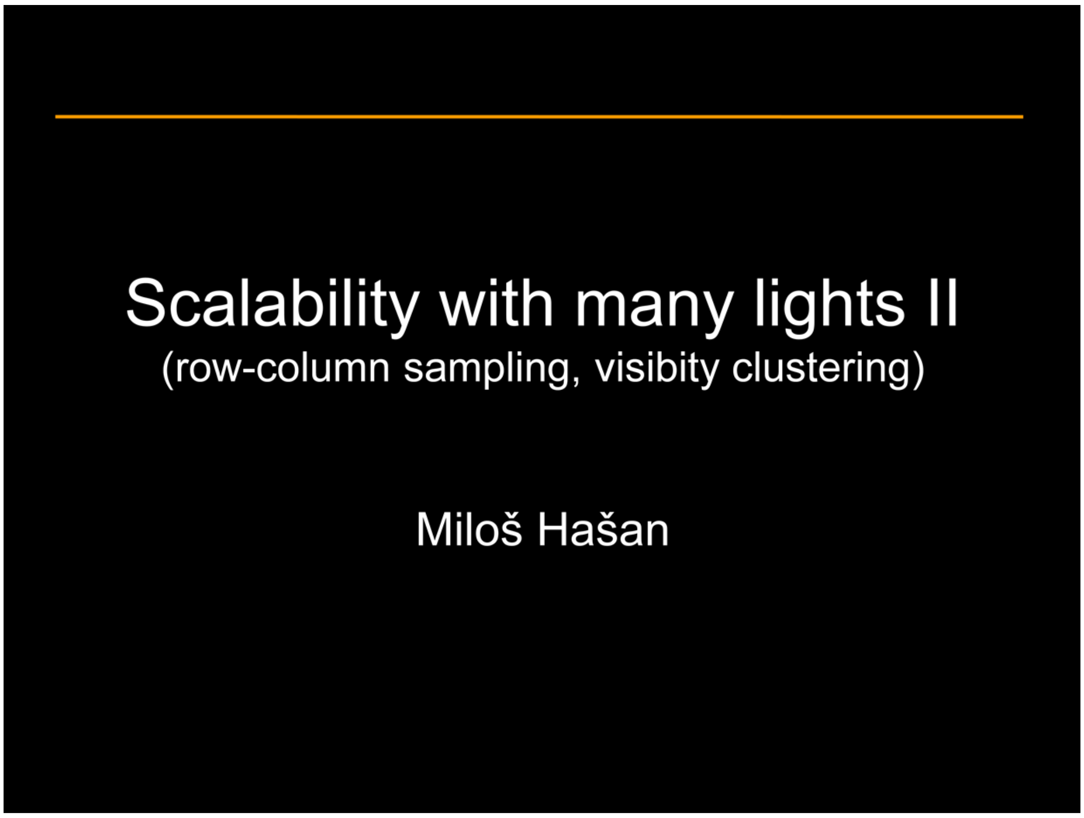
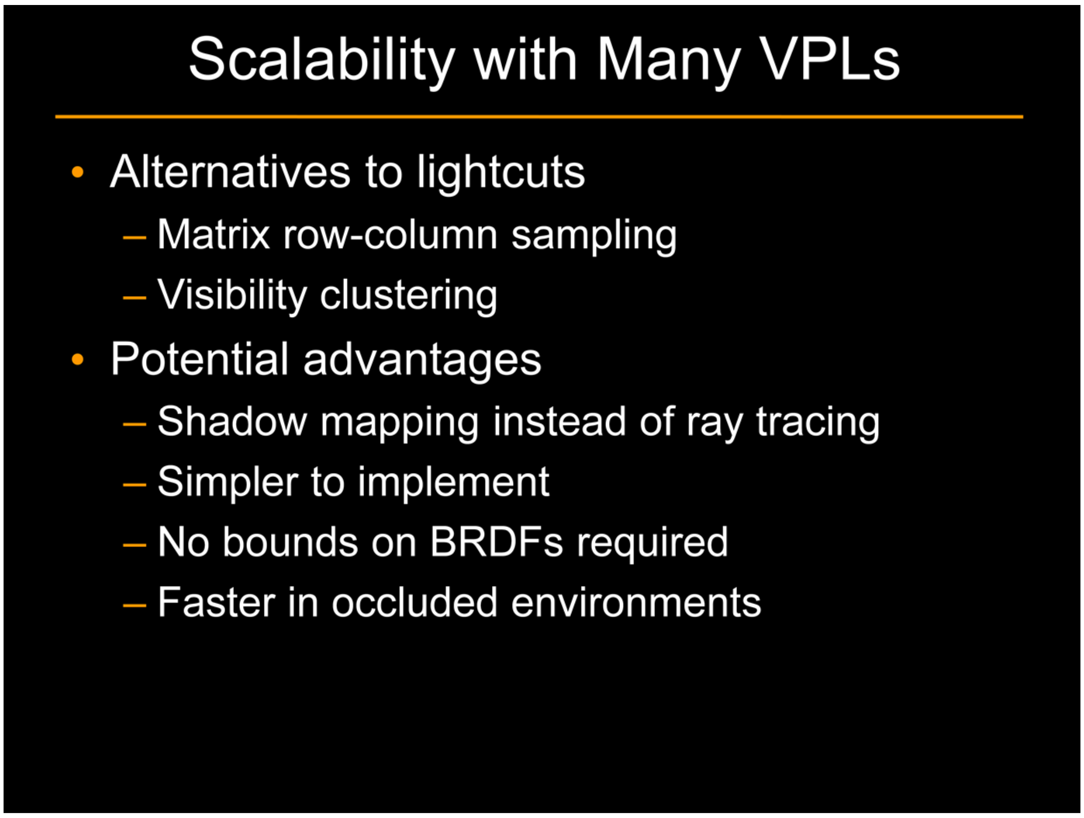


Scalability with many lights II

(row-column sampling, visibity clustering)

Miloš Hašan



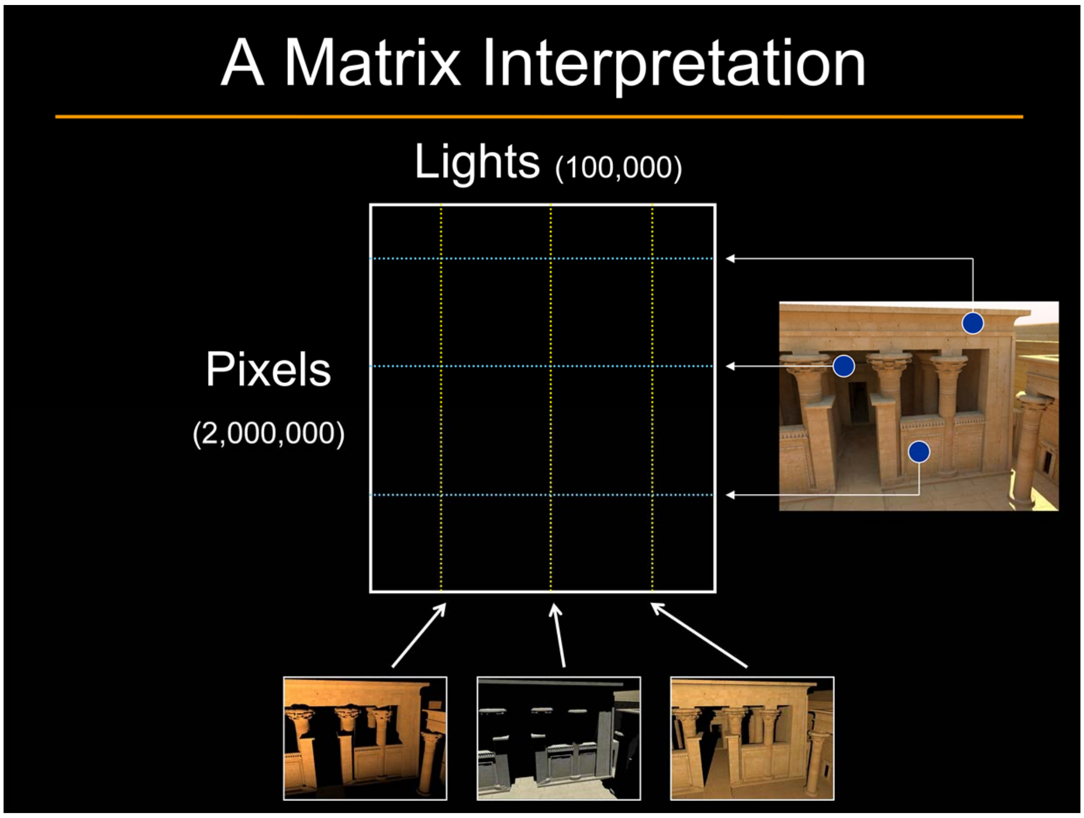
Scalability with Many VPLs

- Alternatives to lightcuts
  - Matrix row-column sampling
  - Visibility clustering
- Potential advantages
  - Shadow mapping instead of ray tracing
  - Simpler to implement
  - No bounds on BRDFs required
  - Faster in occluded environments

In this part of the course, I will talk about several techniques that improve the scalability of many-light methods with the number of virtual lights. In other words – we have reduced our rendering problem to computing the connections between many VPLs and many receivers, or "gather points".

Of course, one way to do it are the lightcuts algorithms that Bruce described, which are very reliable ad high-quality. I will introduce alternative techniques like row-column sampling and visibility clustering, which can have some other advantages.

For example, one may be able ot use shadow maps instead of ray casting for visibility checking, which tends to be faster in most cases. No bounds on shders are required, which can lead to simpler implementations. There may also be advantages in highly occluded environments, where lightcuts will conservatively evaluate illumination assuming full visibility.



A Matrix Interpretation

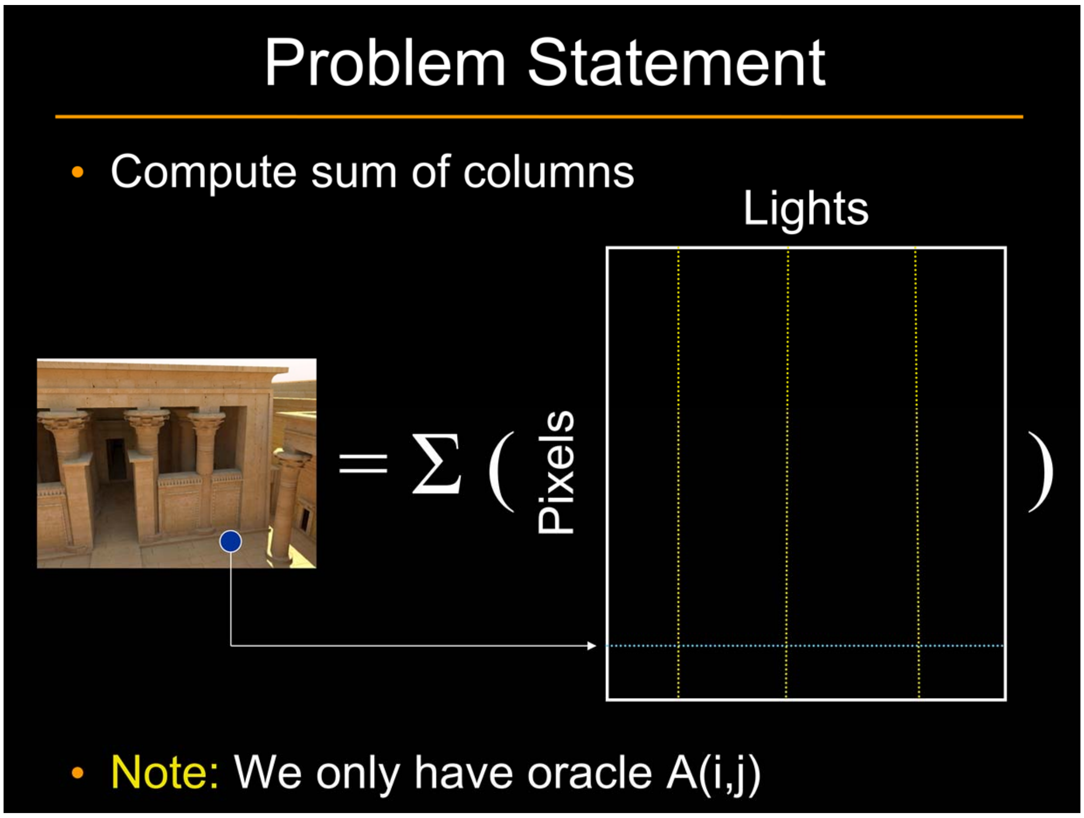
Lights (100,000)

Pixels (2,000,000)

To get started in describing the algorithms, we first need to interpret the many light problem as a matrix of light-pixel interactions. This means that each element of the matrix is the contribution of a single light to a single pixel.

So, the columns of the matrix are really images rendered with a single point light.

The rows represent contributions of all the lights to a particular pixel.



Problem Statement

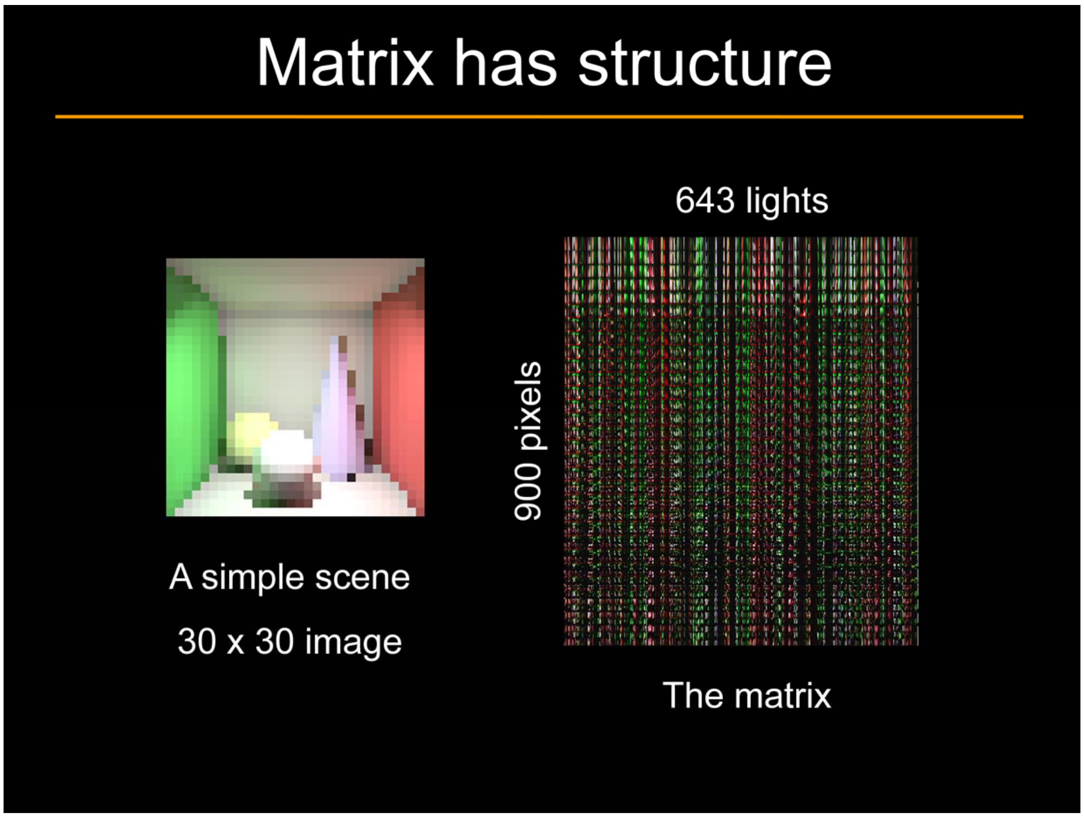
- Compute sum of columns

- Note: We only have oracle A(i,j)

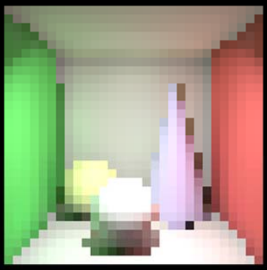
In this setting, the ideal image we would like to render is equal to the sum of the columns of the matrix.

Or, to put it differently, the color of each pixel is equal to the sum of the matrix row corresponding to that pixel.

It is important to note that we’re not given the matrix data, we just have an “oracle” – a function that can evaluate the elements on demand. Our goal is to compute the sum without evaluating most of the elements. How is this even possible?

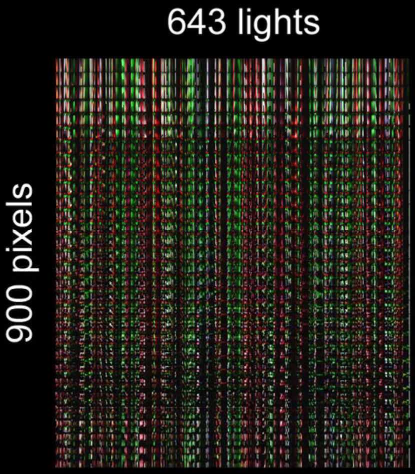


Matrix has structure



A simple scene

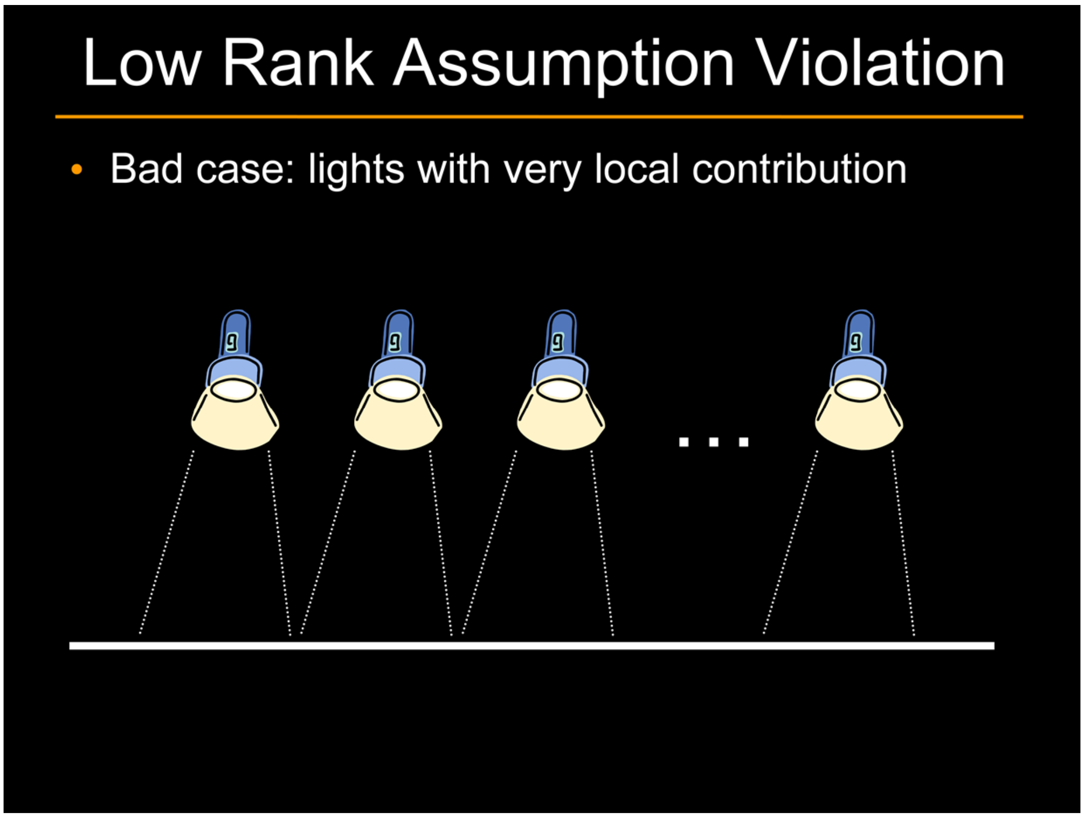
30 x 30 image



The matrix

The trick is that the matrix is highly structured. Here is an example with a Cornell box. with a single direct light, and Numerically, the matrix often close to low-rank: its columns can often be approximated by linear combinations of other columns.

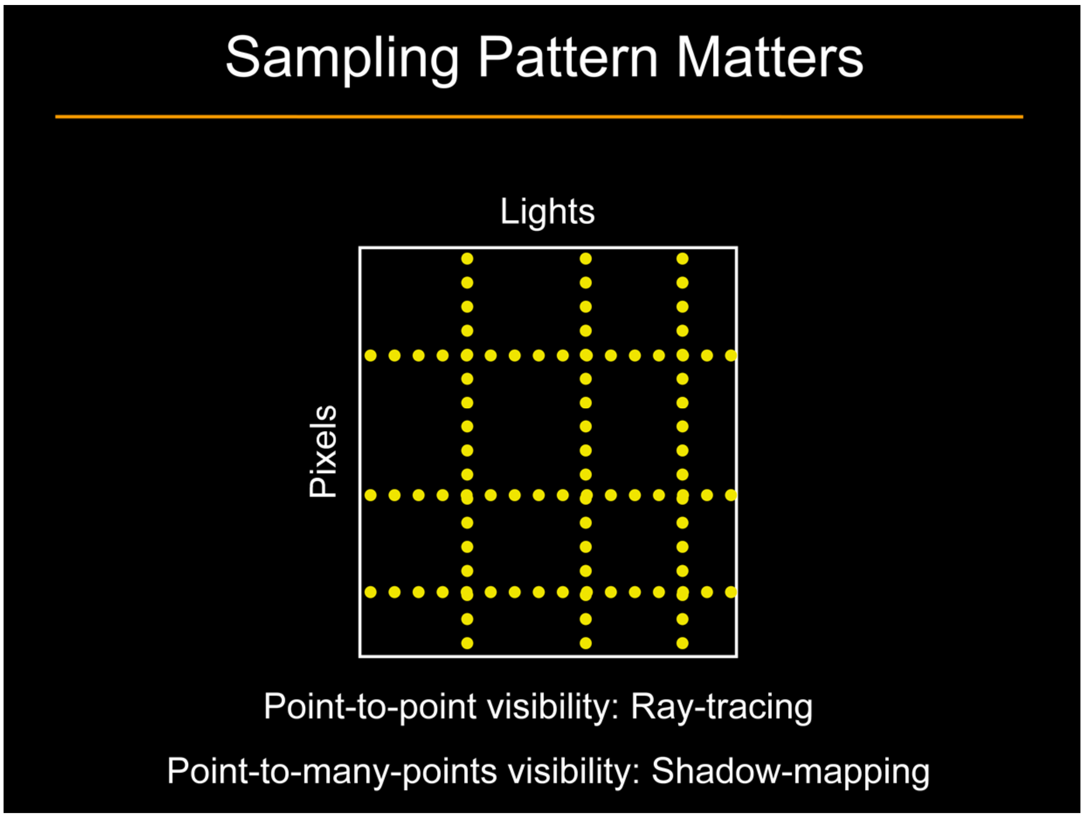
Therefore, we can get away with computing only a very small subset of the elements, and still gather enough information to render an accurate image.



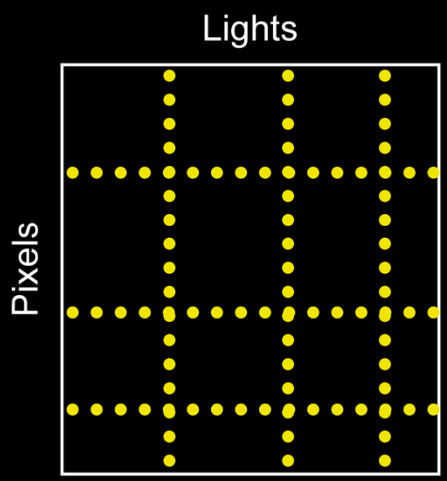
Low Rank Assumption Violation

- Bad case: lights with very local contribution

Of course, the low rank assumption is not always valid – here is an example of a really bad case, where the light’s contributions are linearly independent. Fortunately, this does not happen in practice that often.



Sampling Pattern Matters

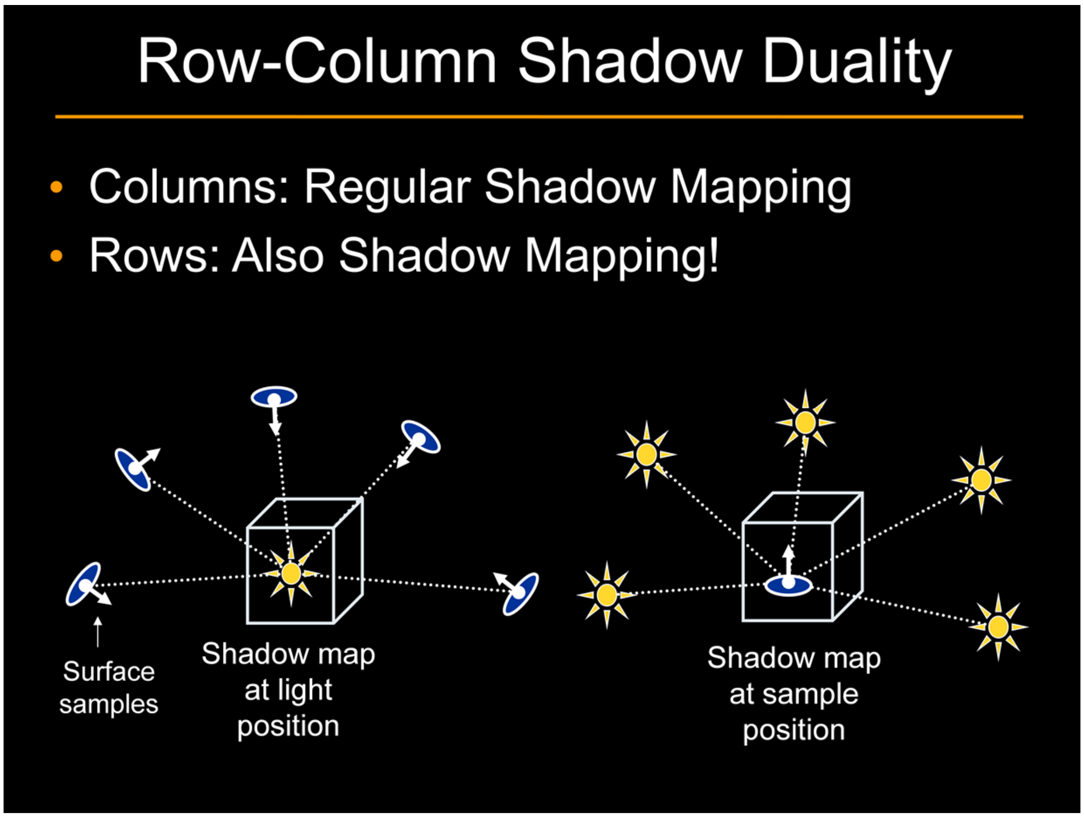


Point-to-point visibility: Ray-tracing

Point-to-many-points visibility: Shadow-mapping

So, we want to sample a subset of the matrix elements, but which ones should we choose?

If we sample complete rows and columns, we can use GPU shadow mapping as our visibility algorithm. This way we can compute elements at very high rate. Futhermore, it is easier to reason about rows and columns, so even if we use a ray-tracer, it may still be an advantage to sample like this.



Row-Column Shadow Duality

- Columns: Regular Shadow Mapping
- Rows: Also Shadow Mapping!

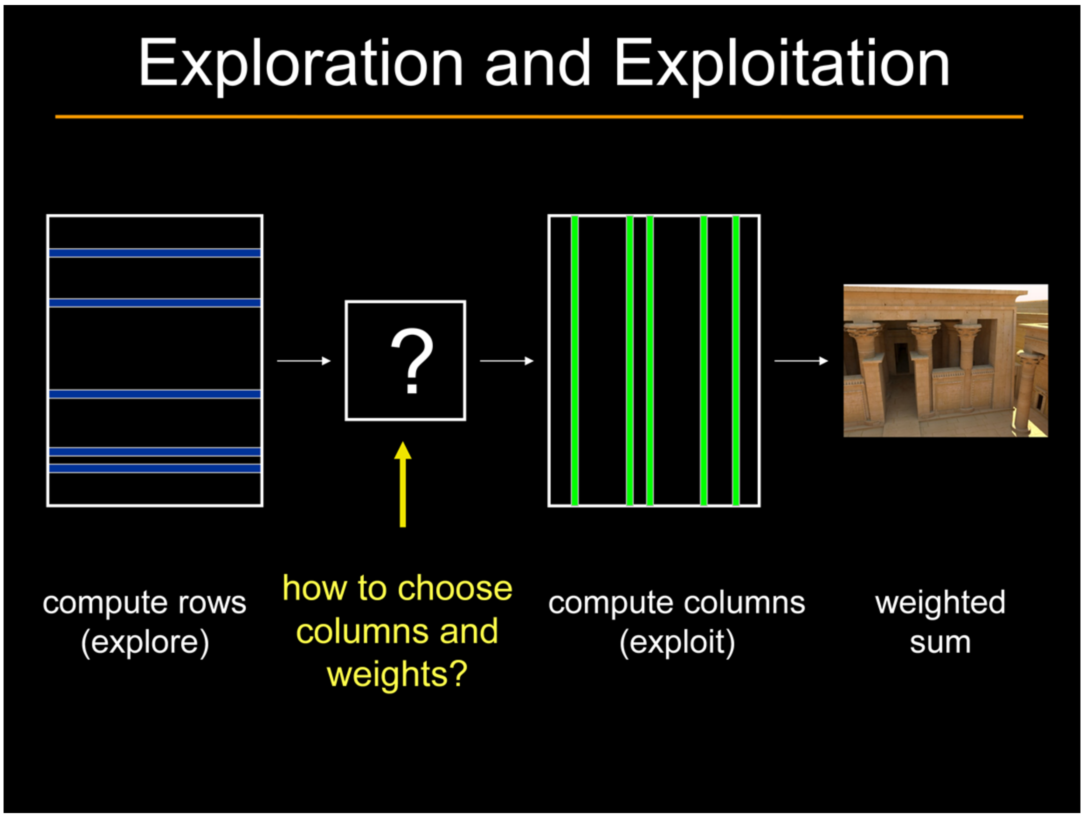
Surface samples
Shadow map at light position
Shadow map at sample position

One may wonder how to compute the row contributions using shadow mapping. After all, shadows normally come from the light!

However, shadow mapping is simply a way to determine the visibility from a point to many other points at once.

We can compute a cube shadow map at the VPL position, and determine the visibility of the surface samples, as usual. Or, we can compute the cube at the receiver position, and query the light positions against it.

However, if you’re implementing any of these algorithms, I would recommend to debug with ray-traced occlusion first.



Exploration and Exploitation

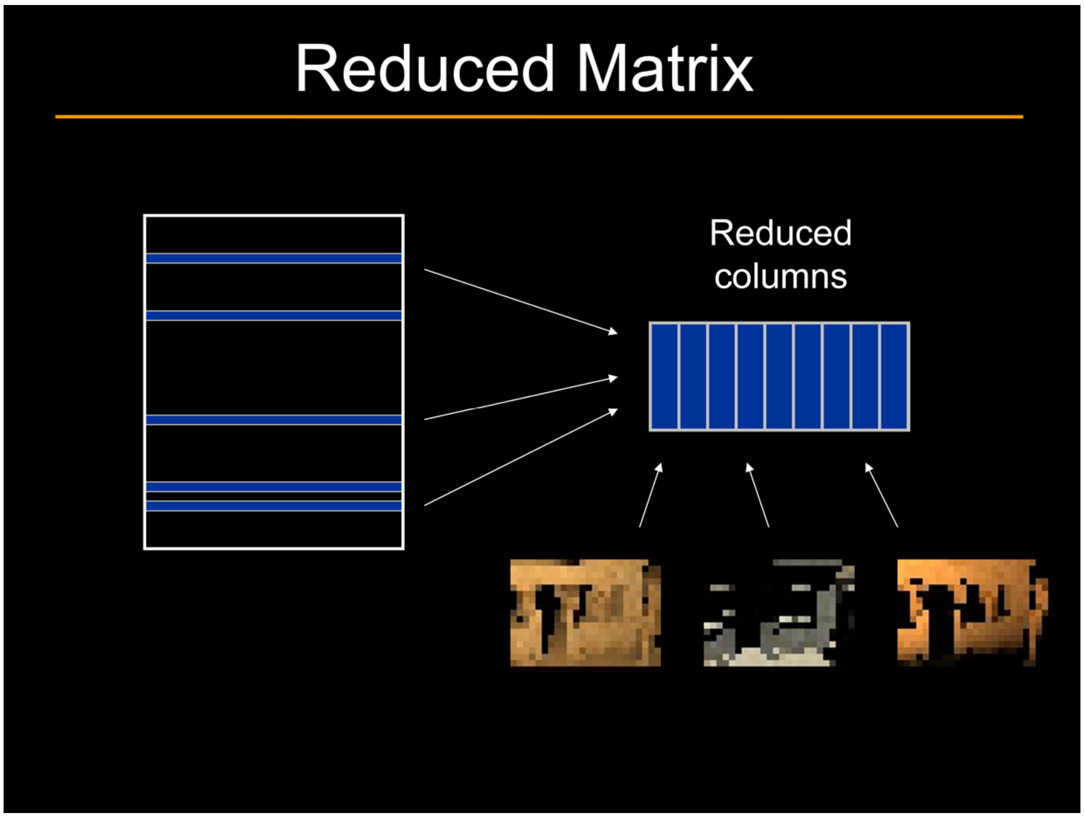
OK, we know how to compute rows and columns. How do we design the rest of the algorithm based on them? We use an idea that combines exploration and exploitation.

As a first step, we explore the structure of the matrix by computing a small, randomly chosen subset of rows.

Next, we analyze the gathered information, and decide which columns to choose and their appropriate weights.

The exploitation step then computes the selected columns, which are finally accumulated into an image.

The only thing missing I the contents of the black box that analyzes the rows and chooses the columns.



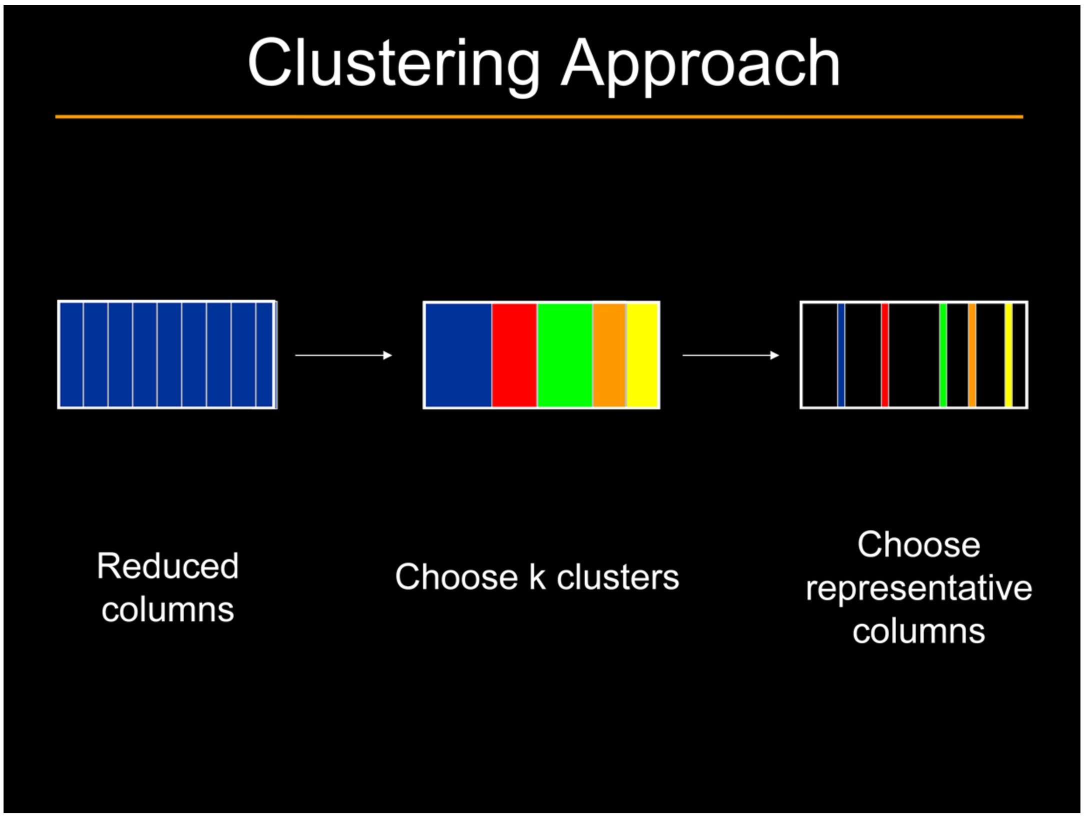
Reduced Matrix

To understand this, let’s take the rows that we computed in the exploration stage,

And assemble them into this long, but not very tall reduced matrix.

Now let’s flip attention to the columns of this matrix, which we’ll call reduced columns.

These can in fact be thought of as tiny images that are sub-sampled versions of the full columns.

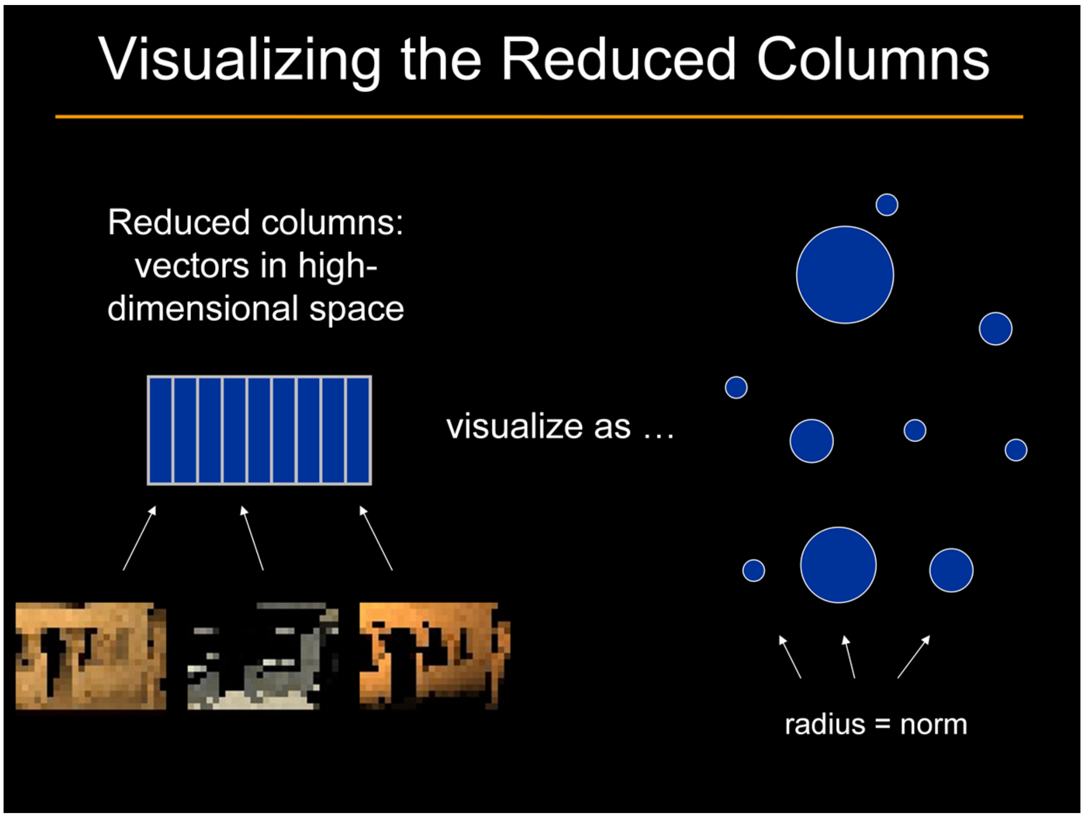


Clustering Approach

We will use a clustering approach to choosing columns: we cluster similar reduced columns, and we pick a representative in each cluster. Of course, we can arbitrarily re-shuffle the columns – there's no such thing as "neighboring columns".

Note that this is actually an unbiased Monte Carlo estimator: each representative, if properly weighted, computed an unbiased estimate of its own cluster. So the whole algorithm is an unbiased estimator of the sum of all lights.

The accuracy of the algorithm is highly dependent on the quality of the clustering, so we should design it carefully.



Visualizing the Reduced Columns

Reduced columns: vectors in high-dimensional space

visualize as …

radius = norm

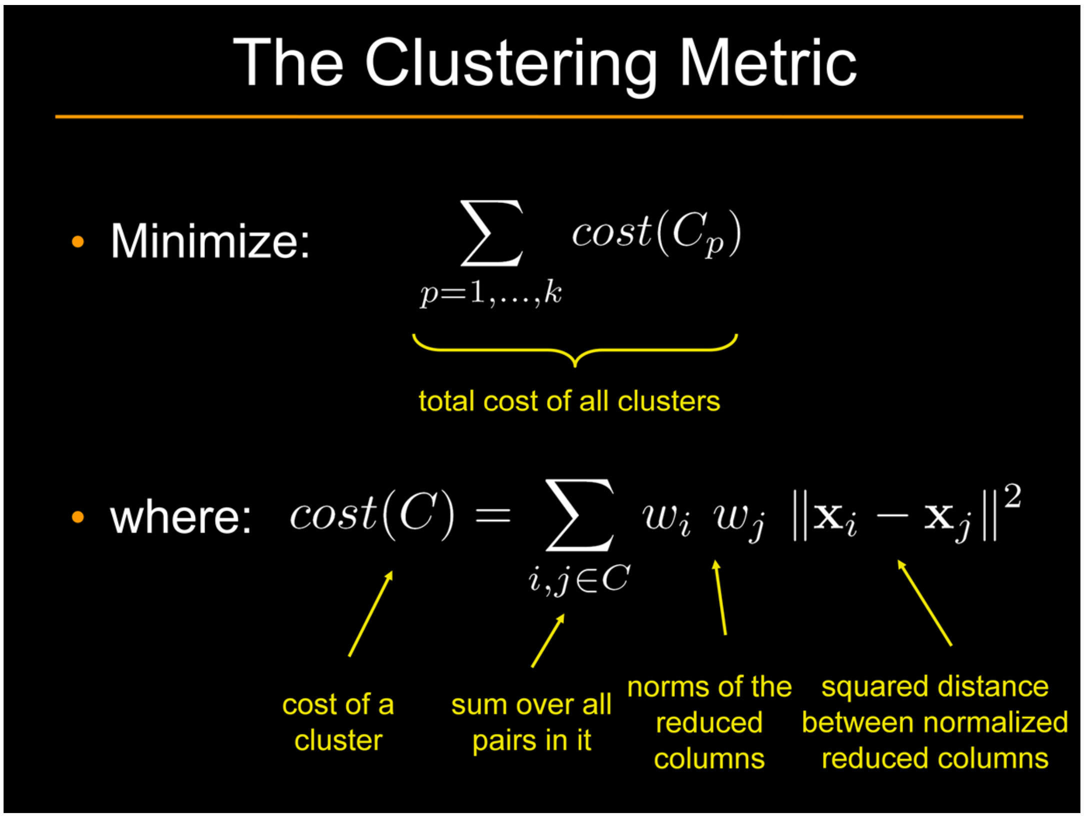
First, let’s think about the reduced columns as vectors in a high-dimensional space.

We’re going to visualize these high-dimensional vectors as circles.

The radius of each circle will correspond to the norm of the reduced column, or equivalently, to the brightness of the little image.

The positions will correspond to the positions of normalized reduced columns in the high-dimensional space.

With a bit of simplification, we can say that circles that are close to each other represent similar lights, and large circles represent lights with strong intensity.



The Clustering Metric

- Minimize: $$\underbrace{\sum_{p=1,\ldots,k} cost(C_p)}_{\text{total cost of all clusters}}$$
- where: $$cost(C) = \sum_{i,j \in C} w_i \; w_j \; \|\mathbf{x}_i - \mathbf{x}_j\|^2$$

cost of a cluster
sum over all pairs in it
norms of the reduced columns
squared distance between normalized reduced columns

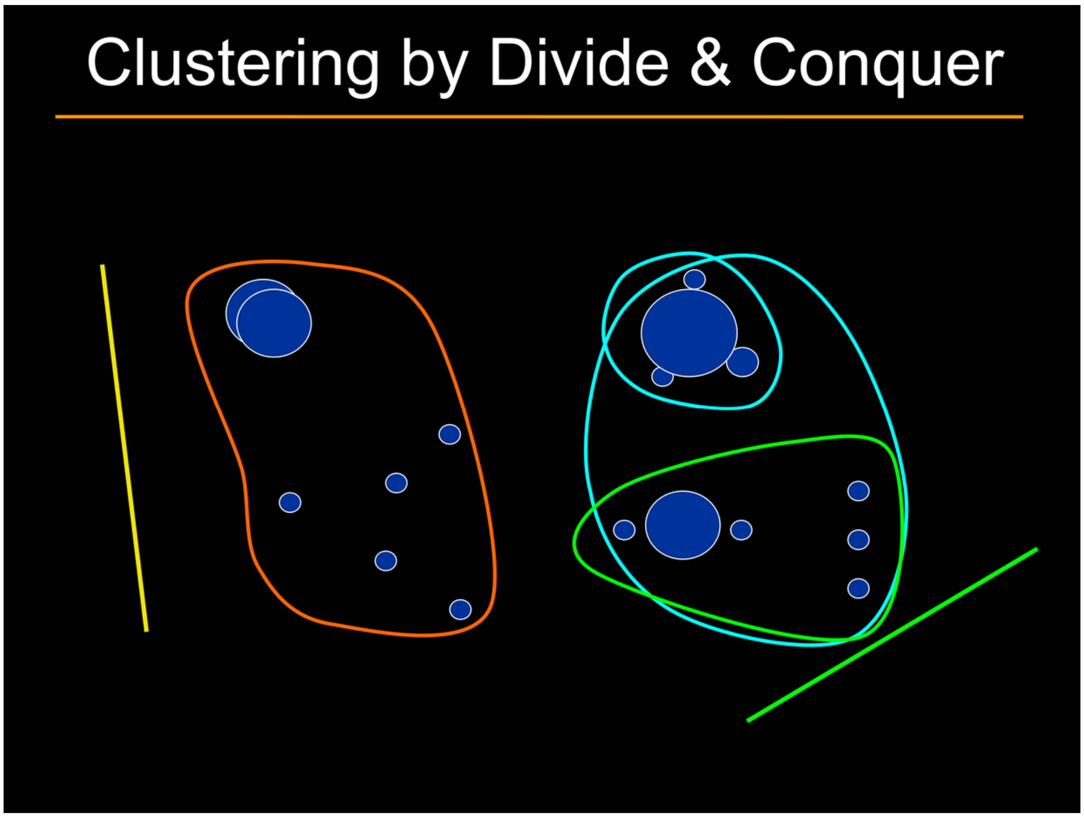
We can prove that the following formula gives the optimal clustering that minimizes the expected error.

So let's look at its meaning. We're minimizing the sum of costs of all clusters,

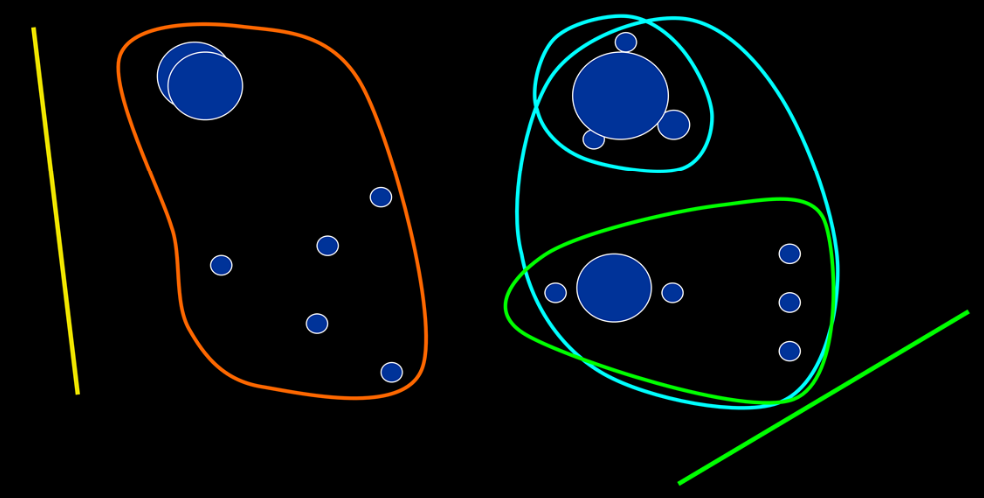
Where the cost of a cluster is defined as the sum over all pairs of elements in the cluster

of the product of norms and squared distance.

This confirms the intuition that strong lights, or lights that are far from each other, should be in separate clusters.



Clustering by Divide & Conquer

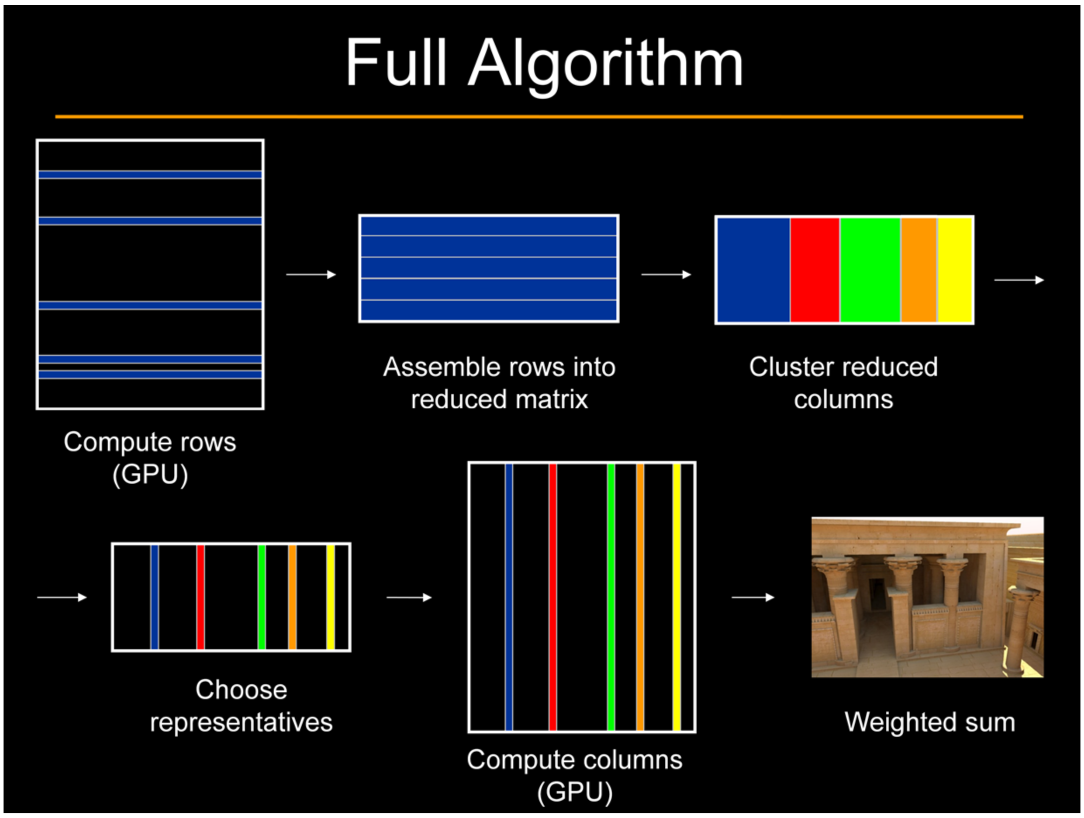


This is an NP-hard problem, but a good approximation can be found by the following divide & conquer algorithm.

We pick a plane with a random orientation. Remember this is in many dimensions, here I'll just visualize it as a line.

Then we move the plane to its optimal position and split the points into two clusters. There are only n possibilities so we can check them all to find the best one.

We then continue this process recursively on the cluster with the currently highest cost.



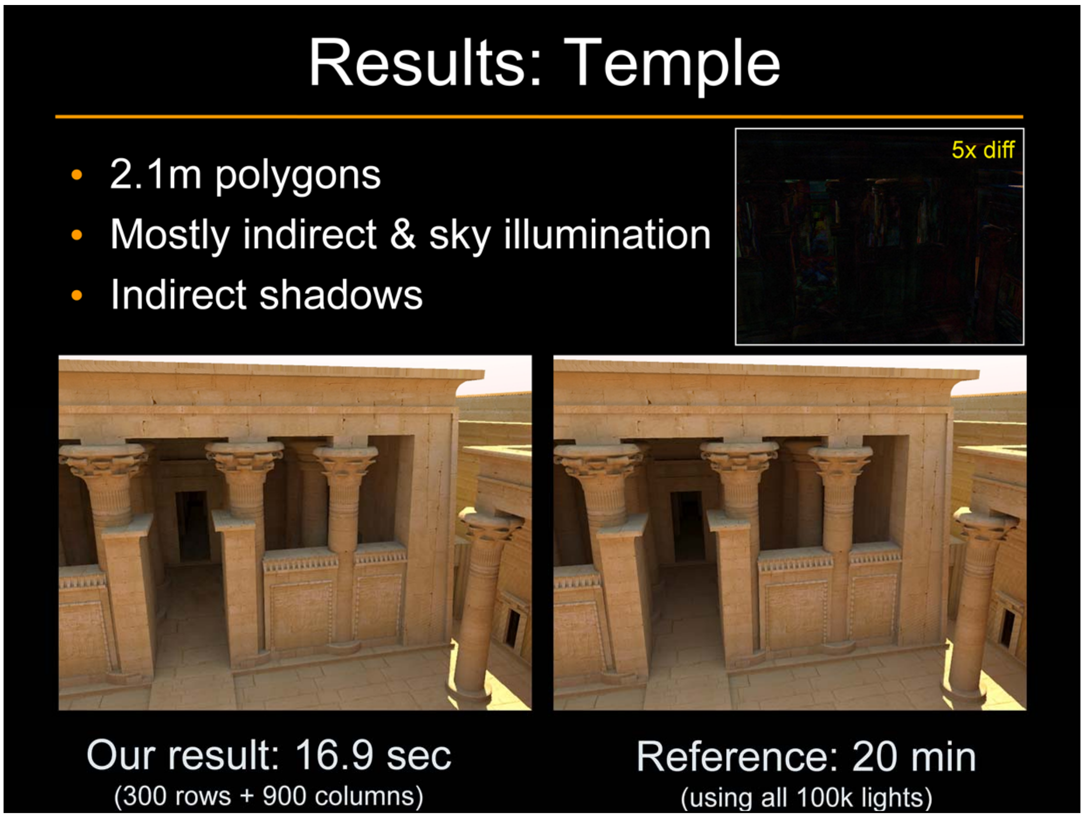
To summarize, here’s the full algorithm:

In the exploration stage, we evaluate some rows on the GPU, then focus on the reduced columns.

We obtain a clustering through the techniques I just described, then pick representatives.

Finally, we accumulate the corresponding columns with the correct weights into an image.

Let’s not lose the big picture: all of this is an abstraction that lets us increase the scalability of an underlying many-light algorithm.



Results: Temple

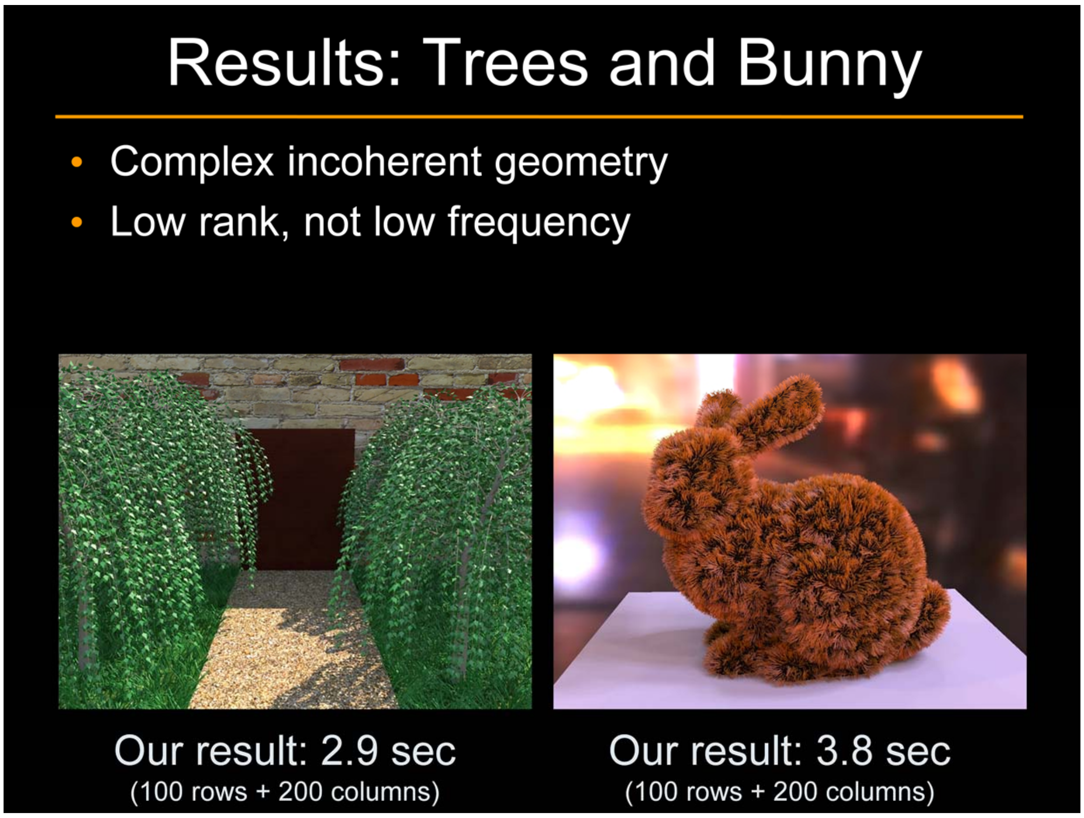
- 2.1m polygons
- Mostly indirect & sky illumination
- Indirect shadows

Our result: 16.9 sec
(300 rows + 900 columns)

Reference: 20 min
(using all 100k lights)

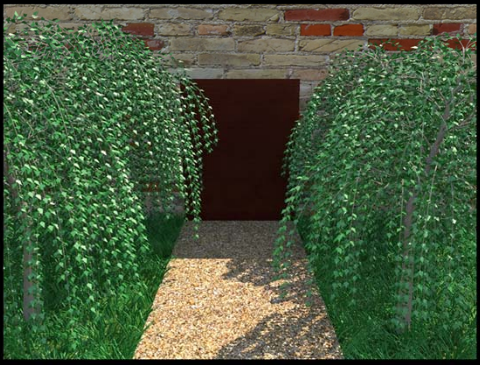
The temple is quite a large scene, not only in terms of the number of polygons, but also its spatial extent. In fact, only a tiny portion of the scene is in view. Most of the illumination in the scene is indirect or sky illumination, with the only pixels lit directly by the sun form the small bright patches on the right.

The method can quite effectively pick the small subset of VPLs that captures the illumination. Many VPLs are either invisible or have a small contribution, and can be aggressively clustered.

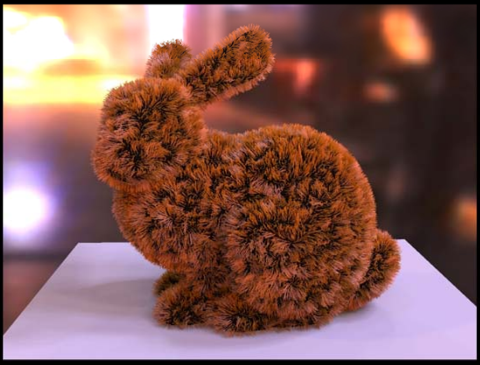


Results: Trees and Bunny

- Complex incoherent geometry
- Low rank, not low frequency



Our result: 2.9 sec
(100 rows + 200 columns)

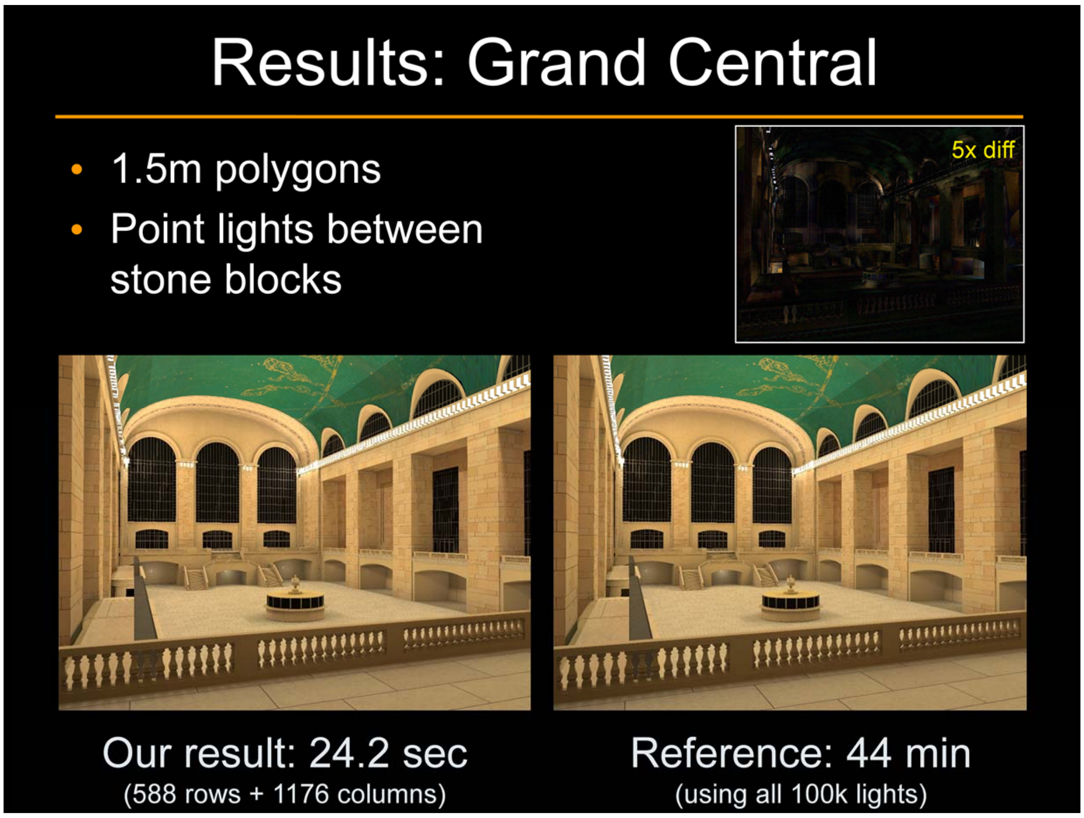


Our result: 3.8 sec
(100 rows + 200 columns)

These scenes are designed to test the algorithm by complex incoherent geometry, and it works quite welll.

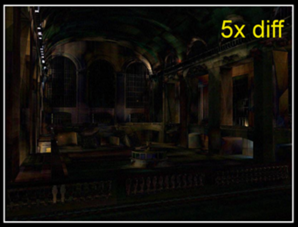
Approaches based on interpolating illumination across coherent surfaces would have a difficult time rendering these images.

However, our algorithm makes no assumptions about image-space coherence. In this sense, low-rank light transport can be a better approximation than smooth, low-frequency irradiance.



Results: Grand Central

- 1.5m polygons
- Point lights between stone blocks

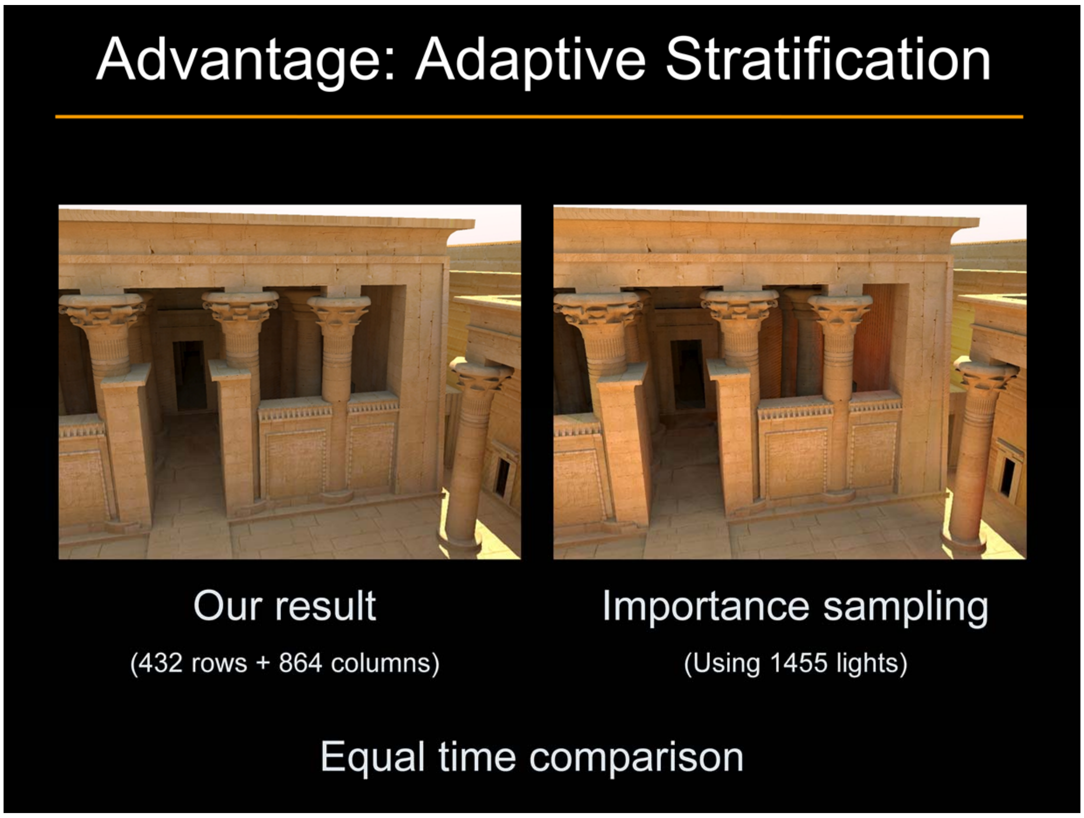


Our result: 24.2 sec
(588 rows + 1176 columns)

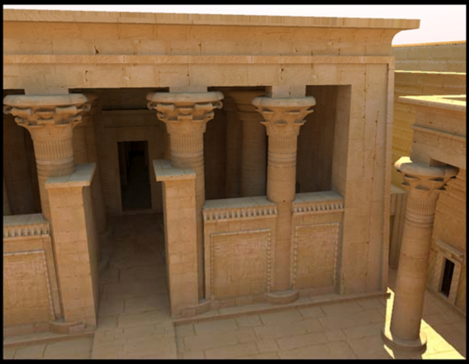
Reference: 44 min
(using all 100k lights)

This scene, the Grand Central station, is a bit of a difficult case. A unique feature of the scene are the omni-directional lights positioned in small recesses between stone blocks.

These lights pose a problem since they violate the low-rank assumption. The columns corresponding to these lights are pretty much linearly independent. Therefore, a larger number of rows and columns will be required here, and some lights are still missing on the left side.

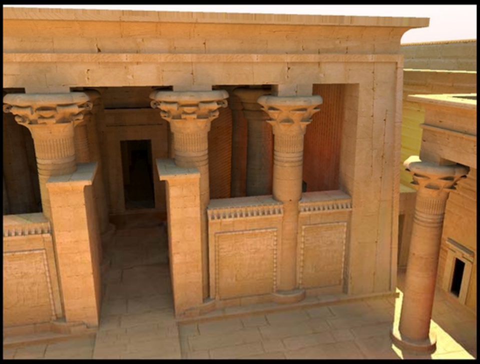


Advantage: Adaptive Stratification



Our result

(432 rows + 864 columns)



Importance sampling

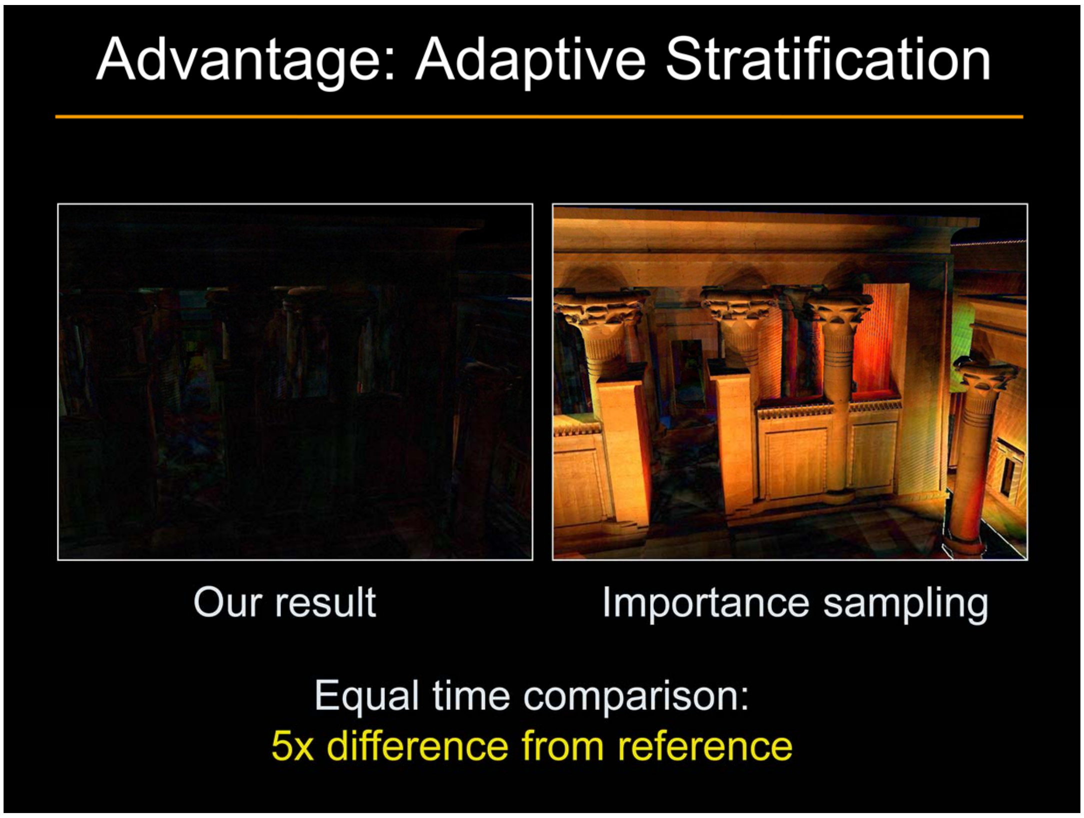
(Using 1455 lights)

Equal time comparison

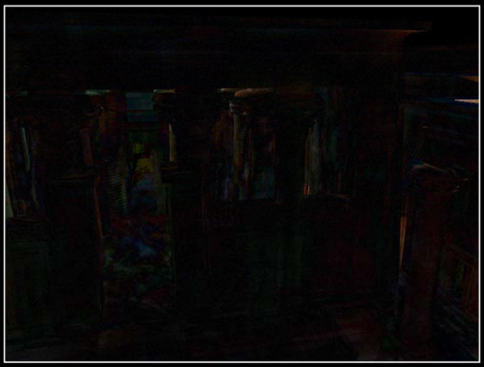
An important question is whether the row sampling step of our algorithm is more useful than brute force.

Indeed, instead of creating 100,000 lights and trying to pick a small subset of them, we could simply create a smaller number of lights and render them all.

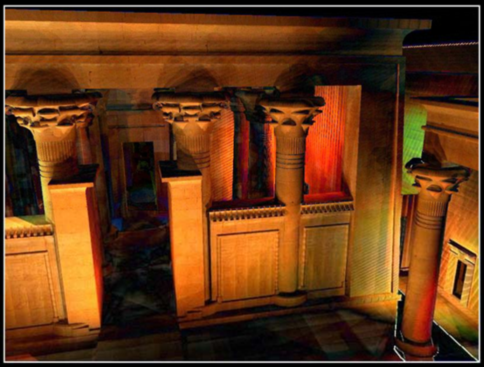
However, if we compare these images, we find that the simpler approach produces much more objectionable artifacts despite taking a bit longer.



Advantage: Adaptive Stratification



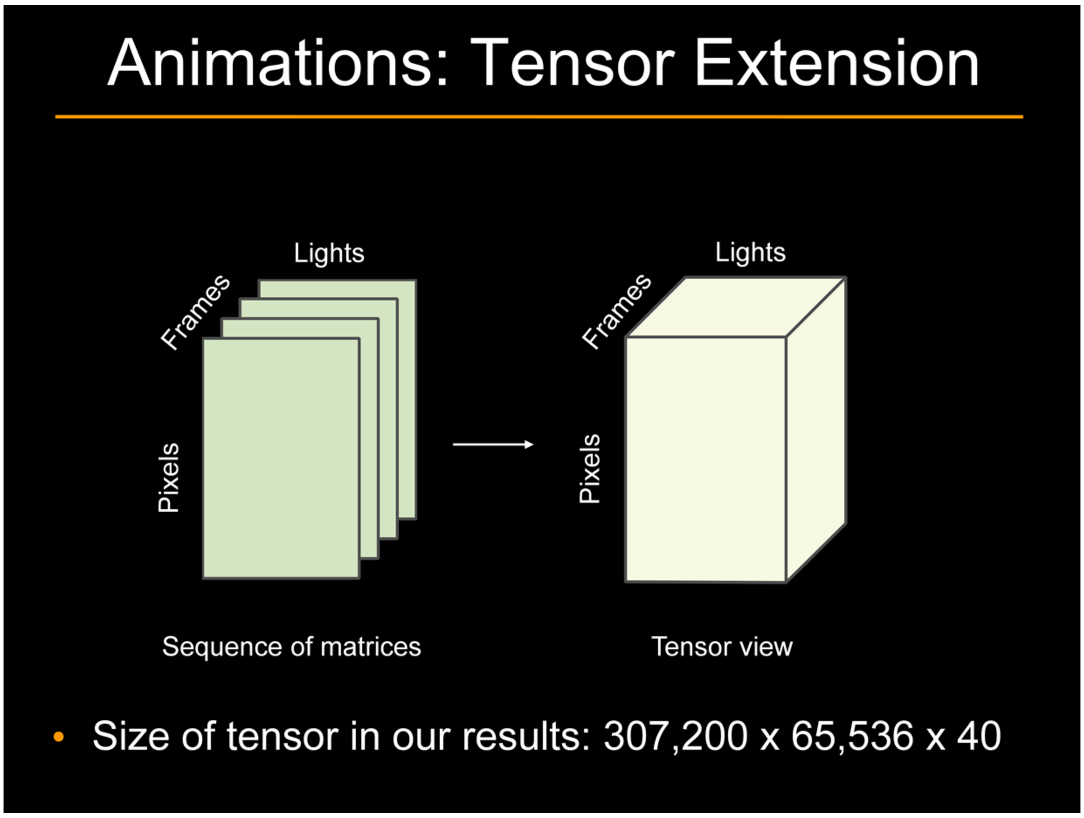
Our result



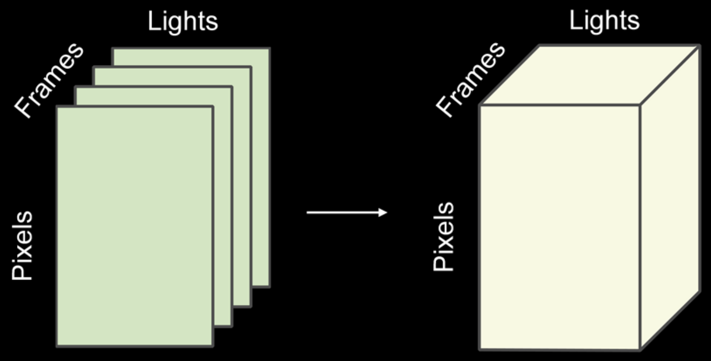
Importance sampling

Equal time comparison:
5x difference from reference

Here are difference images as well.



Animations: Tensor Extension



Sequence of matrices

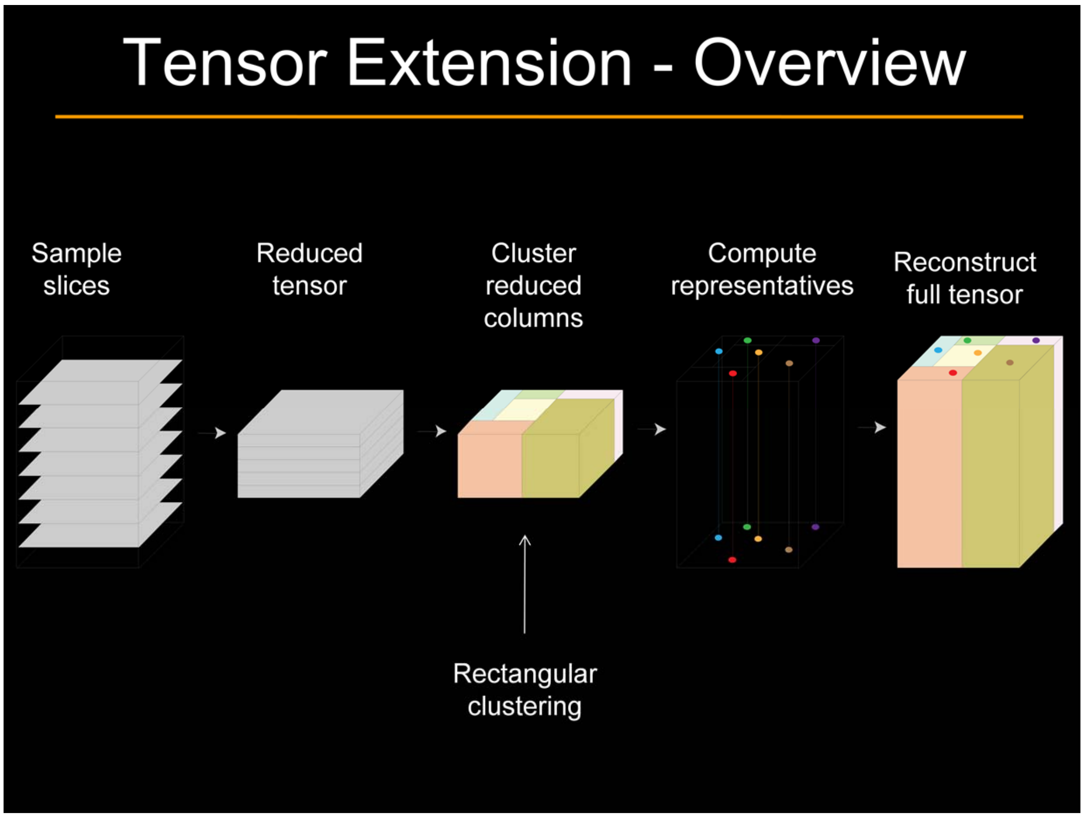
Tensor view

- Size of tensor in our results: 307,200 x 65,536 x 40

In row-column sampling, we had a different matrix in every frame. Let’s concatenate them,

and consider all of them at once as one large 3D array (or tensor) of lights contributing to pixels over frames.

The amount of data is very large, and we need to avoid constructing the whole tensor at any point in the algorithm, similar to the matrix case.

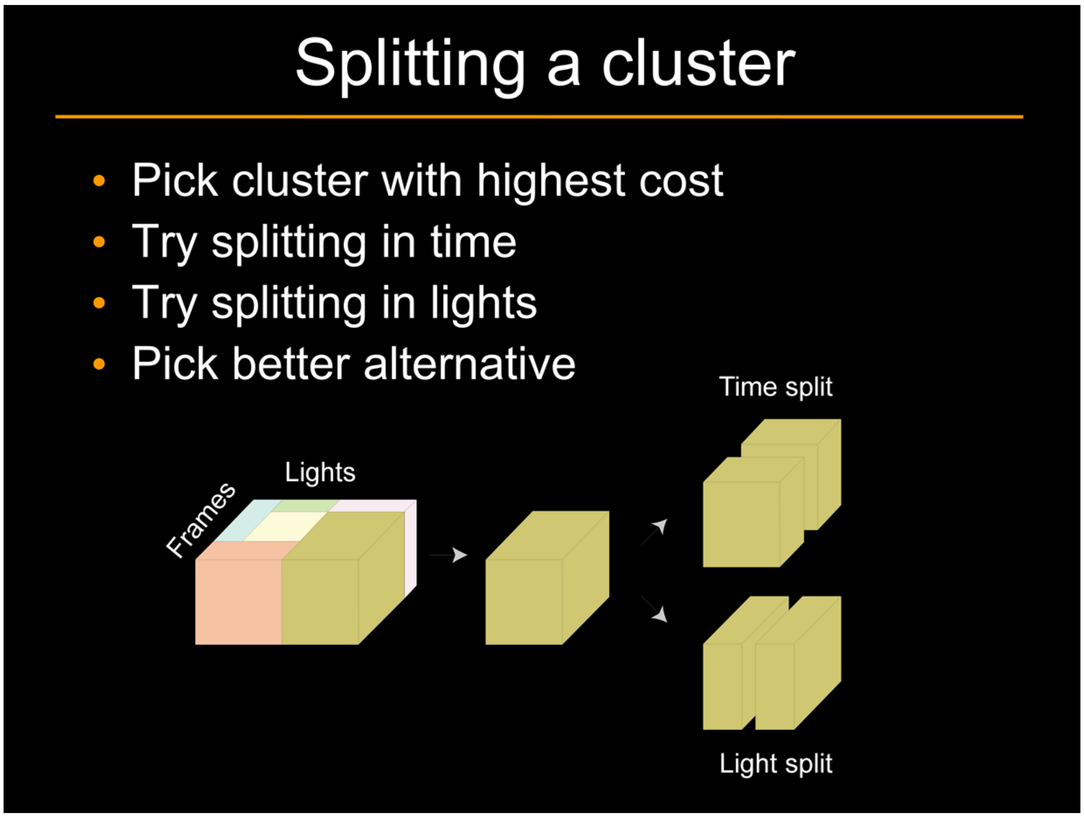


Tensor Extension - Overview

The extension will be done as follows: We compute a set of rows in every frame, resulting in a set of horizontal slices of the tensor.

Then we cluster the columns of this reduced tensor; I call this a “rectangular clustering” since every cluster is a cartesian product of a light subset and a frame subset.

We can then use a similar representative selection and reconstruction as before, except here we have to be careful, because pixels move between frames. A pixel mapping trick similar to optical can be used to warp the representatives to avoid this problem.



Splitting a cluster

- Pick cluster with highest cost
- Try splitting in time
- Try splitting in lights
- Pick better alternative

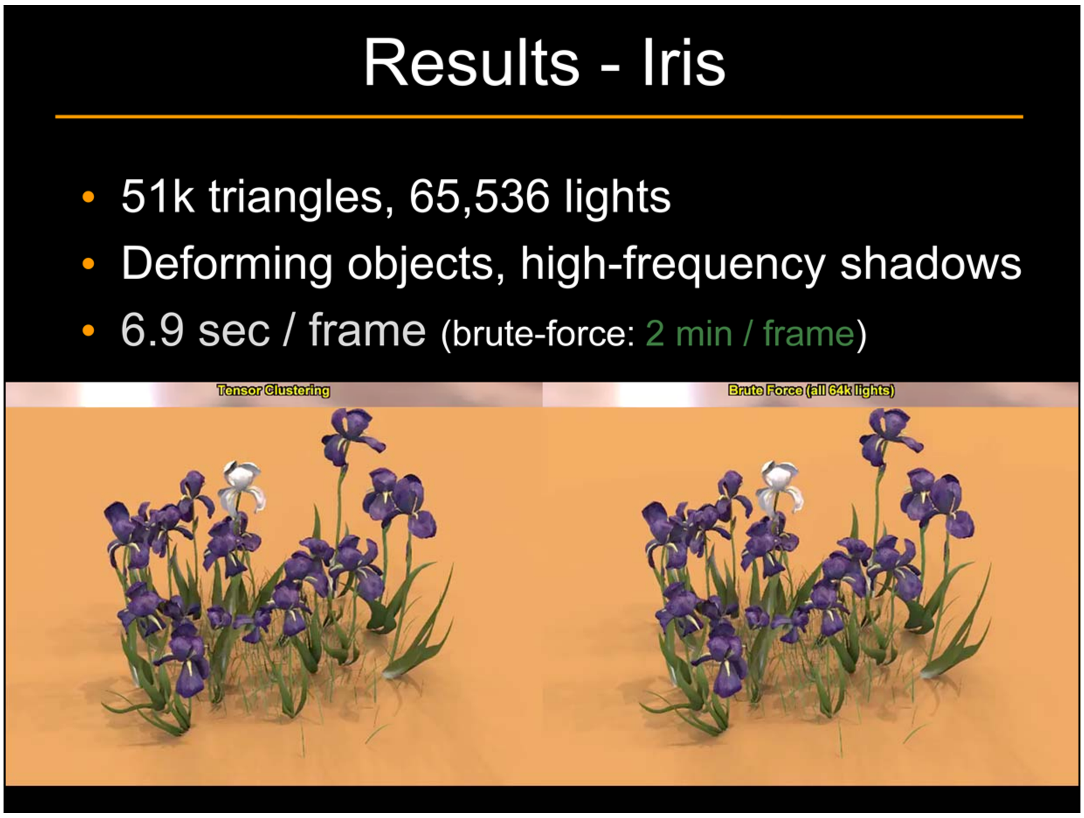
Time split
Lights
Frames
Light split

How do we find a rectangular clustering? This problem is again NP-hard, but we can adapt the divide-and-conquer as follows.

We will start with all light-frame pairs in one cluster, and keep splitting the cluster with highest cost until we reach the desired number of clusters.

We will try splitting the cluster in time and in lights, and pick the split that gives us the better objective function value.

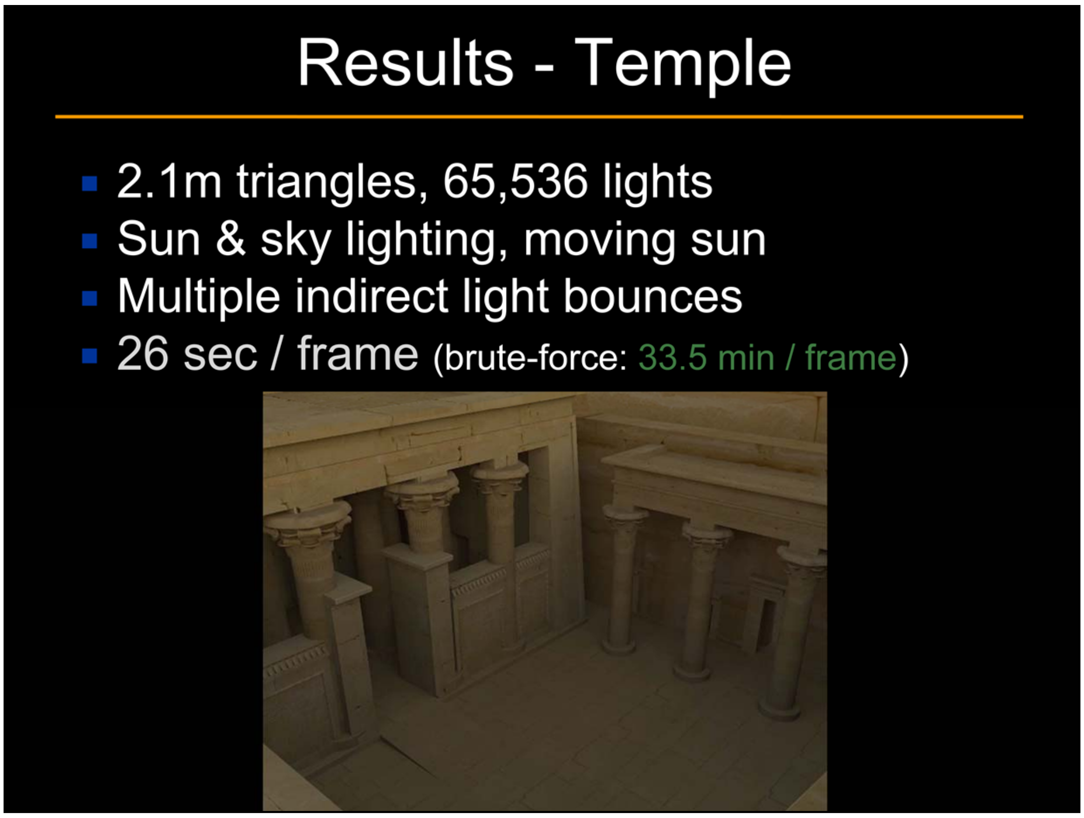
Now the question is, how to split in time and in lights; we can answer this by trying both splits and seeing which one decreases the objective function more.



Results - Iris

- 51k triangles, 65,536 lights
- Deforming objects, high-frequency shadows
- 6.9 sec / frame (brute-force: 2 min / frame)

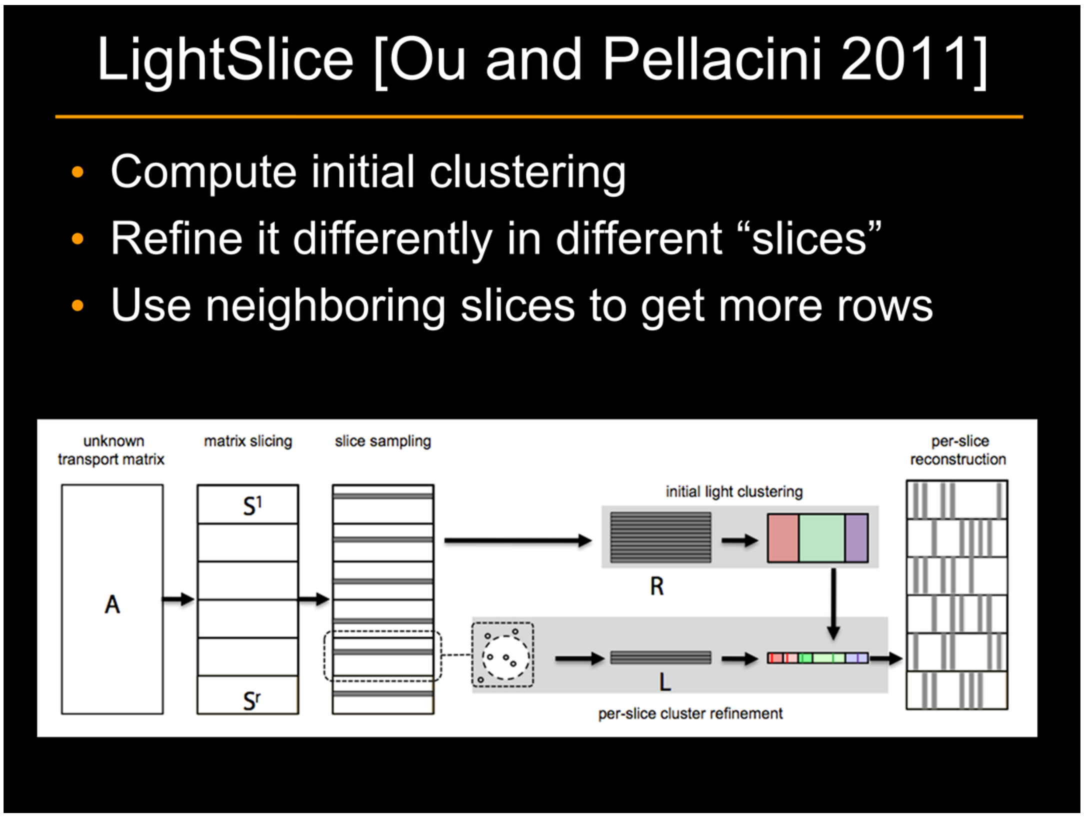
Here is an example animation rendered with this method. On the left is the result, and on the right is reference with all VPLs, which took about 9 times longer.



Results - Temple

- 2.1m triangles, 65,536 lights
- Sun & sky lighting, moving sun
- Multiple indirect light bounces
- 26 sec / frame (brute-force: 33.5 min / frame)

And here is another example with a temple and a moving sun. This one is about 77x faster than the brute-force solution.

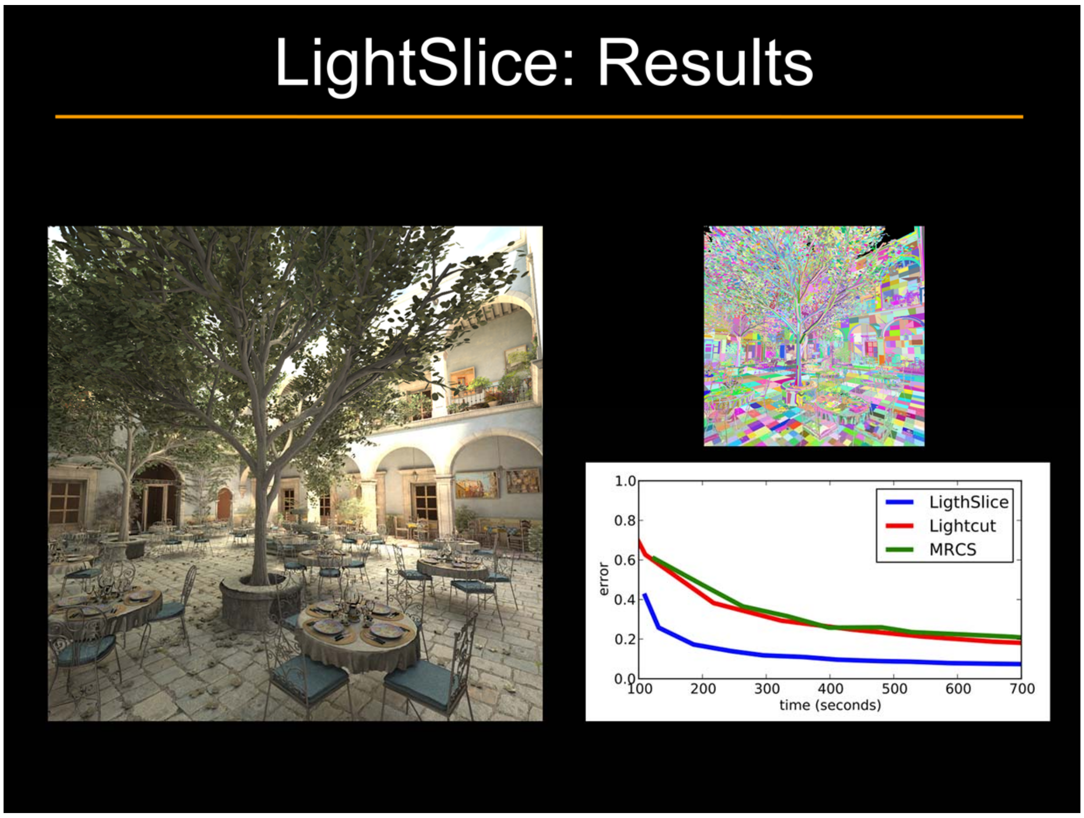


LightSlice [Ou and Pellacini 2011]

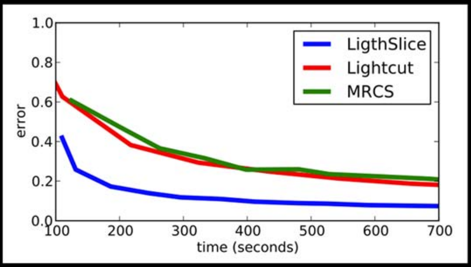
- Compute initial clustering
- Refine it differently in different "slices"
- Use neighboring slices to get more rows

Another nice idea, published recently in the LightSlice paper, is to refine the original clustering within "slices" of the image.

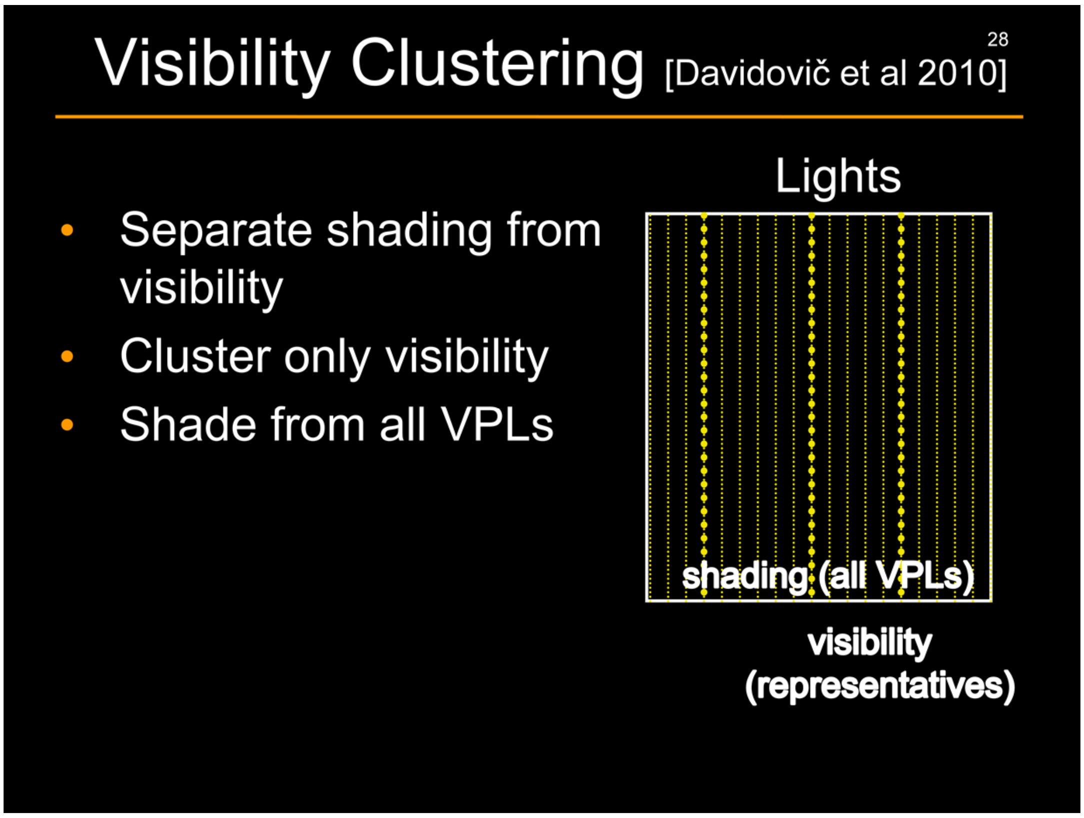
One problem is that each slice will only have one row sample, and one-dimensional data is not enough to run a clustering algorithm. This is addressed by using neighboring row samples to help.



LightSlice: Results

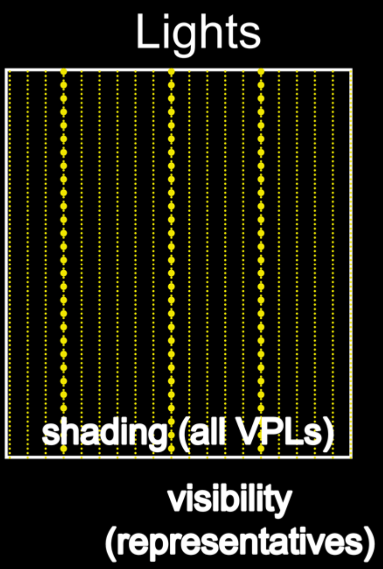


This works very well - in fact, on this fairly complex scene, the algorithm seemsto outperform both lightcuts and row-column sampling.



Visibility Clustering [Davidovič et al 2010]

- Separate shading from visibility
- Cluster only visibility
- Shade from all VPLs

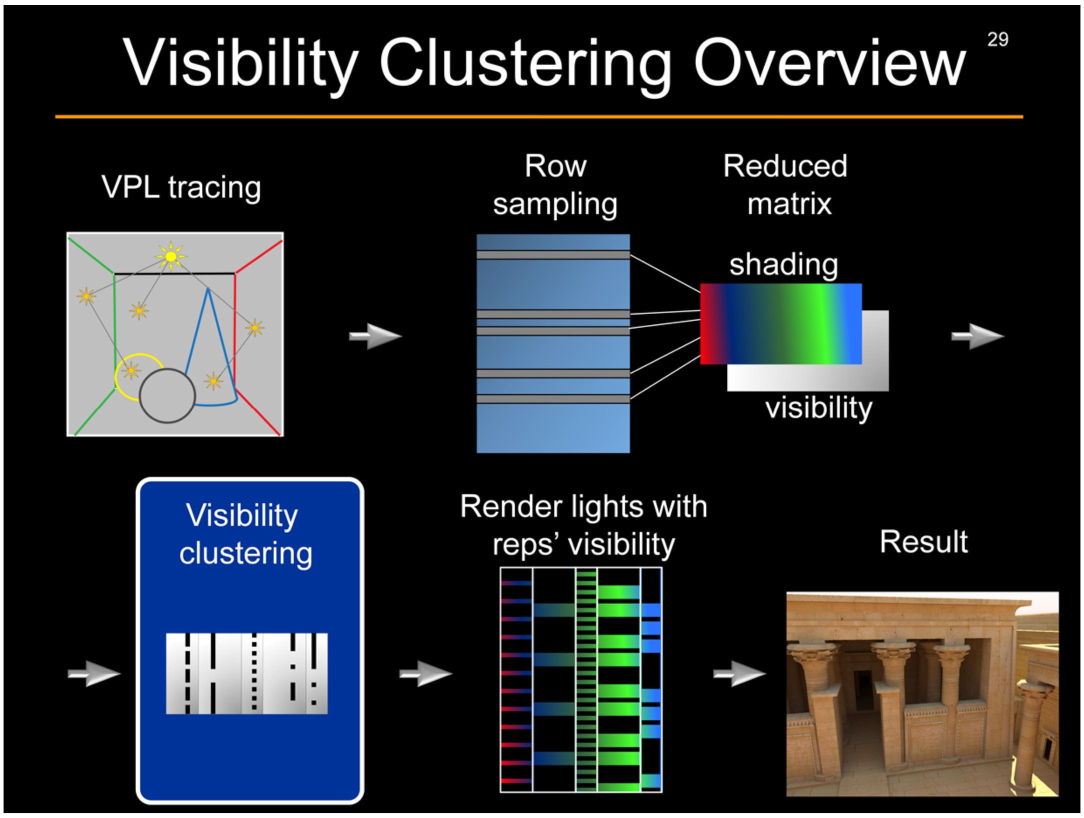


Next, I will introduce an interesting variation of the row-column approach that is some cases more robust.

The design of the algorithm is motivated by several observation on shading and visibility in glossy scenes.

First, we observe that in glossy scenes we cannot easily pick a small subset lights to approximate the solution, because shading from the individual lights is quite different. In other words, we want to compute the shading from all the global VPLs. And that’s actually doable because the GPU is extremely efficient at computing the shading.

But we just cannot afford to evaluate a SM for each of those 200k lights! So the key insight is to separate shading from visibility, use ALL the light for shading, and only a small number of representative lights for visibility.



Visibility Clustering Overview

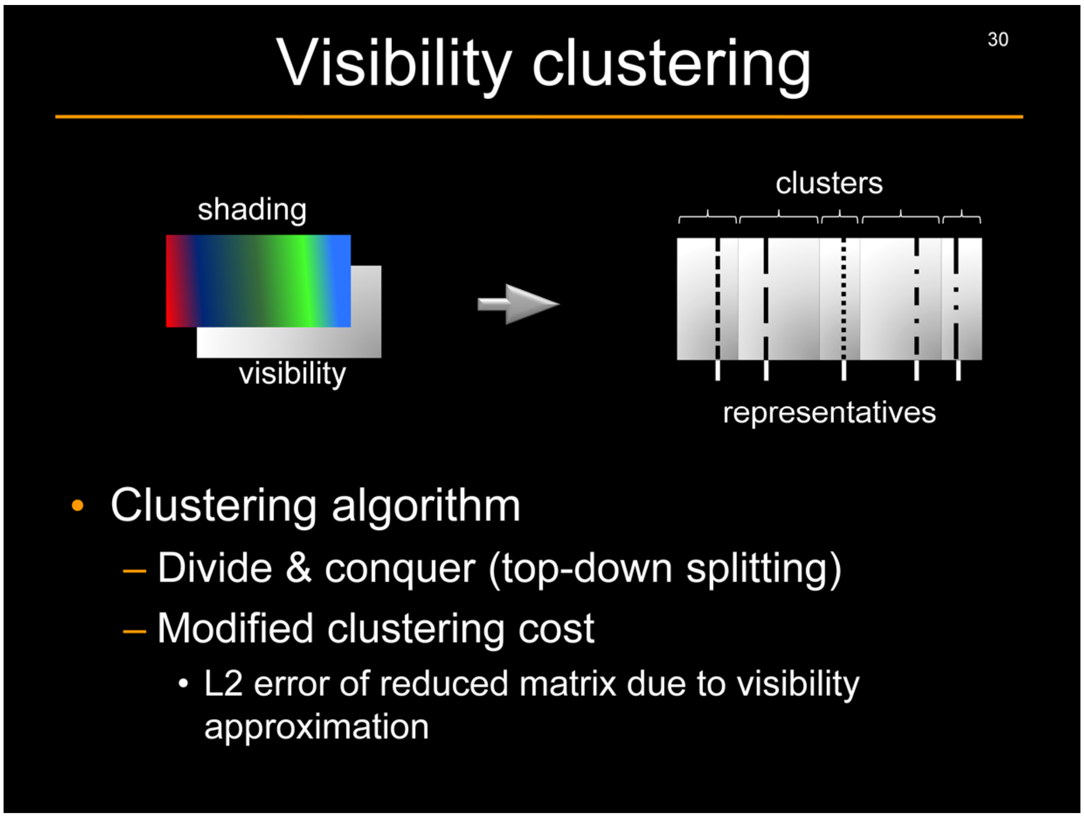
29

Here's the overview of visibilty clustering. We start by distributing the global VPLs in the scene by tracing particles frOm light sources.

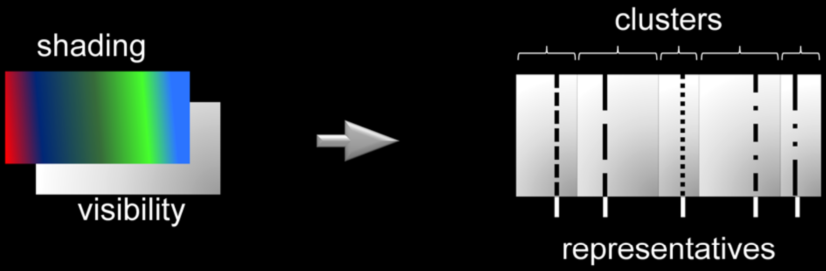
After that, we sample a number of rows of the interaction matrix, which means that we pick a number of pixels and for each of them, we evaluate the shading and visibility for all the lights. That gives us the reduced shading and visibility matrices.

The visibility clustering then analyzes these matrices and yields clusters of lights that will share the same shadow map. After that, we render all the VPLs with the shadow map of the representative light for each cluster. This yields the complete global solution.

There's only one bit that remains to be explained here, and that's the visibility clustering algorithm.



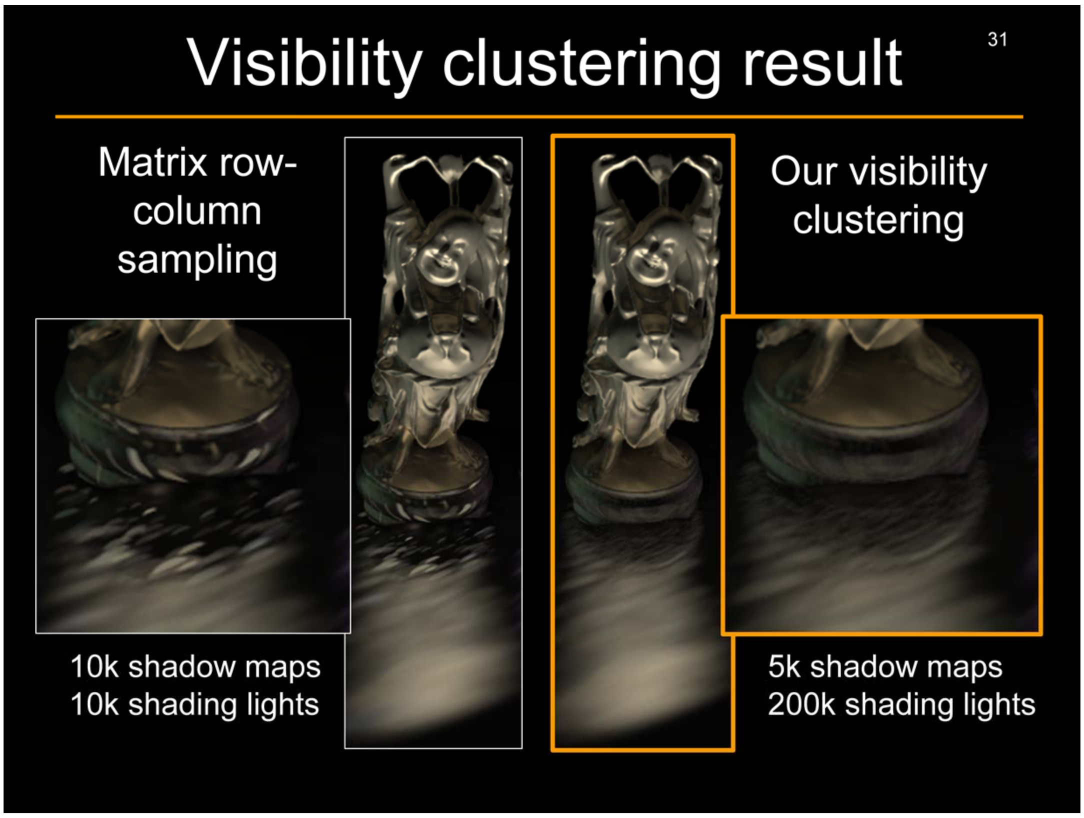
Visibility clustering



- Clustering algorithm
  - Divide & conquer (top-down splitting)
  - Modified clustering cost
    - L2 error of reduced matrix due to visibility approximation

The goal of the visibility clustering algorithm is to group VPLs into clusters that will share the shadow map of its representative VPL. We use a data-driven approach where we analyze the row samples from the light interaction matrix.

The clustering algorithms proceeds in a top-down fashion, splitting clusters with highest cost, which is defined as the L2 error of the matrix incurred by the visibility approximation.



Visibility clustering result

31

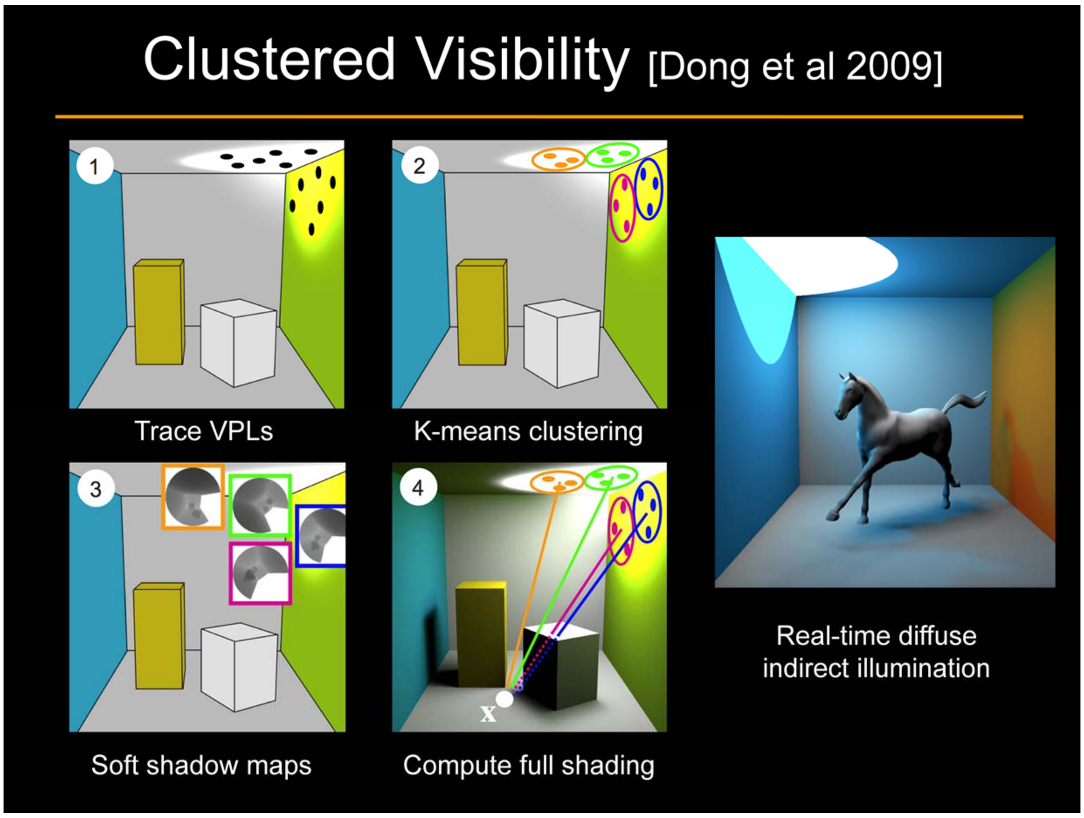
Matrix row-column sampling

10k shadow maps
10k shading lights

Our visibility clustering

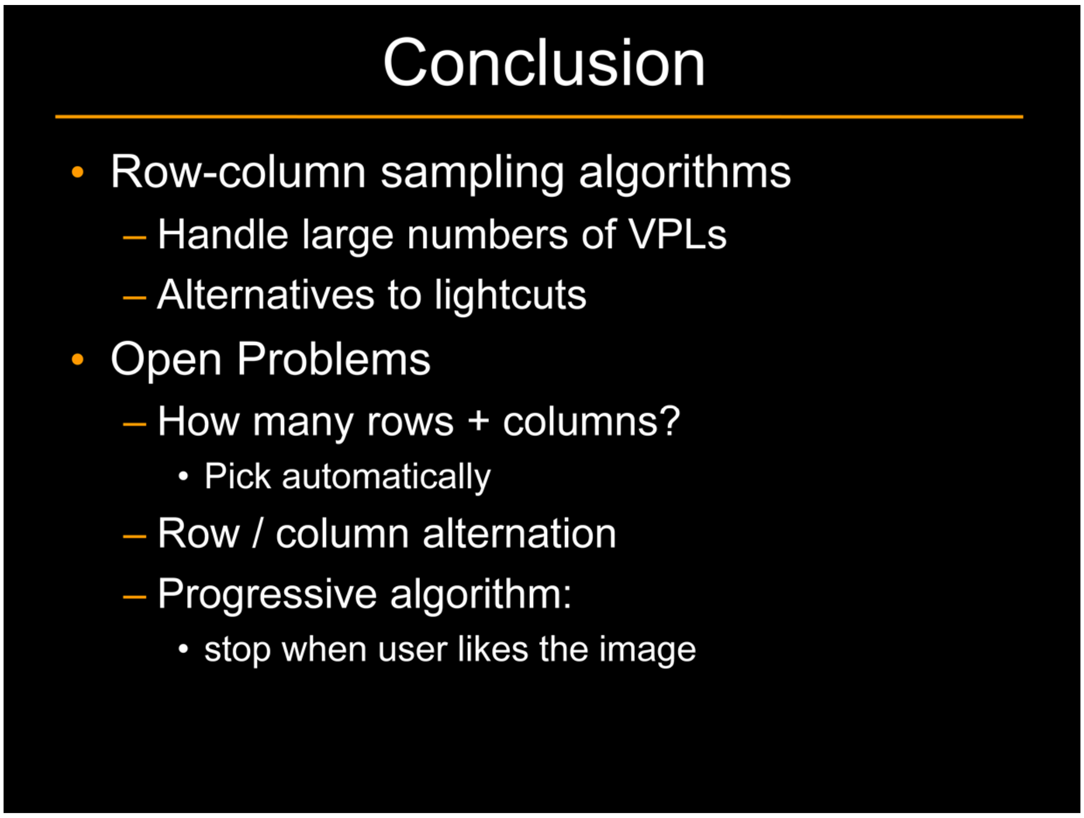
5k shadow maps
200k shading lights

Here’s a comparison of the result of our visibility clustering with MRCS in the same time for a simple scene of the happy Buddha statue reflected in a glossy plane. The MRCS result uses 10k representative lights (i.e. 10k shadow maps and 10k lights for shading), which leaves visible splotches in the image. The visibility clustering, in the other hand, uses only 5k shadow maps for visibility but all the 200k lights for shading, substantially improving the smoothness of the reflections.


Clustered Visibility [Dong et al 2009]
1
Trace VPLs
2
K-means clustering
3
Soft shadow maps
4
x
Compute full shading
Real-time diffuse indirect illumination

Another alternative approach I should mention uses no matrix formulations, but instead uses a simple visibility clustering idea that nevertheless works well in simple scenes and can even achieve real-time performance.

The idea is to use k-means clustering for visibility, and render full shading. One may think that this for highly variable VPL intensities, but for a single bounce, and one lightsource, one can easily make all VPLs same intensity. The approach also uses soft shadow maps, so it can get away with pretty small numbers of shadow maps.



Conclusion

- Row-column sampling algorithms
  - Handle large numbers of VPLs
  - Alternatives to lightcuts
- Open Problems
  - How many rows + columns?
    - Pick automatically
  - Row / column alternation
  - Progressive algorithm:
    - stop when user likes the image

Here are some of the open problems in this area. How to pick the number of rows and columns automatically?

It would be nice to design a variation of the algorithm that alternates row and column sampling, since knowing some columns might tell us which rows to sample, but no one seeknows how to do it.

It might also be useful to make the algorithm progressive, so the user can stop the evaluation when they are satisfied with the image.

A temporal version of the technique that renders multiple frames at once would be great. The problem is – how to do it so that the representatives do not have to be kept in memory?

Finally, there are some other approaches for matrix completion in the literature, but we tried them for rendering and they did not turn out to be better. However, there might be ways to improve them, they just haven’t been studied systematically.