Global Illumination for Point Models

Rhushabh Goradia
Guide : **Prof. Sharat Chandran**

Annual Progress Seminar 2007

Department of Computer Science and Engineering
Indian Institute of Technology, Bombay

16.08.2007

Outline

Outline

Problem Definition

Problem Statement

To compute **a global illumination solution** for complex scenes represented as **point-models**

Things to look out for:

- Point Models
- Global Illumination - Diffuse and Specular
- The Fast Multipole Method
- Graphics Processing Units
- Point-Point Visibility

Problem Definition

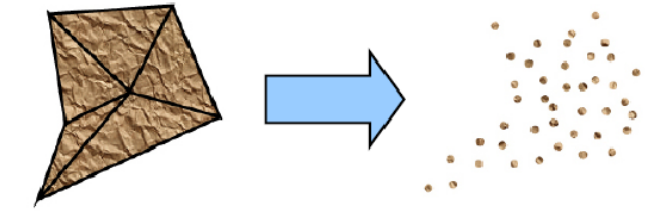
Problem Statement

To compute **a global illumination solution** for complex scenes represented as **point-models**

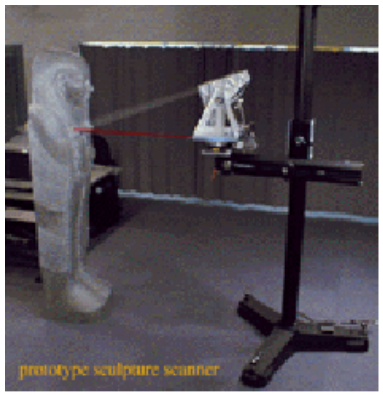
Things to look out for:

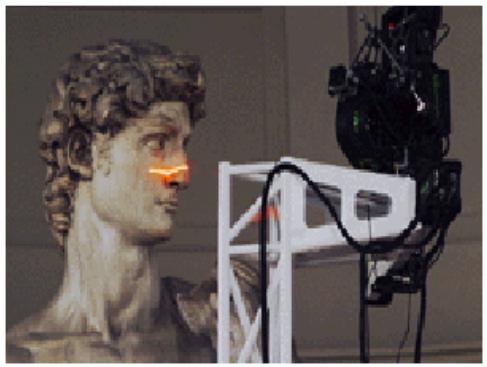
- Point Models
- Global Illumination - Diffuse and Specular
- The Fast Multipole Method
- Graphics Processing Units
- Point-Point Visibility

Point Models ?



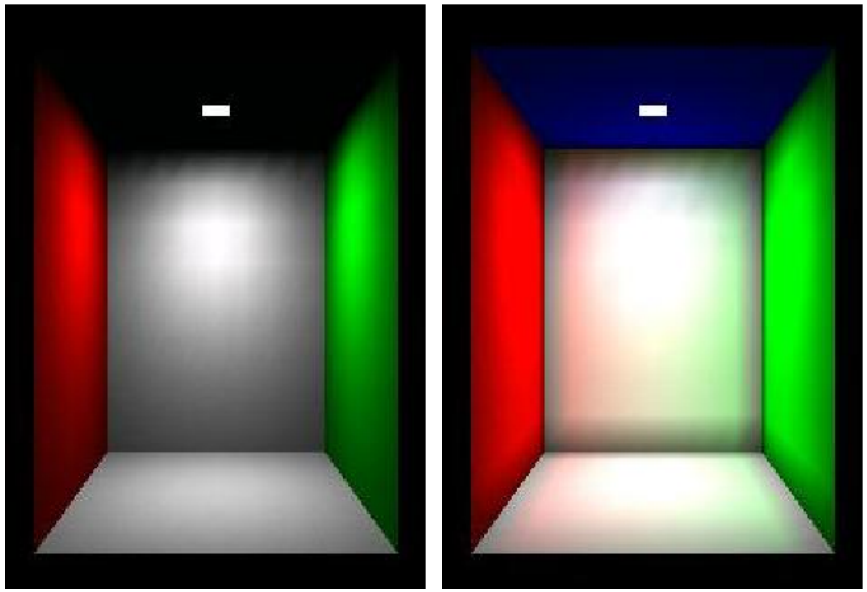
- Model each point as a surface sample representation
- Each point has [co-ordinates, normal, reflectance, emmisivity] values



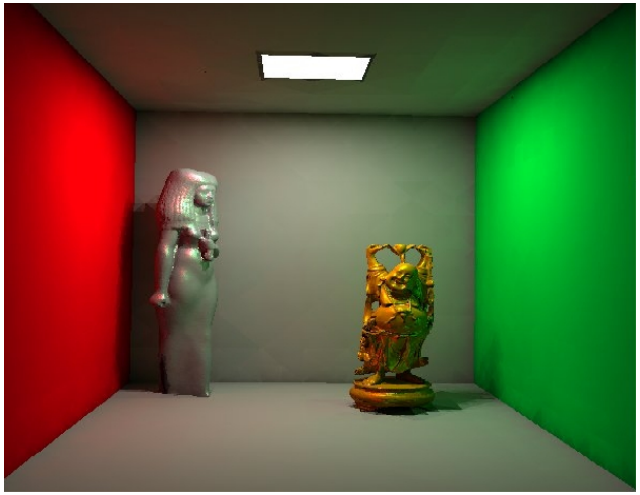


What are Global Illumination Algorithms?

Global illumination algorithms are those which, when determining the light falling on a surface, take into account not only the light which has taken a path directly from a light source (direct illumination), but also light which has undergone reflection from other surfaces in the world (indirect illumination).



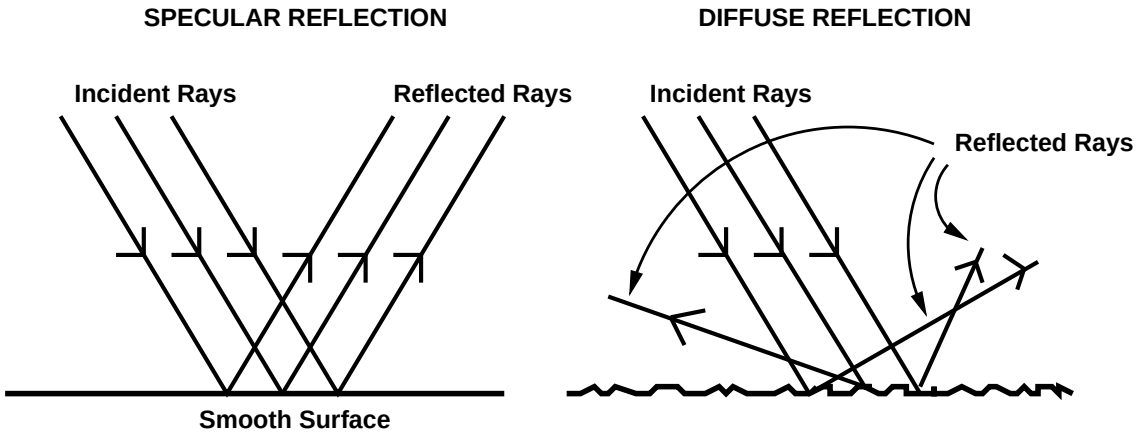
Examples showing GI Effects



Global Illumination effects

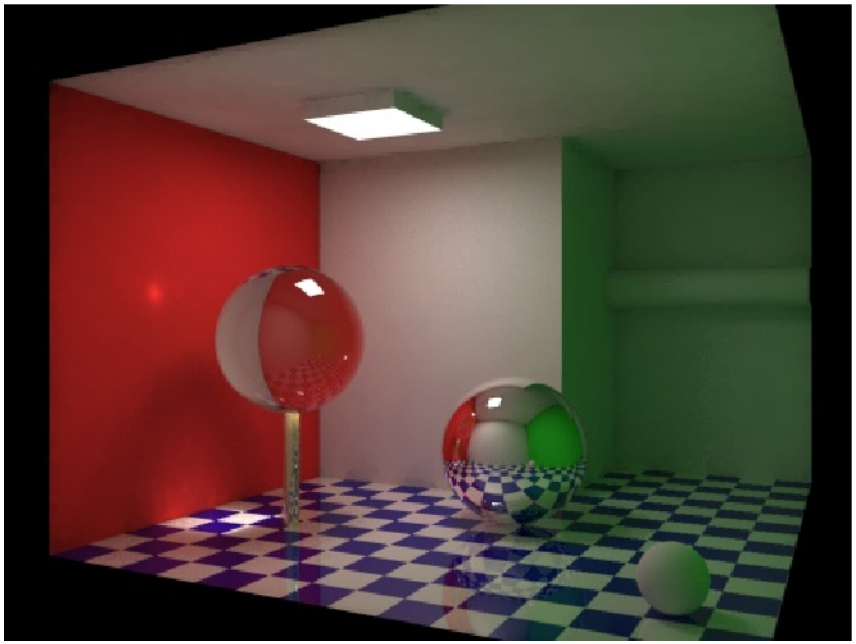
GI effects are the results of two types of light reflections and refractions

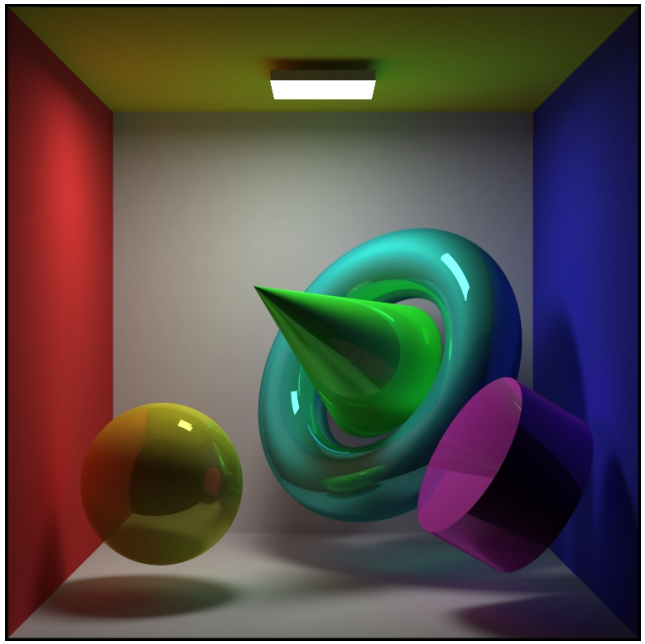
- Diffuse
- Specular



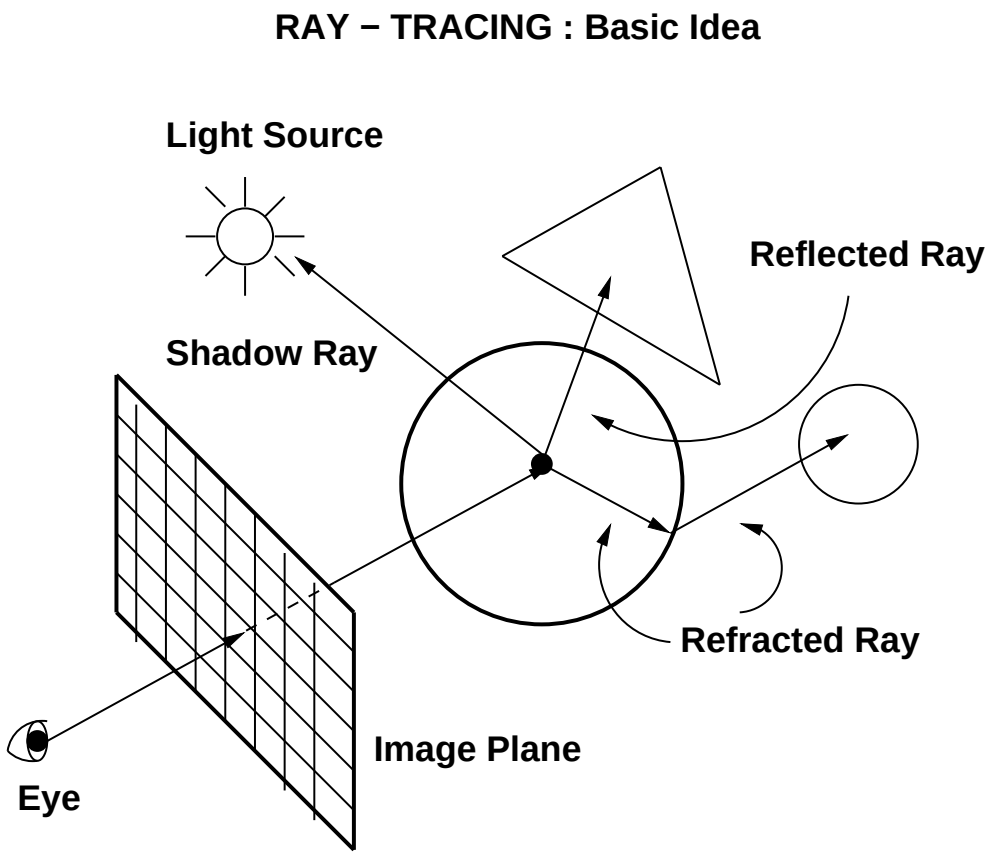
Global Illumination: Diffuse and Specular effects

- Color Bleeding
- Soft Shadows
- Reflections and Refractions
- Specular Highlights and Caustics





Specular Effects using Ray Tracing

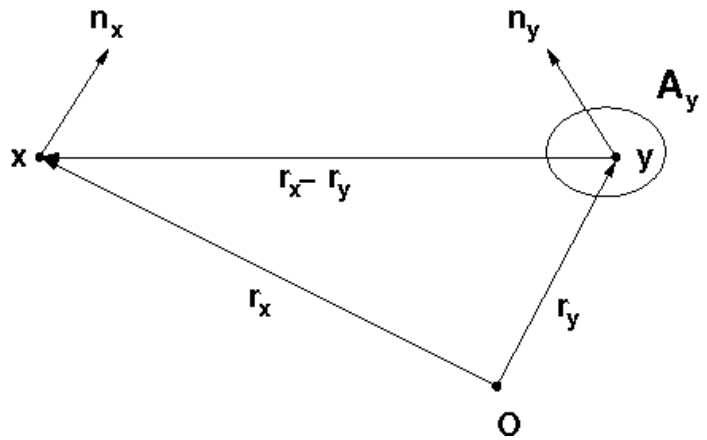


- Color(Pixel) = Direct Color + Reflected Ray Color + Refracted Ray Color

Radiosity as a GI method

Rendering Equation for Radiosity between two points

$$B(x) = E(x) + \rho(x) \int_{A_y} \frac{[\vec{ny}.(\vec{rx} - \vec{ry})][\vec{nx}.(\vec{ry} - \vec{rx})]}{\pi|\vec{ry} - \vec{rx}|^4} B(y)dA_y$$



- **Illumination Maps**(IM) are color values at every point in the model, due to application of Radiosity as the GI algorithm.

The Fast Multipole Method (FMM)

FMM is concerned with evaluating the effect of a "set of sources" $\mathbb{Y}$, on a set of "evaluation points" $\mathbb{X}$.

More formally, given

$$\begin{aligned} \mathbb{X} &= \{x_1, x_2, \ldots, x_M\}, \quad x_i \in \mathbb{R}^3, \quad i = 1, \ldots, M, \\ \mathbb{Y} &= \{y_1, y_2, \ldots, y_N\}, \quad y_j \in \mathbb{R}^3, \quad j = 1, \ldots, N \end{aligned}$$

we wish to evaluate the sum

$$f(x_i) = \sum_{j=1}^{N} \phi(x_i, y_j), \quad i = 1, \ldots, M$$

- Total complexity : $O(NM)$

The Fast Multipole Method

$$f(x_i) = \sum_{j=1}^{N} \phi(x_i, y_j), \quad i = 1, \ldots, M$$

- The FMM attempts to reduce this seemingly irreducible complexity to $O(N + M)$.
- The three main insights that make this possible are
  - **Factorization** of the kernel into source and reciever terms
  - Many application domains do not require that the function f be calculated at very high accuracy.
  - FMM follows a **hierarchical structure** (*Octree*)
- Each node has an associated **Interaction Lists**

FMM: Highly Parallel in Structure

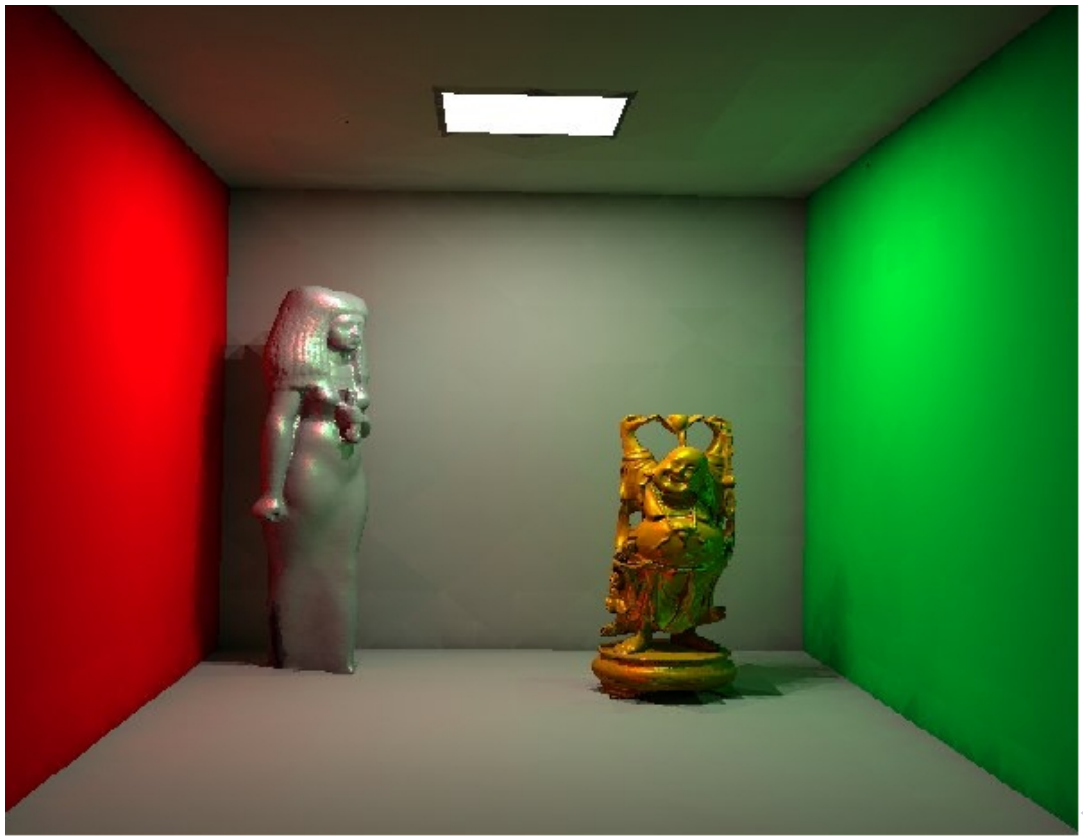
- Besides being very efficient (O(N) algorithm), the FMM is also highly parallel in structure.
- Thus implementing it on a parallel, high performance multi-processor cluster will further speedup the computation of diffuse illumination for our input point sampled scene.
- Our interest lies in a design of a parallel FMM algorithm that uses
  - Static decomposition
  - No explicit dynamic load balancing

FMM and Graphics Processing Units

- Graphics Processing Units (GPUs) are dedicated processors for graphics computations
- Current GPUs have tremendous memory bandwidth, computational power (*much* faster than CPUs), and are programmable
- Architecturally, GPUs are highly parallel streaming processors optimized for vector operations, with both MIMD and SIMD pipelines.
- Harnessing the power of GPUs for general-purpose computing (GPGPU)
- FMM : *One such algorithm*

Visibility Between Point Pairs

Visibility calculation between point pairs is **essential** to give *correct* GI results as a point recieves energy from other point only if it is **visible**



Visibility Between Point Pairs

Its complicated in our case !! Why ?

- Our input data set is a point based model with *no connectivity* information
- Thus, we do not have knowledge of any intervening surfaces occluding a pair of points.
- Theoretically, it is therefore impossible to determine exact visibility between a pair of points.
- We, thus, restrict ourselves to **approximate visibility**.

Application Domains

Problem Statement Revisited

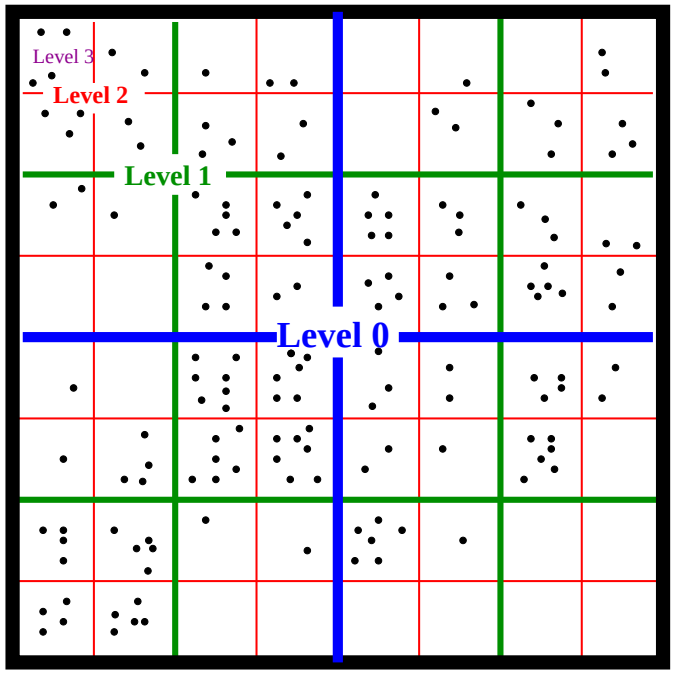
Problem Statement

- **Capturing interreflection effects in a scene when the inputs are point samples of hard to segment entities**
- Mutual visibility between point pairs: A necessary step for achieving correct global illumination effects. **(Done)**
- Inter-reflection effects include both diffuse and specular effects like reflections, refractions, and caustics.
  - We compute diffuse inter-reflections using the Fast Multipole Method(FMM) for radiosity kernel. **(Done)**
  - We desire parallel implementation of visibility and FMM algorithms on GPUs for faster diffuse global illumination solution. **(To be done)**
  - Specular inter-reflections are computed using ray-tracing and caustic map generation. **(To be done)**

Outline

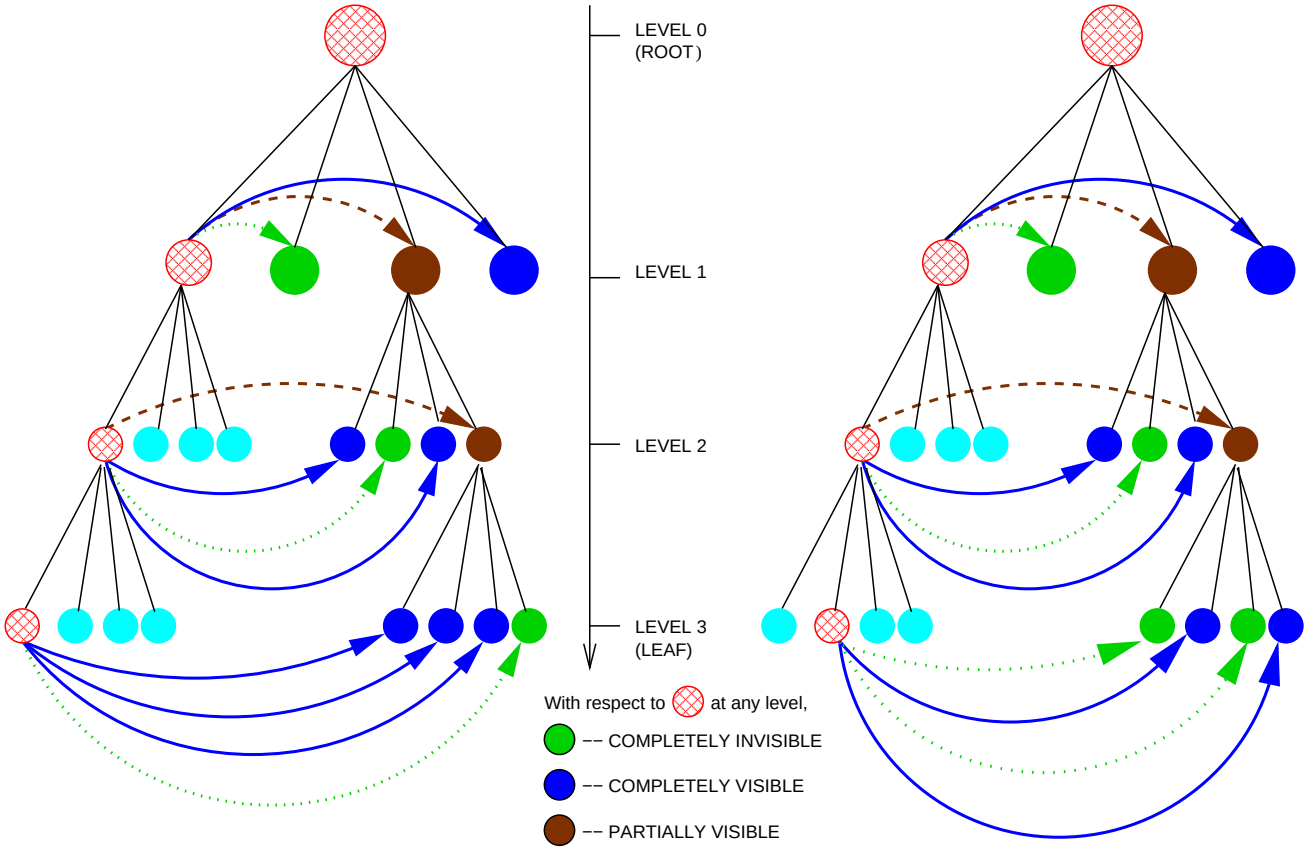
What are Visibility Maps (V-Maps)?

- Assumes a hierarchy is given (Octrees).



- The *visibility map* for a tree is a collection of visibility links for every node in the tree. The *visibility link* for any node p is a list L of nodes; every point in any node in L is guaranteed to be visible from every point in p

What are Visibility Maps (V-Maps)?



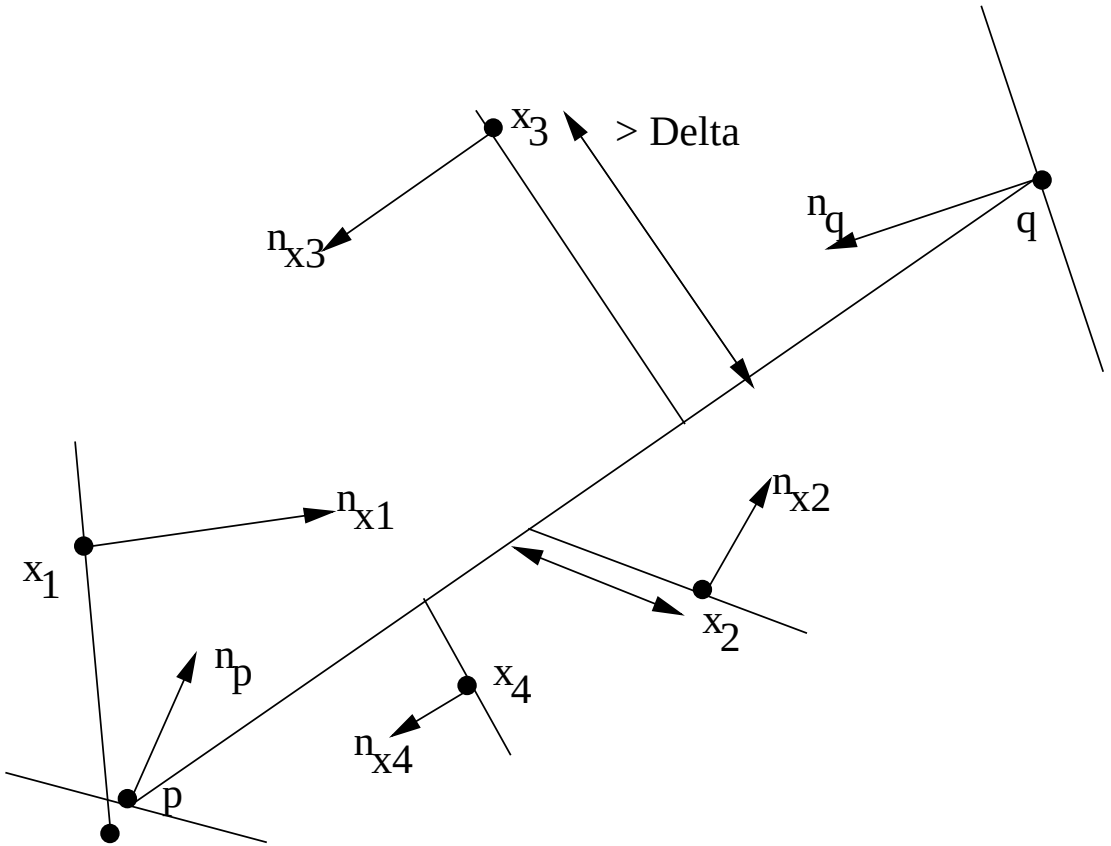
Visibility Maps Queries?

Visibility maps entertain efficient answers to the following queries.

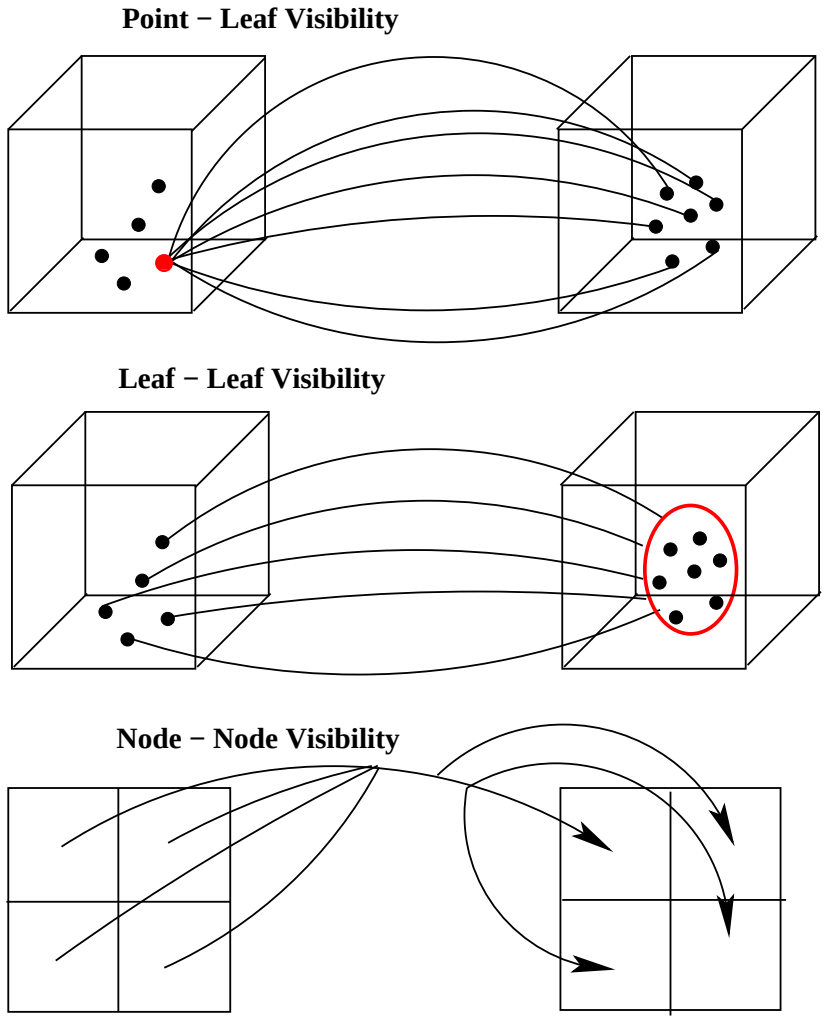
1. Is point x visible to point y?
2. What is the visibility status of u points around x with respect to v points around y?
   - Repeat a "primitive" point-point visibility query uv times
   - V-Map gives the answer with $O(1)$ point-point visibility queries.
3. Given a point x and a ray R, determine the first object of intersection.
4. Is point x in the shadow (umbra) of a light source?

All the above queries are done with a simple traversal of the octree.

Previous Approach: Basic Point-Pair Visibility



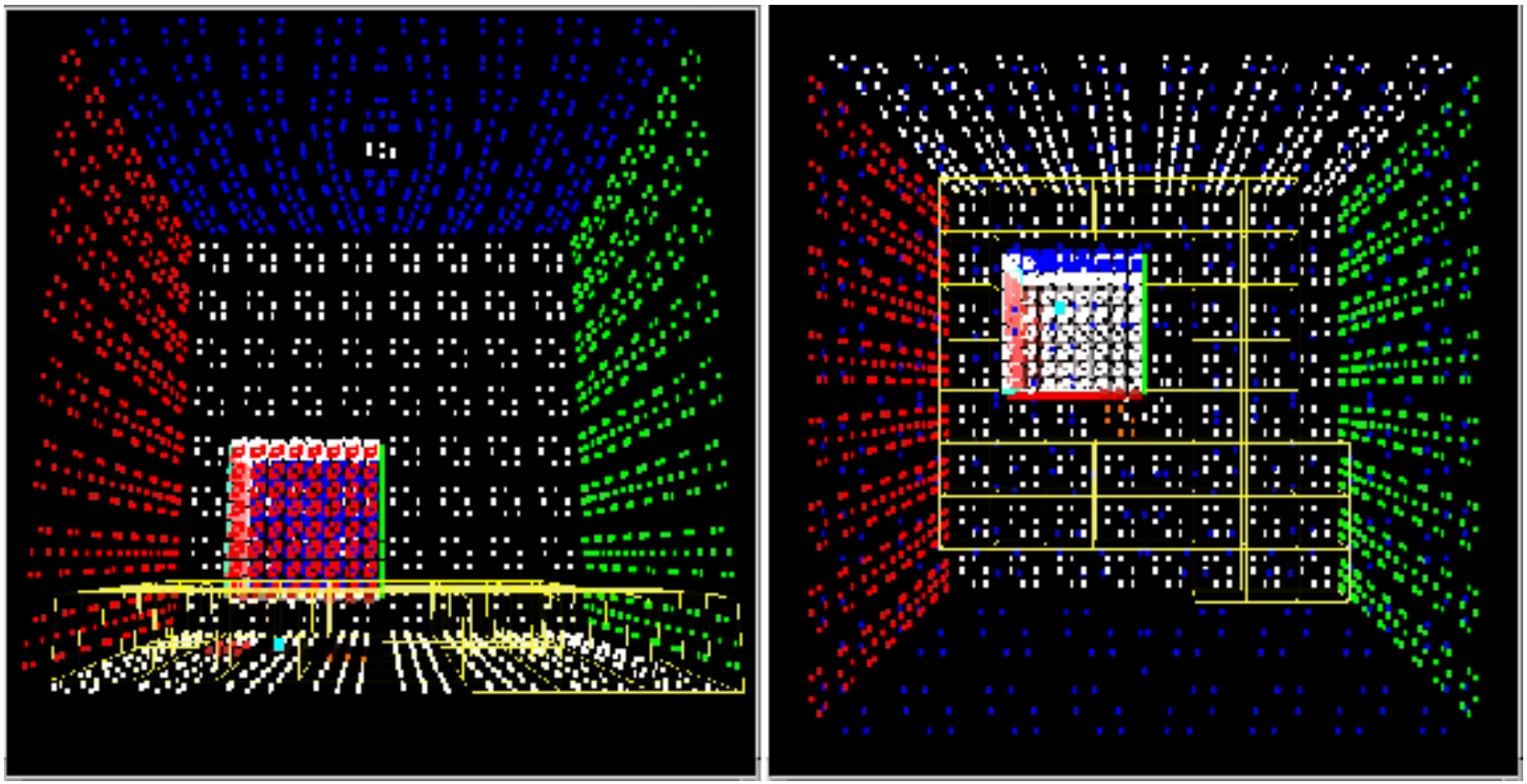
Previous Approach: Hierarchical Visibility Algorithm



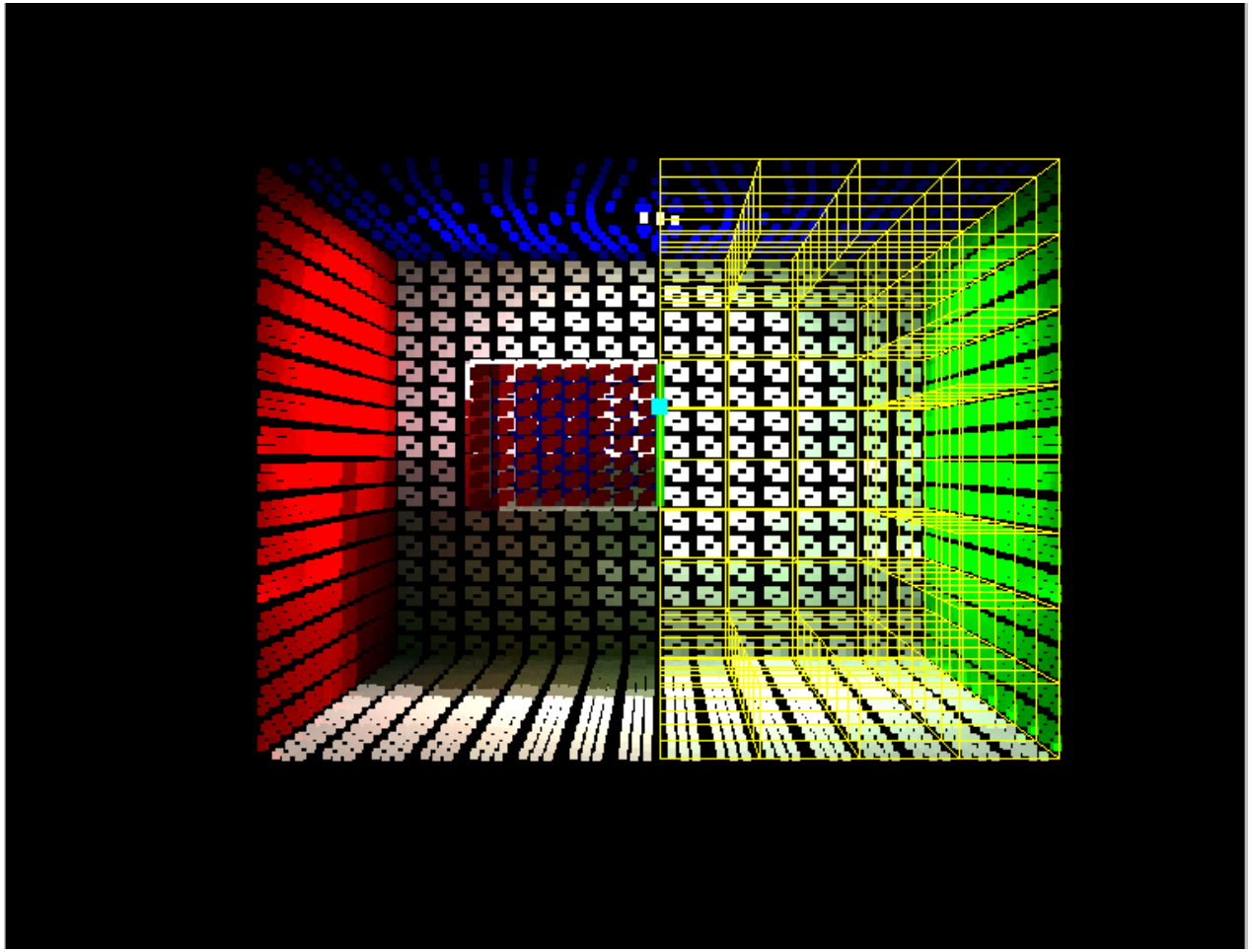
Previous Approach: V-Map Construction Algorithm

```
procedure OctreeVisibility(Node A)
for each node B ∈ interactionlist(A) do
  if notLeaf(A) then
    state=NodetoNodeVisibility(A,B)
  else if Leaf(A) then
    state=LeaftoLeafVisibility(A,B)
  end if
  if equals(state,valid) then
    Retain B in interactionlist(A)
  else if equals(state,partial) then
    for each a ∈ children(A) do
      for each b ∈ children(B) do
        interactionlist(a).add(b)
      end for
    end for
    interactionlist(A).remove(B)
  else if equals(state,invalid) then
    interactionlist(A).remove(B)
  end if
end for
for each R ∈ child(A) do
  OctreeVisibility(R)
```

Correctness of Visibility



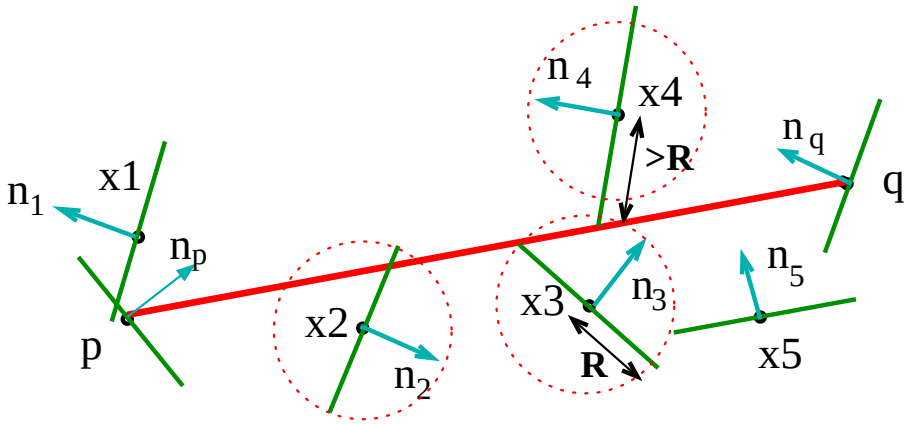
Correctness of Visibility



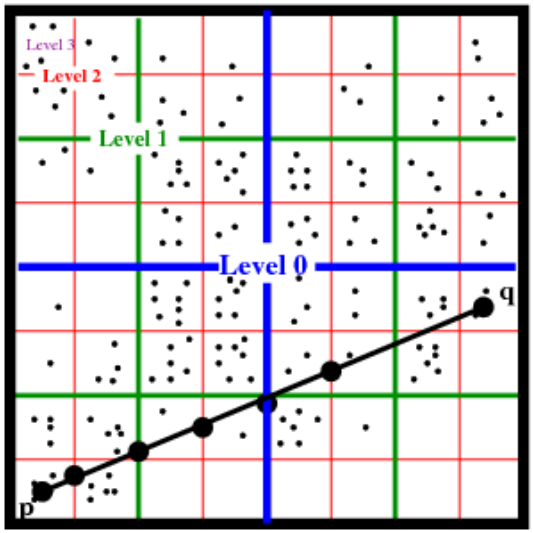
Limitations

- Three different thresholds
- Threshold values are dependent on the input scene complexity
- Finding the k nearest occluders is time consuming
- In the point–pair visibility algorithm, we dont use any conditions which helps us to exit instantaneously as soon as an invisibility case is detected.
- Extra computations performed in case of *partial visibility* case. Introduces a factor of atleast $O(logN)$
- Was implemented for *non-adaptive octrees*

Point Pair Visibility for Non-Adaptive hierarchy

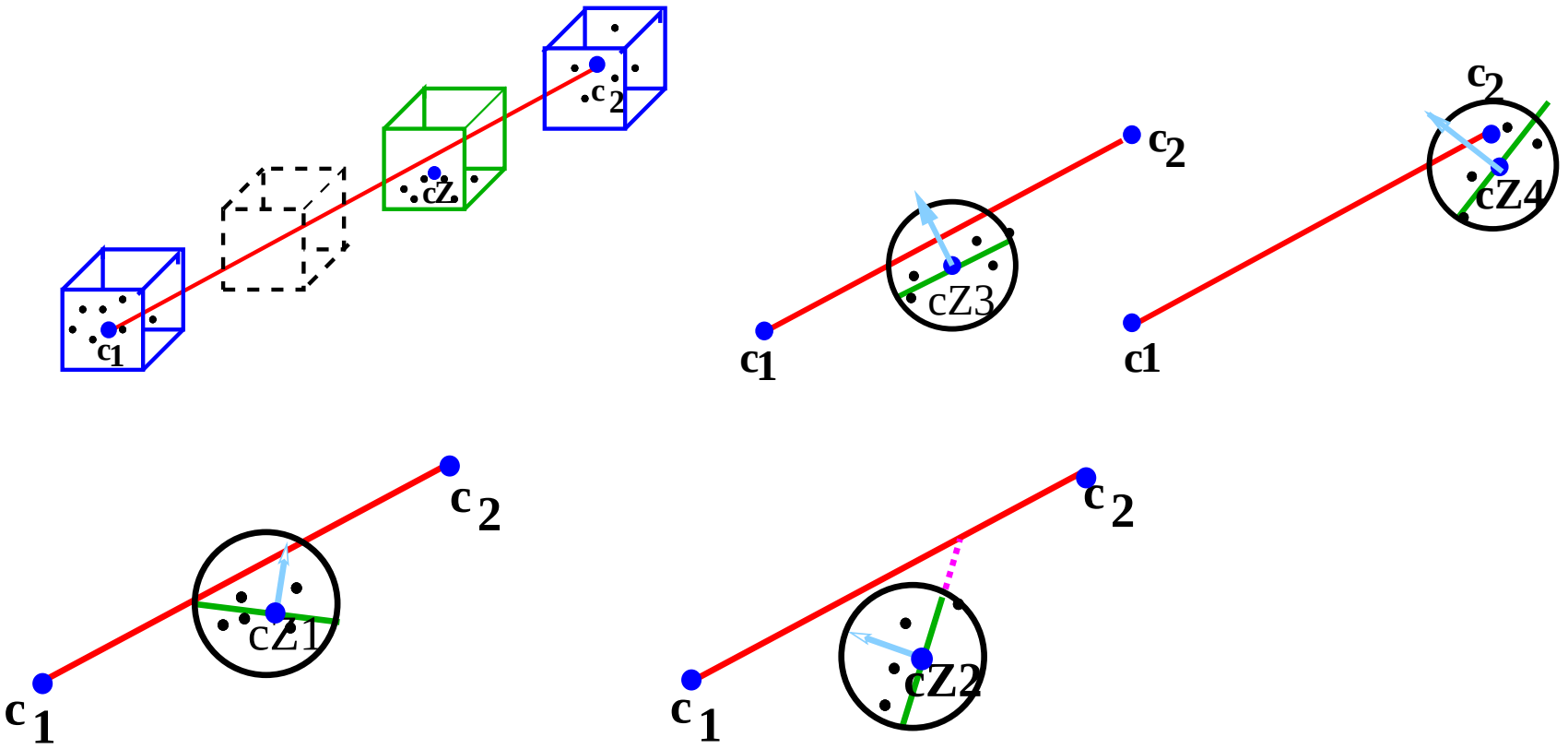


Visibility between points p and q



Finding Potential Occluders using bresenham line algorithm

New Approach: Leaf-Leaf Visibility Algorithm



New Approach: V-Map Construction Algorithm

```
procedure OctreeVisibility(Node A)
for each node B in old interaction list (o-IL) of A do
   if NodetoNodeVisibility(A,B) == VISIBLE then
      add B in new interaction list (n-IL) of A
      add A in new interaction list (n-IL) of B
   end if
   remove A from old interaction list (o-IL) of B
end for
for each C in children(A) do
   OctreeVisibility(C)
end for
```

- Initialize the o-IL of every node to be its seven siblings
- V-Map constructed by calling initially for the root, which sets up the relevant visibility links in n-IL
- NodetoNodeVisibility(A,B) constructs the visibility links for all descendants of A w.r.t all descendants of B (and vice-versa) at the best (i.e. highest) possible level.

New Approach: V-Map Construction Algorithm

procedure ***NodeToNodeVisibility***(Node A)
if A and B are leaf **then**
  return(LeafToLeafVisibility(A, B))
end if
Declare boolean matrix $M(children(A).size * children(B).size)$,count=0
for each a $\in$ children(A) **do**
  for each b $\in$ children(B) **do**
    state=***NodetoNodeVisibility***(a,b)
    if equals(state,visible) **then**
      Store *true* at corresponding location in M; $count = count + 1$
    end if
  end for
end for
if count == $s1 * s2$ **then**
  free M and return VISIBLE
else if count == 0 **then**
  free M and return INVISIBLE
end if

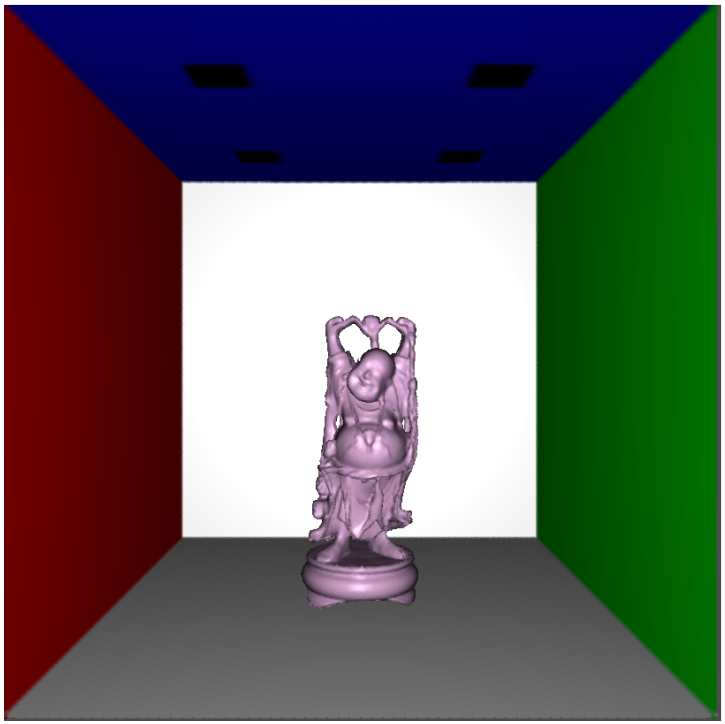
New Approach: V-Map Construction Algorithm

if count < $s1 * s2$ && count > 0 **then**
  for each a $\in$ children(A) **do**
    for each b $\in$ children(B) **do**
      Update n-IL of a w.r.t every *visible* child b (*simple* look up in M) & vice-versa,
      free M
    end for
  end for
  return PARTIAL.
end if

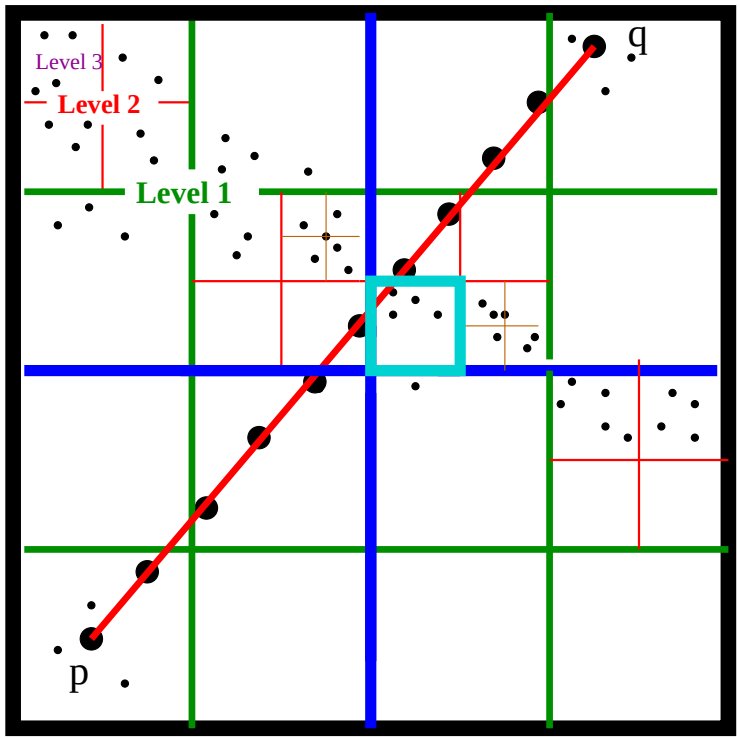
Computational Complexity

- Assume $N = \Theta(n^2)$, n = points in input model.
- Visibility problem provides answer to N pairwise queries. Hence we measure the efficiency w.r.t N
- Octree Visibility has the recurrence: $T(h) = 8T(h-1) + N$ (for a Node A at height h)
- Complexity for *NodetoNodeVisibility(A,B)* is determined by the calls to point-pair visibility algorithm
- Assuming the latter to be $O(1)$, the recurrence relation for the former is $T(h) = 64T(h-1) + O(1)$.
- The overall algorithm consumes a small amount of memory (for storing M) during runtime.

Limitations: Extending to Adaptive Octrees



Cornell room(160,000 points) with buddha model(534000 points)



Problem in finding Potential Occluders using bresenham line algorithm

Ray-Sphere Intersection Algorithm

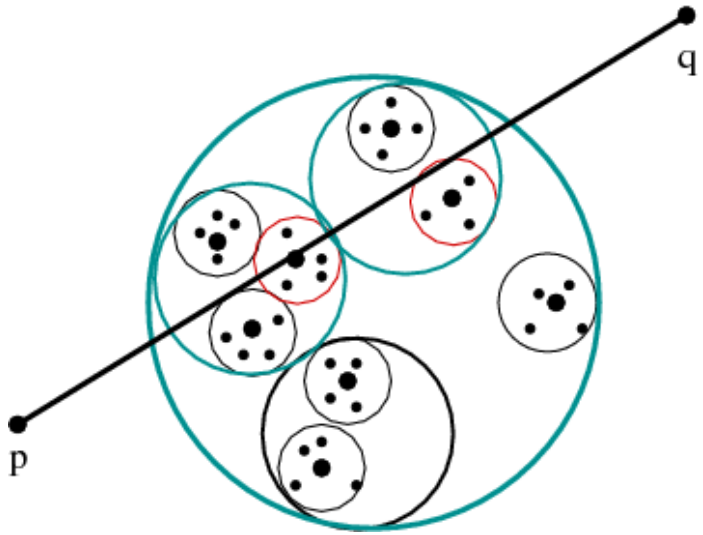
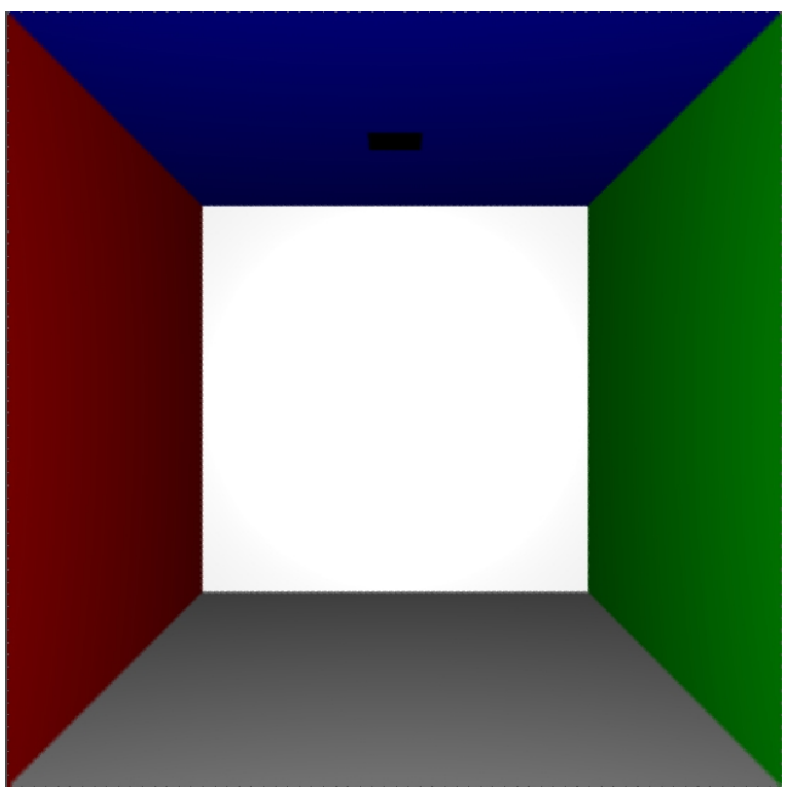
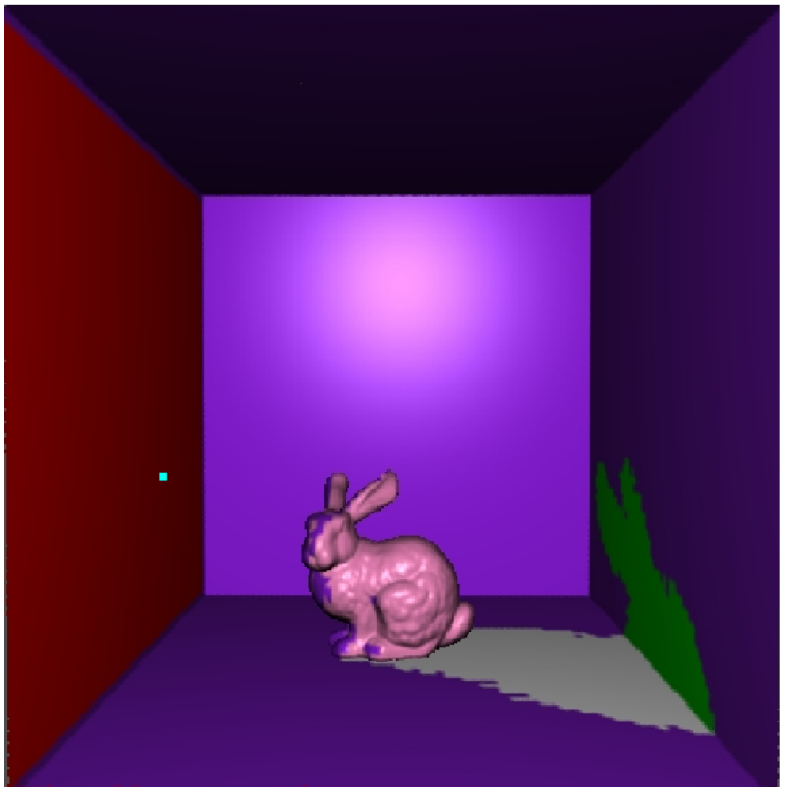


Figure: Ray-Sphere intersection algorithm to determine point-point visibility

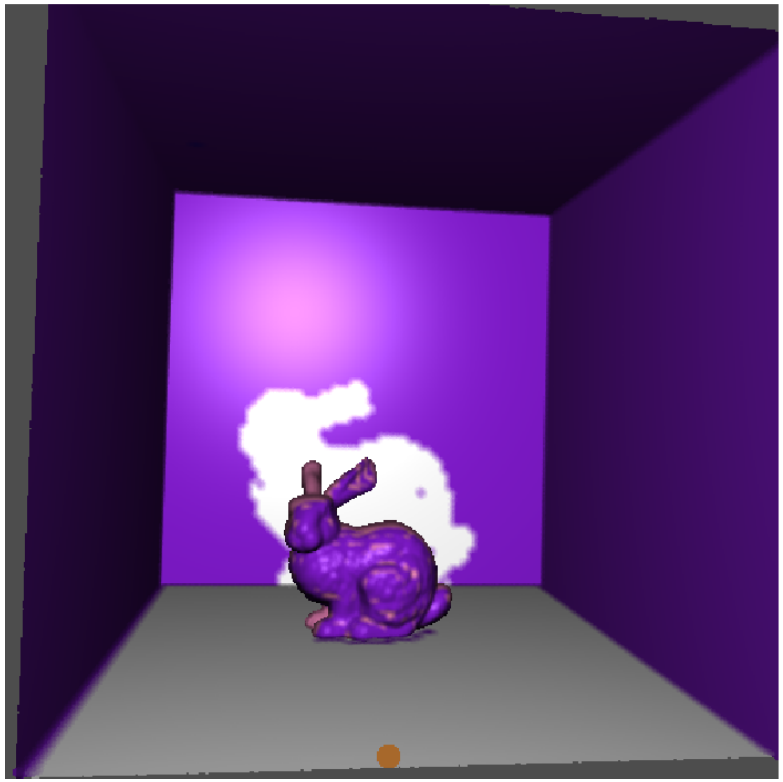
- If node is not a leaf and $\overline{pq}$ intersects the node then traverse its children
- If node is a leaf then check whether tangent plane of that node intersects pq within radius R then node p and q are invisible otherwise declare p and q visible

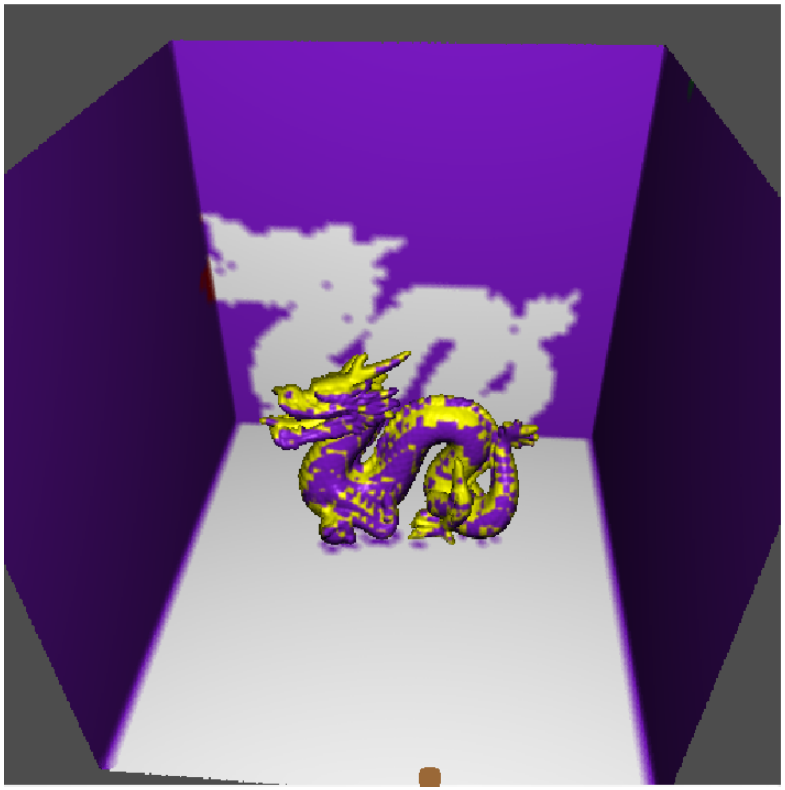
Qualitative Results: Adaptive V-Maps



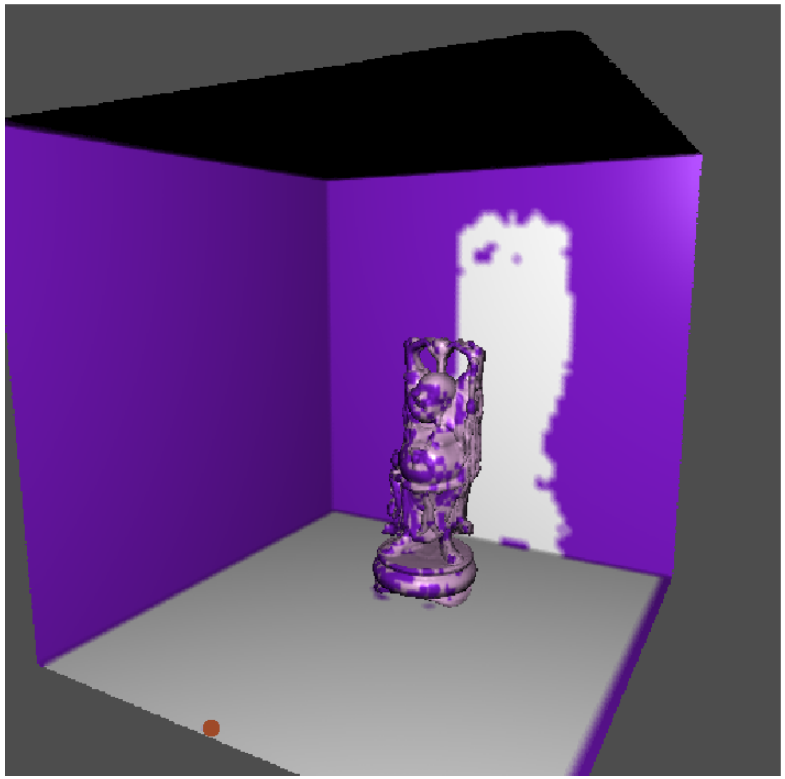


Qualitative Results: Adaptive V-Maps

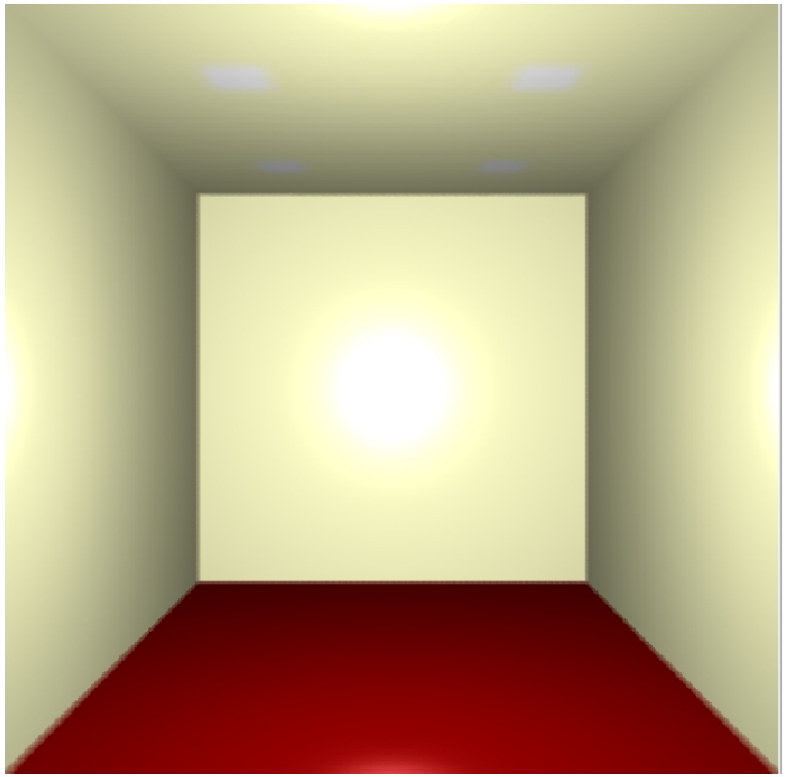


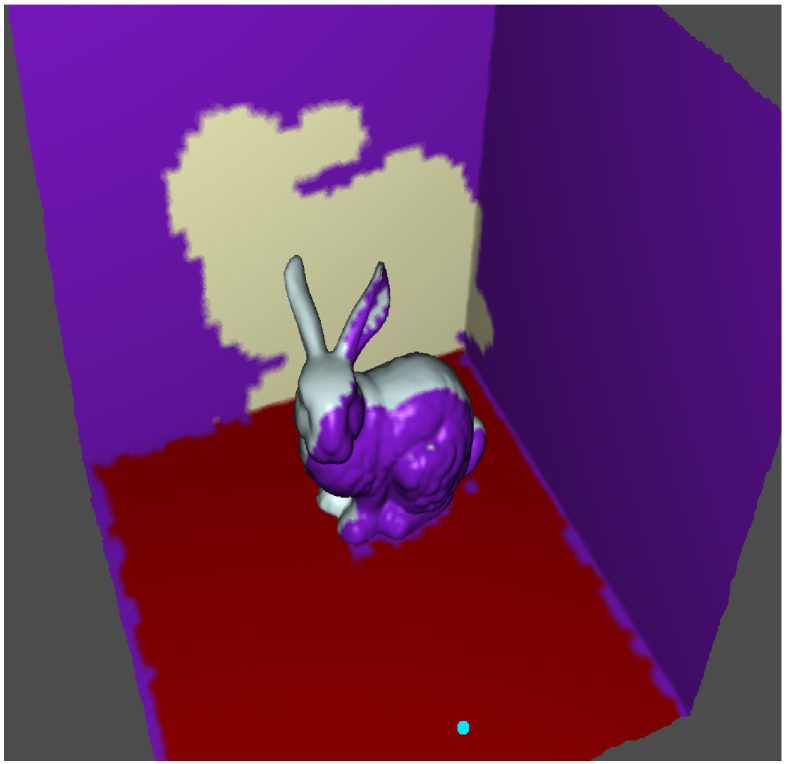


Qualitative Results: Adaptive V-Maps

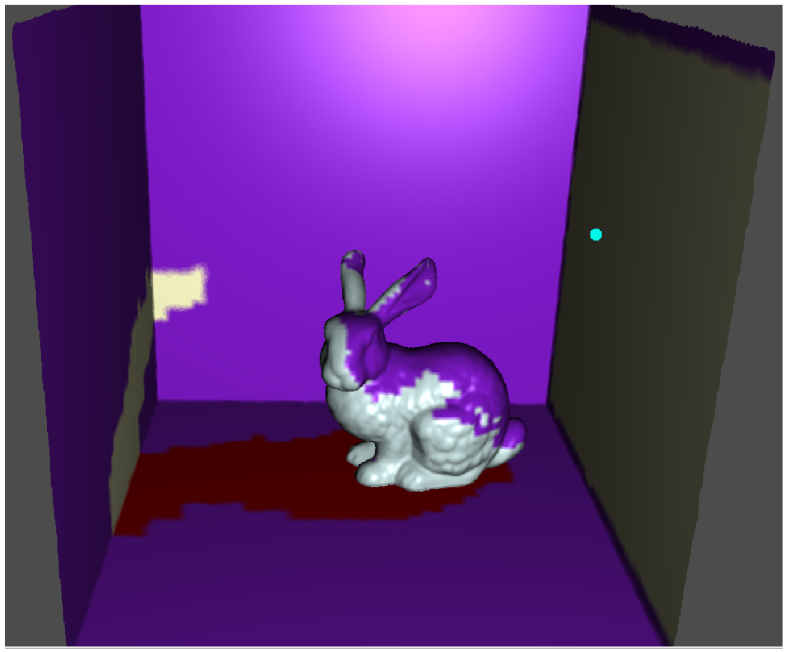


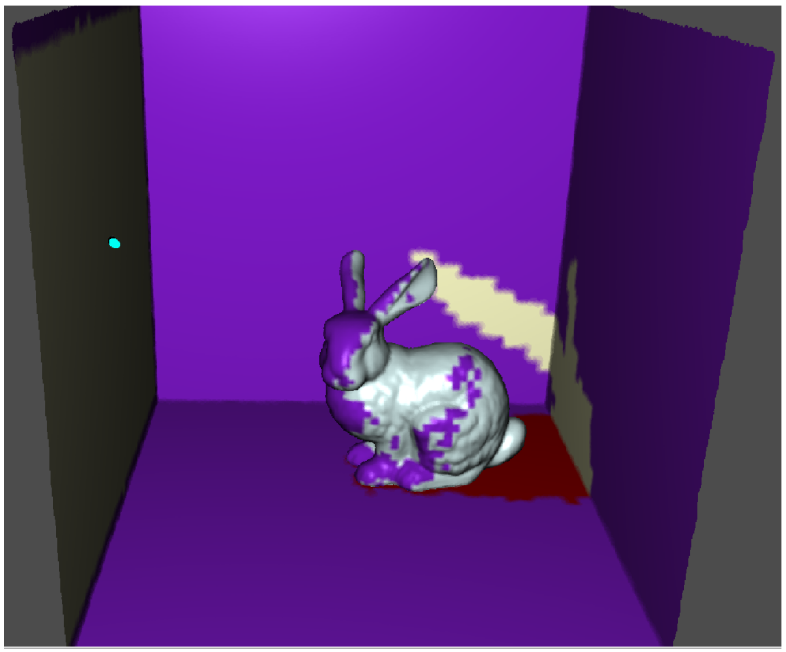
Qualitative Results: Adaptive V-Maps



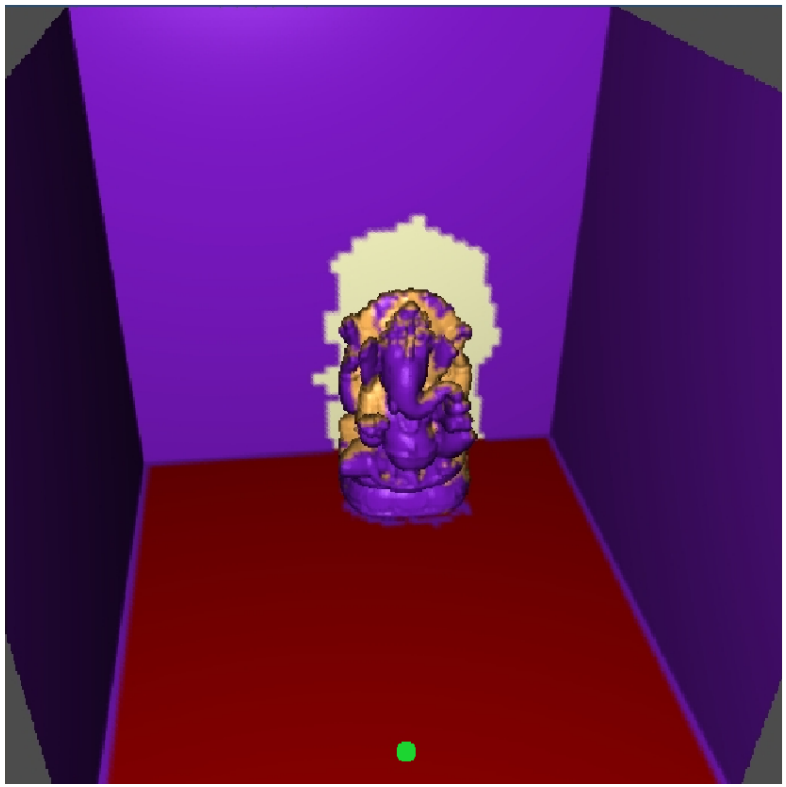


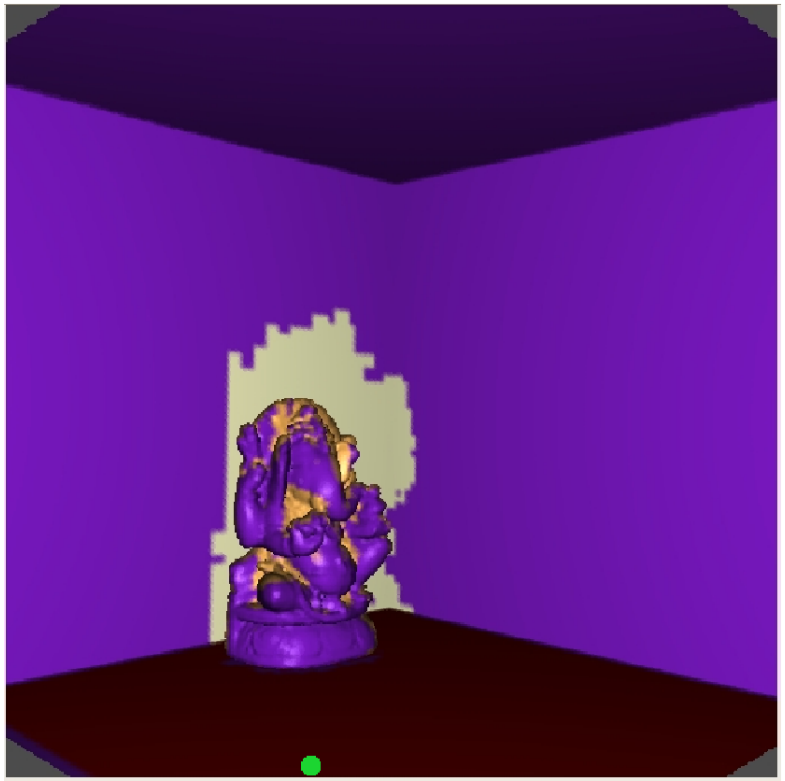
Qualitative Results: Adaptive V-Maps



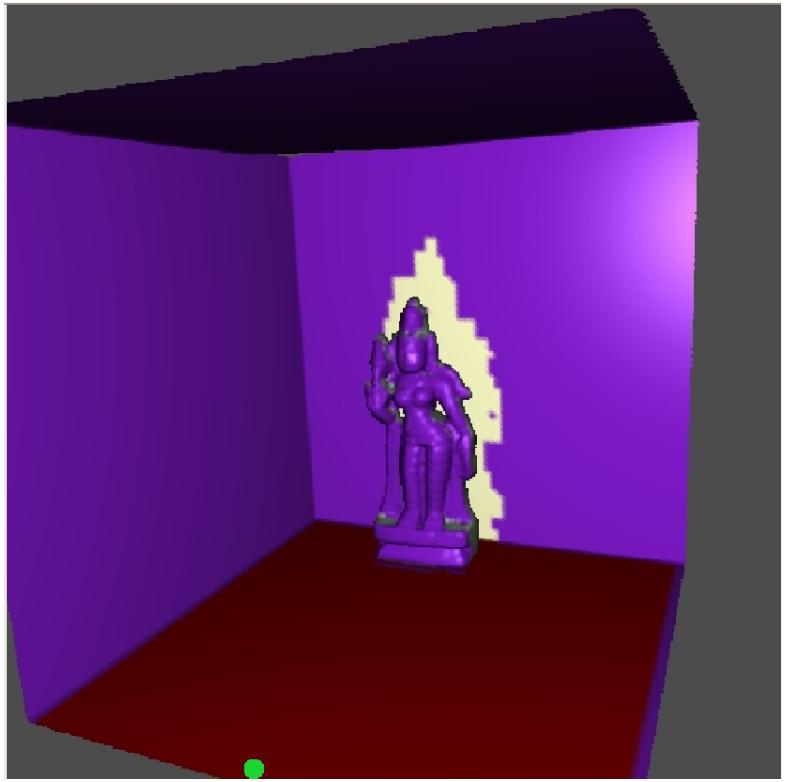


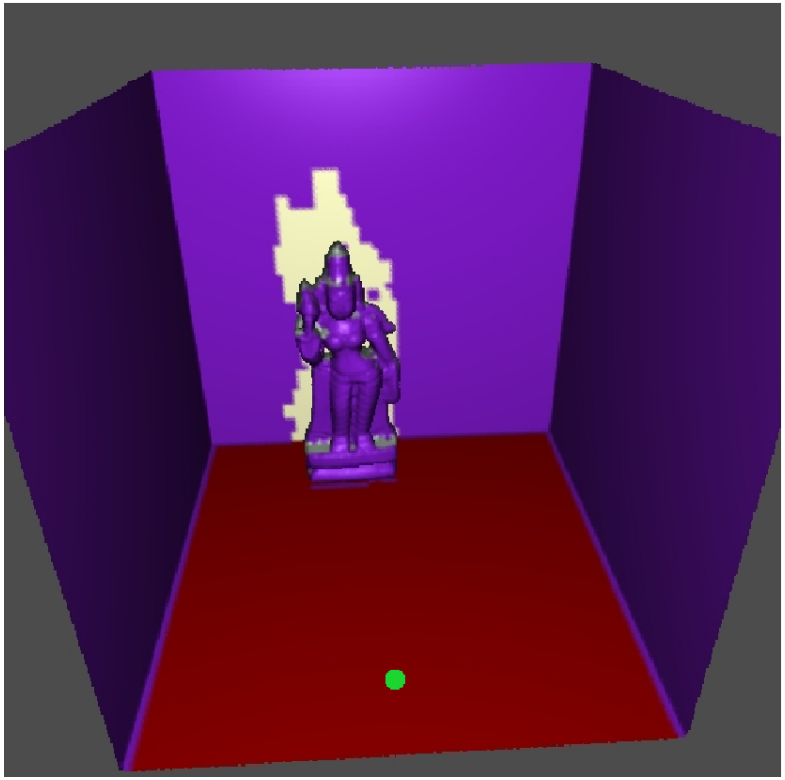
Qualitative Results: Adaptive V-Maps



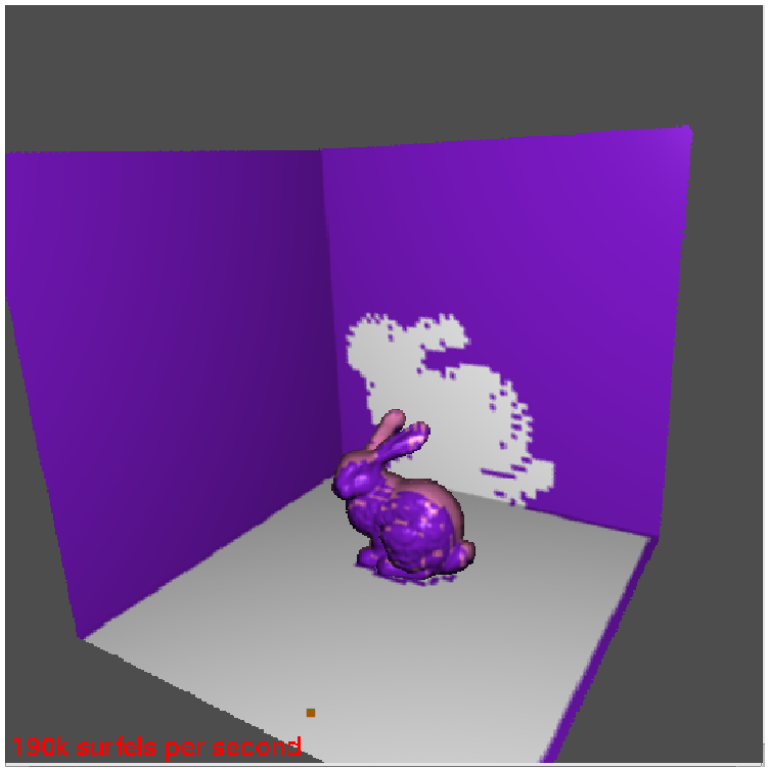


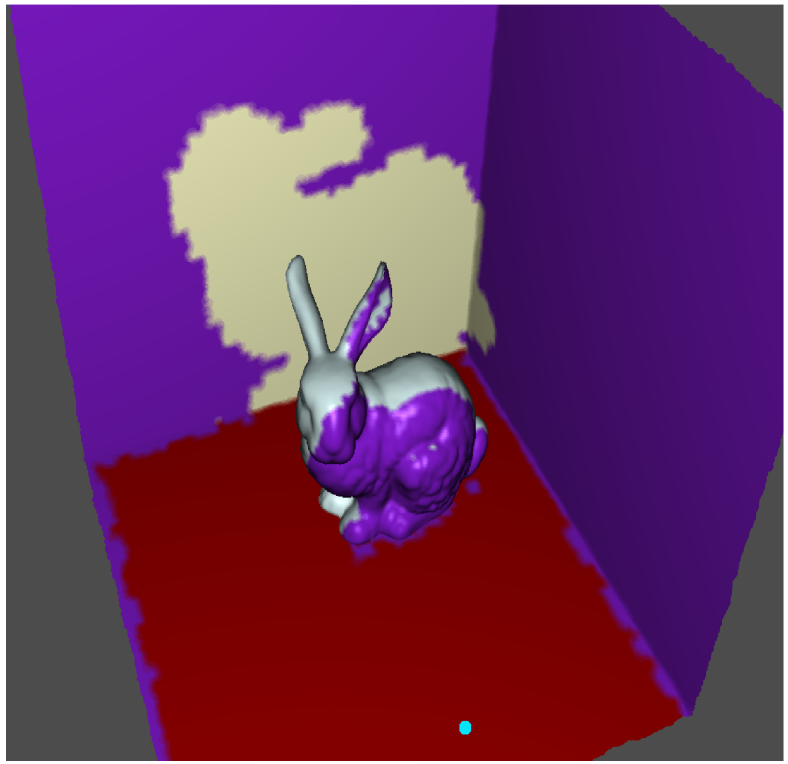
Qualitative Results: Adaptive V-Maps



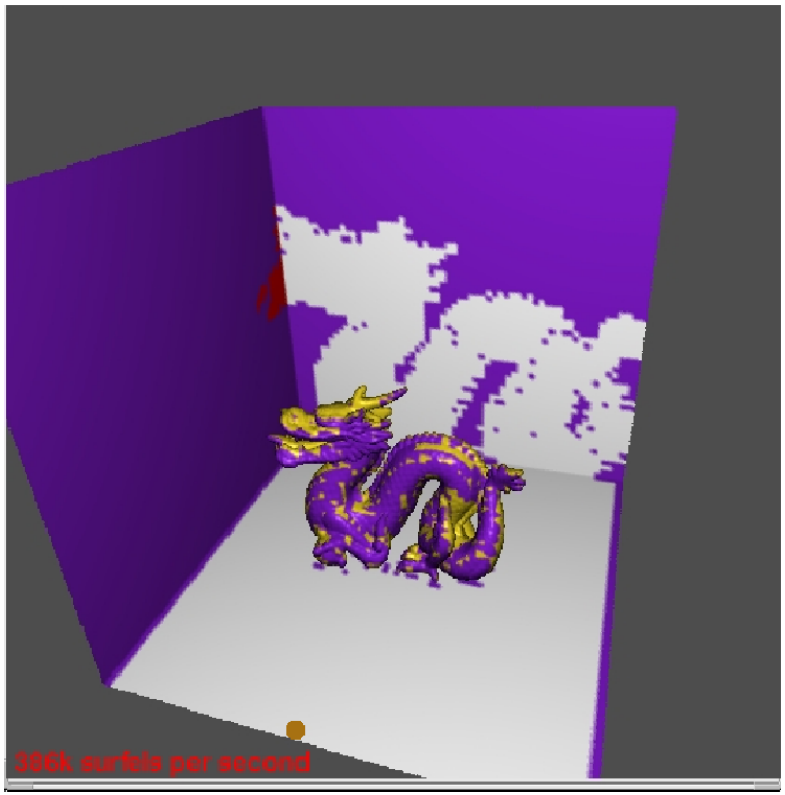


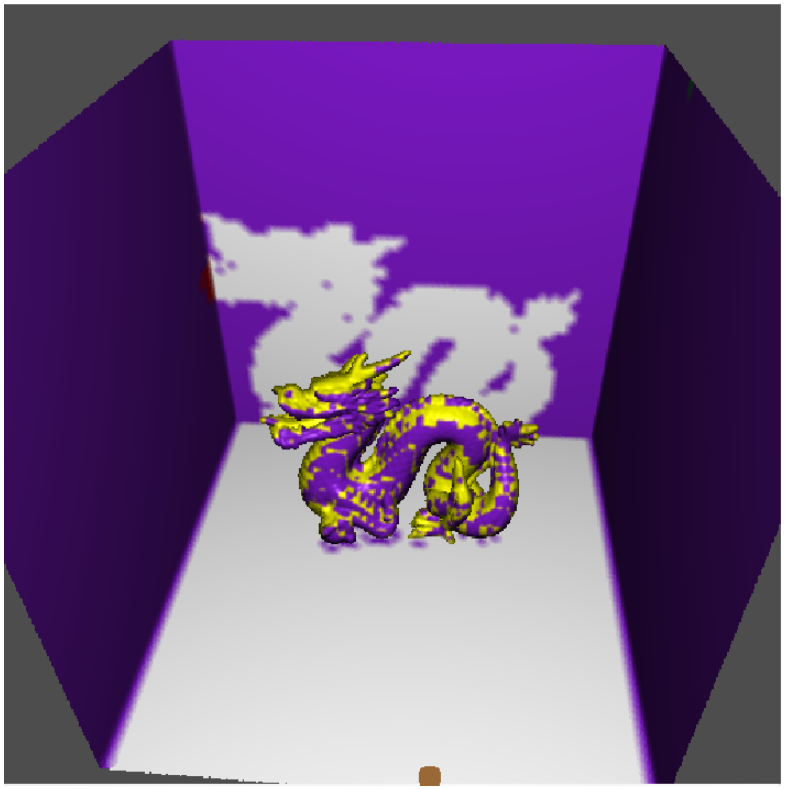
Qualitative Results: Comparisions



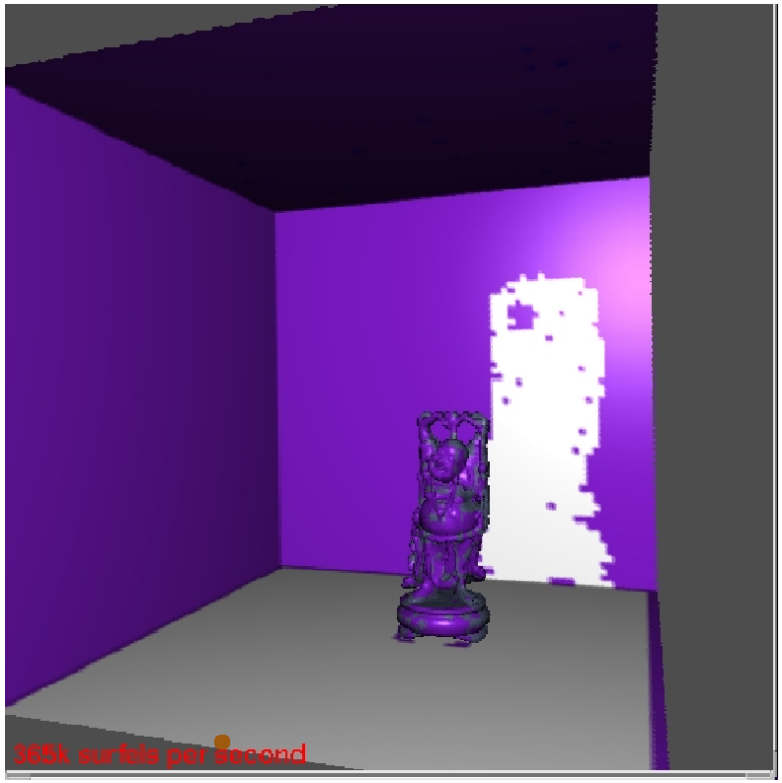


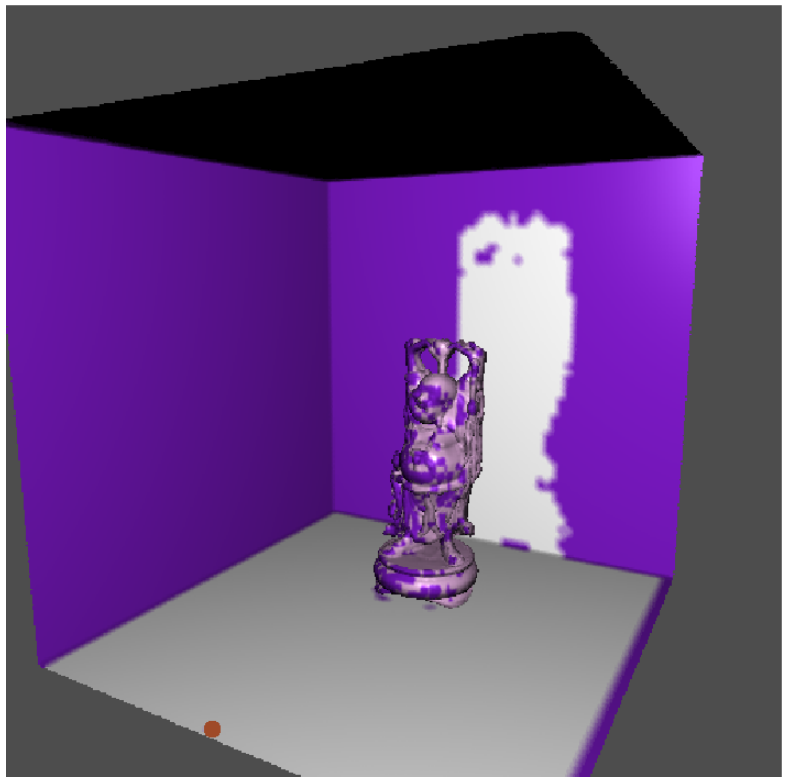
Qualitative Results: Comparisions



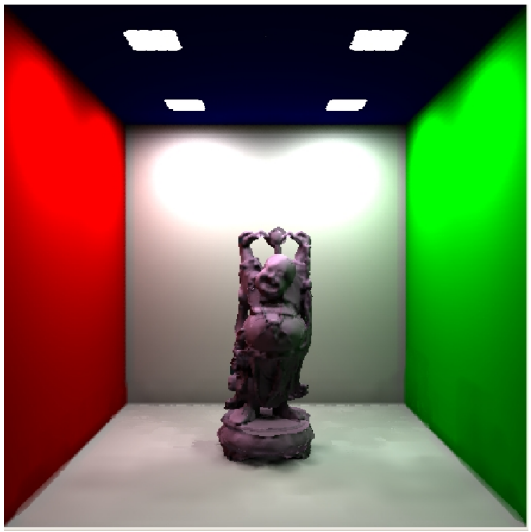


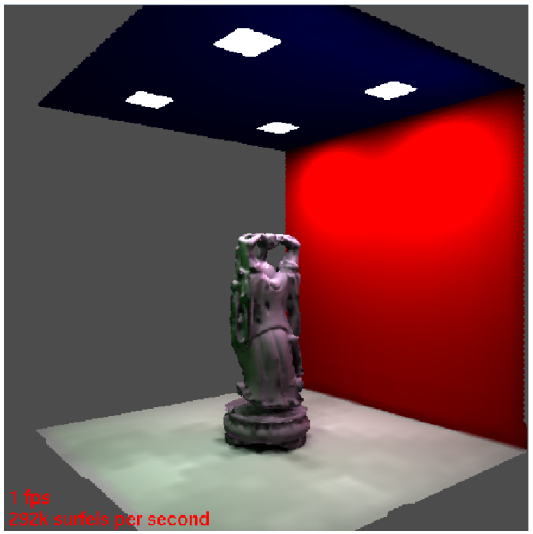
Qualitative Results: Comparisions

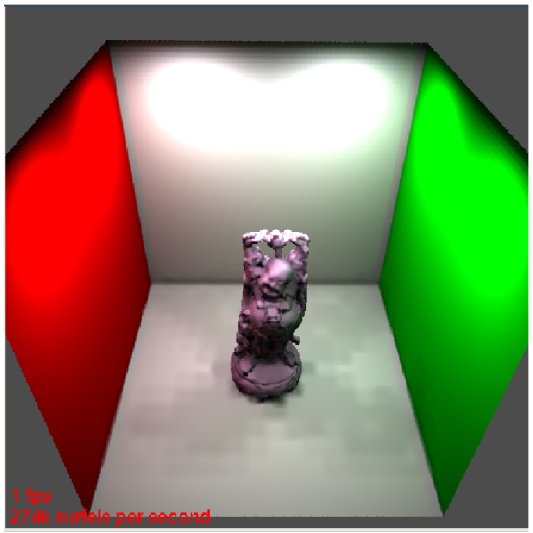




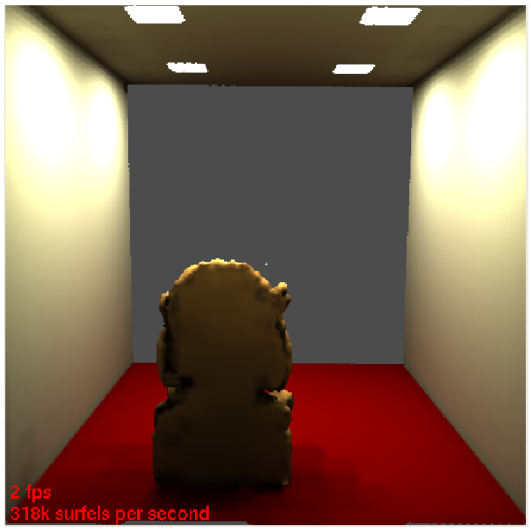
Qualitative Results: Global Illumination



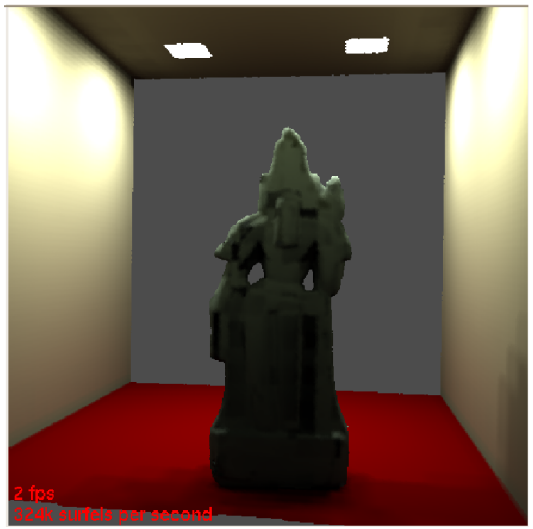




Qualitative Results: Global Illumination

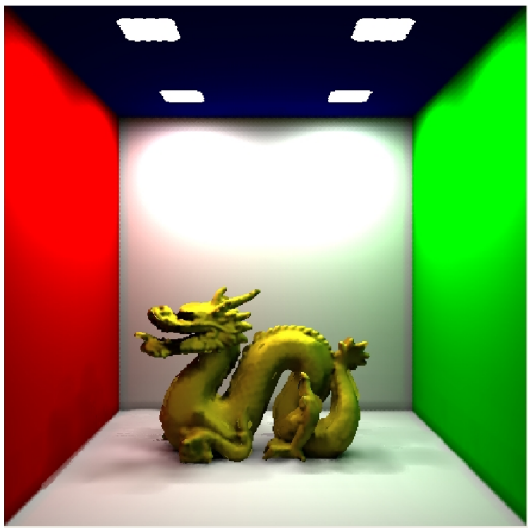


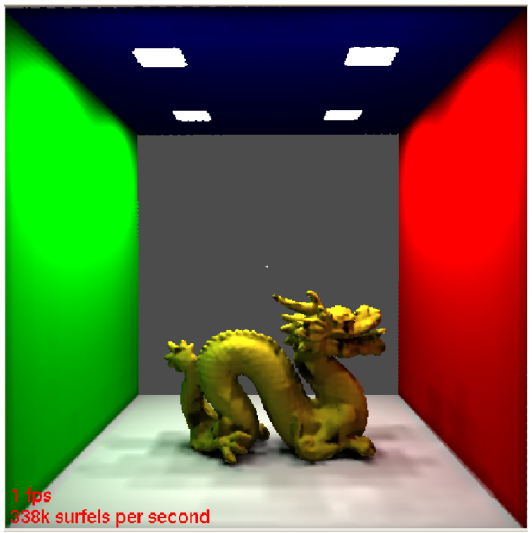
Qualitative Results: Global Illumination

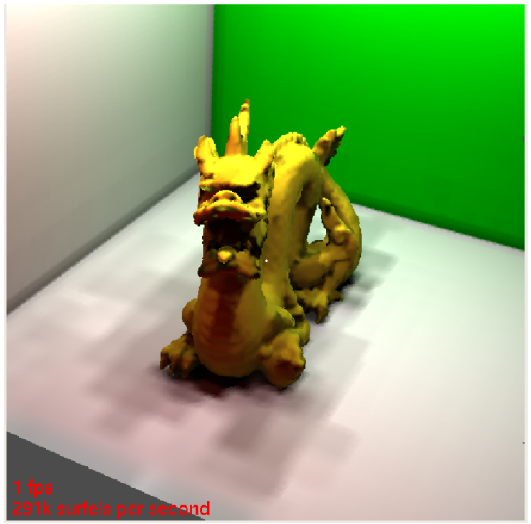


Qualitative Results: Global Illumination

Qualitative Results: Global Illumination







Quantitative Results

Model	Points (millions)	N^2 possible links (millions)	V-Map Links (millions)	% Decrease	Memory(MB) N^2 links	Memory(MB) V-Map links	Build V-Map Time(mins)
ECR	0.1	1.4	0.27	79.5%	5.35	1.09	20.6
PCR	0.14	3.85	0.67	82.62%	15.43	2.68	23.8
BUN	0.15	1.53	0.38	74.64%	6.09	1.5	21.7
DRA	0.55	2.75	0.43	84.54%	11.0	1.7	23.5
BUD	0.67	1.58	0.39	74.75%	6.33	1.6	23.9
GAN	0.15	1.56	0.38	75.64%	6.2	1.55	22.0
GOD	0.17	1.62	0.4	75.31%	6.4	1.63	22.9

Discussion

- The models are not very detailed because of a pre-processing step of point-based simplification
- Also, some roughness and discretization appears due to low level of subdivision used (Octree is divided roughly till **level 7**)
- More sub-division can be done but the processing time increases quite a bit (and hence we require a faster, parallel FMM radiosity kernel solver on GPU so as not to trade off quality for time)
- The FMM solution takes approximate **2 hours per iteration**. We make **3** iterations to get a complete GI solution
- The V-Map computation takes approximately **20-25 minutes** for about a million points, which can further be improvised when implemented in parallel (if analyzed properly, the visibility algorithm is embarrassingly parallel)
- Note that the non-adaptive version with 6 levels in the octree took more than **10 hours** for the Visibility Map computations

Discussion

- The speed of **visibility algorithm** can further be improved if we exploit its **highly parallel** structure
- Visibility algorithm can be implemented on a **GPU**, a highly parallel multi-processor unit **(Details are to be worked out)**
- The timings for global illumination solution via FMM can also be improved if **FMM can be implemented in parallel**
- We can use GPU for the same

Other Related Work

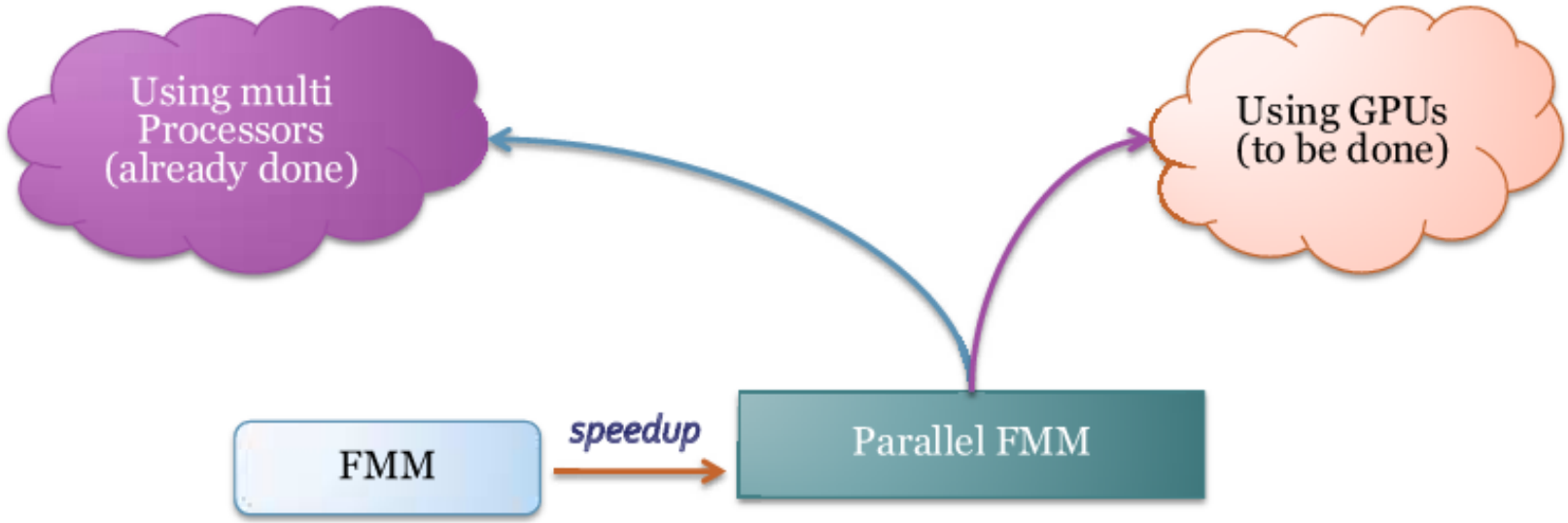
- Worked on **Point Based Simplification** algorithm
- **Smoothing** of results while rendering by effectively detecting points lying in the penumbra and umbra regions from the light sources (Still to be improved on)
- Created new point models
- Moving onto the **adaptive** division of octrees from the *non-adaptive* division was difficult as many things from FMM and Visibility parts needed to be changed

Outline

Problem Definition

Problem Definition

To implement Parallel Fast Multipole Method (FMM) on Graphics Hardware



The Fast Multipole Method (FMM)

FMM is concerned with evaluating the effect of a "set of sources" $\mathbb{Y}$, on a set of "evaluation points" $\mathbb{X}$.

More formally, given

$$\begin{aligned} \mathbb{X} &= \{x_1, x_2, \ldots, x_M\}, \quad x_i \in \mathbb{R}^3, \quad i = 1, \ldots, M, \\ \mathbb{Y} &= \{y_1, y_2, \ldots, y_N\}, \quad y_j \in \mathbb{R}^3, \quad j = 1, \ldots, N \end{aligned}$$

we wish to evaluate the sum

$$f(x_i) = \sum_{j=1}^{N} \phi(x_i, y_j), \quad i = 1, \ldots, M$$

- Total complexity : $O(NM)$

The Fast Multipole Method

$$f(x_i) = \sum_{j=1}^{N} \phi(x_i, y_j), \quad i = 1, \ldots, M$$

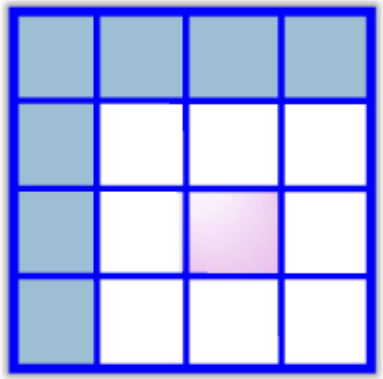
- The FMM attempts to reduce this seemingly irreducible complexity to $O(N + M)$.
- The three main insights that make this possible are
  - **Factorization** of the kernel into source and reciever terms
  - Many application domains do not require that the function f be calculated at very high accuracy.
  - FMM follows a **hierarchical structure** (*Octree*)
- Each node has an associated **Interaction Lists**

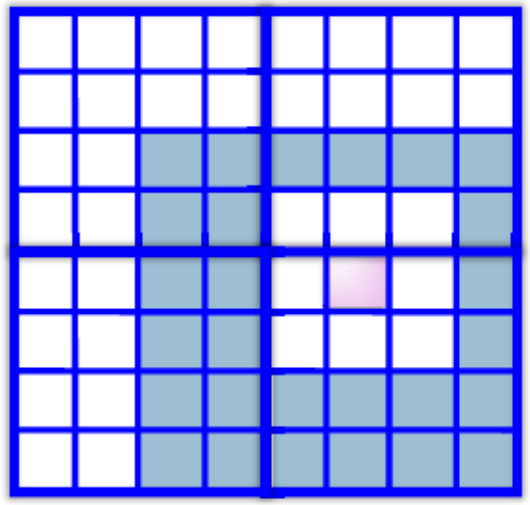
FMM Algorithm: Building Interaction Lists

Building Interaction Lists

Each node has two kind of interaction lists

- Far Cell List
- Near Cell List

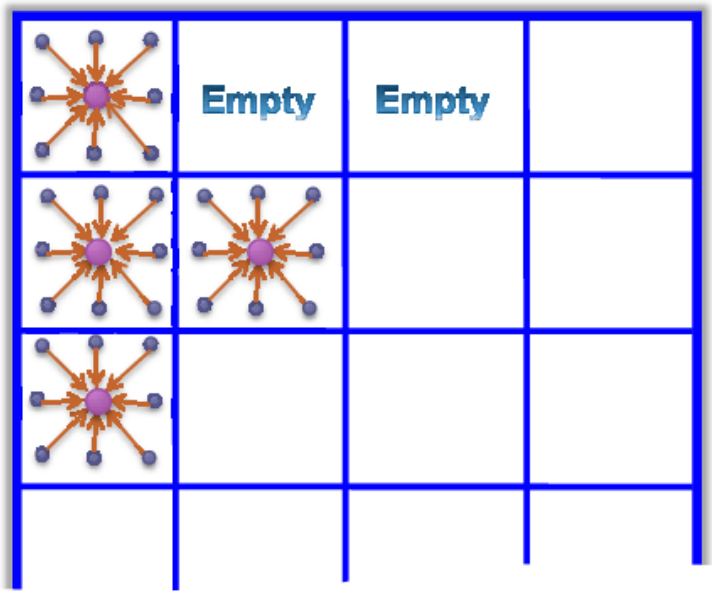




- No far cell list at level 1 and level 0 since everything is near neighbor of other
- Transfer of energy from near neighbors happens only for leaves

FMM Algorithm: Upward Pass

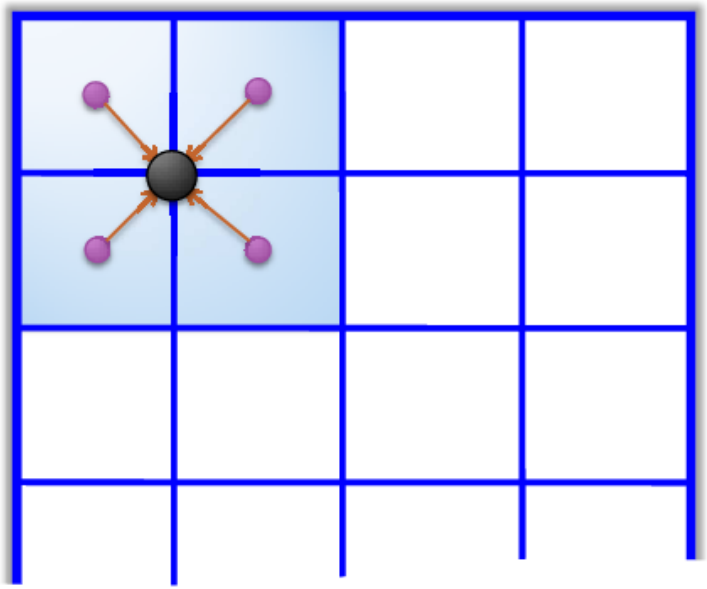
FMM Algorithm
Upward pass Step 1



CALCULATE MULTIPOLE EXPANSION AT EACH LEAF'S CENTER

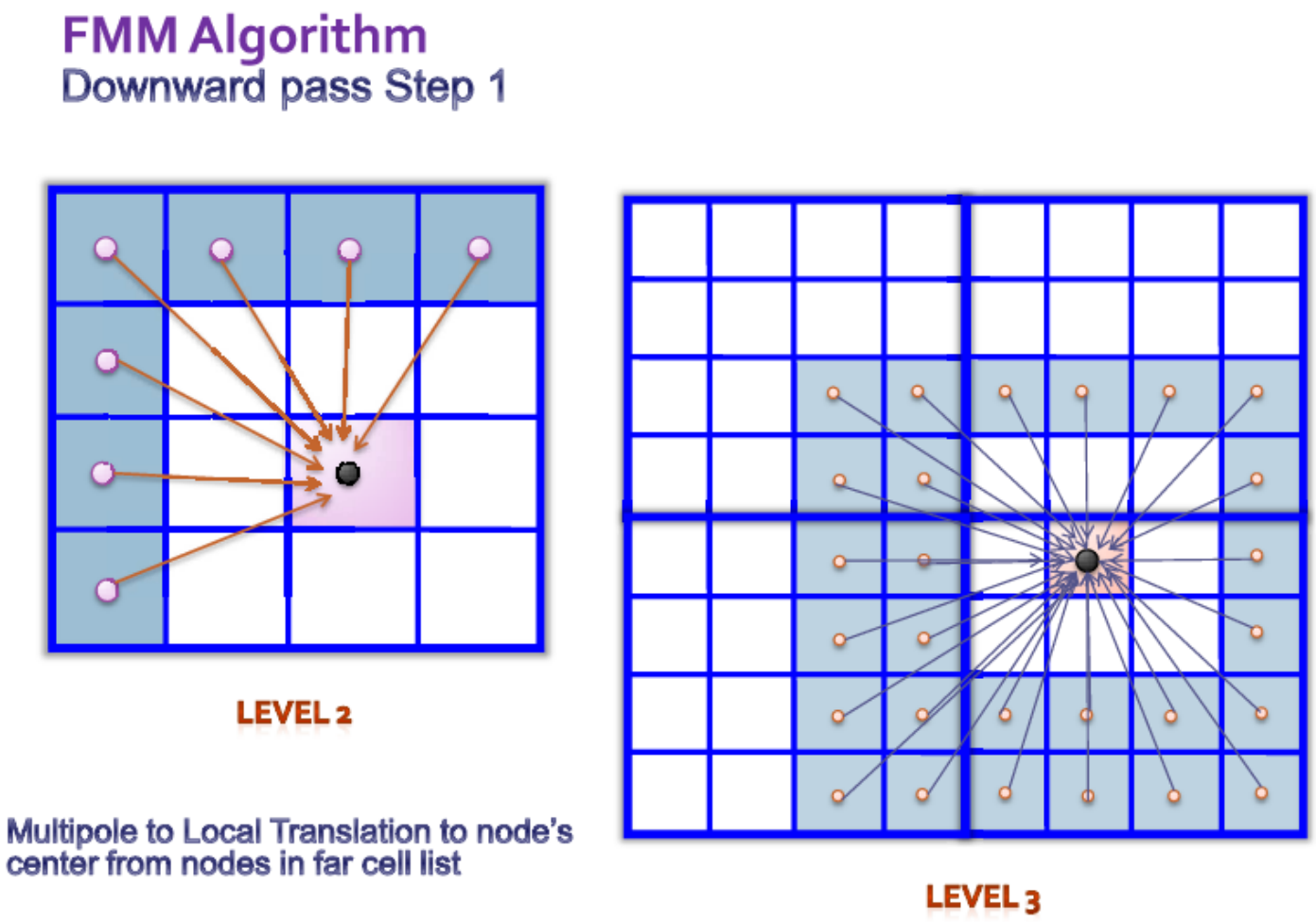
FMM Algorithm: Upward Pass

FMM Algorithm
Upward pass Step 2

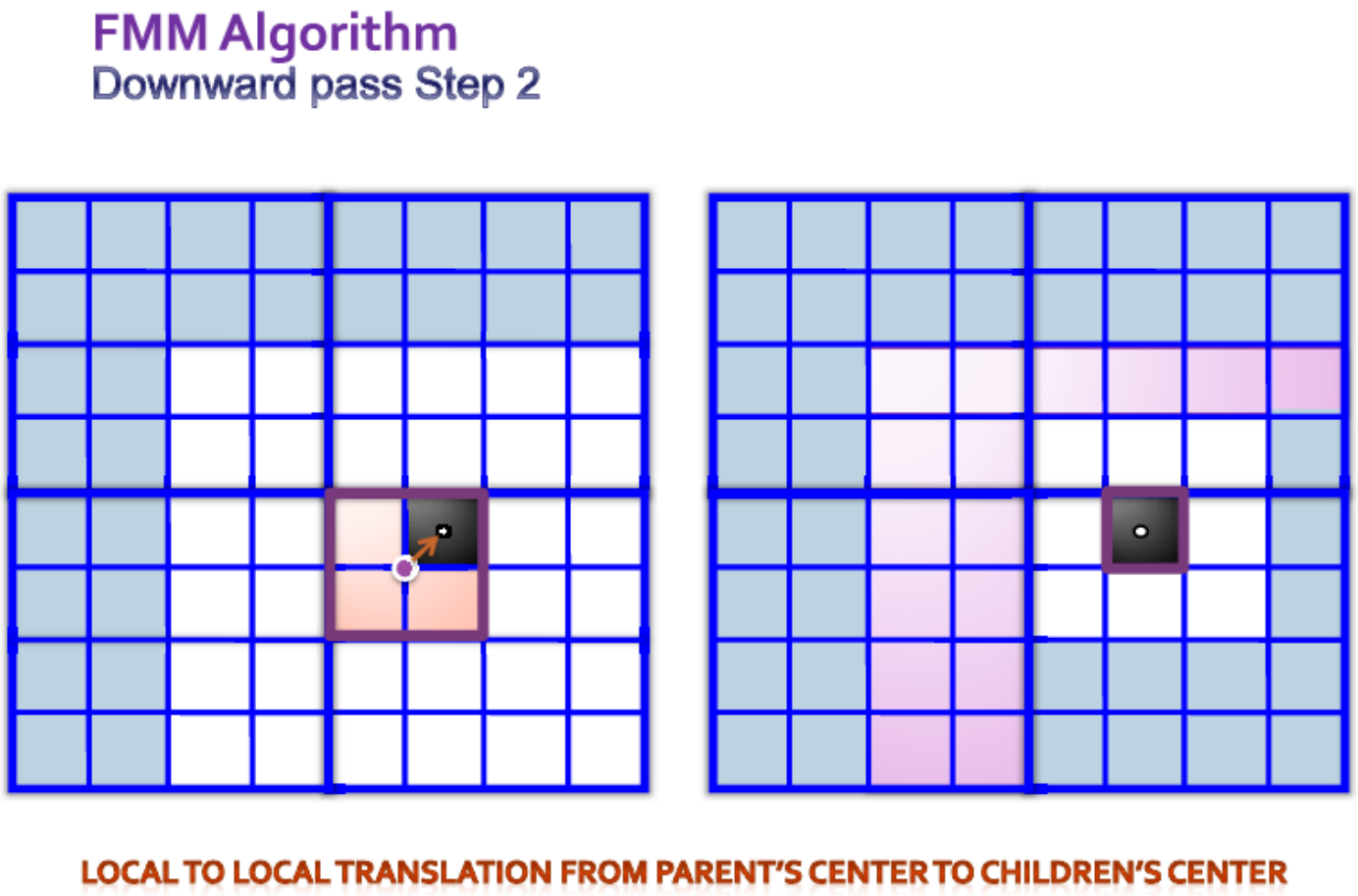


CALCULATE MULTIPOLE EXPANSION AT EACH PARENT'S CENTER

FMM Algorithm: Downward Pass



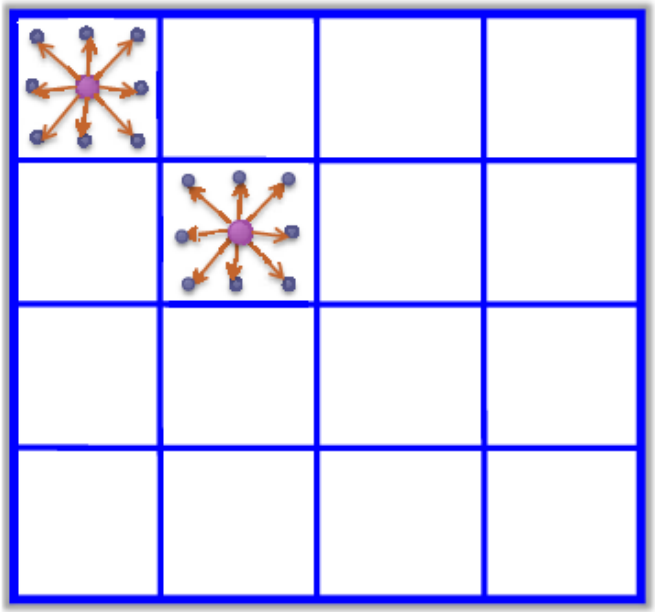
FMM Algorithm: Downward Pass



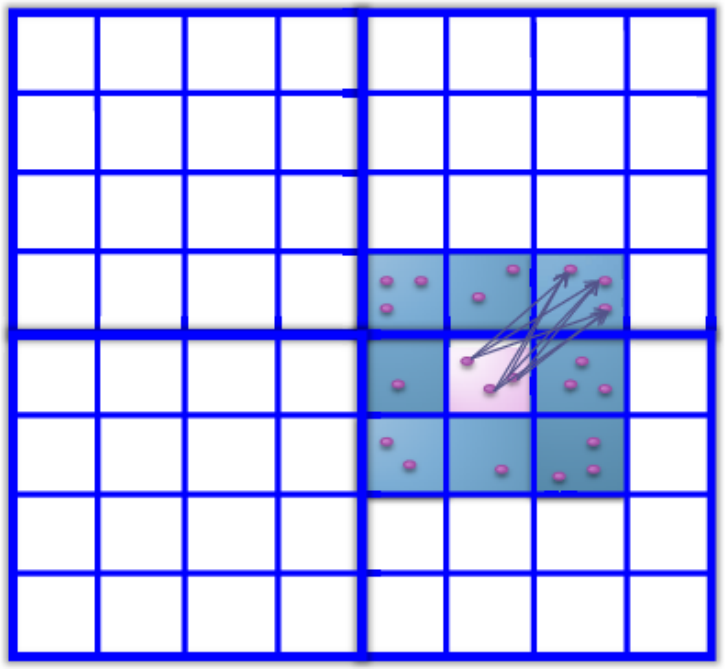
FMM Algorithm: Final Summation

FMM Algorithm
Final Summation Step

Only for leaves of the quadtree

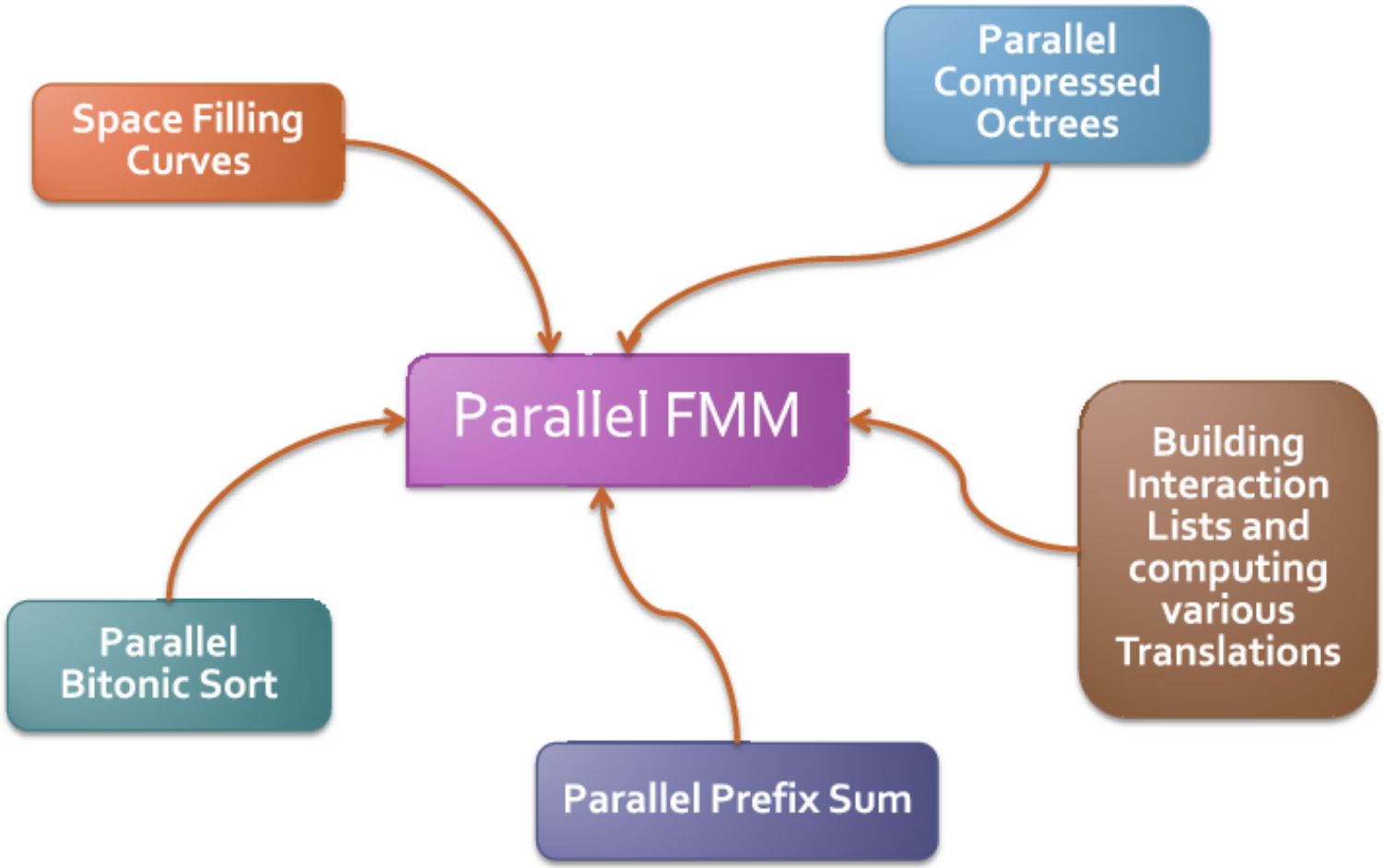


Evaluate Local Expansion at center of leaf at the particles

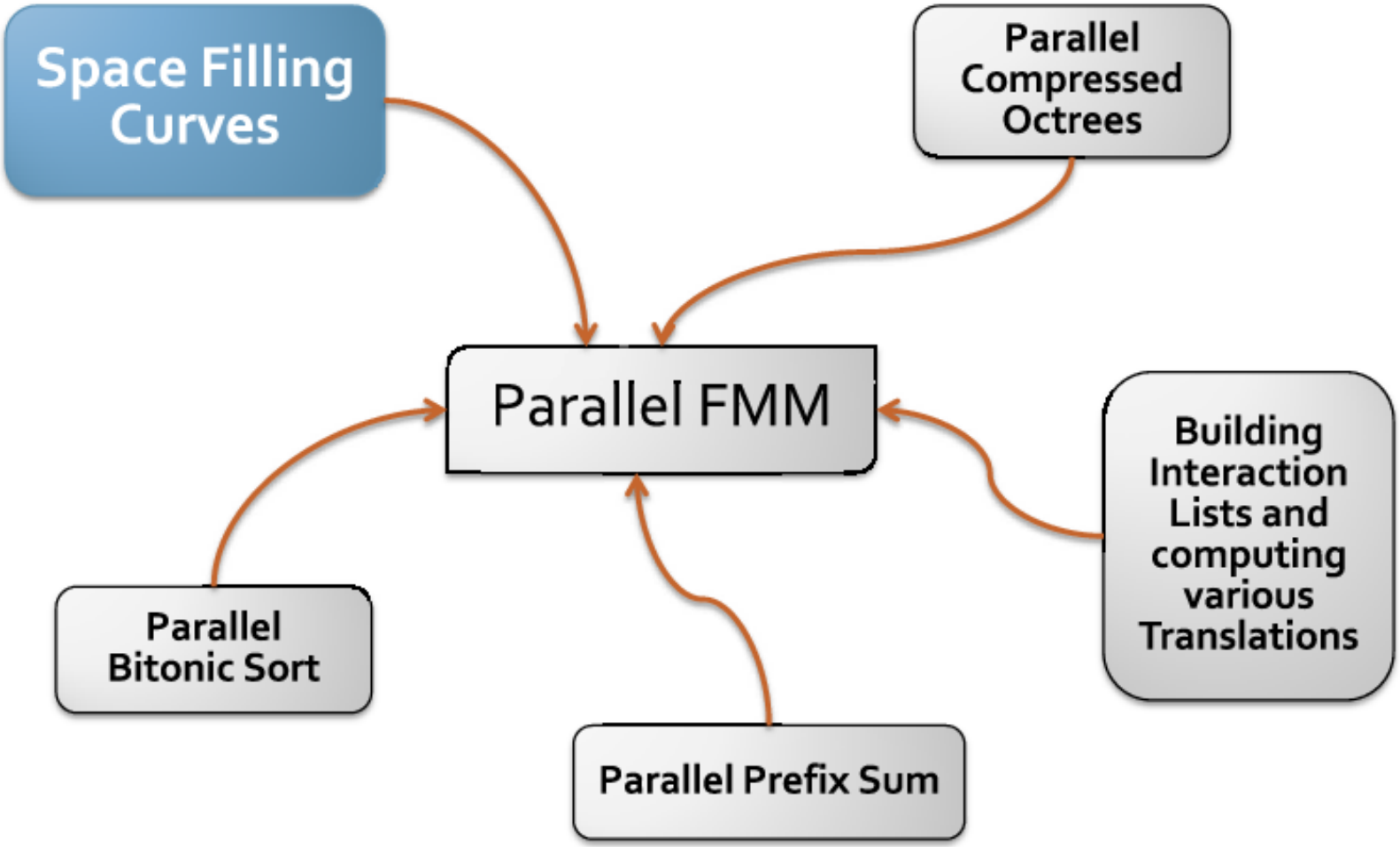


Direct Computation for particles in nodes of near cell list

Parallel FMM: Overview

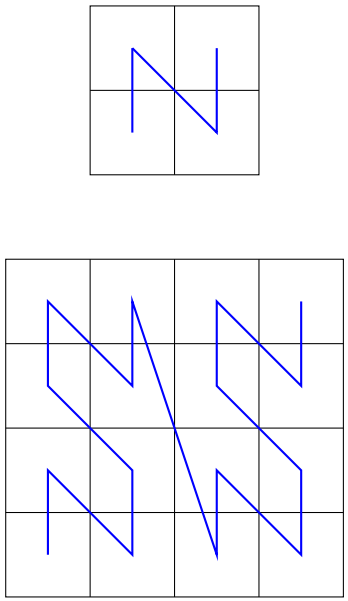


Space Filling Curves



Space Filling Curves: Construction

- Consider a k times recursive bisection of a 2-D area into $2^k 2^k 2^k = 2^{dk}$ non-overlapping cells of equal size.
- A **SPACE FILLING CURVE (SFC)** is a mapping of these cells to a one dimensional linear ordering



(a)

3			**7**
		6	
		4	
1	**2**		**5**

(b)

11	0101	0111	1101	1111
10	0100	0110	1100	1110
01	0001	0011	1001	**1011**
00	0000	0010	1000	1010
	00	01	10	11

(3,1) = (11,01), Index=1011

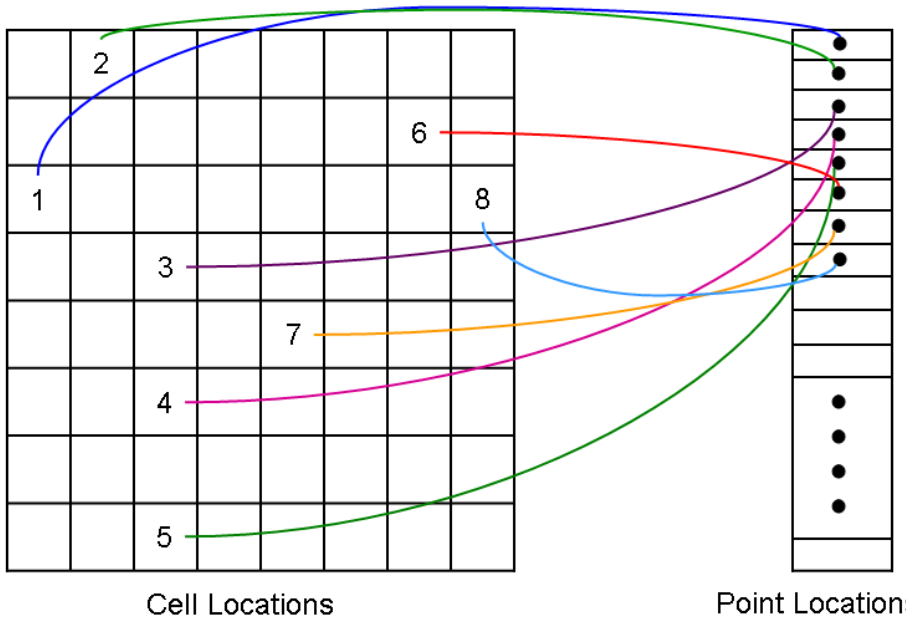
(c)

SFC Linearization Algorithm

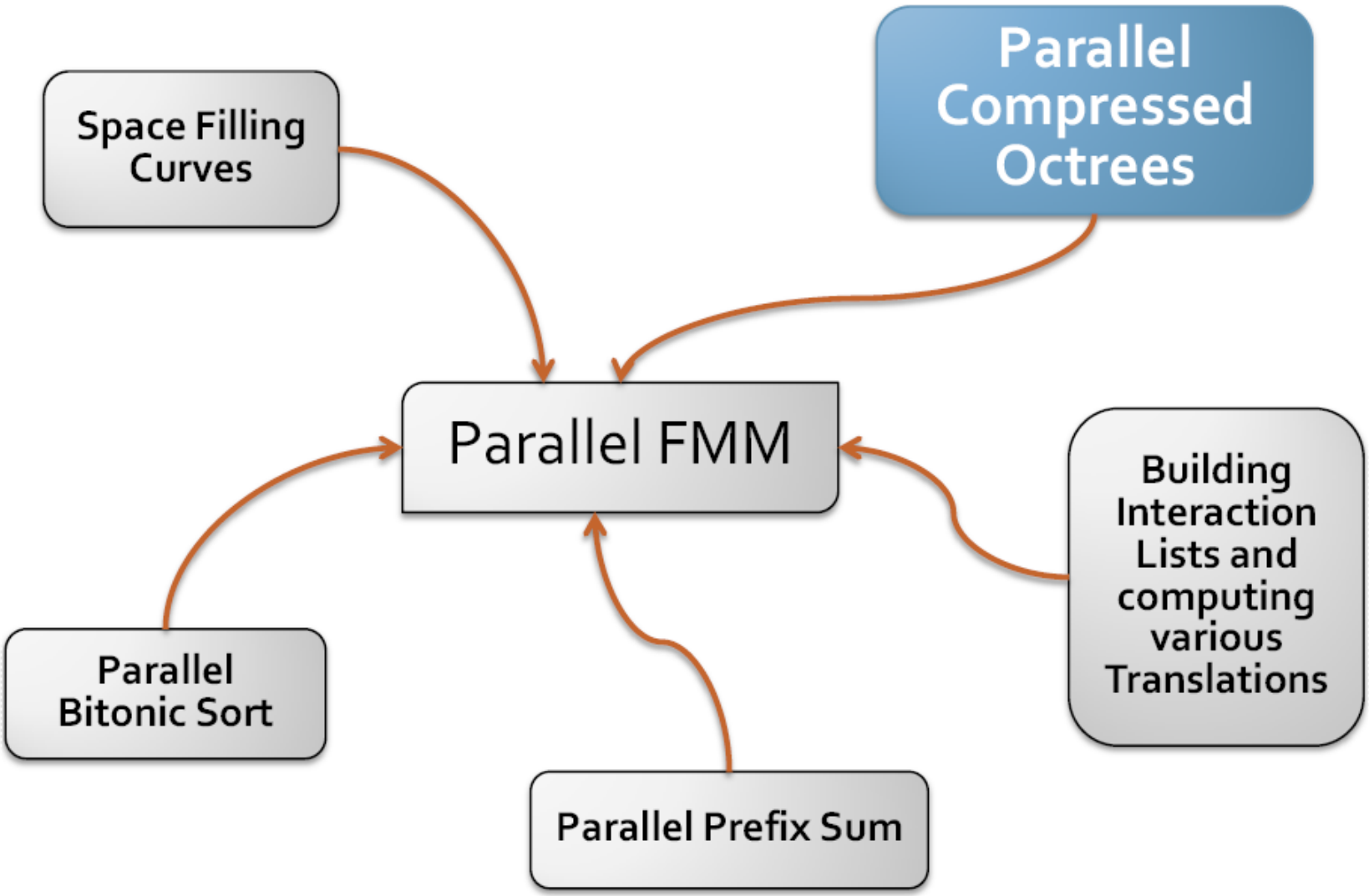
- Choose a resolution k.
- For each point, compute the index of the cell containing the point.
- Parallel sort the resulting set of integer keys.

SFCs on GPU

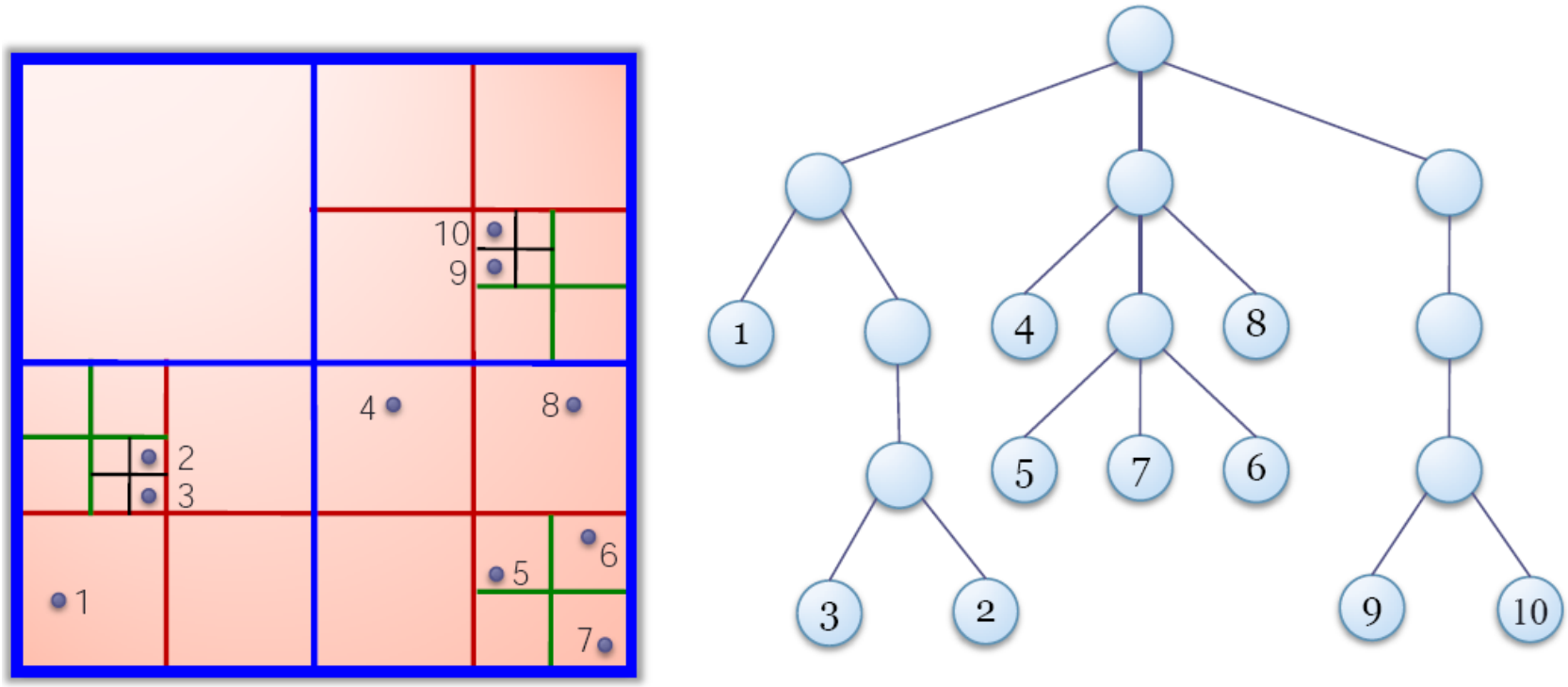
- GPU's memory is designed to store color and texture information
- The *RGBA* value of the cell will index the location of the point contained in it
- Indices sorted using efficient *Bitonic Sort* algorithm on GPU



Parallel Compressed Octrees: Overview

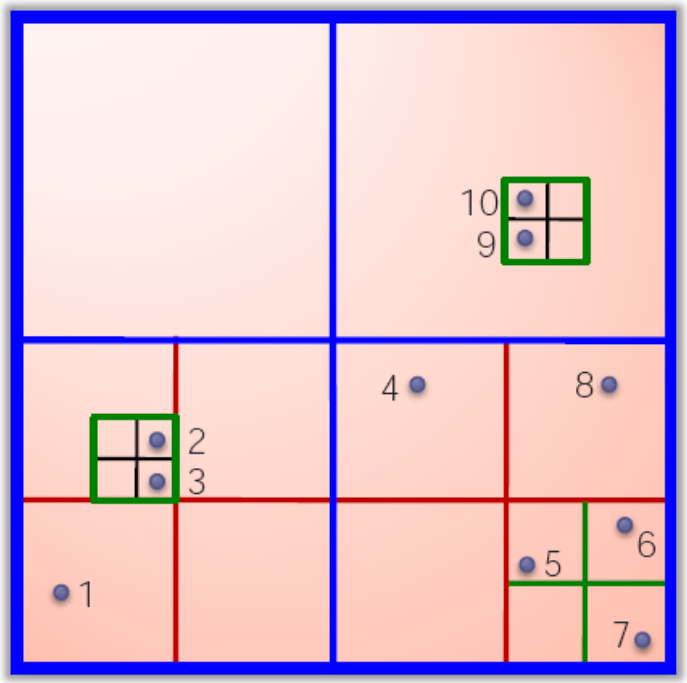


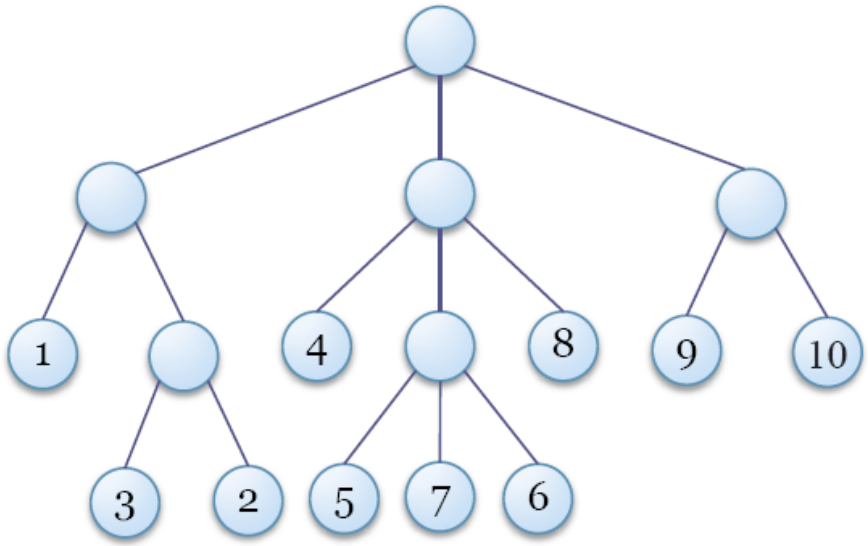
Octrees: Overview



A quadtree built on a set of 10 points in 2D

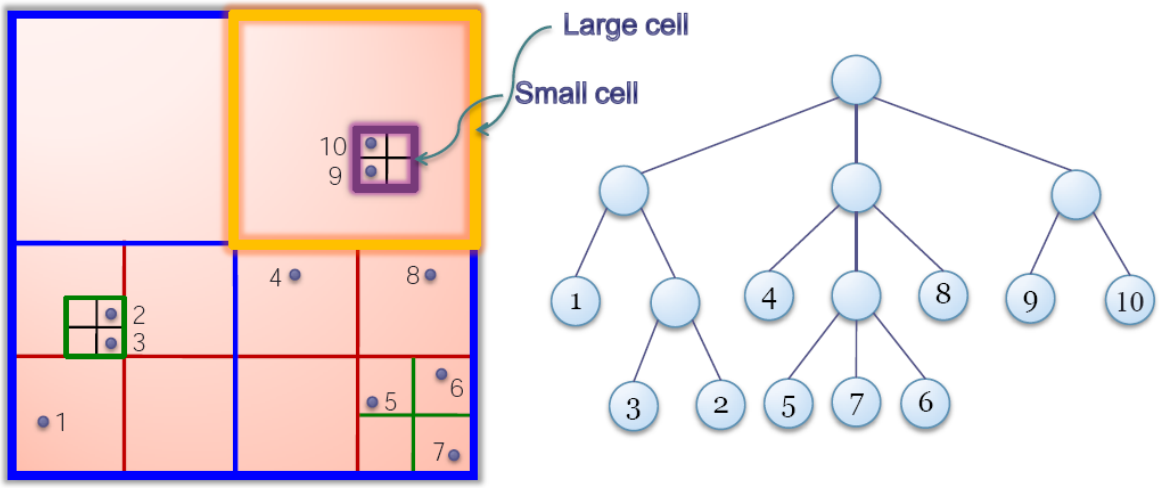
Compressed Octrees



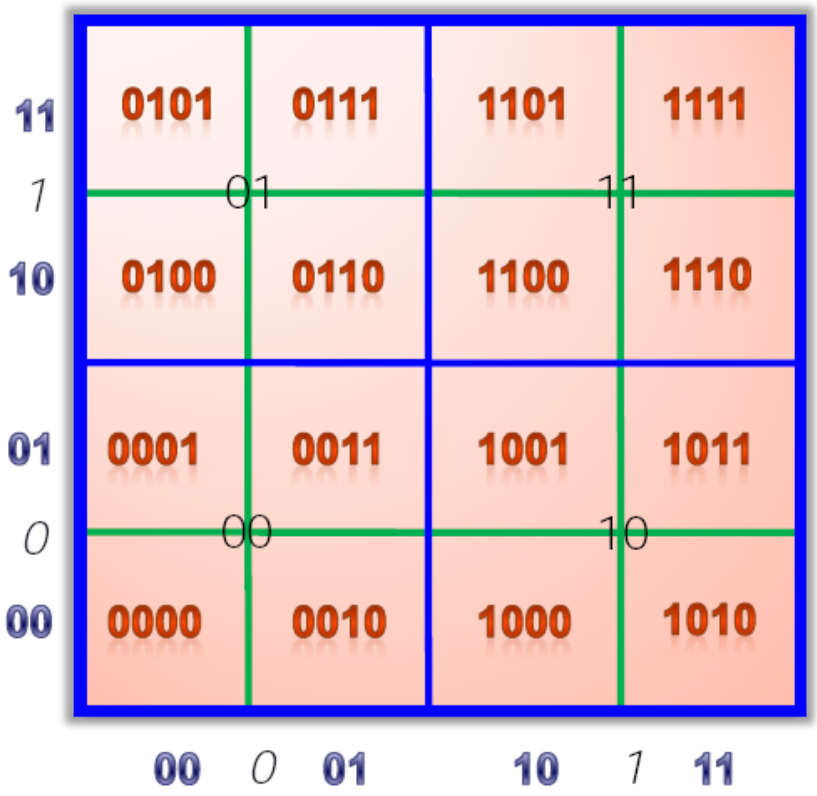


Compressed Octrees

- Need to encapsulate lost spatial information
- Store 2 cells in each node **v** of the compressed octree
  - Large cell **L(v)**:largest cell that encloses all the points the node represents
  - Small cell **S(v)**:smallest cell that encloses all the points the node represents



Octrees and SFCs



Octrees can be viewed as multiple SFCs at various resolutions

To establish a total order on the cells of octree: given 2 cells

if one is contained in the other, the subcell is taken to precede the supercell

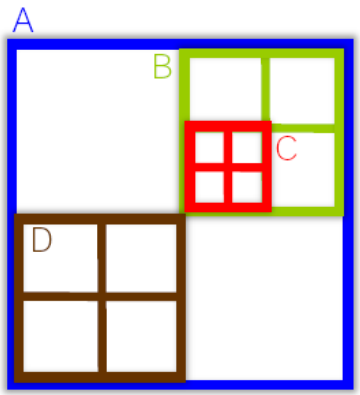
if disjoint, order according to the order of immediate subcells of the smallest supercell enclosing them

The resulting linearization is identical to **post order** traversal

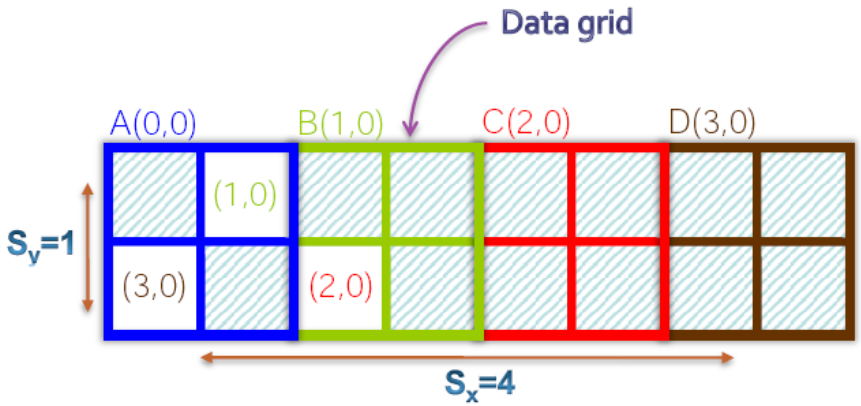
Parallel Compressed Octrees: Construction

- **procedure *ParallelCompressedOctrees***()
- Consider n points equally distributed across p processors
- k = pre-specified maximum resolution
- For each point, generate the index of the leaf cell containing it which is the cell at the max resolution k (can be done using effcient *prefix-sum* algorithm on GPU)
- Parallel sort the leaf indices to compute their SFC-linearization, or the left to right order of leaves in the compressed octree
- Each processor obtains the leftmost leaf cell of the next processor
- On each processor, construct a local compressed octree for the leaf cells within it and the borrowed leaf cell
- Send the *out of order nodes* to appropriate processors
- Insert the received out of order nodes in the already existing sorted order of nodes

Octree Textures on GPU



A quadtree

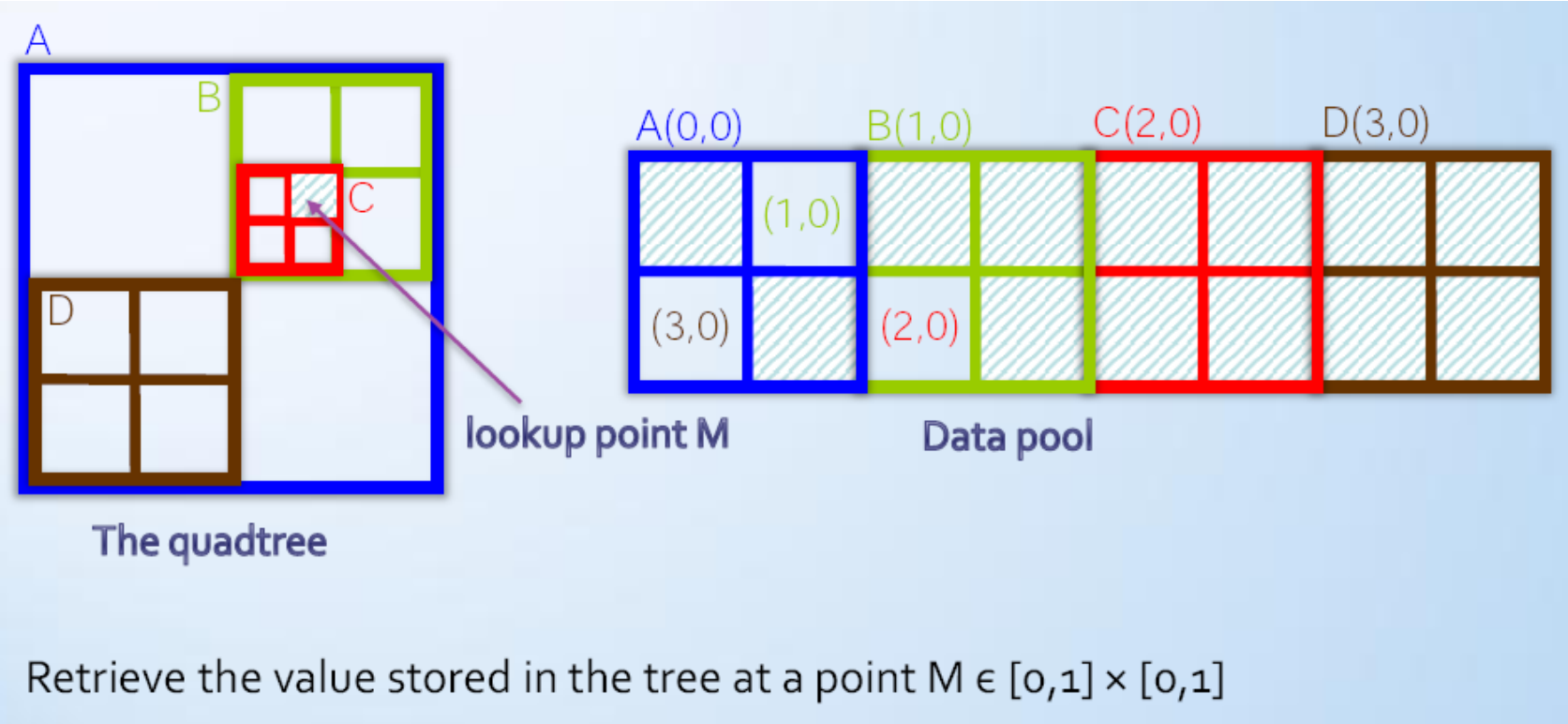


Data pool

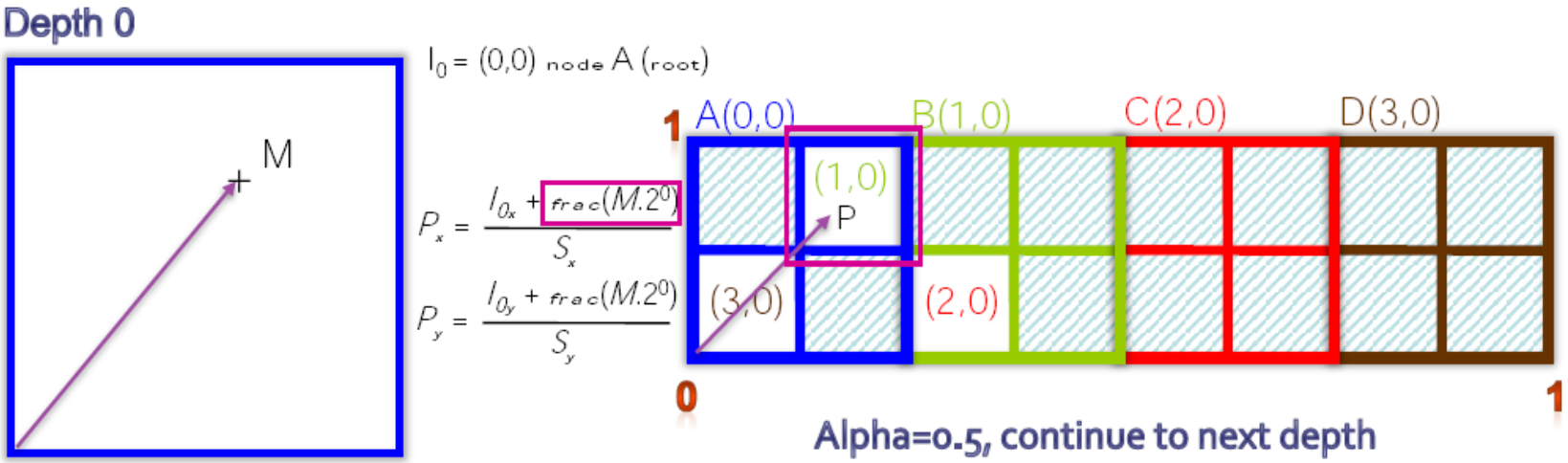
- The content of the leaves is directly stored as an RGB value
- Alpha channel is used to distinguish between an index to a child and the content of a leaf

alpha = 1 ⟹ data
alpha = 0.5 ⟹ index
alpha = 0 ⟹ empty cell

Octree Textures on GPU



Octree Textures on GPU

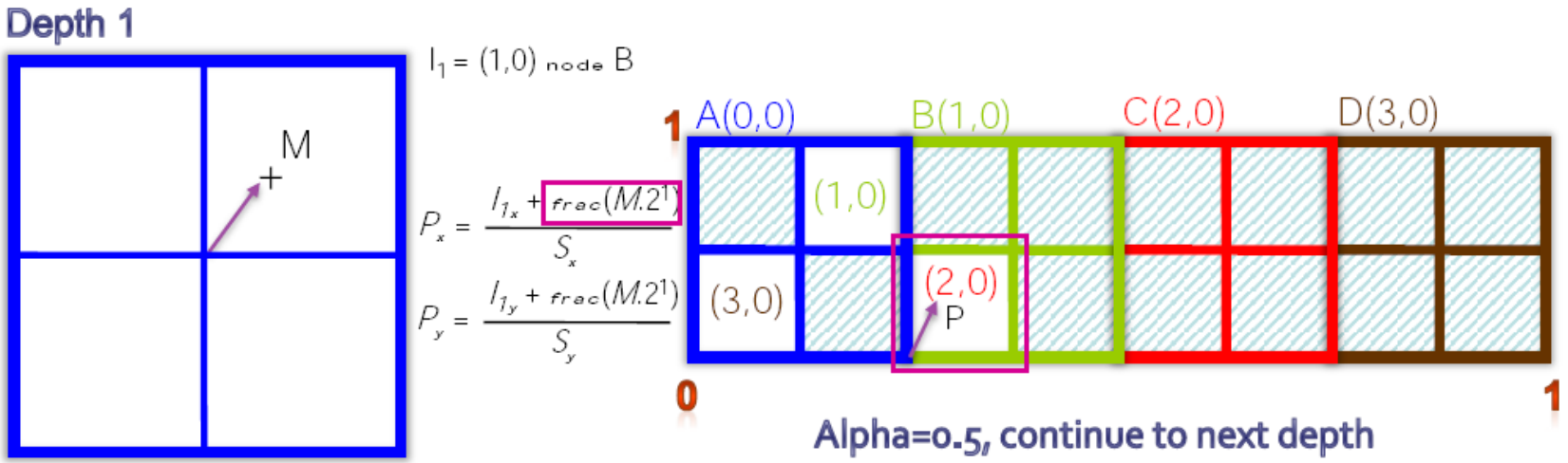


frac(A) denotes the fractional part of *A*

$I_0 = (0,0)$
Let M=(0.7, 0.7)
Coordinates of M within grid A = frac(M·2^0) = frac(0.7x1) = 0.7
x coordinate of the lookup point P in the texture = $P_x = \{I_{0x} + frac(M.2^0)\}/S_x$
$= (0 + 0.7)/4 = 0.175$

y coordinate of the lookup point P in the texture = $P_y = \{I_{0y} + frac(M.2^0)\}/S_y$
$= (0 + 0.7)/1 = 0.7$

Octree Textures on GPU



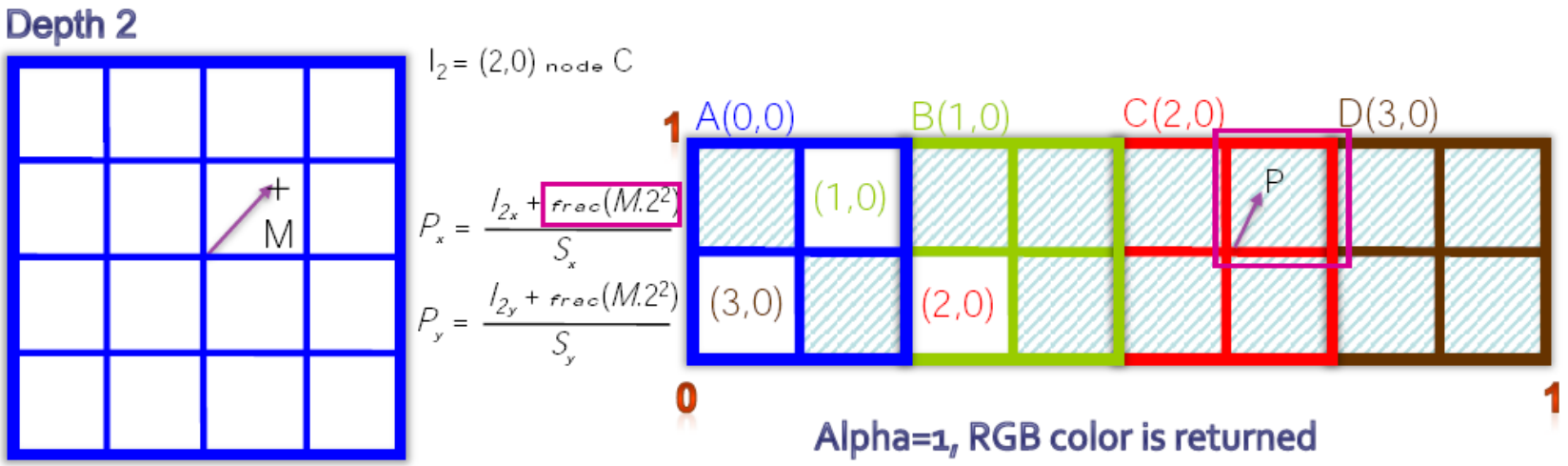
$I_1 = (1,0)$
M=(0.7, 0.7)
Coordinates of M within grid B = frac(M·2^1) = frac(0.7x2) = 0.4
x coordinate of the lookup point P in the texture = $P_x = \{I_{1x} + frac(M.2^1)\}/S_x$

$= (1 + 0.4)/4 = 0.35$

y coordinate of the lookup point P in the texture = $P_y = \{I_{1y} + frac(M.2^1)\}/S_y$

$= (0 + 0.4)/1 = 0.4$

Octree Textures on GPU



$I_2 = (2,0)$
M=(0.7, 0.7)
Coordinates of M within grid C = frac($M \cdot 2^2$) = frac(0.7x4) = 0.8
x coordinate of the lookup point P in the texture = $P_x = \{I_{2x} + frac(M.2^2)\}/S_x$
$= (2 + 0.8)/4 = 0.7$

y coordinate of the lookup point P in the texture = $P_y = \{I_{2y} + frac(M.2^2)\}/S_y$
$= (0 + 0.8)/1 = 0.8$

Queries for Compressed Octrees on GPU

- Apart from point lookup operation, what other operations need to be performed on compressed octrees ?
- How do we modify the octree implementation on GPU to store multiple data corresponding to each leaf-cell ?
- How do we modify the octree implementation on GPU (specifically the memory model) to store information about large-cell and small-cell of every node, along with the indices to its children ?
- How to access the last leaf node belonging to another processor, which is required for construction of local compressed octree ?
- Being a shared memory system, can we ignore the all-to-all communication step required previously ?
- How to identify and take care of out-of-order nodes ?
- How to handle multiple access by processors to same memory location simultaneously ?

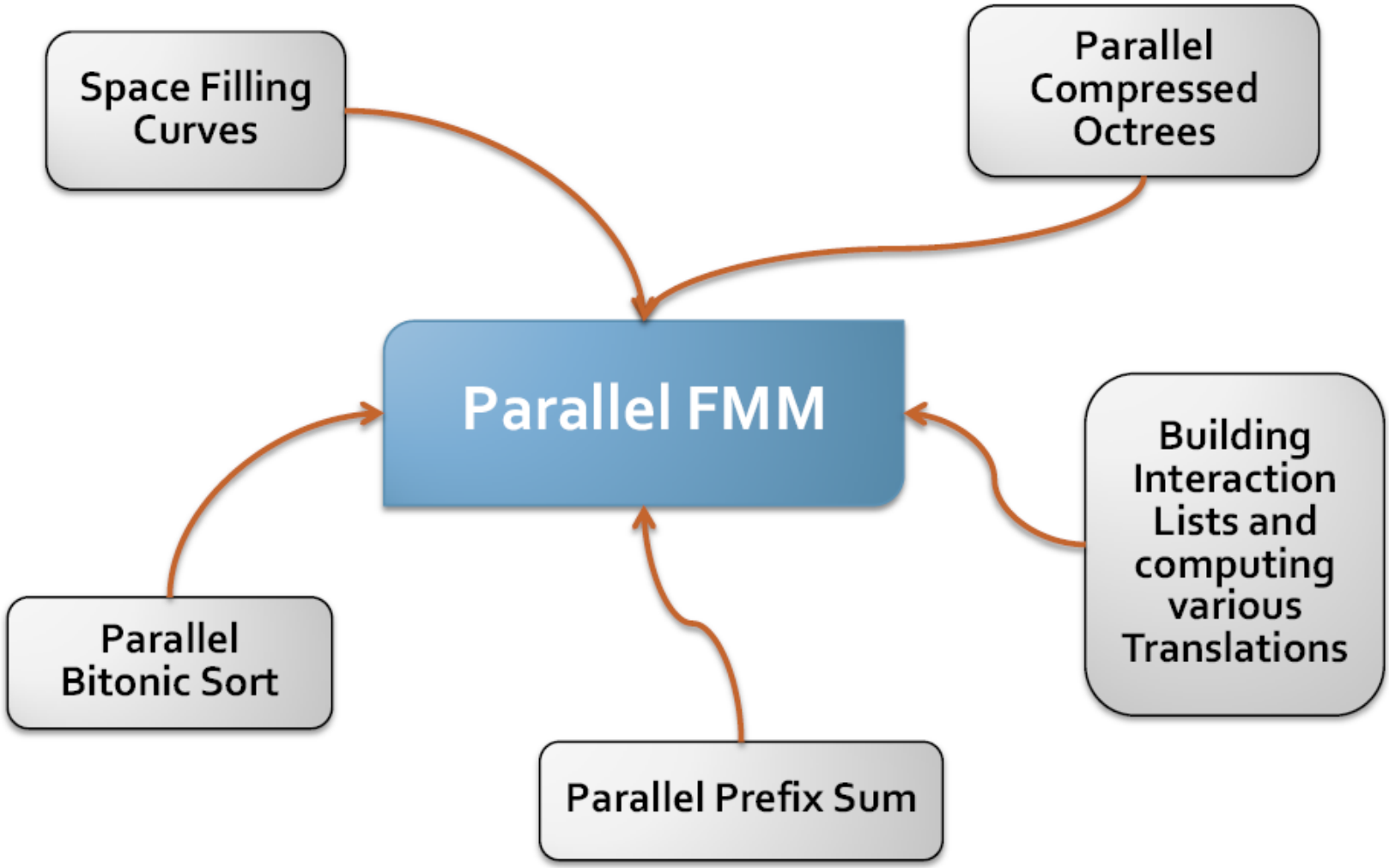
Answer to these hints/queries forms the start point for a parallel compressed octree construction on GPU

Queries for Compressed Octrees on GPU

- Apart from point lookup operation, what other operations need to be performed on compressed octrees ?
- How do we modify the octree implementation on GPU to store multiple data corresponding to each leaf-cell ?
- How do we modify the octree implementation on GPU (specifically the memory model) to store information about large-cell and small-cell of every node, along with the indices to its children ?
- How to access the last leaf node belonging to another processor, which is required for construction of local compressed octree ?
- Being a shared memory system, can we ignore the all-to-all communication step required previously ?
- How to identify and take care of out-of-order nodes ?
- How to handle multiple access by processors to same memory location simultaneously ?

Answer to these hints/queries forms the start point for a parallel compressed octree construction on GPU

Parallel FMM: Overview



Parallel FMM: Phases

The FMM computation consists of the following phases

- Building the compressed octree
- Building interaction lists
- Computing multipole expansions using a bottom-up traversal
- Computing multipole to local translations for each cell using its interaction list
- Computing the local expansions using a top-down traversal
- Projecting the field at leaf cells back to the particles

Computing Multipole Expansions

- Each processor scans its local array from left to right
- If leaf node is reached compute its multipole expansion directly
- If node's multipole is known, shift and add it to parent's multipole expansion provided the parent is local to processor
- **Use of postorder?**
- If the multipole expansion due to a cell is known but its parent lies in a different processor, it is labeled a **residual leaf node**
- If the multipoleexpansion at a node is not yet computed when it is visited, it is labeled a **residual internal node**

- Residual nodes form a tree (termed the residual tree)
- The tree is present in its postordertraversal order, distributed across processors

- Multipole expansions on the residual tree can be computed using an efficient parallel upward tree accumulation algorithm

Computing Multipole to Local Translations

- An all-to-all communication is used to receive fields of nodes from the interaction lists that reside on remote processors
- Once all the information is available locally, the multipoleto local translations are conducted within each processor as much as in the same way as in sequential FMM

Computing Local Expansions

- Similar to computing multipole expansions
- Calculate local expansions for the residual tree
- Compute local expansions for the local tree using a right-to-left scan of the post order traversal
- The exact number of communication rounds required is the same as in computing multipole expansions

Parallel FMM on GPU

- Implementing the discussed algorithm on GPU essentially means shifting to a shared memory system
- Replace the time consuming all-to-all communication steps by calls to local memory
- We saw an idea to start off implementing compressed octrees on GPU
- Once the octrees are implemented, all it remains are traditional memory access operations of GPU to read and write the data at desired locations, and doing computations on those data on each GPU processing element in parallel
- Shared memory makes it much more easier to build and use the interaction lists, store and compute on the residual trees and eliminate the time required for data communication

Just a starting point for parallel implemention of FMM on GPU. Many intricate and vital details are still ignored

Outline

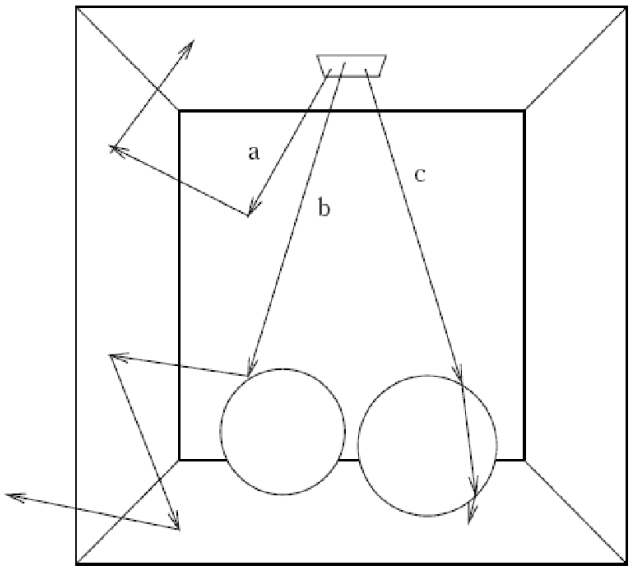
Problem Definition

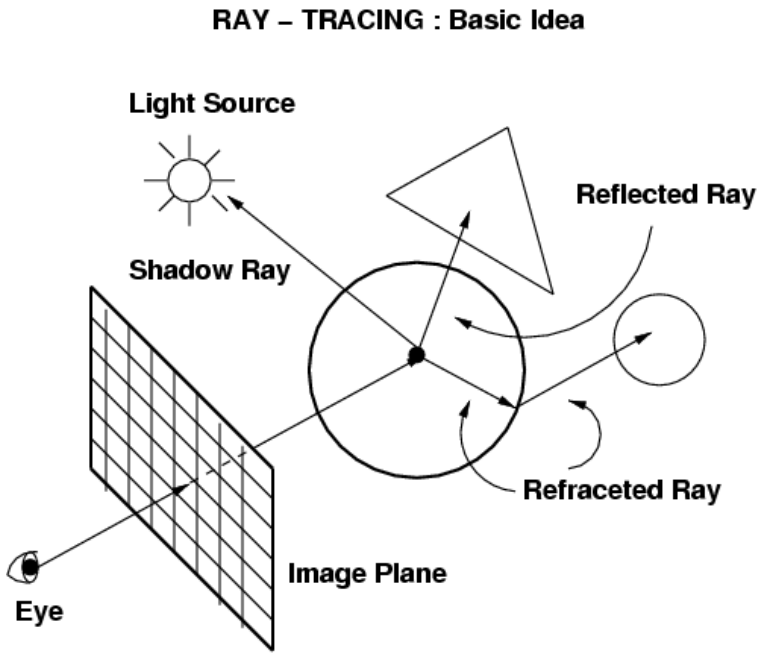
- Computing specular-interreflections, including caustics for point models
- Combining this with the on-going work on computing diffuse inter-reflections for point models, to give a *complete* global illumination solution
- **Approach**
  - Photon Mapping accounts for both diffuse and specular inter-reflections in a scene and produces good results
  - Currently works for *only* polygonal models and is *slow*
  - Analyze it from the perspective of applying it to point models
  - We will see that for point models, we need to make some changes in the *Ray Tracing* Approach
  - Further we try to optimize the slow steps of Photon Mapping Algorithm so as to speed up the whole system and make it as interactive as possible

Photon Mapping

- A two-pass method
  - *First pass* builds the photon map by emitting photons from the light sources into the scene and storing them in a photon map when they hit non-specular objects
    - The distribution of emitted photons must correspond to the emmissive power of the light source
    - Photons can be reflected, refracted and absorbed depending on the solution of the *Russian Roulette*
    - Position, power and incident direction of every photon is stored in the *Photon map*
  - *Second pass*, the rendering pass, makes $kNNqueries$ on the photon map to extract information about the radiance values

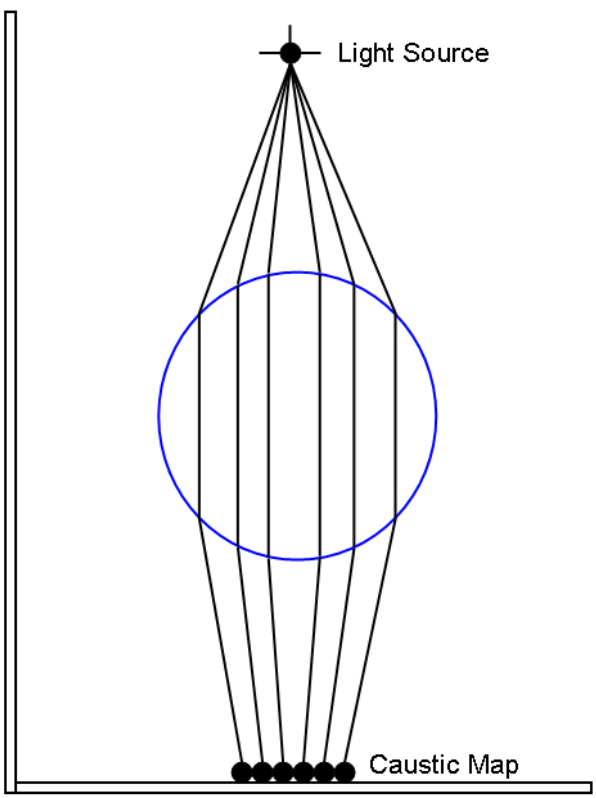
Photon Mapping

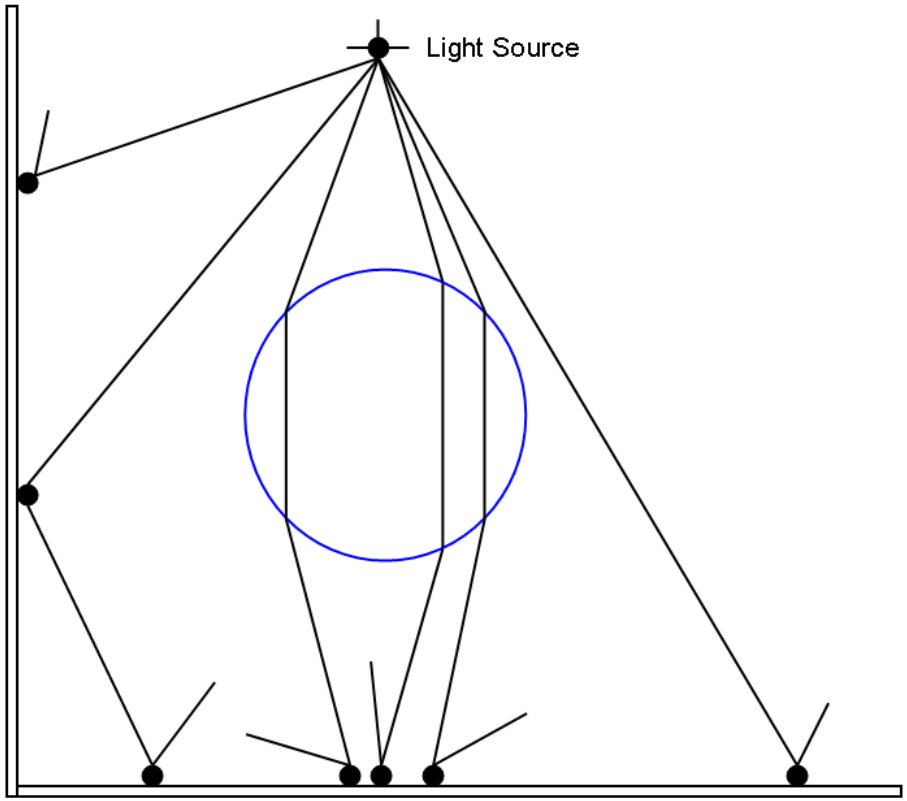




Photon Maps

- *Caustic Photon Maps*: $LS + D$ photons
- *Diffuse Photon Maps*: $LS|D * D$ photons





Example output of Photon Mapping

Splat-Based Ray Tracing(SBRT)

- Based on the concept of Surface-Splatting
- Deals with intersection of rays and *splats* (disks around points)
- Ray-Tracing produces good results if the surface has smoothly changing normals. Splat generation must produce such normals
- Splat with optimal radii are computed so that there is minimum overlap and no holes left in the scene
- A splat may cover many points. Normals (of those points) in a splat used to compute a smoothly varying normal field
- Ray-Splat intersection performed at the time of rendering
- Uses Octrees for traversal
- Responsible for generating reflections, refractions and shadows in the scene using secondary and shadow rays. **No Caustics** !
- Merge with traditional Photon Mapping to get caustics

Interactive Requirement

- Our specular-effects generation algorithm takes as input a point model with diffuse global illumination solution already calculated for it
- Diffuse global illumination solution being view-independent allows us for interactive walk-through of the input scene
- However, specular effects being view-dependent needs to be calculated for every new view-point in the ray-trace rendered frame
- Thus, if specular effect generation takes a lot of time, we loose out of having an interactive walk-through of the scene
- We desire not to loose this advantage, and try to optimize every algorithm required for specular effect generation

Optimizing Photon Generation

- **Only** Caustic photon generation will take place
- Some of the cost factors for photon generation can not be improved on. For example,
  - Rays will be incoherent during photon generation
  - Each light path will require several surface interactions (for reflection and refraction) in order to generate a caustic photon
- However, the number of paths that actually yield caustic photons can be influenced, and should be maximized
- We use **Selective Photon Tracing (SPT)** for the same

Selective Photon Tracing

- *Selective Photon Tracing(SPT)* was originally used to generate fast photon maps for dynamically changing object positions
- We do not consider the temporal domain, but rather use *SPT* for adaptively sampling path space
  - In a first step, a set of "pilot photons" is traced into the scene in order to detect paths that generate caustics
  - For those pilot paths, periodicity properties of the Halton sequence are exploited to generate similar photons around those paths

Optimizing Photon Tracing and Intersections

- The intersection tests performed for generating caustic photon map is similar to those performed while doing *SBRT* (ray-splat intersections)
- Further, SBRT uses Octree data-structure for traversal of the primary and secondary rays during ray-tracing. The use of Octree data structure provides us with quite a few advantages:
  - We can *re-use* the Octree data structure generated for input point model while doing diffuse illumination
  - The same traversal algorithm which SBRT uses on Octrees can be used for photon traversal as well
  - Further more, we can go for an even more optimized algorithm for Octree Traversal using **neighbor finding**. Here we traverse the octree horizontally via neighbor finding instead of traversing vertically starting from the root to the desired node
- Thus, we already have a well-established data structure (Octree) and algorithm (ray-splat intersection) for performing optimal photon traversal and intersection tests of rays and splats around points

Fast Photon Retrieval using Optimized kNN-Query Algorithm

- Although, kd-trees provides for fast kNN queries, they are still slow for interactive settings we desire
- Solving the kNN problem via kd-trees requires a search that traverses the tree downwards
- More importantly, a search-path pruning algorithm, based on the data already examined, is required to avoid accessing all data in the tree. This introduces serial dependencies between one memory look up and the next, consequently slowing down the retrieval process
- So we go for a even more optimised algorithm, **Low Latency Photon Retrieval Using Block Hashing**

Block Hashing

- The algorithm uses the **Locality-Sensitive Hashing (LSH)** function to categorise photons by their positions
- Further, the technique is designed under two assumptions on the behavior of memory systems
  - Its assumed that memory is allocated in fixed-sized blocks
  - Its assumed that access to memory is via burst transfer of blocks that are then cached
- Thus if any part of a fixed-sized memory block is touched, access to the rest of this block will be virtually zero-cost. Therefore, in BH all memory used to store photon data is broken into fixed-sized blocks
- Then, a kNN query proceeds by deciding which hash bucket is matched to the query point and retrieving the photons contained inside the hash bucket for rendering purposes

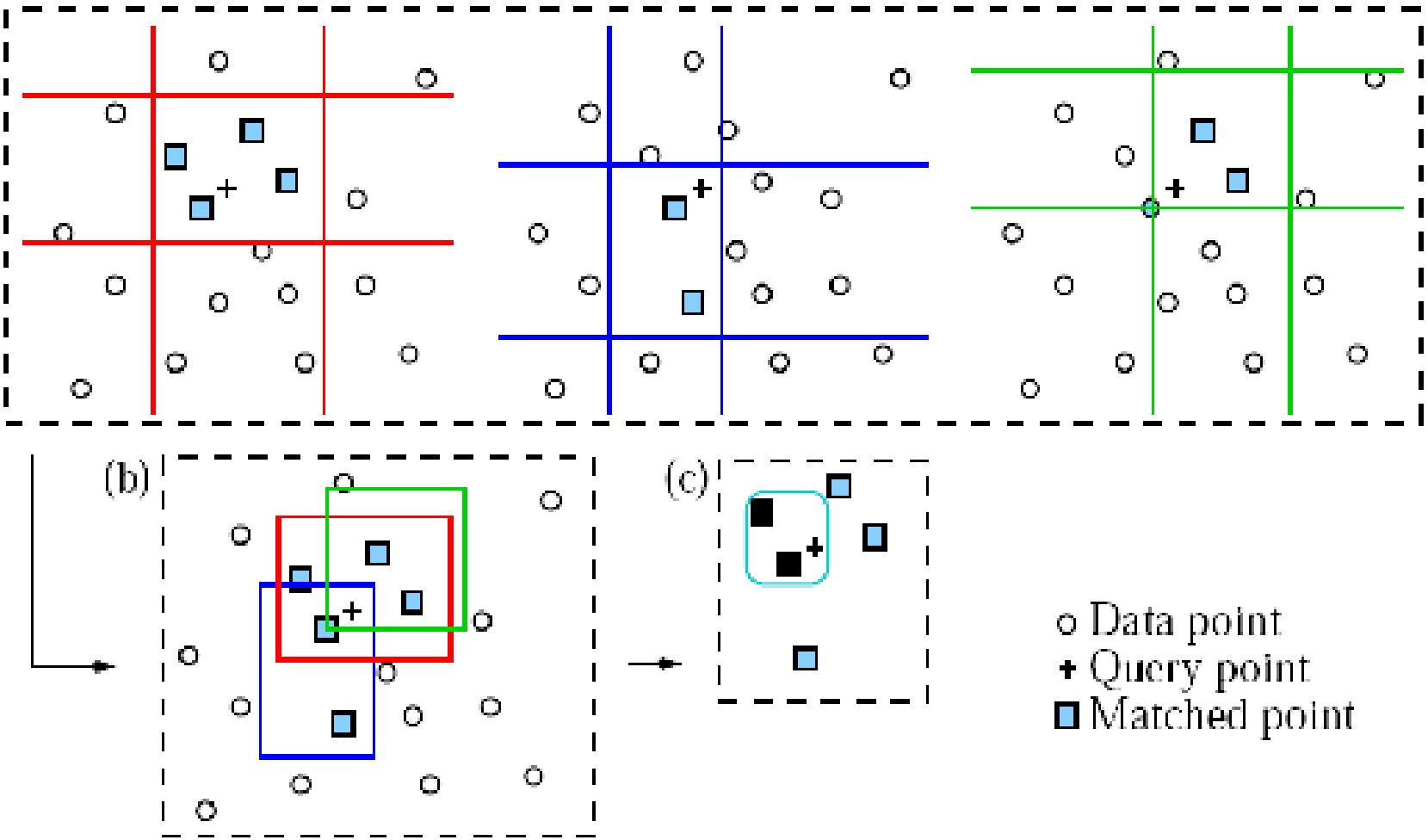
Block Hashing

- One attraction of the hashing approach is that evaluation of hash functions takes constant time
- In addition, once we have the hash value, accessing data we want in the hash table takes only a single access
- These advantages permit us to avoid operations that are serially dependent on one another
- Moreover, the algorithm results in coherent, non-redundant accesses to block-oriented memory. The results of one memory look up do not affect subsequent memory lookups, so accesses can take place in parallel within a pipelined memory system

Block Hashing: Phases

- Its, thus, a two-pass algorithm
  - A preprocessing phase consisting of
    - Organizing the photons into fixed-sized memory blocks
    - Creation of a set of hash tables
    - Inserting photon blocks into the hash tables
  - A query phase where the hash tables are queried for a set of candidate photons from which the $knearest$ photons will be selected for each point in space to be shaded by the renderer

Block Hashing : Querying



Merger into a Single System

- We thus have a fast caustic photon generation code using *Selective Photon Tracing*
- We have ray-splat intersection code from *SBRT* for caustic photon traversing and intersection
- Traversing can be improved by using faster *horizontal* octree traversal using *neighbor finding*
- We use *SBRT* for ray tracing and getting the reflections and refractions
- Usage of caustic photon maps and fast kNN query algorithm using *Block Hashing* provides us with efficient caustics while rendering

We combine all these algorithms in a single system so as to get efficient and high-quality specular effects for point models. This is, however, just a starting point and many issues need to be tackled while actual implementation

Outline

Conclusion and Future Work

- The lack of surface information in point models creates difficulties in operations like generating global illumination effects and computing point-pair visibility
- Point-to-Point Visibility is arguably one of the most difficult problems in rendering since the interaction between two primitives depends on the rest of the scene
- One way to reduce the difficulty is to consider clustering of regions such that their mutual visibility is resolved at a group level (V-Map)
- Visibility Map data structure we propose enables efficient answer to common rendering queries
- In this report, we have given a novel, provably efficient, hierarchical, visibility determination scheme for point based models
- By viewing this visibility map as a 'preprocessing' step, photo-realistic global illumination rendering of complex point-based models have been shown

Conclusion and Future Work

- Further, parallel implementation of FMM, used for diffuse illumination, is a difficult task with load balancing, data decomposition and communication efficiency being the major challenges
- We discussed an algorithm which removes these disadvantages.
- **We now aim to exploit the parallel computing power of GPUs for implementation of the FMM as well as the point-pair visibility determination algorithm using V-Maps**
- We gave necessary hints for the same
- Further, we saw how various algorithms from the literature were combined under a single domain to get us a time-efficient system designed to generate the desired specular effects for point models
- **We now aim to implement these algorithms, merge them together and get the specular effects solution for point models**

Conclusion and Future Work

- We, thus, will have a two-pass global illumination solver for point-based scenes consisting of both diffuse and specular models
  - First pass will calculate the diffuse illumination maps
  - Second pass for specular effects
- Finally, the scene will be rendered using *splat-based ray-tracing* technique
- **However, a question remains that since we are parting the diffuse and specular effect calculations for the scene, how would we handle specular objects (and their effects on diffuse objects) while calculating only diffuse global illumination in the first pass of the global illumination solver**
- *This important issue needs to be investigated thoroughly*

Thank you for your time !

Questions ?