Supplementary Material: Implementation Details

This is a supplementary document for our SIGGRAPH paper entitled *An Efficient GPU-based Approach for Interactive Global Illumination*. This document contains additional implementation details and pseudo-code for several major algorithms.

Overview Algorithm details and various parameters we use can be found in Sections 3.1 to 3.4 of the paper. We have implemented all our algorithms using BSGP [Hou et al. 2008], which is a general-purpose C programming interface suitable for many-core architecture such as the GPU. BSGP builds on top of NVIDIA's CUDA, and it can be downloaded as a package from `http://www.kunzhou.net/2008/BSGP-package.zip`, which includes the BSGP compiler, editor, library, and example code.

In the algorithm listings of this document, the parallel primitives (such as **segmented reduction**) and code fragments marked by **in parallel** refer to GPU code; the other parts are executed on the CPU. Details on standard parallel primitives (such as scan, reduction, and list compaction) can be founded in previous work such as [Harris et al. 2007], [Hou et al. 2008] and [Zhou et al. 2008]. The CUDA programming model requires us to organize the computation into blocks of threads. We use a default number of 256 threads per block.

1 Selecting Irradiance Sample Points

Given the shading points, we select irradiance sample points in two stages. First, we use an adaptive strategy to seed initial sample points; we then apply k-means clustering to refine these samples and obtain the final sample locations.

Adaptive Seeding of Sample Points We use a histogram-based method to adaptively distribute initial sample points according to the local geometric changes in the scene. The pseudo-code of this step – ADAPTIVESEEDING – is given in Algorithm 1. Before it starts, we need to construct a static quadtree of all the shading points. This is straightforward to build as the shading points have a one-to-one mapping to screen pixels.

The ADAPTIVESEEDING procedure takes as input the positions and normals of all shading points (stored in hit_list), and outputs the resulting seed points ($cluster\_list$). In the first step, we assign each shading point to all quadtree nodes that it belongs to, and then reorganize the data in the order of the quadtree nodes, so that points belonging to the same node are stored together. In the second step, we compute the geometric variation for all quadtree nodes in parallel. This is computed in two passes. The first pass computes the average position x_k and normal $\vec{n}_k$ of each quadtree node by using a segmented reduction; the second pass then computes the geometric variation (ε) per shading point according to Eq 2 in the paper, and sums it up to obtain the variation per node (ε_q).

Given the variation computed for all nodes, we seed sample points using an adaptive, top-down approach. We start with the root node, and repeatedly distribute the sample budget at the current node to its children nodes. The distribution is proportional to the geometric variation term computed above.

K-Means Clustering We follow standard k-means algorithm to refine the irradiance sample points. We use an error metric defined by Eq 2 in the paper, and the pseudo-code – CLUSTERSHADINGPOINTS – is provided in Algorithm 2. This procedure takes all the shading points (hit_list) and adaptive seed samples $cluster\_list$ as input, and outputs the refined sample points $cluster\_list$. Using k-means, we update each cluster repeatedly until either the clustering converges, or a maximum number of iterations is reached. In the update stage, we need to classify a shading point to its nearest cluster with respect to the error metric. A brute force approach would require a linear search of all cluster centers. To accelerate this step, we build a kd-tree of all cluster centers, and apply a range search to quickly eliminate clusters that are too far away that do not need to be tested. At every iteration, we compute the average position and normal of each cluster by using a parallel segmented reduction.

Algorithm 1 Adaptive Sample Seeding

```
procedure ADAPTIVESEEDING(in hits_list,out cluster_list)
begin
  id_list ←new list
  error_list ←new list
  // Compute the geometric variation of each quadtree node
  for each shading point i in hits_list in parallel
    for quadtree node level j = 1 to n_level
      id_j = the j-th level parent node of shading point i
      id_list.add(make_pair(id_j,i))
  Perform sort on id_list by id_j
  for each quadtree node in parallel
    Compute the average position and normal of the node using
    segmented reduction applied on the order list id_list
  for each shading point i in hits_list in parallel
    Compute the geometric variation ε to every parent node q
    error_list.add(make_pair(ε,i))
  for each quadtree node q in parallel
    Compute the geometric variation of node, ε_q, from the
    error_list by using a segmented reduction
  // Distribute seeding samples
  for quadtree node level j = 1 to n_level
    Normalize geometric variation ε_q by segmented reduction
    Compute number of seeds n_q in proportion to ε_q
    for each node in the level in parallel
      Distribute n_q sample points randomly
      cluster_list.add(seed_k)
end
```

2 Constructing the Illumination Cut

To reduce the density estimation cost, we compute an illumination cut to approximate the entire photon tree. The pseudo-code is provided as BUILDILLUMINATIONCUT in Algorithm 3. This procedure takes the photon map as input and outputs the illumination cut. We start by selecting a coarse tree-cut in two steps: first, nodes with an approximated irradiance value ($\widetilde{E}_p$) larger than a predefined threshold E_min are added as candidate nodes for the cut; then, we traverse the different levels of the tree to ensure the construction of a complete tree-cut. This is done by deleting nodes whose children are already in the cut, and adding nodes to leaf nodes that do not have parents representing them in the cut.

Given the coarse tree cut, we perform an optimization to refine its accuracy. This is done by comparing the accurate density estimation E_p, which is evaluated at node p's center using the full photon tree, with the approximated irradiance $\widetilde{E}_p$ computed above. If the difference between the two is larger a given threshold Δ_E, the node

Algorithm 2 K-Means Clustering

```
procedure ClusterShadingPoints(
      in hits_list,
      in_out cluster_list)
begin
   id_list ←new list
   while not max_iteration and not converged
      Build a kd-tree, cluster_kdtree, on the cluster_list
      for each shading point i in hits_list in parallel
         Apply a range search in cluster_kdtree to find the cluster id_c with the minimum error
         id_list.add(make_pair(id_c,i))
      Perform sort on id_list by id_c
      Update cluster centers by using segmented reduction
   for each cluster j in cluster_list in parallel
      Pick the shading point with the smallest error as the irradiance sample point, cluster_list.add(s_j)
end
```

Algorithm 3 Constructing Illumination Cut

```
procedure BuildIlluminationCut(
      in node_list,
      out cut_list)
begin
   work_list ←new list
   for each photon map node p in parallel
      if Ẽ_p > E_min then
         work_list.add(p)
   for each node p in work_list in parallel
      if p.children is in work_list then then
         work_list.remove(p)
   for each tree level l of node_list from bottom-up
      for each node p in level l in parallel
         if p.chilren not both in work_list then
            work_list.add(p)
   while not work_list.empty()
      for each node p in work_list in parallel
         Compute E_p by a KNN search
         if ||E_p − Ẽ_p|| ≤ Δ_E then
            work_list.remove(p) and cut_list.add(p)
         else
            work_list.remove(p)
            work_list.add(p.children)
end
```

is removed from the cut and replaced by its children nodes. We perform 3∼5 iterations of this refinement.

References

HARRIS, M., SENGUPTA, S., AND OWENS, J. 2007. *GPU Gems 3 – Parallel Prefix Sum (Scan) with CUDA*. 851–876.

HOU, Q., ZHOU, K., AND GUO, B. 2008. BSGP: bulk-synchronous GPU programming. *ACM Trans. Graph. 27*, 3, 1–12.

ZHOU, K., HOU, Q., WANG, R., AND GUO, B. 2008. Real-time kd-tree construction on graphics hardware. *ACM Trans. Graph. 27*, 5, 126:1–11.