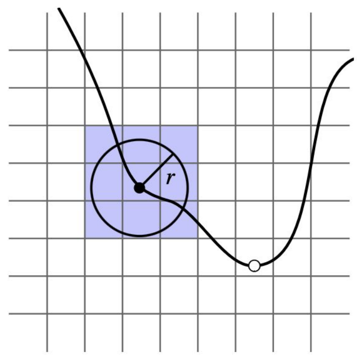
- Implementing a rendering system with ***complex data structures*** from scratch
- Complex numerical algorithm with ***stochastic features***
  - Impossible to count flops and examine cache misses analytically
  - Adjacent rays have varying behavior, making it hard to exploit locality
    - *hit on different material leads to differing bouncing logic, lots of if statements*
  - Validation needs further considerations
    - *fix random seed to generate exactly the same pixels, better error metric than MSE, etc.*
- Inherently an algorithm with ***memory bottleneck***

Challenges

- Implementing a rendering system with ***complex data structures*** from scratch
- Complex numerical algorithm with ***stochastic features***
- Inherently an algorithm with ***memory bottleneck***
  - Large hash table with entries that reference different memory locations
  - Need careful consideration of data structure layouts and access patterns

Bottlenecks

- Building and looking up hashtable data structure

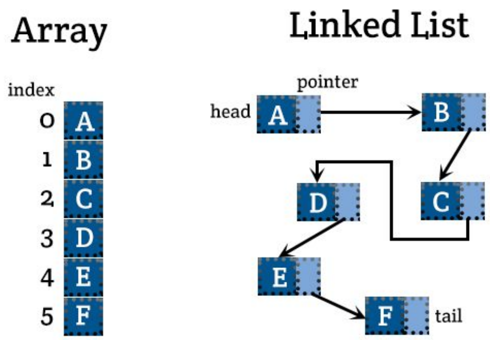


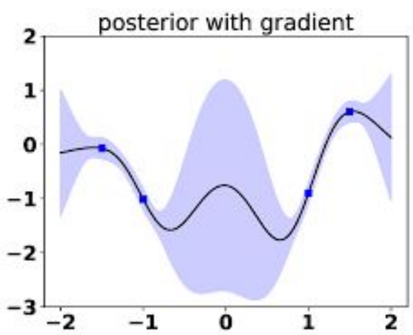
- Ray intersection with the scene (~10%)

- Vector math computations

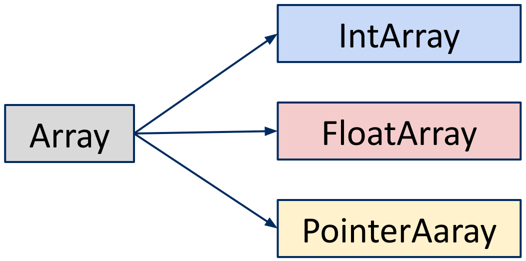
Optimization: Stage 1

- Inline vector math function
  - Remove procedural calls
- Improve spatial hashtable
  - Use array instead of linked-list, enhance spatial locality
  - Allocated space once at the start instead of reallocating for each iteration
- Apply Bayesian optimization to find suitable radius parameter
  - As an auto-tuning infrastructure
  - Radius determines spatial data structure resolution





Optimization: Stage 2



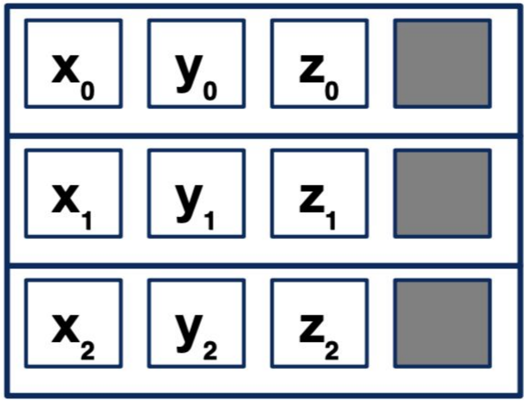
- Type-specific array
  - Type: Int, Float, Pointer, Vector
  - The general array structure stores pointers to the target data
  - Reduce memcpy overhead
- Avoid unnecessary branching
  - Same if statement may appear in consecutive for loops
  - Store if statement results to circumvent redundant computation
- Inline several short but frequently-called functions
  - e.g. functions for array abstraction, ray generation, etc.

Optimization: Stage 3

- Replace built-in rand() with self-implemented RNG to gain speed up
  - Implemented **xorshift** generators
  - Non cryptographically secure but extremely efficient
  - Less than 10 cycles per random number
- Used **SIMD** to implement xorshift
  - Generates 8 32-bit numbers per SIMD __m256i register
  - Store in a pool, take from the pool when needed
  - Unrolled generation loop 8 times (total of 64 floats per generation)

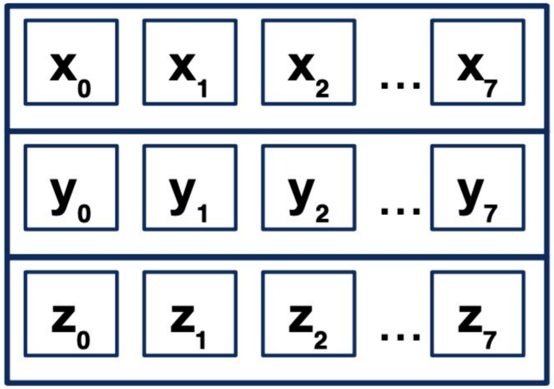
Optimization: Failed

- First attempt on vectorization: **horizontal layout**
- Each vector represented by one __m128
  - Vectorisation implemented on vector math operations only
  - Straightforward, ease of implementation
  - Requires constant loads/stores
  - Problematic for flattening operations: dot, cross, norm, …
  - 15% speed up on ARM Mac, but speed reduction on Skylake



Optimization: Stage 4

- Second attempt on vectorization: **vertical layout**
- Change array of structures (AoS) into structure of arrays (**SoA**)
  - Exploit more memory locality for sequential operations
  - Ease of loading for vectorisation
- Used **SIMD** to implement the entire rendering algorithm
- Able to process **8 rays** at the same time
- Naive implementation slower than scalar version of code



Optimization: Stage 4

- Slowdown due to visible points traversal
- Many gather and scatter operations
- Gathered values are non-contiguous
  - This triggers many cache misses
  - Exacerbated by structure of array layout
    - *e.g. when 16 elements are read, potentially 16 cache misses!*
- Scattering values are equally slow

Camera pass	0.056631 sec
Build lookup	0.020133 sec
Photon pass	1.728950 sec
Consolidate	0.002215 sec

Camera pass	0.016865 sec
Build lookup	0.016023 sec
Photon pass	2.655217 sec
Consolidate	0.000722 sec

Individual passes before and after optimisation

Optimization: Stage 5

- Gathered elements transposed into AoS before use
  - With 16 elements gathered, reduces cache misses by a factor of 16!
  - Read from hash table entries
- Scattered elements stored as array of structures directly
  - With 4 elements scattered, can have only a quarter of cache misses
  - Possibly due to only 1 sequential access to the underlying data
- Introduce small overhead when transforming SoA to AoS
  - Require **transpose** operation on input & output
  - Input: 8*8 transpose twice for 16 elements
  - Output: 4*8 transpose for 4 floats
  - Performance gain worth the overhead

Optimization: Failed

- Attempted to reduce memory transfer further by sorting the hash table by mesh type
  - Number of meshes is typically small
  - When inserting into the hash table, insert into different bins
- Together with precomputation of variables, only need to gather 8 floats
  - Same number of cache misses, but fewer loads
- Required looping over all possible meshes and checking each bin
- Ended up 10% slower
  - Increased malloc requirements
  - Increased non-SIMD computations
  - Increased loop iterations

Optimization: Stage 6 (ongoing)

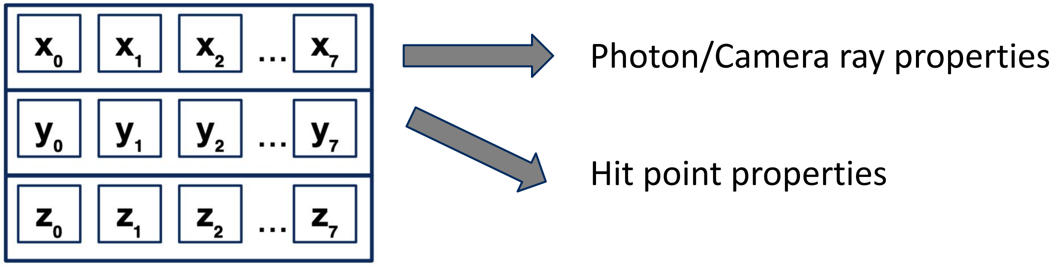
- Precomputed common variables to remove memory overhead
  - Since hotspot is the gather and scatter operations, trades reduced memory store/load for increased computational complexity
- Reduce aggressiveness of joins
- Common variable extraction
- …

Optimization: Stage 7 (ongoing)

- Precomputed come variables to remove memory overhead
  - Since hotspot is the gather and scatter operations, trades reduced memory store/load for increased computational complexity
- Common variable extraction
- …

Optimization: Stage 5 (Cont.)

- Shoot rays from camera & photons from light source
  - **8 rays** intersect with the scene and bounce at the same time
  - require to vectorize all involved operations
    - *intersection, bouncing direction, lighting computation*
  - different material properties lead to different bouncing and termination logic
  - perform masking to deal with rays terminated in the middle of iteration
  - check if all rays are not moving forward for early termination
- Store & retrieve information from hashtable data structure
  - compute target grid locations for **8 ray hit points** at a time

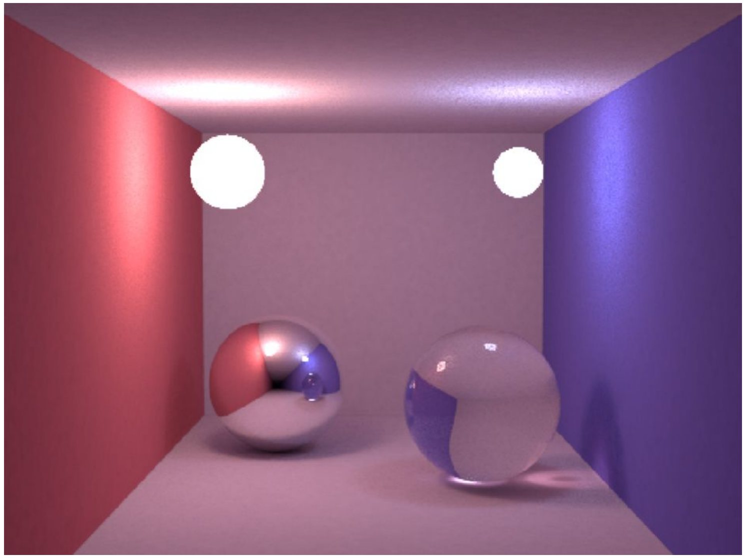


Optimizations

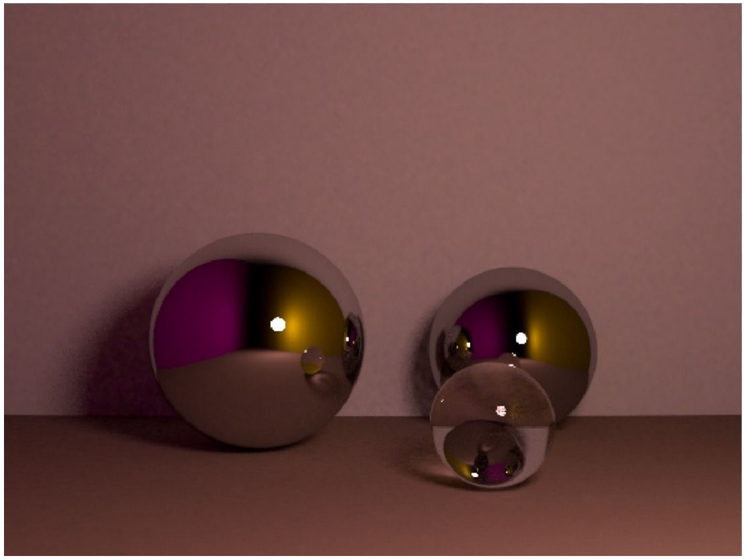
1. Inlining vector math function calls
2. Improve spatial hashtable: using array instead of linked-list
3. Speed up random float generation using SIMD
4. Use structure of arrays (**SoA**) instead of array of structures (AoS) to exploit more memory locality
5. Vectorize vector math library using SIMD
6. Refactor rendering algorithm for employing SIMD

Test Scenes (1)

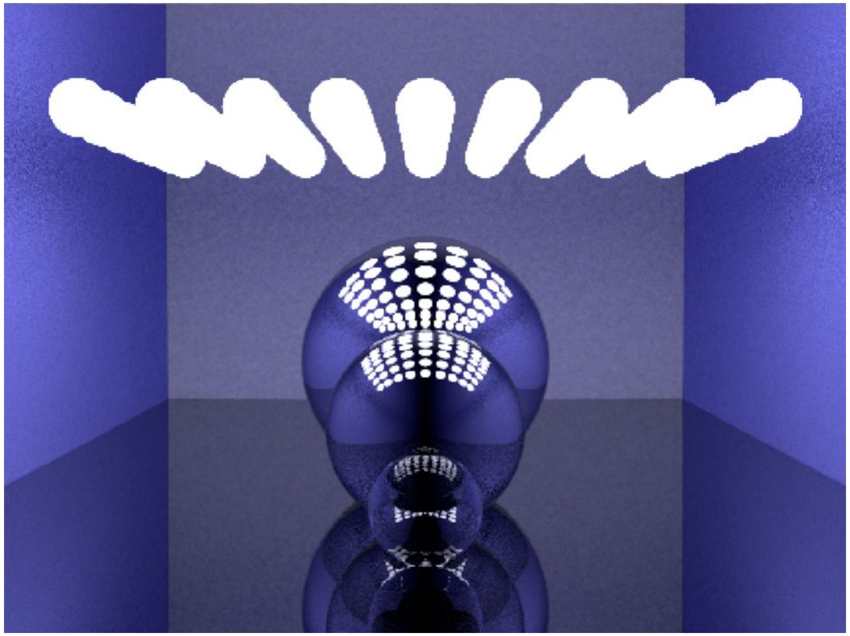
- Cornell Box



- Large Box

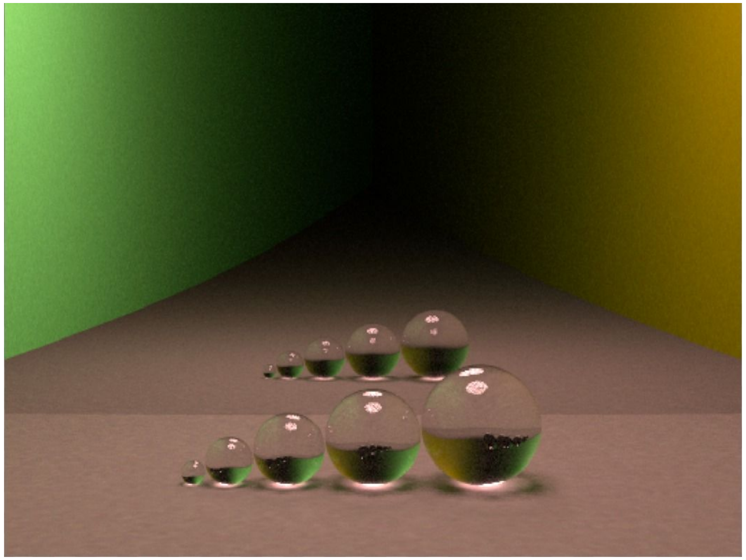


- Surgery Box

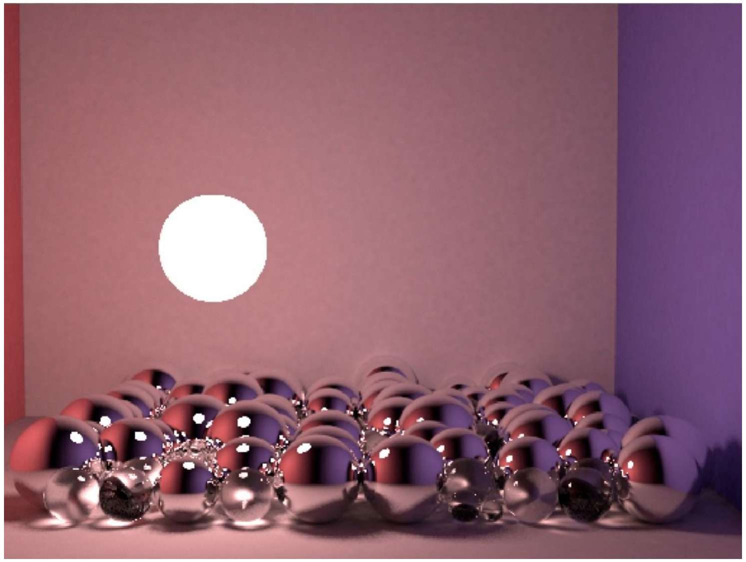


Test Scenes (2)

- Mirror Box



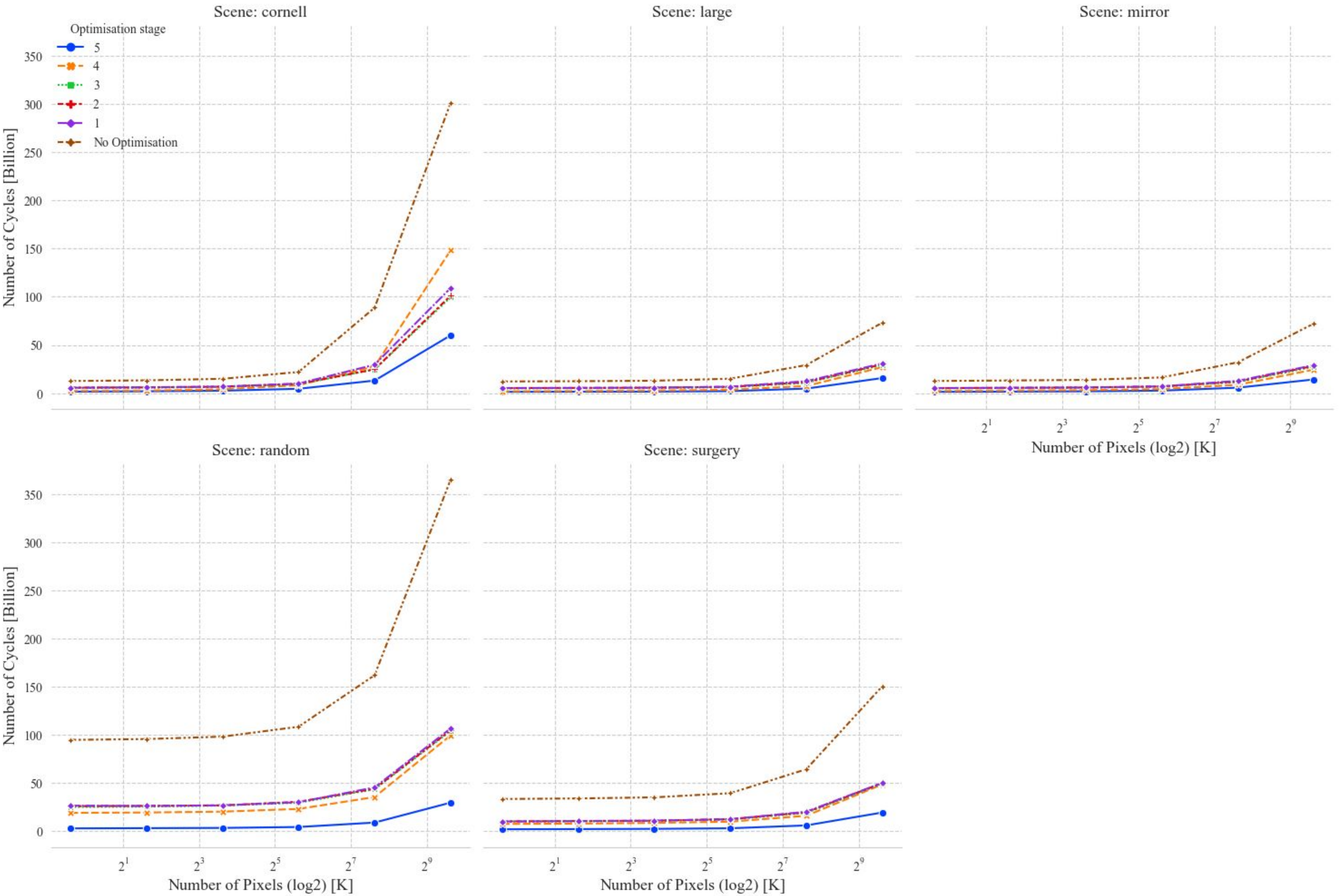
- Random Box



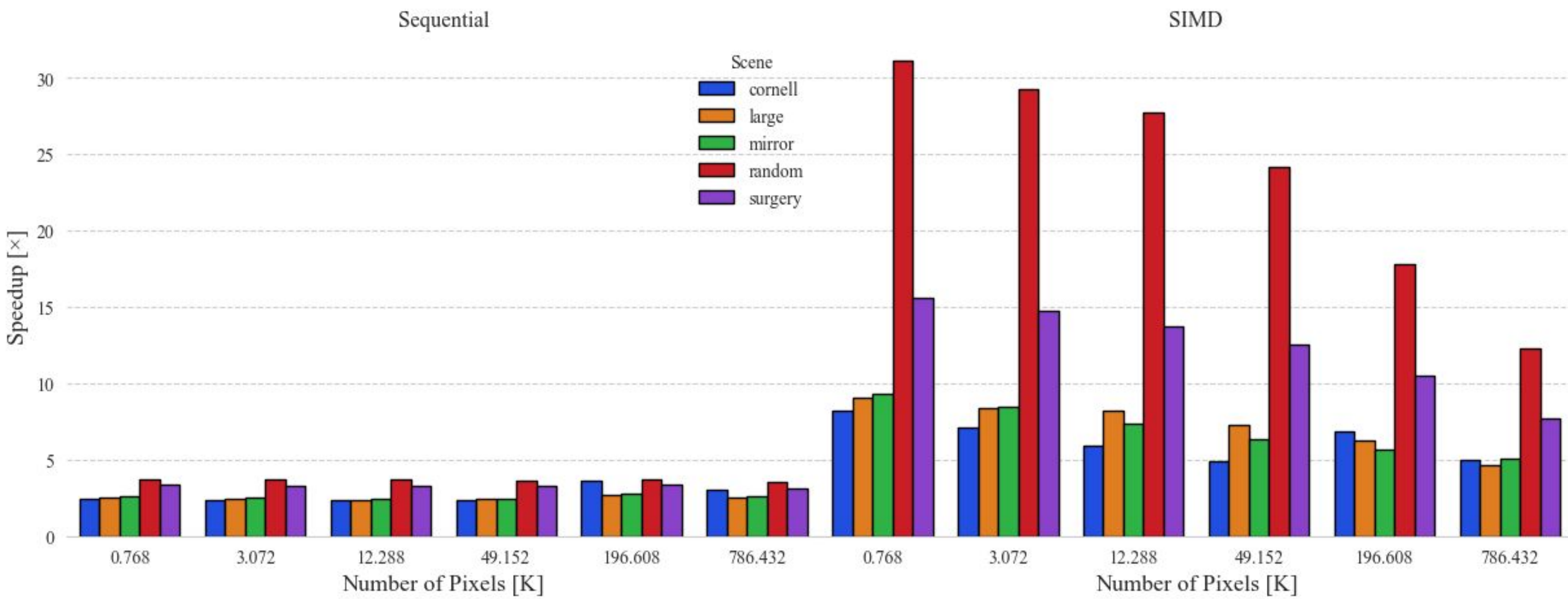
Experiment Setup

- Cold cache
- Turbo-boost and hyperthreading disabled
- Testbed:
    - CPU: Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz
    - Number of cores: 16 (2 sockets)
        - *L1d cache: 512 KiB*
        - *L1i cache: 512 KiB*
        - *L2 cache: 4 MiB*
        - *L3 cache: 40 MiB*
    - Memory: 64GiB

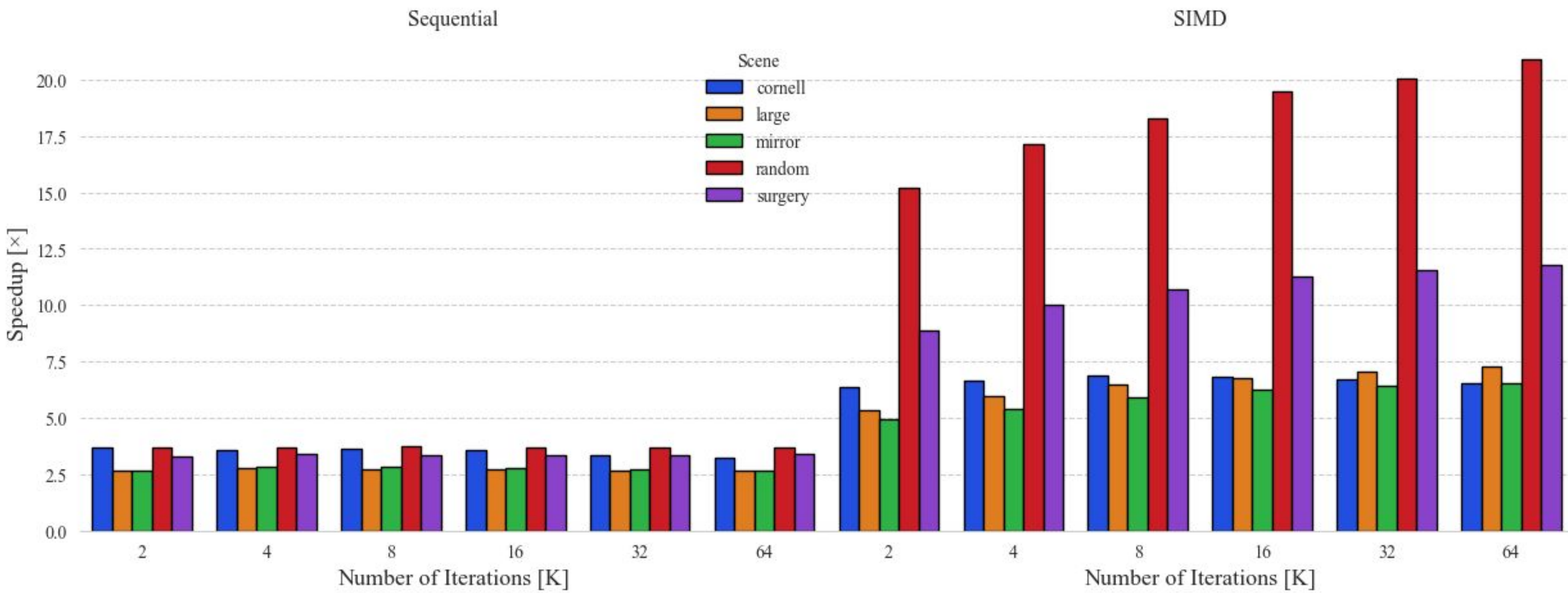
Results: Stages (image sizes)



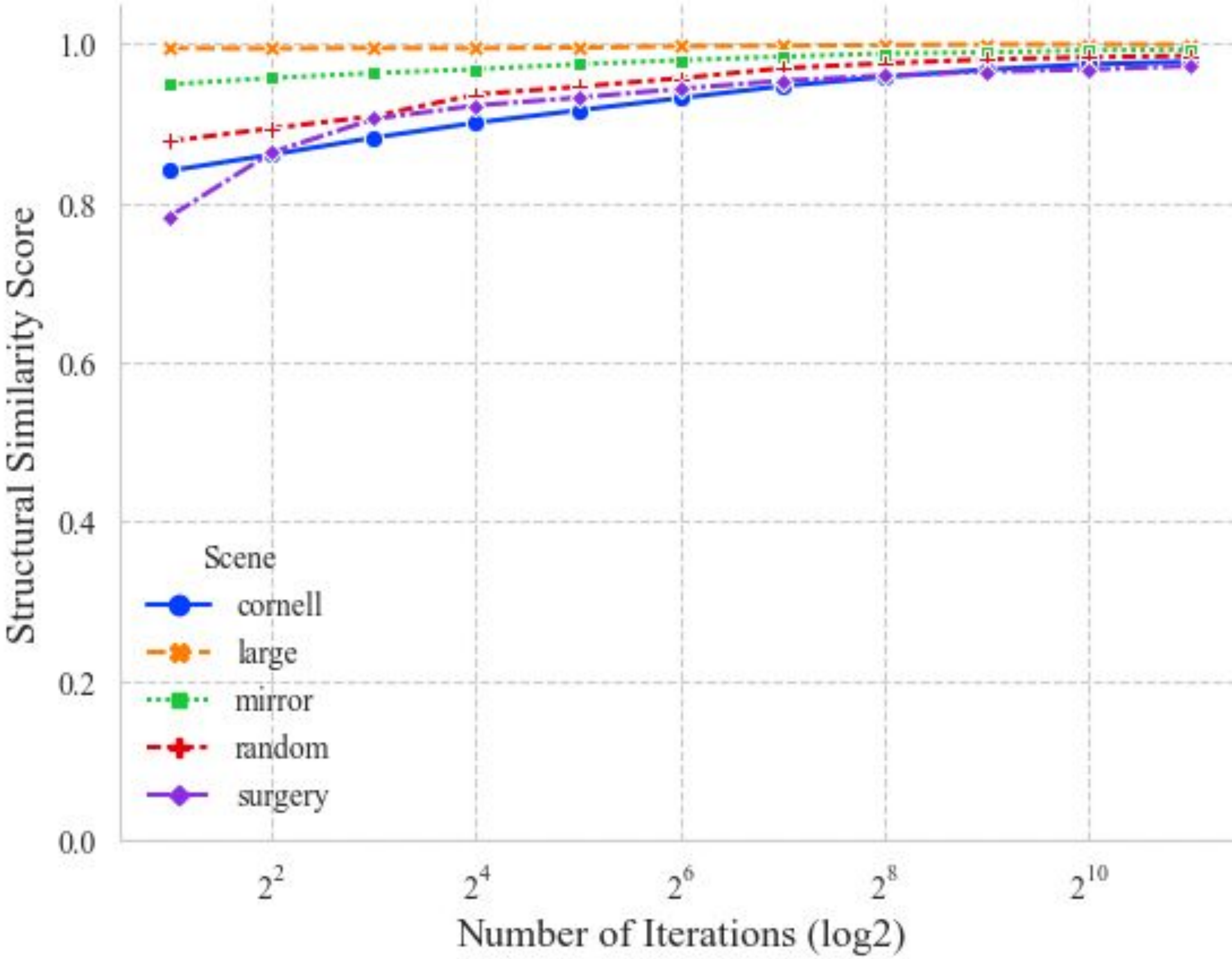
Results: Speedup (image size)



Results: Speedup (#iterations)



Validation

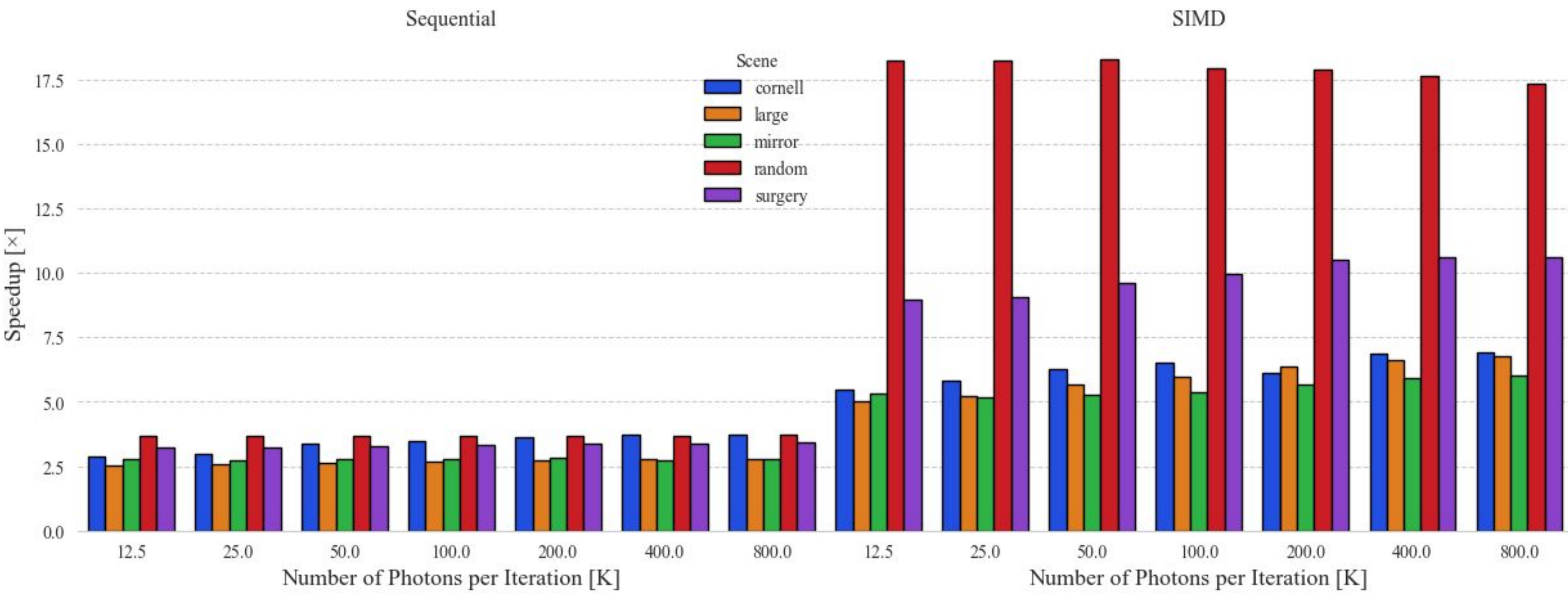


Structural Similarity Score
1.0
0.8
0.6
0.4
0.2
0.0
2^2
2^4
2^6
2^8
2^{10}
Number of Iterations (log2)
Scene
cornell
large
mirror
random
surgery

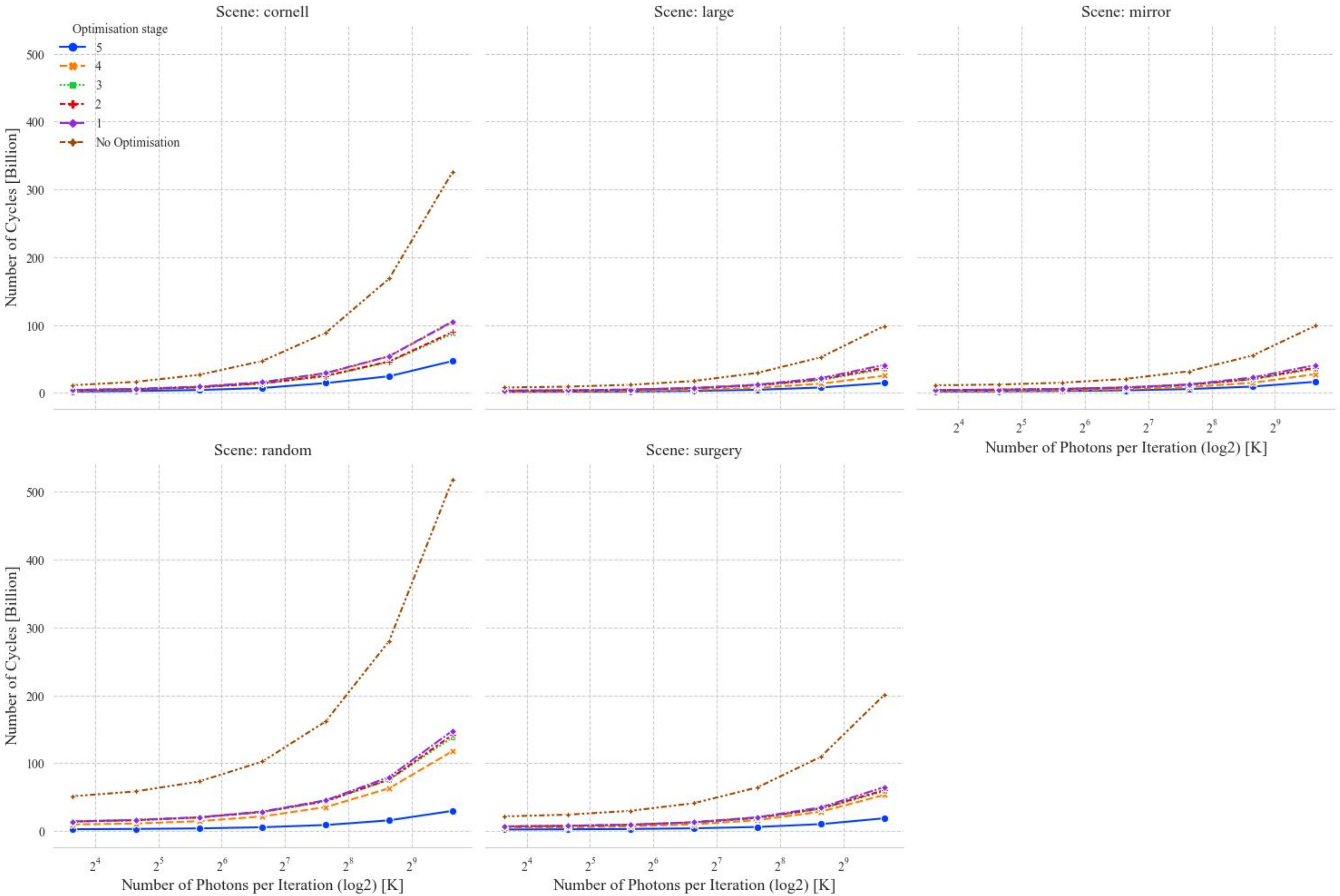
Thank you

Q&A

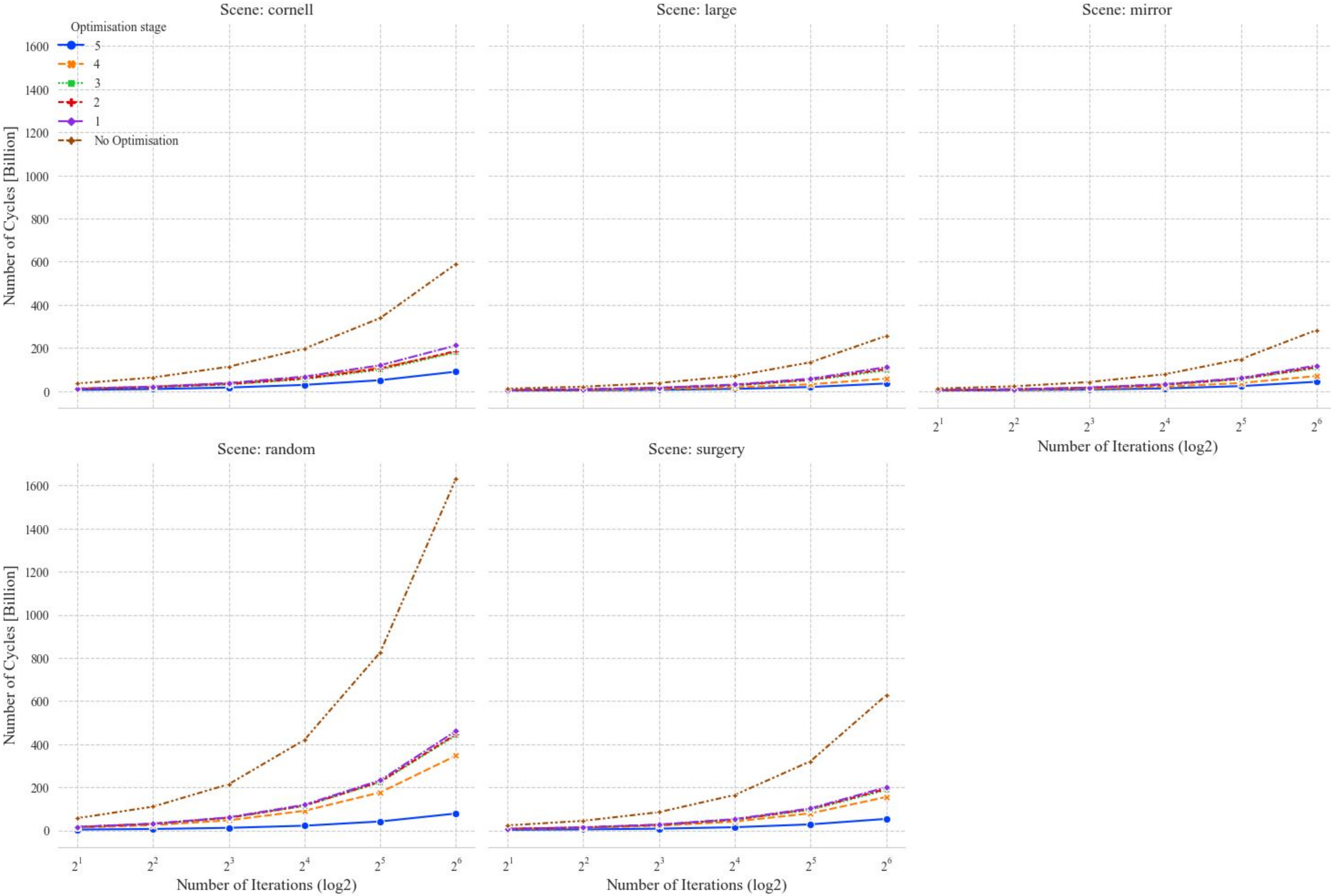
Backup: Speedup (#photons)



Backups: Stages (# of photons)



Backups: Stages (# of iterations)



Potential Q&A

1. **What are the bottlenecks that have and/or haven’t been resolved?**
2. **What’s the speedup over the baseline?**
3. **What are the major optimizations/code versions?**
    a. Explain the most interesting part in depth.
4. **Why do you think the input sizes have been pushed to the limits?**

Results: Speedup