FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Fast Radial Search for Progressive Photon Mapping

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Technische Informatik

eingereicht von

Alexius Rait
Matrikelnummer 11777774

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Dr.techn. Michael Wimmer
Mitwirkung: Stefan Ohrhallinger, PhD

Wien, 01.10.2022

(Unterschrift Verfasser) (Unterschrift Betreuer)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Fast Radial Search for Progressive Photon Mapping

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Computer Engineering

by

Alexius Rait
Registration Number 11777774

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Dr.techn. Michael Wimmer
Assistance: Stefan Ohrhallinger, PhD

Vienna, 01.10.2022

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Alexius Rait
Barmhartstalstraße 109

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Kurzfassung

Um das Problem der Global Illumination effizient zu lösen, wurden über die Jahre zahlreiche Algorithmen entwickelt, einer davon – Photon Mapping – dient speziell dafür, Kaustikeffekte zu simulieren. Das wird im Algorithmus erreicht, indem man Photonen nicht direkt von einer Lichtquelle bis zum Auge verfolgt, sondern zuerst deren Auftrefforte auf diffusen Flächen in einer Datenstruktur speichert, um später Photonen nahe bestimmter Suchpunkte zu finden. Diese Schritt ist besonders rechenintensiv, da für jede Iteration Millionen von Suchvorgängen gestartet werden müssen.

In dieser Arbeit werden wir die Performance zweier verschiedener räumlicher Datenstrukturen und deren in CUDA geschriebenen Suchalgorithmen vergleichen.
Beide Algorithmen wurden in einem Open-source Progressive Photon Mapping Projekt [11] implementiert und getestet. Für die Implementierung wurden Teile von S. Reinwalds Bachelorarbeit Fast-kNN verwendet [9].

Abstract

Global illumination is critical to realistic rendering and as such many different algorithms were developed to solve it, one of which – photon mapping – was designed to efficiently render caustics and indirect lighting. It achieves this by saving photons in a data structure during the first step and then using the result by collecting photons near specific search origins. This step is very time intensive due to the sheer amount of searches performed each iteration.

In this paper, we will compare the performance of two spatial data structures and parallelized search algorithms written in CUDA for the GPU by execution time and memory usage for the photon-gathering use case. The algorithms were implemented in an open-source progressive photon mapping project [11] and are using parts of S. Reinwald's fast-KNN as a basis [9].

Contents

CHAPTER 1

Introduction

Photon Mapping is a two-pass rendering algorithm developed by Henrik Wann Jensen [3] in the early 2000s designed to more efficiently simulate refraction of light. Instead of tracing the complete path from eye to light source, light is collected on diffuse surfaces and stored in a photon map, before being used to calculate luminance on said diffuse surfaces in the photon gathering step. This method has an advantage over other Monte-Carlo ray tracing algorithms such as path tracing [4] when it comes to light traveling through multiple specular surfaces (eg. light shimmering at the bottom of a swimming pool), which otherwise would only have a small chance to be sampled.

In 2008, two of his colleagues (Toshiya Hachisuka, Shinji Ogaki) and Henrik improved on the algorithm by allowing the user to execute an arbitrary amount of photon mapping passes to render scenes more accurately without requiring memory to store all photons at once [2].

1.1 Motivation

During photon gathering a search for nearby photons is made for each screen pixel. For this reason photons should be stored in a spatial data structure in a way that allows quick local access. The most straightforward way to achieve this is to partition space into a 3D grid and use the search coordinates as indices. Each grid element then references an either sufficiently large or dynamic array with all photons at that grid cell.

This solution is not ideal from a memory locality standpoint however as cells next to each other are not necessarily stored in a close-by location in memory. In practice the arrays of each cell are combined into one big array, while a hash table stores the starting memory location or index and the size of each (previously) individual array as shown in Figure 1.1. Doing this while arranging cells based on their position in space is called spatial hashing.

After this change photons that are in neighbouring cells on the x coordinate are close together in memory and we can read a whole line at once but changing either the y or z coordinate can still lead to jumps in memory location as seen in Figure 1.2.

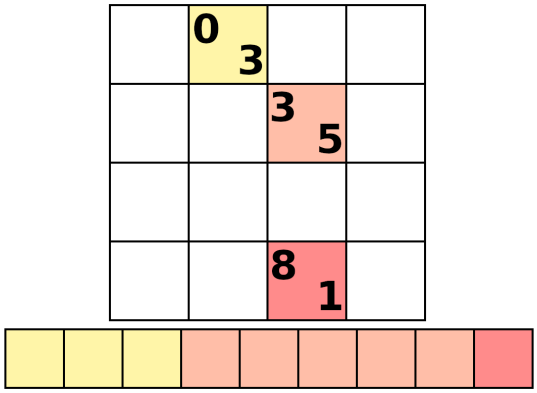


Figure 1.1: Example of a packed array, here a 2D coordinate is used to find a cell which then contains an index (upper left) and length (lower right) to describe an area in the packed array, empty cells would contain a length of 0

Alternatively, we can project photons on a 2D grid and use the projected coordinates for our search. This way the memory locations for a search in the packed array are put into bigger contiguous groups, since they are ordered by 2 coordinates instead of 3, thereby improving the cache usage. Conversely, the search overhead is increased by projecting photons that are far from our search origin based on the projection's z-axis into the same cell as the photons we want to find.

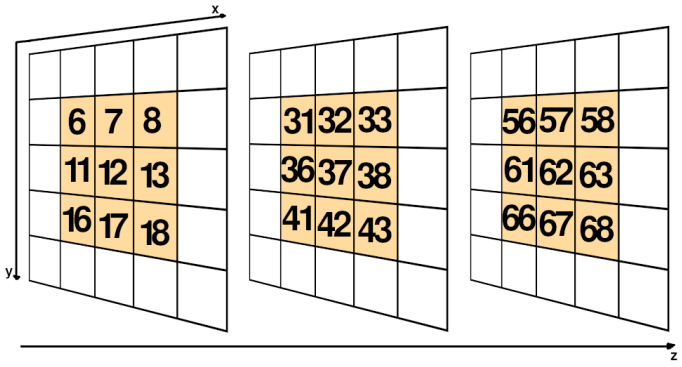


Figure 1.2: Spatial locality of the 3D grid. Searching through a 3 by 3 by 3 area sequentially, the memory location jumps 3 times as much compared to in a similar 2D grid.

Since photon gathering is a memory-intensive operation and a big time loss during photon mapping the goal of this paper is to compare the efficiency in terms of execution time of both methods, 3D spatial hashing and a new 2D-based radial search.

1.2 Method

We have the following problem statement: *Given a set of points and search origins in* $\mathbb{R}^3$ *and a threshold for each search origin, let* ***A*** *denote the set of points that are nearer to the search origin in euclidean distance than the threshold. The algorithm should return a super-set of* ***A*** *for each search origin.*

Finding points that are farther will not affect the produced image, as their effect is removed by the distance attenuation factor, but will still slow down the query and should be avoided where possible. Since it is used for progressive photon mapping [2] the algorithm is run once per iteration, each time with a different set of points.

As stated in the previous section the goal is to project the points into a 2D grid. Before rendering, a projection is chosen based on the scene. Using that projection, all points are assigned to specific cells in the grid based on their projected x- and y-coordinates as seen in Figure 1.3 (cells cover equal areas of equal form in the projection; a cell can contain multiple points).

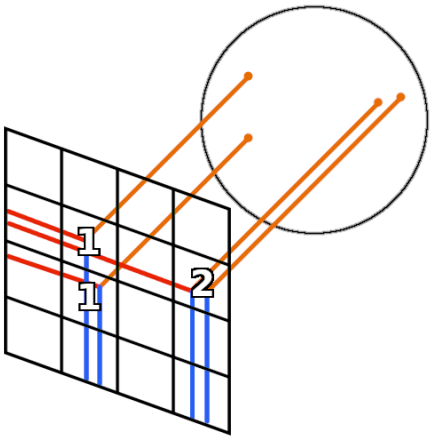


Figure 1.3: Projecting photons into a 2D grid. After projecting into screen-space (screen width and height of projection in the figure are set to 4, same as grid size), a point's x- and y-coordinates are rounded to the nearest integer to find the cell position.

The content of all cells is packed into an array based on cell position with a specific ordering scheme (see section 3.1 Spatial hash grid). This data structure was chosen over kd- or quad-tress to offer better control over the searched areas (see section 5 Optimizations). As the last step of data structure construction, an additional helper data structure containing information necessary to index specific cells is also created.

Finally, by projecting the search areas (spheres) and calculating which cells intersect, it becomes possible to find the super-set of A.

The existing 3D grid algorithm which we compare to works much in the same way except that instead of doing a projection the grid extends in 3 dimensions with cubic cells.

A more detailed explanation can be found in Chapter 4 Implementation.

CHAPTER 2

Related Work

Finding neighbours in a data set is a common problem in numerous applications and has been worked on by many over the years. If the problem consists of finding data points under a certain distance from a search origin for example, the primitive solution would be to compare every data point to every search origin, yielding a time complexity of $O(kn)$, where k is the number of search origins. To optimize the resulting algorithm we can parallelize it to k independent searches. Since photon mapping requires multiple searches per rendered pixel this number scales up quickly (in the order of millions). A GPU is ideal for doing these types of calculations. We used S. Reinwald's CUDA implementation for the k-nearest neighbour search as a reference for our program [9].

The other possibility would be to reduce the n factor by limiting the search area. Using a spatial data structure makes it possible to quickly skip areas that lie outside of the search bounds. Vincent C. H. Ma et al. designed an approximate kNN (k-nearest neighbours) search based on a Hilbert curve hash function [5]. It works by ordering photons based on the curve, creating contiguous blocks with a fixed amount of photons in each and creating multiple slightly different hashing tables to refer to every block intersecting specific areas. A search is done per hashing table (skipping over blocks that were read beforehand) to make sure neighbouring photons are found in edge cases.

In their work *Toward Practical Real-Time Photon Mapping: Efficient GPU Density Estimation* M. Mara et al. compared many different ways of realizing photon density estimation for photon gathering [6] and categorized these approaches.
The first categorization is based on what is iterated through first. Solution one consists of going through each photon and drawing its effect on the neighbouring surfaces. This can be done by creating a bounding volume around each photon representing its area of effect and issuing a draw call for it. The advantage of this method consists of making use of the graphics mode of a GPU instead of a compute shader and being generally better supported, since it is what GPUs are designed for. The bounding geometry may be fully 3D (simpler to program since only the size changes not the shape, but slower due to how much geometry is created) or a billboard (which has to be skewed and scaled first depending on projection).

The other solution is what is typically referred to as *gathering*, where for each pixel nearby photons are found and their contribution is summed up. This is done with a compute shader and allows more flexibility on how much to optimize for performance or quality by sub-sampling. Subcategories of this solution most notably include a 3D hash grid and a 2D tiled algorithm. For the 2D tiled algorithm, their use of a perspective projection with a similar field of view and direction as the scene camera improved cache usage and yielded the best results. In our test application however it made execution time dependent on camera placement – looking at large surfaces at a tight angle slowed the algorithm down substantially – so we chose to use a different projection instead. This effect can be avoided if only one photon is stored by cell as stated in the following paragraph.

For further optimization of the hash data structure, Wann Jensen and Hachisuka stipulated that one can avoid packing an array altogether by simply only using one photon per cell at random and multiplying its flux by the number of photons that would have been mapped to that cell [1]. The resulting image will need more iterations to converge but the algorithm will use much less time per iteration. We will however not go into further detail on that alternative as the resulting algorithms would converge at different speeds depending on how the grid is laid out, thereby making comparison difficult.

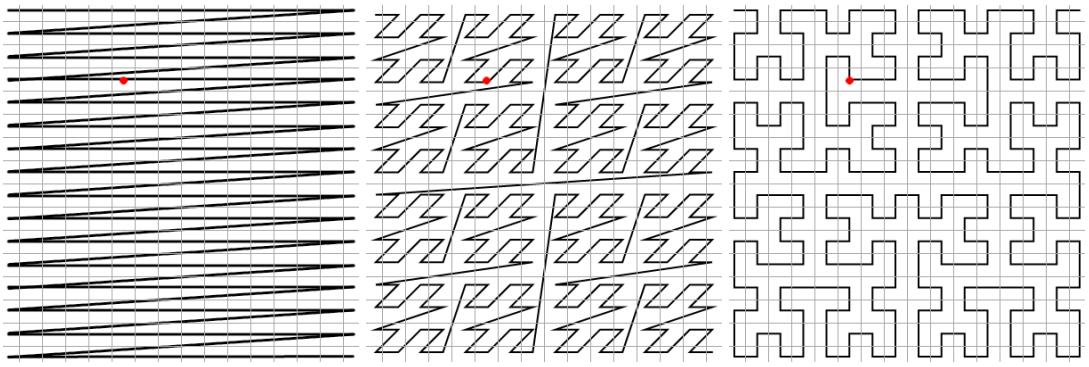
To implement and compare our search algorithm we used the git project *Accelerated Stochastic Progressive Photon Mapping On GPU*, which uses a fixed size 3D hash grid (hash function orders by x then y then z) to store photons [11].

CHAPTER 3

Technical Background

3.1 Spatial hash grid

A spatial hash grid is a data structure to efficiently store and retrieve data using 2 or more coordinates as an index. The hashing function defines in which order the grid elements are stored and is used to retrieve them. Depending on the application a reverse hashing function needs to be found which converts an index back to coordinates. Possible ways to order elements include left to right and top to bottom, z-order, or Hilbert curve order.



(a) Left to right and top to bottom, index: 53

(b) Z-order, index: 27

(c) Hilbert-curve order, index: 28

Figure 3.1: Different hashing function output for coordinates (5,3)

Since the previously implemented 3D grid in the test program uses the first ordering scheme despite being worse for spatial locality and could be implemented with any of those we chose to make use of the same one to offer a better comparison.

3.2 Fast-KNN

Fast-KNN is a CUDA algorithm to efficiently solve the KNN problem in two dimensions developed by S. Reinwald [9] and D. Schoerkhuber [10]. It does this by partitioning space in a quad-tree, the elements of which are saved in a packed array and creating additional buffers that contain index and element count of each quad-tree cell similar to a hash grid.

Using the element count buffers a search radius is then estimated for efficient searching. If the element count does not match the minimum and maximum required the algorithm performs another iteration with an adjusted radius. [9]

This was used as a basis for the radius search. The quad-tree aspect was removed in our implementation since we wanted to compare performance with a fixed radius algorithm and a radius estimate was as such not necessary. We also changed the ordering of cells in the packed array from Z-order to left to right, top to bottom for the same reason.

CHAPTER 4

Implementation

In this chapter, we will first go over the existing 3D spatial hash algorithm briefly and then explain our projection-based radial search in more detail.

4.1 3D Spatial hash

The existing algorithm separates its photons based on a grid with cubic cells, changing the dimensions based on the bounding box of the scene (in our test cases they are at 32x32x32). These photons are stored in a spatial hash grid (see Chapter 3.1).

Construction of data structure

Photons are first stored in a dense array and receive a key based on their rounded down position ($i + x_{dim} * j + x_{dim} * y_{dim} * k$ where $x_{dim}, y_{dim} \in \mathbb{N}$ denote the grid dimensions and $i, j, k \in \mathbb{N}$ stand for the position based on grid coordinates). Using the CUDA function thrust::sort_by_key the photons are then sorted, which means they are ordered by their respective cell's z then y then x position (elements of the same cell in no particular order). An array containing the starting index of each cell is created by comparing each element's grid position g with the previous one's and setting the starting index of the g cell to that element if it is different (for empty cells that number stays at -1).

Search

For a search, the cell containing the search origin and 26-connected cells are inspected sequentially using loops.

4.2 Radial search

Our algorithm separates its elements based on a 128x128 grid (testing showed this to be the best subdivision in powers of 2 for our test setup) and also uses a spatial hash grid (see Chapter 3.1).

Overview

While there are a few differences in the implementation, the algorithm generally can be thought of as a 3D spatial hash (see Chapter 4.1) with a changed hashing function (due to the orthographic projection) and a different search area (instead of a 1 by 1 by 1 cube, a cylinder spanning the near and far plane of the projection is searched).

The following arrays are used to store photons:

- radialSearchNumBuffer[$gridDimX * gridDimY$] : uint
  Contains the number of photons that fall in each cell, ordered left to right, top to bottom
- radialSearchPackedIndices[$gridDimX * gridDimY$] : uint
  For each cell contains the starting index for elements of that cell in radialSearchPackedPhotons.
- radialSearchPackedPhotons[$numPhotons$] : photon
  Contains all photons in a packed array.

Program initialization

Once per program execution a projection is selected based on the diffuse objects in the current scene. The projection should be chosen in such a way that $\sum_F |F_{normal} \bullet view| * F_{area}$ over all diffuse faces F is maximized. Choosing a bad projection, for example orthographic with (0,0,-1) view direction in our test scene, increases the execution time by around 2 orders of magnitude.

In our solution, we forego calculating the ideal projection and use (1,1,1) as a view direction and an orthographic projection instead. To make sure all objects fit in the image fully, the height and width of the projection are chosen by computing a bounding box, rotating it with the view matrix and selecting the maximum and minimum of the x and y coordinates. The transformation is saved in a view projection matrix.

Since the grid we project on stays the same size, we need to compensate our search radius for different scene sizes to make sure the search area in world space stays the same. The search radius is divided by the projection size and multiplied by the grid size.

At this point, the search radius is further reduced to compensate for the area difference between the circle itself and the area of intersected grid cells to make for a better comparison of both methods. The formula was determined by a regression analysis tool using area ratio data from a python script. The script tested circles of radius from 1 cell length to 10 in 0.25 increments and changed their center in 0.1 increments. The formula for area ratio between a circle's surface and the intersected cells was tested to be around $1 + 1.2727/radius$ for that range.

Construction of data structure

For every photon, we multiply its position with our view projection matrix and then transform the coordinates from view space to a screen space that represents our grid (128x128). For each grid cell, photons that would fall into it are counted and the result is written into the numBuffer.

In the next CUDA kernel, the starting index of each cell is computed by summing the numBuffer elements of previous cells and written into the packedIndices array. This is done sequentially in a for loop, but since this part is only executed once per iteration the effect on execution time is negligible.

Photons are written into the packedPhoton array in parallel. To avoid collisions, their index is set to be the starting index plus the result of an atomic increment operation.

```
// Photon packing function
photon p;
width, height = GRIDSIZE;
pos = projectIntoScreenSpace(orthoMat, width, height, p.position);
if (onScreen(pos)){
    // hashing function; GRIDSIZE is used for both dimensions
    idx = pos.x + GRIDSIZE * pos.y;
    //the writeBuffer used for packing starts with all zeros
    offset = atomicAdd(writeBuffer[idx], 1);
    targetIdx = radialSearchPackedIndices[idx] + offset;
    radialSearchPackedPhotons[targetIdx] = p;
}
```

Search

The search origin is projected into our 2D grid space and we define a search area around that origin with the radius set at initialization. For every grid row that is intersected, we get the index of the first photon of the leftmost intersected cell and the index of the last photon of the rightmost intersected cell and go through every photon in-between including those two. This is done to limit additional memory accesses to radialSearchNumBuffer and radialSearchPackedIndices.

To find the first and last cell of a row, we calculate the maximum circle width in the bounds of that row by inserting into the circle equation $x = \sqrt{r^2 - (y - v.y)^2}$ (where v stands for the search origin) using y in grid coordinates (whole numbers). We substitute $y + 1$ for y if the current row is above the search origin (this represents the y position of the lower border of that row) and 0 for $(y - v.y)$ if the search origin is on the same row (effectively making that case $x = r$). The case differentiation is shown in Figure 4.1. The first and last intersected cells of that row lie on $\boldsymbol{v} \pm \begin{bmatrix} x \\ 0 \end{bmatrix}$.

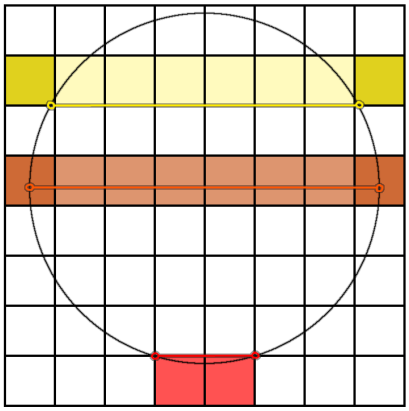


Figure 4.1: Width calculation; yellow: since we need the lower border we need to add 1 to get the y position with the largest width; orange: at the center we use the full radius

CHAPTER 5

Optimizations

This chapter contains all optimizations that were either implemented or at least tested during this project's lifetime.

Search Area

Early versions of the algorithm searched an n by n area on the grid. The higher resolution per direction of our data structure compared to the 3D grid (128 against 32) allowed us to skip cells on the edges of that n by n area that would have been too far from the search origin entirely. This improved execution time by about 10ms for the dragon scene (Figure 6.1). As stated in Chapter 4.2 the radius was later also reduced to better reflect what the search area of the previous algorithm would be in 2D (3 by 3 out of 32 by 32). To make sure the generated images stay correct we compared them after a set number of iterations to a reference picture using peak signal-to-noise ratio.

Hashing function

When creating the algorithm we started by using a z-order hashing function because fast-KNN was built with a quad-tree data structure and had packed them this way for that reason. During development one of the optimizations done consisted of reducing read accesses to the numBuffer and indexBuffer by only looking at the first and last photon indexes of cells neighbouring each other in memory. This coupled with the search area change from before was not a viable option for a z-ordered grid as shown in Figure 5.1. Changing from z-order to left-to-right, top-to-bottom did not incur a significant execution time increase however. We also tested ordering by the y coordinate first but no significant time loss or gain was measured.

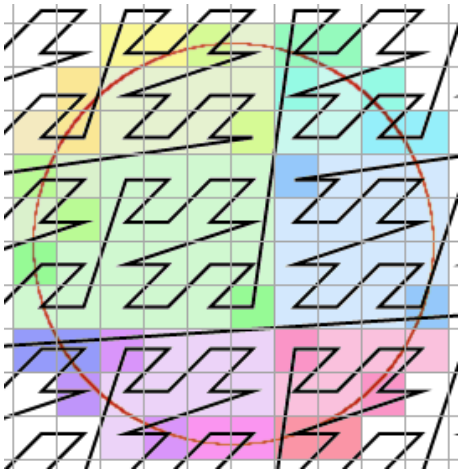


Figure 5.1: Neighbouring cells in z-order when searching in a circle, each different hue (15) represents a separate area with start and end being a darker color.

Grid size

Over the course of development, a few different grid sizes have been tested. At first, due to the algorithm using a quad-tree only dimensions in powers of two were possible. Now any grid size divisible by 4 is valid but due to variation between scenes and only small differences around 128 we chose to stay at a power of 2, the execution time comparison can be seen in Figure 5.2. The optimum used to be at 64 but we increased the photon count per iteration to offer more accurate results and had to change the grid accordingly.

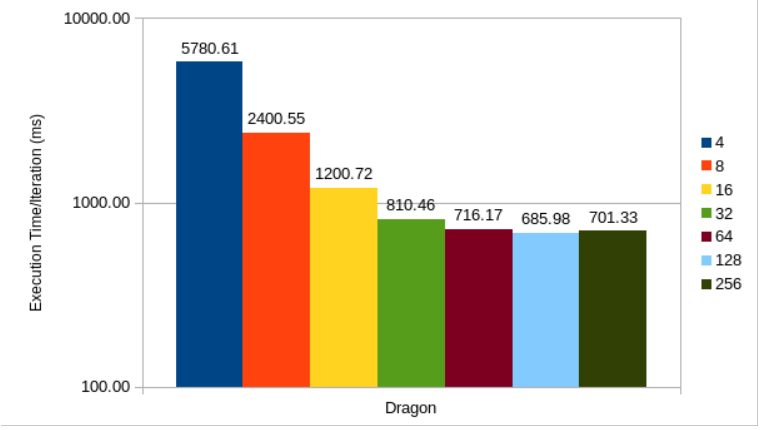


Figure 5.2: Different powers of 2 for grid size compared in terms of execution time per iteration.

CHAPTER 6

Performance

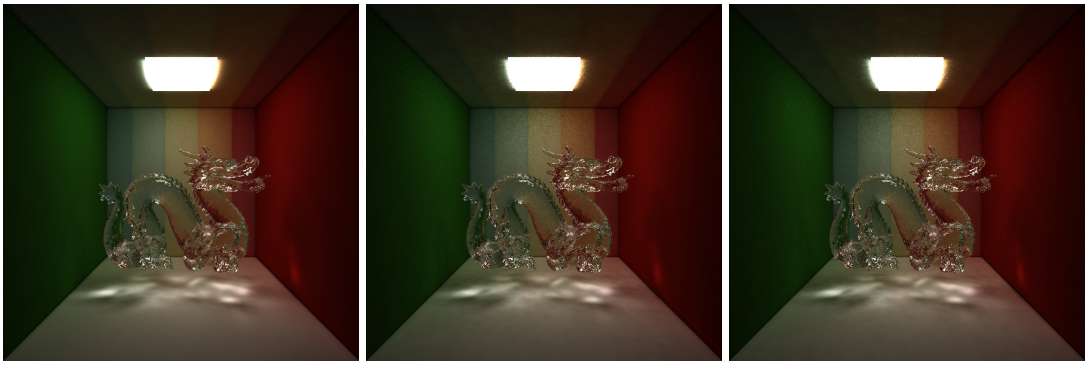
Test setup

In each scene, photons are bounced until 3.200.000 positions are recorded every iteration of which we measured the first 50 iterations. Only photons that bounced at least once before and are diffusely scattered in that bounce are kept for future calculations, which makes the final number of photons after stream compaction around 300.000. We sample once for each rendered pixel in each iteration, adding up all photons that the search returns that fall on the object the raycast ended on (weighted based on distance).

To compare the correctness of our algorithm we used images that were rendered for 250 iterations using the 3D grid and a larger search area as a reference. The relative peak signal-to-noise ratio difference between images rendered using the previous 3D grid search and our 2D projection-based search stayed under 1/1000 for all test scenes.

We had the following test scenes:

- Buddha (Figure 6.4): a scene with a high polygon count ($\approx 1.000.000$)
- Checkerboard (6.7): the geometry in the background is alternating between surfaces that are sufficiently far from each other to not fall into neighbouring cells, which is bad for caching
- Cow Bunny Sphere (6.5): both reflective and transmissive glass textures on objects and diffuse spheres
- Dragon (6.1): base scene on which most time was spent optimising
- Diffuse Dragon (6.2): scene to show weaknesses of projection
- Mega (6.3): due to higher resolution more searches are made
- Tetrahedron (6.6): very simple geometry



(a) Reference picture, rendered for 250 iterations

(b) The scene after 50 iterations using the 3D grid hashing

(c) The scene after 50 iterations using our algorithm

Figure 6.1: Dragon Scene

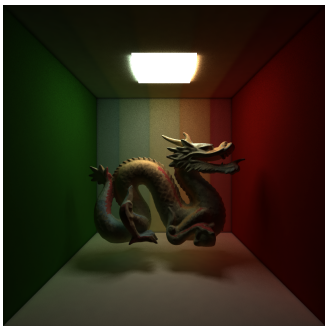


Figure 6.2: Diffuse dragon

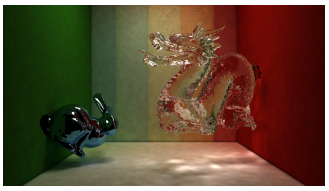


Figure 6.3: Mega: scene rendered in full hd instead of 800x800 resolution

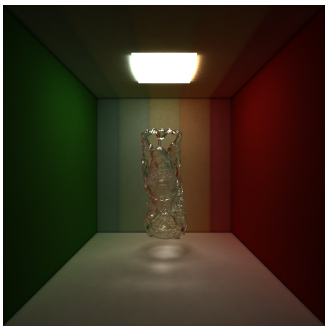


Figure 6.4: Buddha

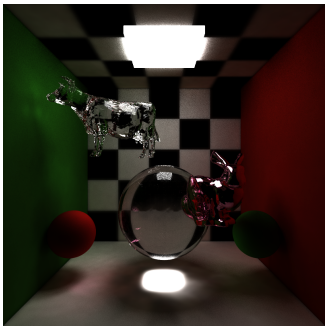


Figure 6.5: Cow bunny sphere

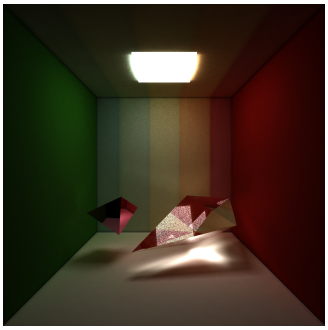


Figure 6.6: Tetrahedron

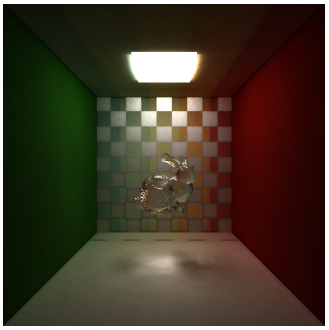


Figure 6.7: Checkerboard

Measurements

For every scene, the median and average execution time per iteration were measured. We also counted how many photons the search had to go through per iteration on average and the maximum memory usage of the program.

All results were measured on a test machine using a Geforce GTX 1070 Ti GPU. Since the

search is executed as part of another kernel and not easily separable we measured the execution time with and without search, the graphs presenting search time contain the difference. Figure 6.8 shows time spent constructing and filling the data structures, 6.9 contains time spent searching with the construction times subtracted and 6.10 contains both summed up.

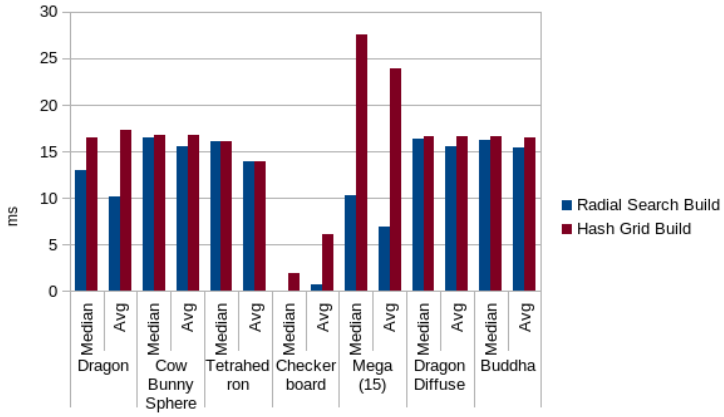


Figure 6.8: Time required per frame to construct and fill the data structures, since thrust::sort is used by the hash grid algorithm during construction it takes slightly longer.

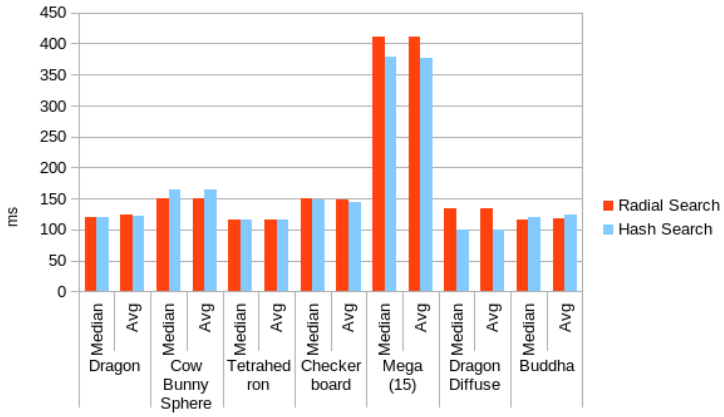


Figure 6.9: Time required per frame to search for photons (construction times subtracted)

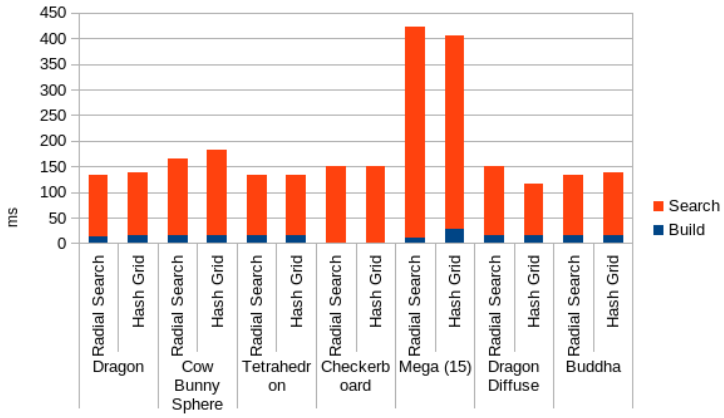


Figure 6.10: Time required per frame for photon gathering step summed

While our algorithm takes less time to fill its data structure this part is an order of magnitude shorter in both cases and doesn't affect the final result by a lot. For the search itself, both methods seem to have similar execution times with the exception of the diffuse dragon scene, where a 3D grid proved more effective.

As was expected due to overlapped surfaces in the projection our algorithm reads through more photons in total than the 3D counterpart (between 20% and 50% for the test scenes) as shown in Figure 6.11. This depends on how the scene is structured, Tetrahedron and Buddha for example show the least difference since most photons are projected on the same surface.

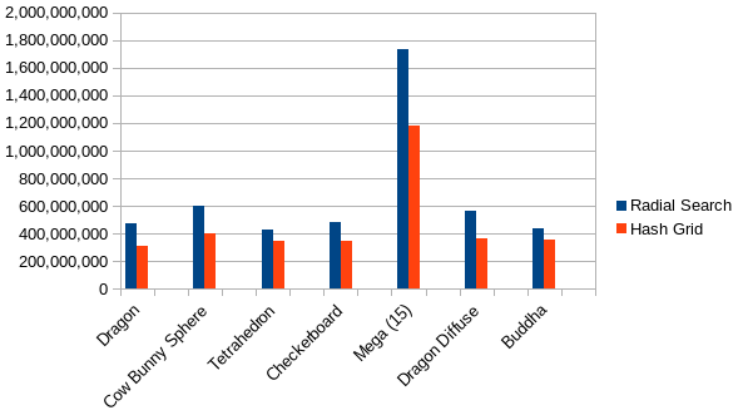


Figure 6.11: Read accesses per frame to the photon data structure during search

The memory usage was consistently 10% higher due to the packed photon structure being copied as part of construction instead of being sorted in place like for the counterpart.

In the two following subsections we will provide some insights on cache usage for both methods, but due to our test GPU not fulfilling the architecture requirements for a GPU trace with NVIDIA Nsight Graphics [8] we can not provide statistics on the matter.

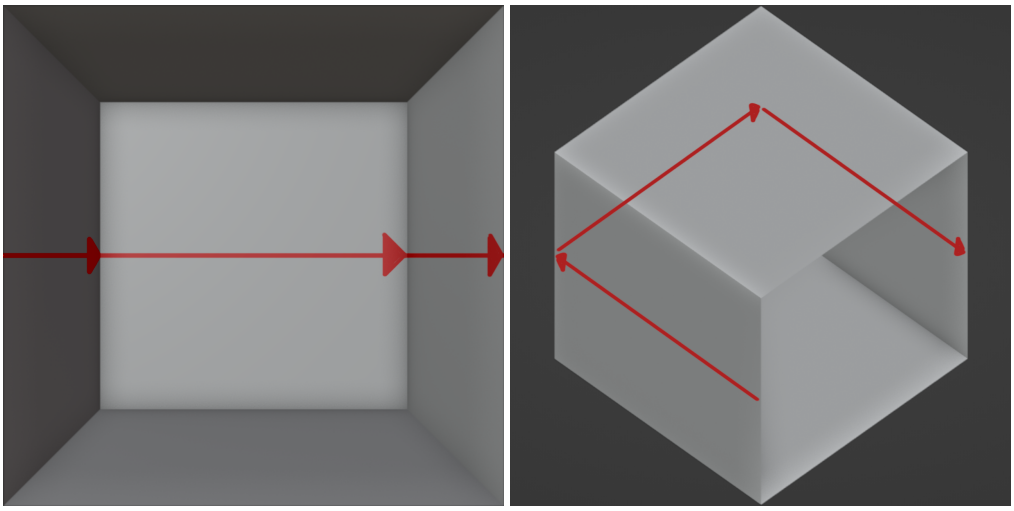
Cache lines

Since the photon data structure used is unoptimized and contains data not relevant to the gathering step each photon takes 60 bytes of storage instead of the 20 described by Henrik Wann Jensen [3]. An L1 cache line is 128 bytes long [7], which means we only load 2 photons in at once on average. This causes the machine to execute a full load operation (no cache hits) for most photons during a first read instead of reusing the contents of the cache line and slows down the algorithm with better spatial locality dis-proportionally.

Cache reuse

Assume a scene with a diffuse cube-shaped exterior with 1 missing side and a light source that uniformly distributes photons on those 5 surfaces. Using the 3D grid there is an average of ca. 58.5 photons ($300.000/(32^2 * 5)$) in each cell containing surfaces and 0 elsewhere and a search goes through about 9 non-empty cells. For the 2D grid (leaving out parts that are not projected onto) we average 30.5 and search through ca. 60 cells, this higher density is advantageous for caching since each thread does its searches in close proximity to each other.

This advantage is offset slightly due to the way the data structure is traversed during consecutive searches. The order in which rays are cast is based on the rendered pixels going left-to-right top-to-bottom. Depending on the current y coordinate each line traverses between 32 (start and end) and 94 cells, while the same operation in projected space traverses between 90 (52 in x, 38 in y; counting cells intersected by a line is done by simple addition instead of the Pythagorean theorem) and 270 cells as shown in Figure 6.12. This affects how quickly photons in our cache become obsolete.



(a) For the 3D grid 94 cells are traversed (32 positive z, 31 positive x then 31 negative z)

(b) In the projected space a large amount of cells are traversed (ca. 250), making cached photons become obsolete more quickly

Figure 6.12: Traversal of data structure during search for one line (disregarding refraction and reflection)

CHAPTER 7

Conclusion

During this research, we compared the efficiency of 2 data structures for the use of photon mapping. This paper shows that neither method is superior in terms of execution time in our current test environment. While our algorithm seemingly processes more photons in the same time window, this advantage is offset by extra reads on the main data structure. The following sections will provide some possible ways to improve the algorithm.

Further optimizations

Finding a better projection should be the first priority of anyone trying to improve on the algorithm. With our (1,1,1) orthographic projection our test scenes only occupied about 60% of the grid we projected on while choosing a projection with too much self overlap as is the case with (0,0,-1) slows down the search algorithm immensely.

To that end, it might also be worth to look into splitting the data structure into one for each diffuse object, which is an option since photons that stop on objects other than the one the camera ray intersected with are not taken into account for a search.

Improving the photon data structure to only use 20 instead of 60 bytes would allow 6 photons in a single cache line and maybe yield a stronger execution time improvement on radial search compared to grid hash.

By implementing the change suggested in *Parallel progressive photon mapping on gpus* [1] of only using one photon per cell an index and numbuffer array wouldn't be needed anymore thereby saving time during construction and 3 reads per searched line to those data structures and allowing for a different ordering scheme without having to deal with finding contiguous areas as shown in Figure 5.1. This change would also enable the more cache-efficient option of using the camera's perspective projection as a basis for the algorithm.

Finally changing the grid order and the order in which rays are cast from row first to something more consistent like the Hilbert curve order could also improve cache usage for photon gathering.

Bibliography

[1] Toshiya Hachisuka and Henrik Wann Jensen. Parallel progressive photon mapping on gpus. In *ACM SIGGRAPH ASIA 2010 Sketches*, New York, NY, USA, 2010. Association for Computing Machinery.

[2] Toshiya Hachisuka, Shinjii Ogaki, and Henrik Wann Jensen. Progressive photon mapping. In *SIGGRAPH '08: Special Interest Group on Computer Graphics and Interactive Techniques Conference*, pages 1–8, New York, NY, USA, 2008. Association for Computing Machinery.

[3] Henrik Wann Jensen. *Realistic Image Synthesis Using Photon Mapping*. Taylor & Francis Group, New York, NY, 2001.

[4] James T Kajiya. The rendering equation. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, pages 143–150, 1986.

[5] Vincent C. H. Ma and Michael D. McCool. Low latency photon mapping using block hashing.

[6] Michael Mara, David Luebke, and Morgan McGuire. Toward practical real-time photon mapping: Efficient gpu density estimation. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '13, page 71–78, New York, NY, USA, 2013. Association for Computing Machinery.

[7] Nvidia. Memory transactions. `https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/sourcelevel/memorytransactions.htm`. Accessed: 2022-03.

[8] Nvidia. Nsight graphics supported gpus. `https://developer.nvidia.com/nsight-graphics-gpus-full-list`. Accessed: 2022-03.

[9] Siegfried Reinwald. Fast KNN in Screenspace on GPGPU. Bachelor Thesis, Vienna University of Technology, 2019.

[10] Dominik Schörkhuber. Fast KNN in Screenspace on GPGPU. Bachelor Thesis, Vienna University of Technology, 2016.

[11] Xiaoyan Zhu and Yingting Xiao and Ishaan Singh. Accelerated stochastic progressive photon mapping on gpu. https://github.com/ishaan13/PhotonMapper. Accessed: 2022-03.