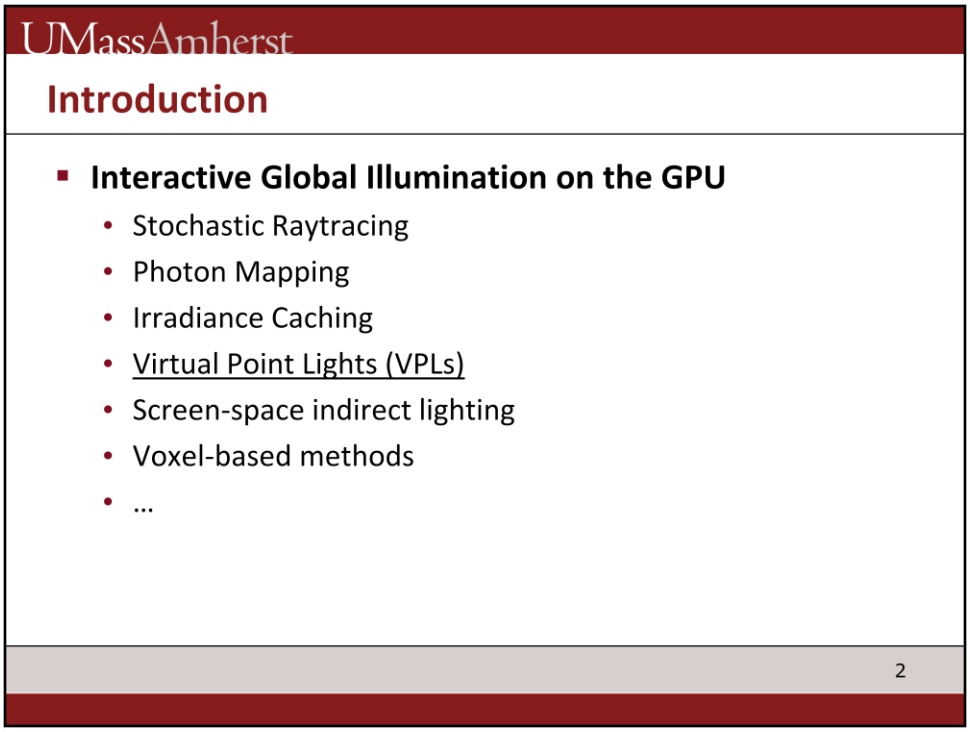
UMassAmherst

Importance Point Projection for GPU-based Final Gathering

David Maletz and Rui Wang

Univ. of Massachusetts Amherst

In this work, our goal is to achieve global illumination on the GPU using an importance-driven point projection method.

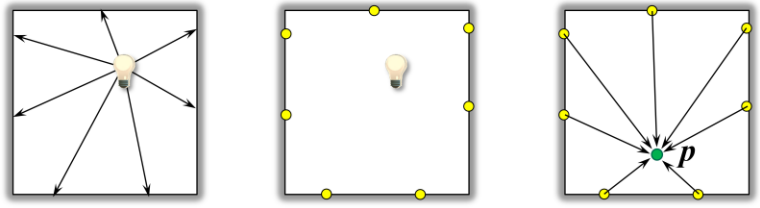


Global illumination is a classic problem in graphics research, and many recent techniques exploit the GPU to achieve interactive speed. Among them, virtual point lights are quite popular because of its simplicity and suitability for parallel processing.

UMassAmherst

Point-Based Global Illumination

- **Instant Radiosity** [Keller 97]



Direct Lighting *VPLs* *Final Gather*

Here is a simple illustration of the idea. Assume the direct illumination of the scene can be represented >>> onto a discrete set of points called the VPLs; then >>> given a shading point p, the contributions of the VPLs will be accumulated at that point to approximate indirect lighting. The accumulation is typically referred to as the final gather step.

UMassAmherst

Point-Based Global Illumination

- **Selection/Clustering VPLs**
  - Hierarchical clusters [WFA*05, Chr08, REG*09]
  - Importance sampling [SIMP06, SIP07, GS10]
  - Matrix row-column sampling [HPB07]
  - Incremental instant radiosity [LSK*07]
- **Solving Visibility**
  - Shadow maps [Kel 97, HPB07]
  - Imperfect shadow maps [RGK*08]
  - Voxel/volume based [NPW10, KD10, THGM11]
  - Rasterization z-buffer [Chr08, REG*09]

Most VPL-based techniques are focused on solving three problems. First, the number of VPLs directly affect the computation cost of the final gather, therefore there must be an efficient way to either select representative VPLs or group them into cluster to reduce the computation cost.

Second, the visibility between each VPL and the shading point must be resolved, which can be done through a variety of techniques, such as shadow maps, z-buffers, or voxel-based approaches.

UMassAmherst

Point-Based Global Illumination

- **Glossy VPLs**
  - Virtual Spherical Lights [HKWB09]
  - Combine global and local VPLs [DKH*10]
  - Reflectance filtering [LWDB10]

Finally, although traditional VPLs are assumed to be diffuse, there are several recent papers that study how to incorporate glossy VPLs in order to produce glossy-to-glossy reflection effects.

UMassAmherst

Point-Based Global Illumination

- **Glossy VPLs**
  - Virtual Spherical Lights [HKWB09]
  - Combine global and local VPLs [DKH*10]
  - Reflectance filtering [LWDB10]
- **Hierarchical Point Rasterization** [Chr08, REG*09]
  - A single point cloud to represent both illumination and geometry.
  - Rasterize points to microbuffers by traversing a hierarchical structure (i.e. tree) of the points.

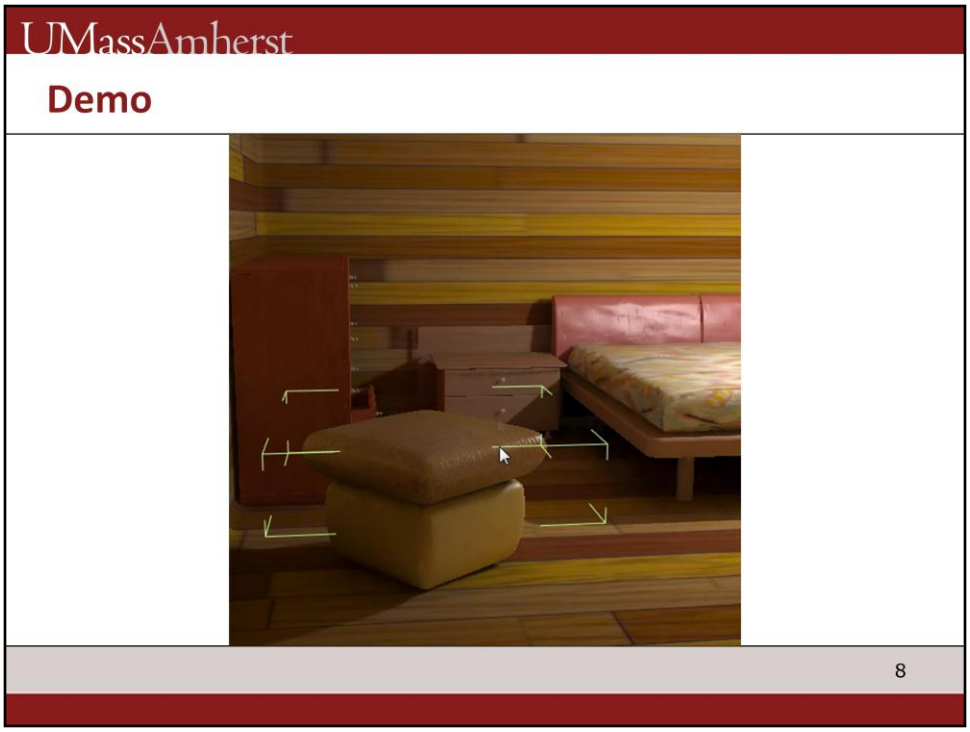
Among the existing work, the most relevant to ours are two papers published recently, where they use a single point cloud to represent both illumination and geometry. The main idea is to rasterize a microbuffer at every shading point by traversing a hierarchical structure of the points, and the microbuffer also serves as a z-buffer to resolve visibility.

UMassAmherst

Our Contributions

- **Our Goal:**
  - An alternative method that uses importance sampling and projection of points instead of a hierarchical approach.
- **Benefits:**
  - Simplicity.
  - Incorporation of glossy VPLs.
  - No requirement to build a hierarchy of poitns on the fly.

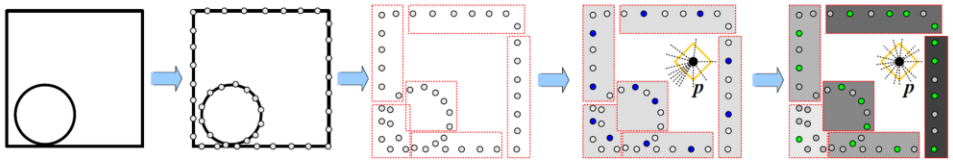
In this work, our main goal is to study an alternative method, where instead of building and traversing a point hierarchy, we use importance sampling and projection of the points to generate microbuffers. The reason we want to study such a method is because it provides several benefits, including its simplicity, its capability to directly incorporate glossy VPLs. In addition, it does not require building a tree of the points on the fly, therefore it can reduce the computation cost for fully dynamic scenes.



This video demo shows our algorithm captured in real time. The user can modify any part of the scene, including the objects, light sources, spatially varying material properties and the viewpoint. Our method can faithfully capture indirect reflections as well as shadows, such as the area under the mattress shown here. And this is a Cornell box scene that contains some glossy objects. As the rendering is updated each frame, some noises are noticeable, which is due to the random sampling and progressive rendering as I will explain later. But you can see that overall the rendering quality converges very quickly within a second.

UMassAmherst

Algorithm Overview



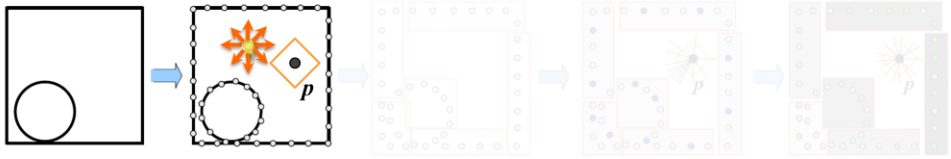
1. Represent scene as points;
2. Split points into clusters;
3. ***Random sampling*** to evaluate cluster importance;
4. ***Importance sampling*** to exploit cluster importance;
5. ***Project*** both random and importance samples to a microbuffer.

This is a one slide quick overview of the our algorithm. We first discretize the scene into a set of uniformly distributed points, which are then partitioned into clusters; next, for each cluster we use random sampling to evaluate its importance; finally we draw importance samples from each cluster, and both the random and importance samples are projected into the microbuffer, which will be integrated with the BRDF to compute indirect lighting.

4 minutes here.

UMassAmherst

Algorithm Overview



- **Represent scene and illumination as points**
  - The scene geometry is sampled into 256K points (**scene points**) using parallel surface Poisson disk sampler [BWWM10].
  - Compute direct illumination radiance L_D for each point, consisting of 1 diffuse radiance and up to 4 glossy lobes.

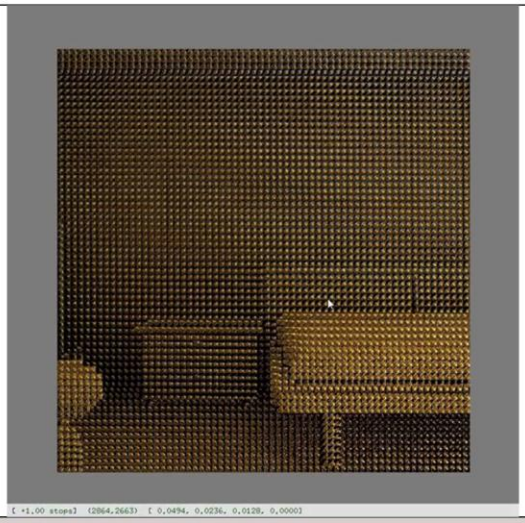
Now let me explain each step in detail. In the first step, we discretize the scene geometry into 256K points which we call the scene points. This is done using a parallel Poisson disk sampler on the surface. For each point >>> we compute its direct illumination radiance received from the main light sources. This includes 1 diffuse radiance and up to 4 glossy radiance lobes if the point is on a glossy material.

Our ultimate goal is >>> project these points into the microbuffer of a shading pixel.

UMassAmherst

Examples of Microbuffer

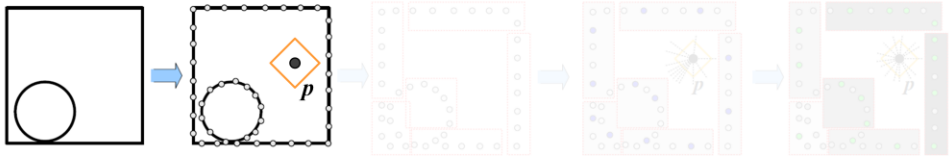
- *Tiny* environment map at every pixel.
- Hemi-octahedral mapping [*WNLH06*]
- 32x32 texture.



Here is a visualization of the microbuffer computed at every pixel in this image. You can see that each microbuffer is essentially a tiny environment map that stores the incident radiance field at every pixel. It is parameterized using hemi-octahedral map and stored as 32x32 texture in GPU memory.

UMassAmherst

Algorithm Overview

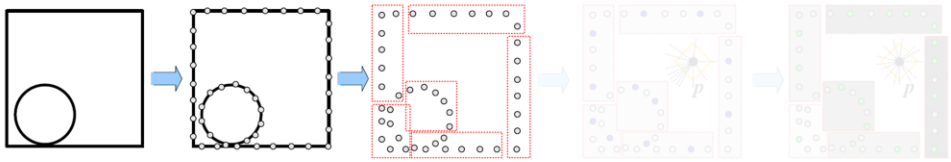


- **Importance sampling and projection**
  - To efficiently generate the microbuffer, we select and project points by using an importance function.

Obviously it is not feasible to project all scene points, therefore our goal is to selectively choose some points for projection using an importance function. Intuitively, this importance function is defined as the estimated contribution of every scene point to the microbuffer.

UMassAmherst

Algorithm Overview



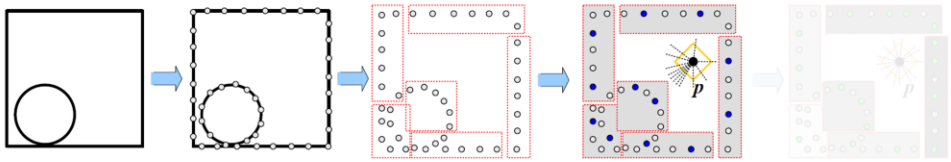
- **Clustering scene points**
  - Partition scene points into 512 clusters using k-means
  - Estimate one importance value for each cluster.
  - This results in a piecewise constant importance function.

13

In our case, it is constructed as follows. First, we exploit the spatial coherence of the scene points by partitioning them into 512 clusters. This is done using a k-means clustering that considers both the position and normal of points. Next, we estimate one importance value for each cluster, and points within the same cluster are treated with equal importance. So essentially this results in a piecewise constant importance function.

UMassAmherst

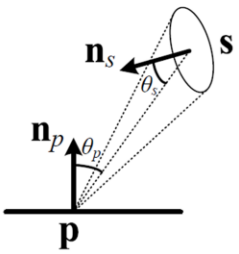
Algorithm Overview



- **Evaluate per-cluster importance (diffuse)**
  - Importance value for of a single point **s**:

$$p = \frac{\max(\cos\theta_{\mathbf{p}}, 0) \cdot |\cos\theta_{\mathbf{s}}|}{|\mathbf{s}-\mathbf{p}|^2} \Delta A_{\mathbf{s}}$$

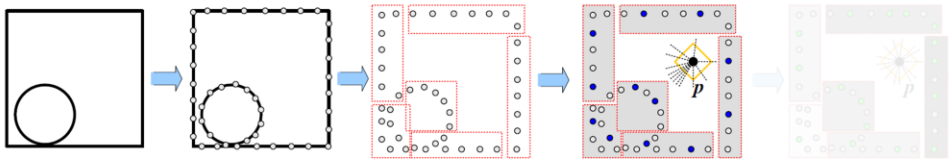
Delta projected solid angle



To define the per-cluster importance, we first realize if everything is diffuse, the importance value of a single point s can be defined as its delta projected solid angle at the shading pixel p. Notice that unlike bidirectional importance sampling, this definition does not consider the illumination radiance at the source points. This is because even if a point has no illumination radiance, it can still have an importance value because it may block the radiance of other points therefore casting indirect shadows.

UMassAmherst

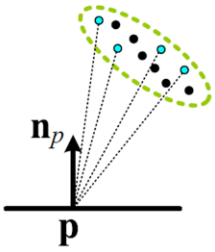
Algorithm Overview



- **Evaluate per-cluster importance (diffuse)**
  - Importance value for cluster $\boldsymbol{k}$:

$$p_k = \sum_{\mathbf{s}} \frac{\max(\cos\theta_{\mathbf{p}}, 0) \cdot |\cos\theta_{\mathbf{s}}|}{|\mathbf{s}-\mathbf{p}|^2} \Delta A_{\mathbf{s}}$$

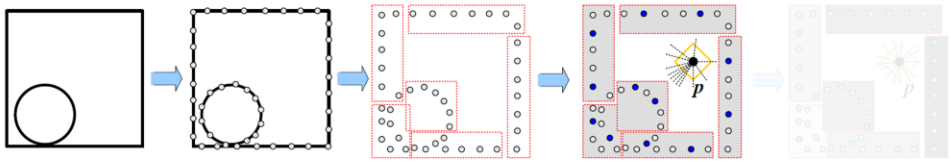
  - Sum of $N_{rnd} = 4$ random samples drawn from the cluster.



Given the importance value of a single point, the importance of a whole cluster can be estimated by drawing random samples from the cluster, and summing up the contribution of each individual sample. In practice, we usually select 4 random samples per cluster, which we found to be sufficient. In the diagram shown here, these random samples are indicated by blue dots. The number of samples can be increased if we need higher accuracy to estimate the cluster importance.

UMassAmherst

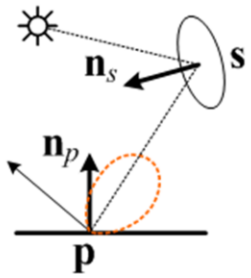
Algorithm Overview



- **Evaluate per-cluster importance (glossy)**
  - Importance value for cluster $\boldsymbol{k}$:

$$p_k = \sum_{\mathbf{s}} \frac{\max(\cos\theta_{\mathbf{p}}, 0) \cdot |\cos\theta_{\mathbf{s}}|}{|\mathbf{s}-\mathbf{p}|^2} f_r(\mathbf{s} \to \mathbf{p}, \omega_o) \Delta A_{\mathbf{s}}$$

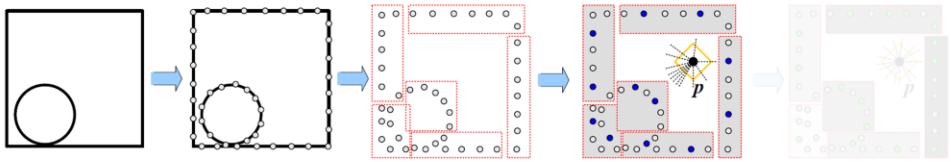
  - Sum of $N_{rnd} = 4$ random samples drawn from the cluster.



For glossy surfaces, we can easily modify this equation by adding a BRDF term. This way, higher importance will be given to scene points that fall along the reflected direction at the shading pixel p.

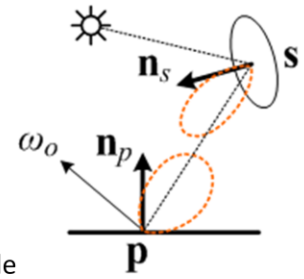
UMassAmherst

Algorithm Overview



- **Evaluate per-cluster importance (glossy)**
  - Importance value for cluster $\boldsymbol{k}$:

$$p_k = \sum_{\mathbf{s}} \frac{\max(\cos\theta_{\mathbf{p}}, 0) \cdot |\cos\theta_{\mathbf{s}}|}{|\mathbf{s}-\mathbf{p}|^2} f_r(\mathbf{s} \to \mathbf{p}, \omega_o) \Delta A_{\mathbf{s}}$$

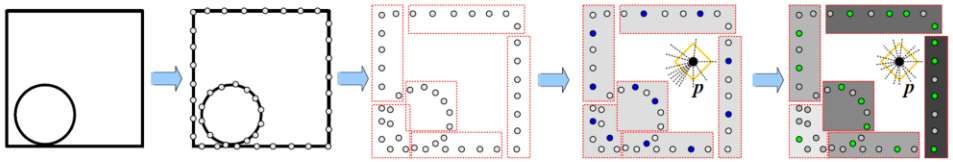


  - However, this is only suitable for relatively smooth BRDFs, due to the number of sample

We can further include the glossy radiance lobe at the scene points into this equation, which can then lead to more efficient sampling of glossy-to-glossy reflection paths.

UMassAmherst

Algorithm Overview



- **Importance Sampling**
  - ***Select a cluster*** by sampling the PDF of the importance function.
  - ***Draw a uniform random point*** within the cluster.
  - ***Project and splat*** the selected point to the microbuffer.

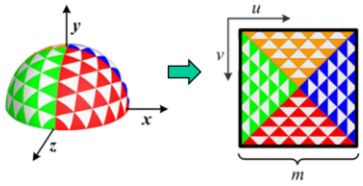
18

After we are done evaluating each cluster, our final step is to draw importance samples by applying the importance function that we have just obtained. This is achieved by first selecting a cluster based on the importance PDF, then drawing a uniform random sample within the selected cluster. In the diagram shown here, the cluster importance is indicated by the darkness of the box, and the selected samples are indicated as green dots. As you can see, clusters with higher importance will have proportionally more samples drawn from them.

UMassAmherst

Projecting a Point to Microbuffer

- ***Microbuffer***: 32x32 texture; each pixel stores the radiance and depth of the closest projected point.
- ***Projection:***
  Hemi-octanedral mapping.

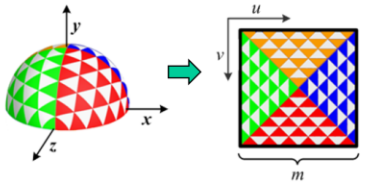


Each selected point will be projected into the microbuffer, which is a 32x32 texture parameterized using hemi-octahedral mapping. Every pixel of the microbuffer stores the radiance as well as the depth value of the closest projected point. When a new point is added, we first find its texture coordinate using the mapping formula, which can be found in the paper.

UMassAmherst

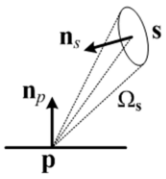
Projecting a Point to Microbuffer

- ***Microbuffer***: 32x32 texture; each pixel stores the radiance and depth of the closest projected point.
- ***Projection:***
  Hemi-octanedral mapping.

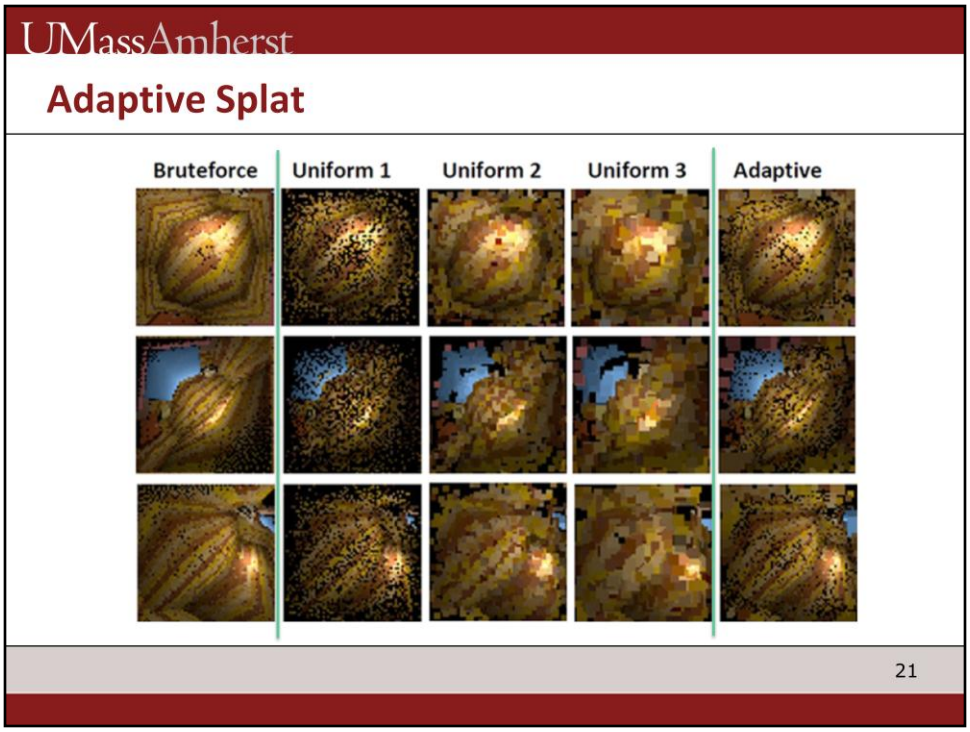


- ***Adaptive Splat***:
  Update microbuffer if depth test succeeds.

*Total solid angle represented by **s**:* $\Omega_s = \frac{\Delta\Omega_s}{p_s}$



Then, if the point passes a depth test, its color and distance values will splatted into the microbuffer. Now, the reason for splatting is because the support size of a sample may be larger than one pixel, therefore we must calculate an appropriate splat size, which we call the adaptive splats. This is achieved by estimating a total solid angle represented by the sample, defined as its delta solid angle divided by the probability of selecting this sample. Then the total solid angle will be converted to a square splat to be pasted into the microbuffer.



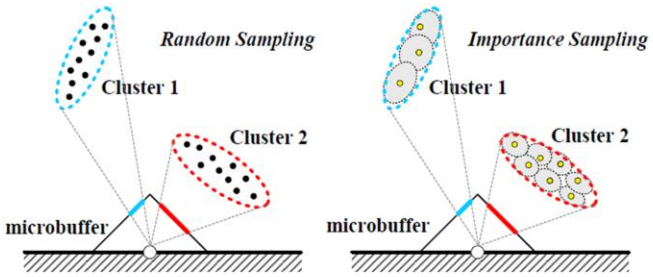
This slide shows a few examples of the microbuffers we generated. Each row corresponds to a different scene pixel. The leftmost column is the reference obtained by projecting all scene points in a brute force way. The rightmost column shows our results obtained using importance sampling and adaptive splats. The middle three columns are results obtained by fixed splats of 1, 2 or 3 pixels in size. As you can see, the uniform splat either under fills or overfill the micorbuffers; while the adaptive splats produce the most faithful results.

10 min

UMassAmherst

Sampling Algorithm Summary

- Both random samples and importance samples are projected to the microbuffer.



22

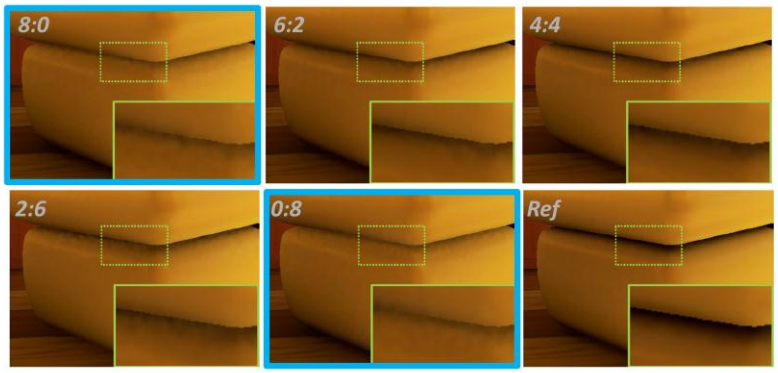
The adaptive splat is more intuitive to explain with this simple example. Here during the random sampling step, cluster 2 is found to be more important than cluster 1, therefore more points will be drawn from cluster 2 during the importance sampling step. And consequently each such point represents a smaller support size. This is similar to Monte Carlo integration, where the estimator is inversely proportional to the sample probability.

Also, notice that because the random sampling step already involved some computation that we can reuse for projection, for efficiency reasons, we project both the random and importance samples into the microbuffer.

UMassAmherst

Sample Allocation

- Given a ***fixed*** total sample budget (e.g. 8 per thread), how many do we allocate to random vs. importance samples?

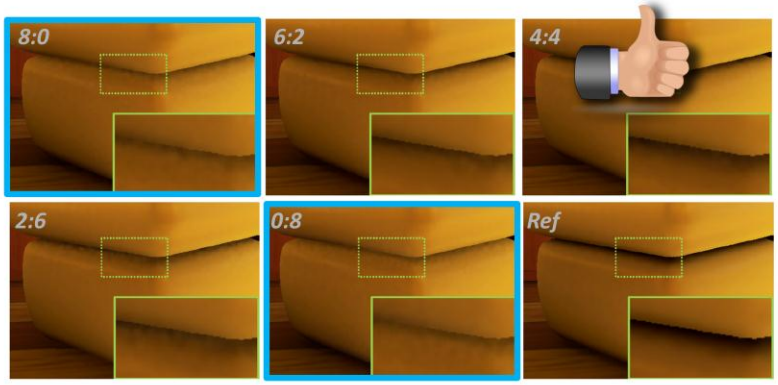


Now, the question is given a fixed total sample budget, such as 8 per thread, how many do we allocate to each category? Obviously if we allocate >>>> the budget completely to just one category, the algorithm will degenerate to uniform random sampling. This will lead to poor efficiency, indicated by the lack of indirect shadows in the outlined examples here.

UMassAmherst

Sample Allocation

- Given a ***fixed*** total sample budget (e.g. 8 per thread), how many do we allocate to random vs. importance samples?

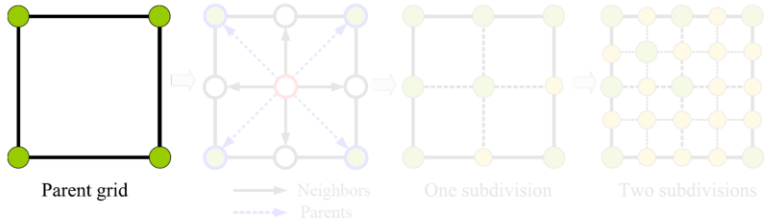


Empirically, we have determined that an equal number of allocation to each category provides the best result. This make sense intuitively, as we need some samples to help build the importance function, and then some samples to exploit the importance function. But keep in mind that the exact ratio might change if we increase the total sample budget.

11:00 here

UMassAmherst

Image-Space Adaptive Sampling

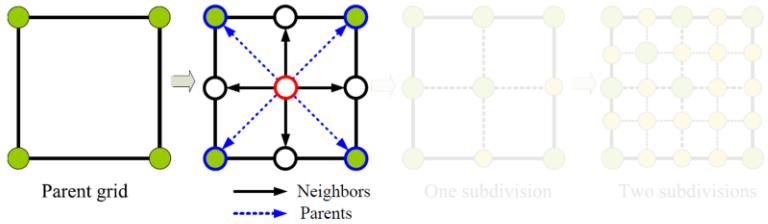


- **Start by sampling all pixels on the 4x4 grid**

Ok, so far we are done with the point sampling and projection part. Now, in order to provide reasonable interaction speed, we have to also exploit the image-space coherence to avoid computing a microbuffer at every screen pixel. We do so by using an image-space adaptive sampling algorithm, which I will briefly explain here. To start, we sample all pixels on the 4x4 grid.

UMassAmherst

Image-Space Adaptive Sampling



- **Check pixels on the 2x2 grid** (e.g. the red pixel here)
  - Compute the ***coherence metric:***

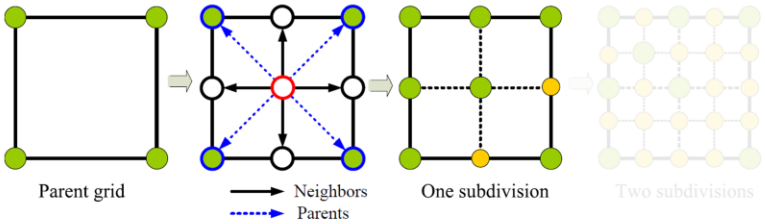
$$\max_i \left(\frac{|\mathbf{p}_i - \mathbf{p}_c|}{d/2} + \sqrt{2 - 2(\mathbf{n}_i \cdot \mathbf{n}_c)} \right) + \lambda \sum_{j,k} |L_o(\mathbf{p}_j) - L_o(\mathbf{p}_k)| \quad > \varepsilon \, ?$$

Local geometric changes *Local radiance changes*

and then proceed to pixels on the 2x2 grid, such as the red pixel in the center here. For each of them, we compute a coherence metric that involves two terms. The first is a geometric term, which evaluates geometric changes in the local region. The second term is a radiance term, which checks the pixel's four parents computed from the previous level to see if there are significant radiance changes, typically due to indirect shadows or glossy reflections. The weighted sum of the two terms is compared against a predefined threshold.

UMassAmherst

Image-Space Adaptive Sampling

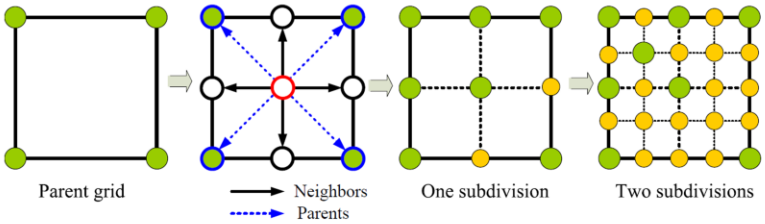


- **Based on the coherence metric:**
  - Pixels whose coherence values $> \varepsilon$ are sampled
  - The others are interpolated using joint bilateral upsampling [SGNS07]

If it's larger than the threshold, the current pixel must be sampled if it's not already sampled; otherwise, it will be interpolated from the four parent pixels using joint bilateral upsampling. So the green dots here indicate pixels that must be sampled, and the orange are the ones that will be interpolated.

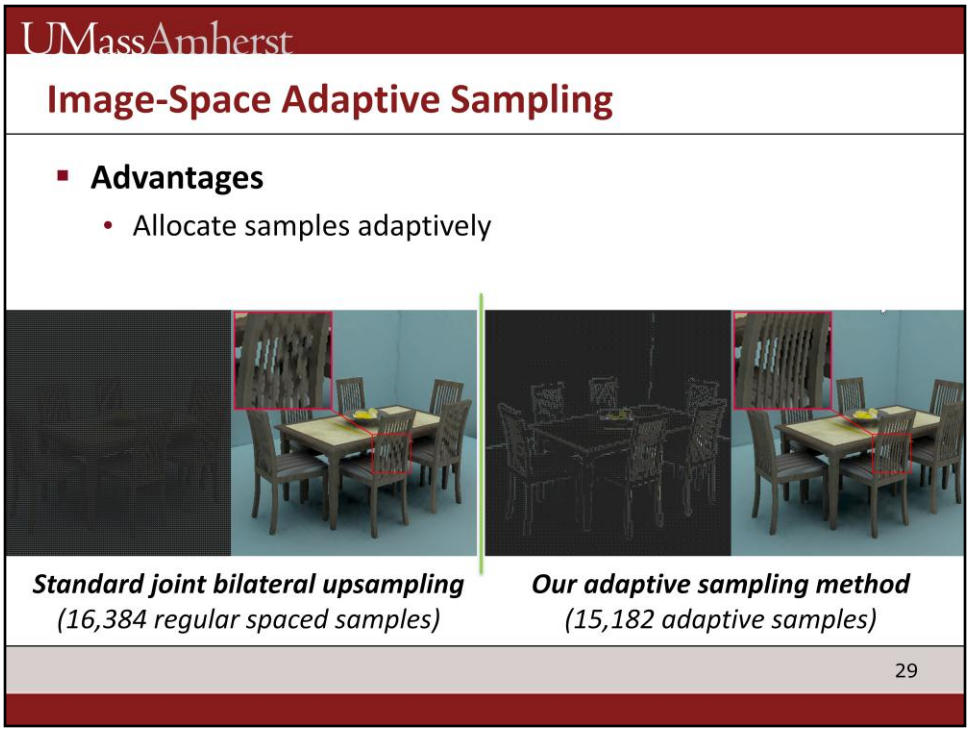
UMassAmherst

Image-Space Adaptive Sampling



- **Repeat for the 1x1 grid**
- Continue to ***subpixel*** level, if anti-aliasing is enabled.
- A pixel may be requested for sampling even if the previous level says it's ok to interpolate.

We then repeat the process for the 1x1 grid, and the process can continue to subpixel level if anti-aliasing is enabled. Although this algorithm sounds similar to standard adaptive sampling, it does have a couple of differences. First, because the same pixel will be independently checked at each level, it may be requested for sampling even if the previous level says it's ok to interpolate.

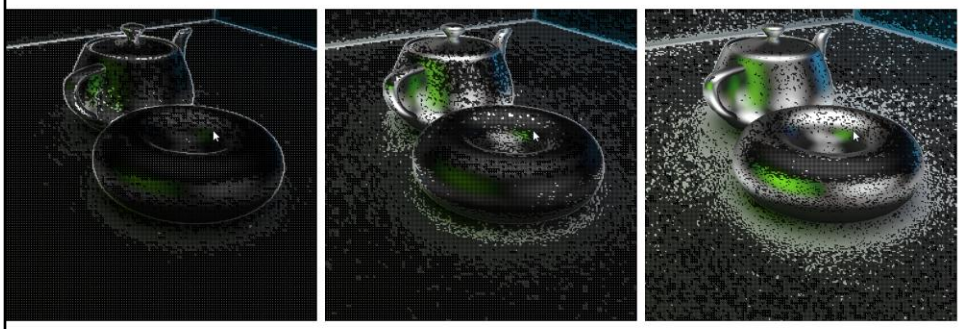


And this property turns out to be important for preserving small geometric details. Here is an example where we compare our method with standard joint bilateral upsampling that uses a 4x4 regular sampling grid. As you can see from the left image, regular sampling leads to aliasing artifacts on the chair's back, because here the feature size is close or smaller than the spacing between the grid points. In comparison, our adaptive sampling method on the right can faithfully preserve these features. For this example, we have modified our algorithm to start from an 8x8, instead of 4x4 grid, and the image is captured when the total number of samples is actually less than the one on the left.

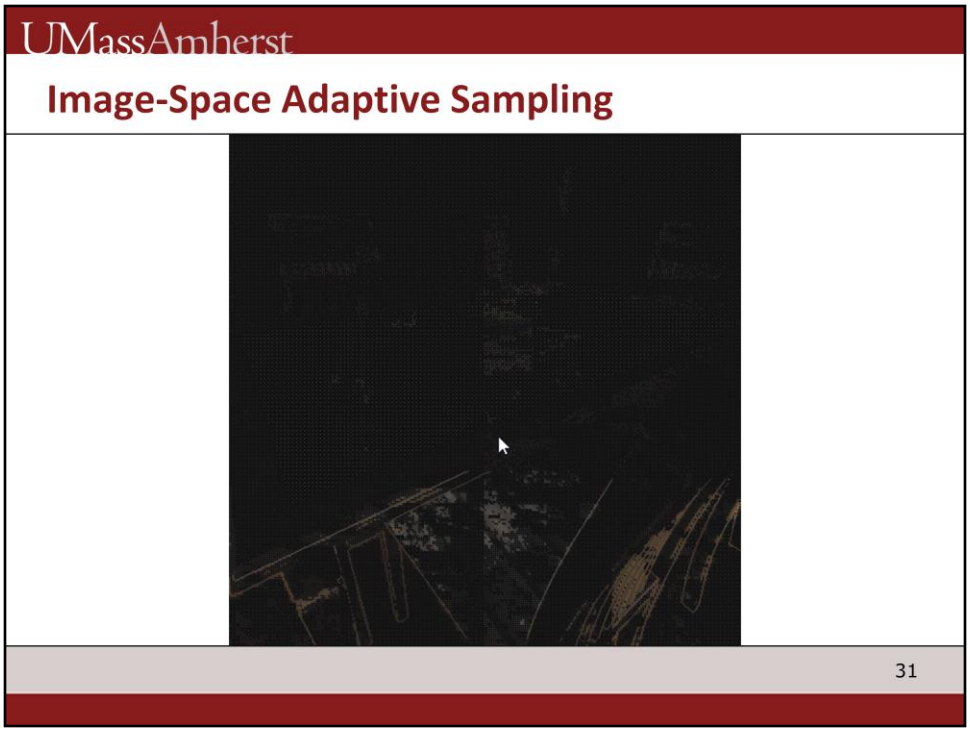
UMassAmherst

Image-Space Adaptive Sampling

- **Advantages**
  - Allocate samples adaptively
  - Accounts for both geometric and radiance changes



The second difference of our algorithm is that it considers both geometric and radiance changes. Therefore the samples can quickly snap to regions under indirect shadows and glossy reflections, even when the local geometry is flat. This property makes our algorithm more efficient than those that only consider geometric changes.



One purpose of the adaptive sampling is to enable progressive rendering, where we gradually reduce the coherence metric threshold, forcing new pixels to be sampled each frame. With the progressive rendering mode, the user can interrupt and make scene changes interactively; and the efficiency of our adaptive sampling allows the image quality to stabilize very quickly before fully frame convergence.

UMassAmherst

Implementation Details

- Implemented using **NVIDIA CUDA** and **CUDPP** library.
- **Primary lights:** up to 4 point lights
  - Allows glossy lobes to be stored with scene points
- **Microbuffer:**
  - 32x32 size, 3 bytes diffuse, 3 bytes glossy, 2 bytes depth.
  - 8KB total per microbuffer, stored in GPU shared memory

Now I will describe some implementation details. We use NVIDIA CUDA and CUDDP library to implement the entire algorithm, and we allow up to 4 point lights as the primary lights, because this way we can store the glossy reflected lobes with the scene points, to speed up the computation of glossy-glossy reflections. For the microbuffer, we quantize the colors and depth values, so that each microbuffer is 8KB in size, which can fit comfortably in GPU shared memory to allow for fast access.

UMassAmherst

Implementation Details

- **Random Sampling**
  - 512 threads, each draws $N_{rnd} = 4$ random samples.
- **Importance sampling**
  - 512 threads, each draws $N_{imp} = 4$ samples $\sim$ CDF.
- **Total projected points**: (4+4)*512 = 4K
- **5x5 bilateral filter** to reduce sampling noise
- **Anti-aliasing**

The sampling algorithm will launch 512 threads twice, the first time using each thread to evaluate cluster importance, and the second time using each thread to draw importance samples. The total projected points are four thousand. At the end of each frame, we apply a 5x5 bilateral filter on the image buffer to help reduce sampling noise, and finally, anti-aliasing can be easily enabled by using a supersampled pixel buffer. Again, our adaptive image sampling algorithm allows the image quality to stabilize quickly for a supersampled buffer.

Rendering Quality

(d) Ours (e) Reference (f) 2× diff

Now I will present some results. First, as usual, we need to check the rendering quality by comparing our results, shown on the left, with raytraced reference in the middle; and we also show the 2 times difference image on the right. On close examination, some differences are clearly noticeable, which are mainly due to color quantization and the limited resolution of the microbuffers. On the other hand, the visual quality is reasonably good given the amount of computation that we have reduced.

UMassAmherst

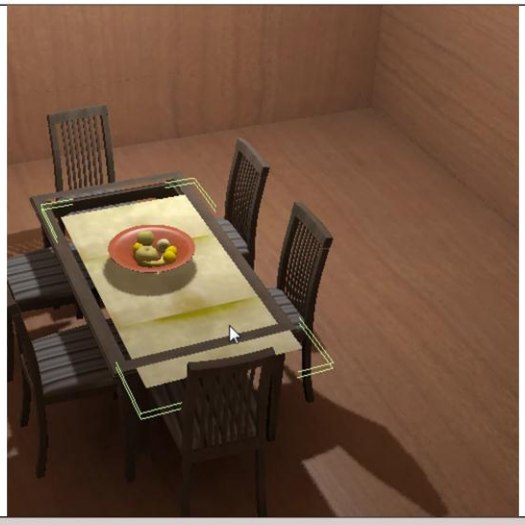
Performance

- **On NVIDIA 480 GTX**
  - $\mathbf{512} \times \mathbf{512}$ image resolution.
  - **3~4 seconds** for fully rendered image (i.e. every pixel is sampled)
  - **2~3 frames per second** in progressive rendering mode (the visual quality is acceptable for many scenes.)

In terms of performance, on NVIDIA 480 GTX, a fully rendered 512 square image takes 3-4 seconds, that is when every pixel is sampled. Our progressive rendering mode can perform at 2-3 frames per second, and for many scenes the visual quality is already acceptable under the progressive rendering mode.

UMassAmherst

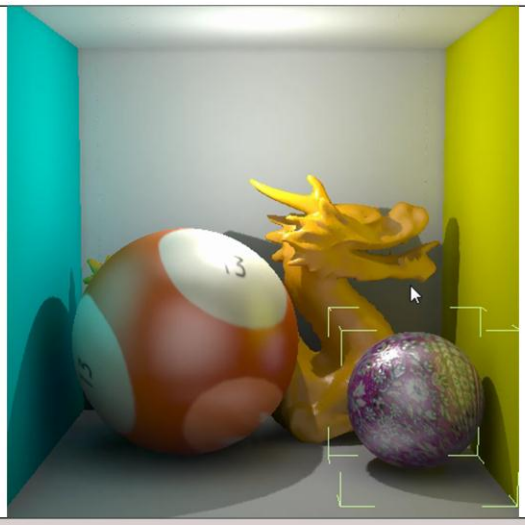
Scene Manipulation



This is a demo we captured in real-time that demonstrates interactive scene editing. When objects are moved, we will re-cluster the scene points, and this computation time is included in the reported frame rates.

UMassAmherst

Scene Manipulation

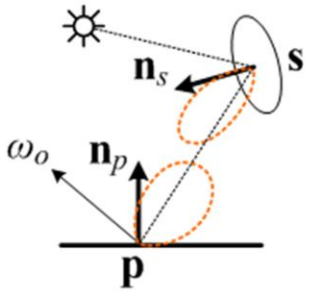


This is another video clip that shows interactive moving of objects and editing of material properties, which can include bumpmaps and spatially varying BRDFs.

UMassAmherst

Glossy-Glossy Reflections

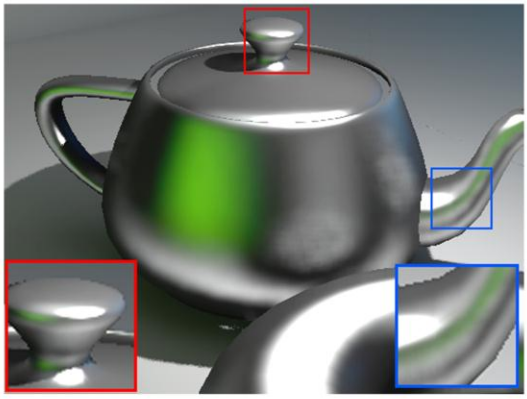
- *Use glossy radiance lobes stored at the scene points.*



This is an example that demonstrates the effect of glossy to glossy reflections. This is enabled by making use of the glossy radiance lobes stored at the scene points when we project them into the microbuffer. >>>> The effects can be seen on the specular highlights of the torus that's reflected from the teapot.

UMassAmherst

Glossy-Glossy Reflections



With Glossy-Glossy

Here I am switching between turning on or off the glossy lobes stored at the scene point so that you can see the difference it makes.

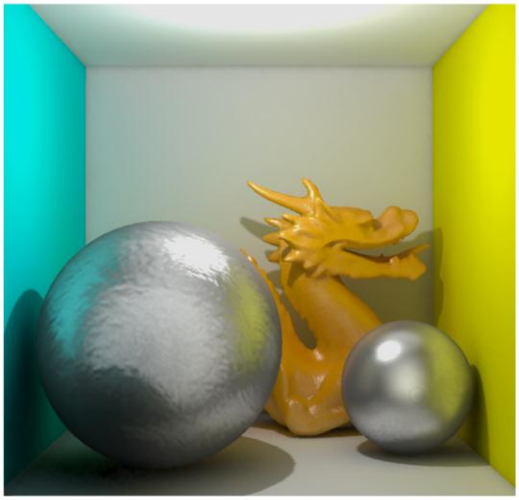
UMassAmherst

Multi-Bounce Indirect Lighting

We can also enable multi-bounce indirect lighting by treating the scene points as shading points and performing the rendering computation in the same way as before. Then we use the updated radiance of the scene points to compute the final image. By switching between one-bounce and two-bounce indirect lighting, we can observe the difference in the brightness and the color bleeding effects in this scene.

UMassAmherst

Multi-Bounce Indirect Lighting



This is another scene. Again, I am switching between one and two bounces of indirect lighting to compare the difference.

UMassAmherst

Scene Manipulation

This is the final scene that shows the editing in a kitchen scene. Here you can see the direct, … indirect… and the global illumination effects. And also the effects of editing the color of the wall, and the texture on the floor.

UMassAmherst

Summary

- A new method for final gathering by using importance point selection and projection.
- A GPU implementation that provides near-interactive rates.

To summarize, we have presented a new method for final gathering by using importance point selection and projection. We have provided a GPU implementation that can achieve near-interactive rates under arbitrary scene editing and manipulation.

UMassAmherst

Summary

- **Limitations and Future Work**
  - Prone to stochastic sampling noise.
  - High depth complexity may lead to aliasing artifacts (such as when a cluster contains multiple layers of geometry).
  - Not suitable for sharp BRDFs.

The main limitations of our work are: first, as we use stochastic sampling, our rendering result is prone to sampling noise particularly in progressive rendering mode. Also, scenes with high depth complexity may lead to aliasing artifacts, especially if a cluster happens to contain multiple layers of geometry. This is mainly because we treat the points within the same cluster equally with no relative importance, so if a cluster has multiple layers of geometry, it can cause visibility aliasing artifacts. But fortunately, this issue usually doesn't happen because the clustering algorithm is quite successful at separating different layers. Also, we can always increase the number of clusters or use an improved clustering algorithm to reduce this potential artifact. Finally, our algorithm is not suitable for very glossy BRDFs, because of the small number of samples it uses. We are planning to address this issue by using glossy reflectance filtering techniques, or by separating the computation into a near-field and far-field component and use a different approach to handle each component.

Acknowledgement

- EGSR anonymous reviewers
- John C. Bowers, Oskar Akerlund
- NSF Grant CCF-0746577