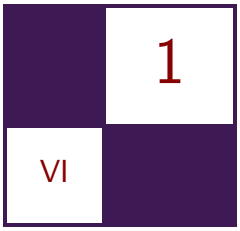

Octree Mapping from a Depth Camera

Dave Kotfis and Patrick Cozzi

1.1 Overview

To render artificial objects with consistent shading from arbitrary perspectives, a 3D scene needs to be constructed from the camera frames. Data parallel GPU computing allows for real-time 3D mapping of scenes from depth cameras such as the Kinect sensor. Noise in the camera's depth measurements can be filtered over multiple image frames by representing the scene as a voxel-based map rather than as a collection of raw point clouds. However, a dense voxel grid representation is not suitable for large scenes or live rendering for use in games.

In this chapter, we present our method that uses CUDA to reconstruct 3D scenes from depth cameras at near real-time speeds. A scene is represented by a *sparse voxel octree* (SVO) structure that scales to large volumes. We render these scenes with CUDA and OpenGL using methods that eliminate the slow process of generating meshes from point clouds or voxel grids. We will describe an SVO

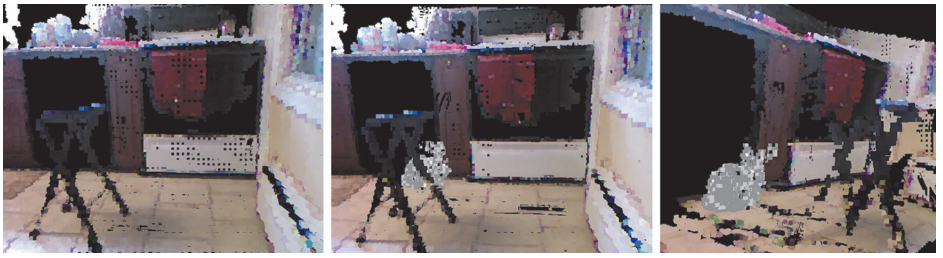

Figure 1.1. Augmented reality: A kitchen scene rendered with voxel cone tracing (left); rendering a textured Stanford Bunny sitting behind a stool in the kitchen (center); and the augmented scene rendered from an alternative view (right).

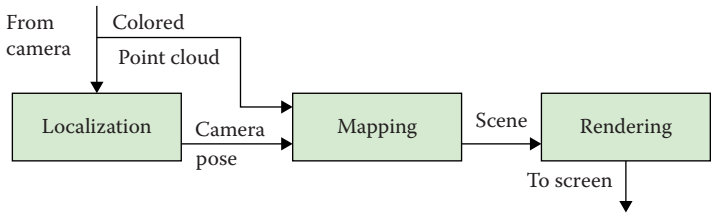

Figure 1.2. Top-level system view of an augmented reality system that simultaneously maps and renders a scene.

representation of a scene and data parallel methods to update and expand from incrementally received colored point clouds. While real-time 3D mapping has a variety of applications ranging from robotics to medical imaging, this chapter will focus on applications to augmented reality. (See Figure 1.1.)

1.1.1 Augmented Reality

In recent years, low-cost depth cameras using structured light or time of flight methods have become commonplace. These *RGB-D* (color + depth) cameras directly measure additional 3D information that previously could only be generated through sophisticated computer vision algorithms in software. These cameras are useful for creating models for 3D printing, computer vision for robotics, and creating immersive and interactive video game experiences.

Augmented reality (AR) is a field that lives on the boundary between computer graphics and vision to create experiences that blend artificial graphics with the real world. AR systems today typically render virtual objects in front of a raw depth camera frame. Future AR systems will seamlessly render virtual graphics blended with a live camera scene. Real scenes could be viewed with artificial lighting conditions and with virtual objects that cast shadows. (See Figure 1.2.)

There are many AR applications where raw color and depth data provides sufficient 3D data. This is generally the case where the application does not need to make use of information that is outside of the current physical camera view. A few example applications of mapping include object collisions with occluded surfaces and casting shadows from objects outside of view. Multiple nearby cameras could interact with the same AR application, by registering and merging their maps to establish a cohesive operating picture. Even without mapping, some AR applications may need at least a localized estimate of the camera's motion. This is required for a moving camera to maintain fixed virtual object locations. Many current AR systems use inertial sensing available on smartphones to track orientation changes. With this sensing, the absolute positioning will drift over time, but a more robust visual motion estimate can improve performance.

1.1.2 Localization

To reconstruct a scene, the movement of the camera between each frame must be determined so the points in each frame can be spatially correlated. GPU computing enables dense camera pose tracking techniques that match every pixel in 640×480 frames at 30 frames per second to track the motion of the camera without the need for a motion capture system. Previously, sparse techniques required detection of a smaller set of invariant features to track, which are not always available [Dryanovski et al. 13].

RGB-D cameras provide enough information to generate 3D positions and surface normals. The *iterative closest point* (ICP) algorithm attempts to align one frame to the previous by iteratively reducing the error between the points of each frame and the surfaces of the scene. Visual odometry with depth is a similar process that minimizes a photometric (color) error term rather than a geometric one [Steinbrucker et al. 11]. In different scenes, either geometric or photometric detail may be more prominent, so recent approaches use a combined error function that mixes the two [Whelan et al. 12].

The hard part is computing the error gradient fast enough to keep up with the camera's motion for the solution to converge. If that rate cannot be maintained and frames are skipped, the space of possible transformations that must be searched to align the frames grows. This increases the computational burden, slowing the computation down even further and creating a vicious cycle that makes the process fail. GPU computing that exploits the parallelism of the computation is critical to achieve the speeds required to avoid this downward spiral.

The methods presented in this chapter focus on mapping and rendering techniques. However, a localization method for tracking a camera's motion is a necessary part of any mapping application involving a moving camera. The ICP techniques described above offer real-time localization solutions using camera data, though alternative methods exist. An alternate approach requires the use of an external motion capture system, and many commercial *virtual reality* (VR) systems use this method for localization of a user's head pose.

1.1.3 Mapping

Reconstructing a scene requires a map representation to incrementally update and store data from each camera frame. There are many possible representations to do this, the simplest of which would be to concatenate each new point cloud by transforming all points according to the pose of the camera, assuming it is known. However, the size of this map would grow linearly with time, even when observing the same part of the scene, so it is not a suitable candidate for concurrent rendering. A standard RGB-D camera can generate several GB of raw data within only a minute. This data explosion could easily be avoided by fixing the

map size to a maximum set of frames, though this can create undesirable effects when parts of the map become forgotten over time.

We will focus our discussion on mapping methods that accumulate information over the full life of a program rather than a fixed history of frames. If the camera used for mapping remains in a finite volume of space, the map size will be finite as long as spatially redundant information is never duplicated in the representation. To do this, 3D bins at a maximum resolution can be used to identify and filter duplicate points. However, this will result in loss of detail, and the map will contain any noise produced by the camera data. While the binning of the points is trivially data parallel, the removal of point duplicates requires parallel sorting and reduction.

1.2 Previous Work and Limitations

1.2.1 KinectFusion

KinectFusion is a 3D reconstruction technique that attempts to filter the noise of incoming depth data by representing the map as a 3D voxel grid with a truncated signed distance function (TSDF) data payload storing the distance from a surface [Newcombe et al. 11]. The values are truncated to avoid unnecessary computations in free space as well as reduce the amount of data required for surface representation. Building this grid is far more maintainable than storing a raw point cloud for each frame, as the redundancy enables the sensor noise to be smoothed. It also avoids storing significant amounts of duplicate data and is highly data parallel for GPU acceleration.

However, the memory footprint of a voxel grid approach scales poorly to large volumes. The dense representation requires voxel cells allocated in memory for the large amount of free space that will almost always be prominent in scenes. Also, while the voxel grid and TSDF are an appropriate representation for the surface function, it is inefficient for any color data. The rendering process either requires ray marching to directly render the grid, or a slow surface extraction and remeshing process, neither suitable for concurrent real-time rendering.

1.2.2 OctoMap

OctoMap is a probabilistic framework where the log-odds of occupancy are stored in an octree data structure [Hornung et al. 13]. Log-odds is a quantity directly related to the probability, though it is in a form that provides the convenience of an update rule that uses addition and subtraction to incorporate information from new observations. The sparse octree structure overcomes the scalability limitations of a dense voxel grid by leaving free space unallocated in memory. OctoMap also filters sensor noise by assigning probabilities of hit and miss that

represent the noise of the sensor. Nodes in the tree are updated by logging each point from a point cloud as a hit. All points along the ray from the camera position to the end point are logged as a miss. This process takes place serially on a CPU, looping over each point in each frame.

The OctoMap is rendered by iterating through the leaves of the tree and extracting cells that have a probability greater than 0.5 of being occupied. These voxels are rendered as cubes with edge length determined by the depth of the corresponding node in the octree. This framework is most commonly used with LIDAR sensors, which have only a few points per scan, which has little benefit from parallelization. An RGB-D sensor would provide millions of points per frame that could be parallelized. However, the pointer-based octree structure used by OctoMap is less suitable for GPU parallelization than a stackless linear octree.

1.3 Octree Scene Representation

1.3.1 Data Format

We developed a sparse octree representation of a scene on a GPU, along with methods to efficiently update and expand it from incrementally received colored point clouds. The GPU data structure is based on the work of GigaVoxels [Crassin et al. 09] that uses a node pool in linear memory and a brick pool in texture memory. The nodes are composed of two 32-bit words. The first word has two 1-bit flags and 30 bits for the index of the first child node. The second word holds either an RGBA value or the location in the brick pool to be used when interpolating values within the node.

Although the sparse octree does not allocate every node of the tree in memory, we use Morton codes as keys for unique identification of voxels. Here is an example key: `1 001 010 111`. The key starts with a leading 1 to identify the length of the code, and thus the depth in the tree. After that, the key is made up of a series of 3-bit tuples that indicate a high or low value on the binary split of the x-, y-, and z-dimensions, respectively.

Using a 32-bit integer, this can represent 10 levels of depth in the tree. However, this is insufficient for mapping with a Kinect camera. The Kinect has a range of 0.3–5.0 meters in depth resolution, and doing a back-of-the-envelope calculation for the horizontal resolution (480 pixels, 5-m range, 43 degree field of view) shows that the camera will typically provide sub-centimeter resolution. A 10-meter edge volume can only achieve 1-cm resolution using 10 levels of depth. Therefore, we have transitioned to representing these keys with long integers (64 bit), which could represent more than kilometers of volume at millimeter precision, if needed. Figure 1.3 and Listing 1.1 provide descriptions of our data format.

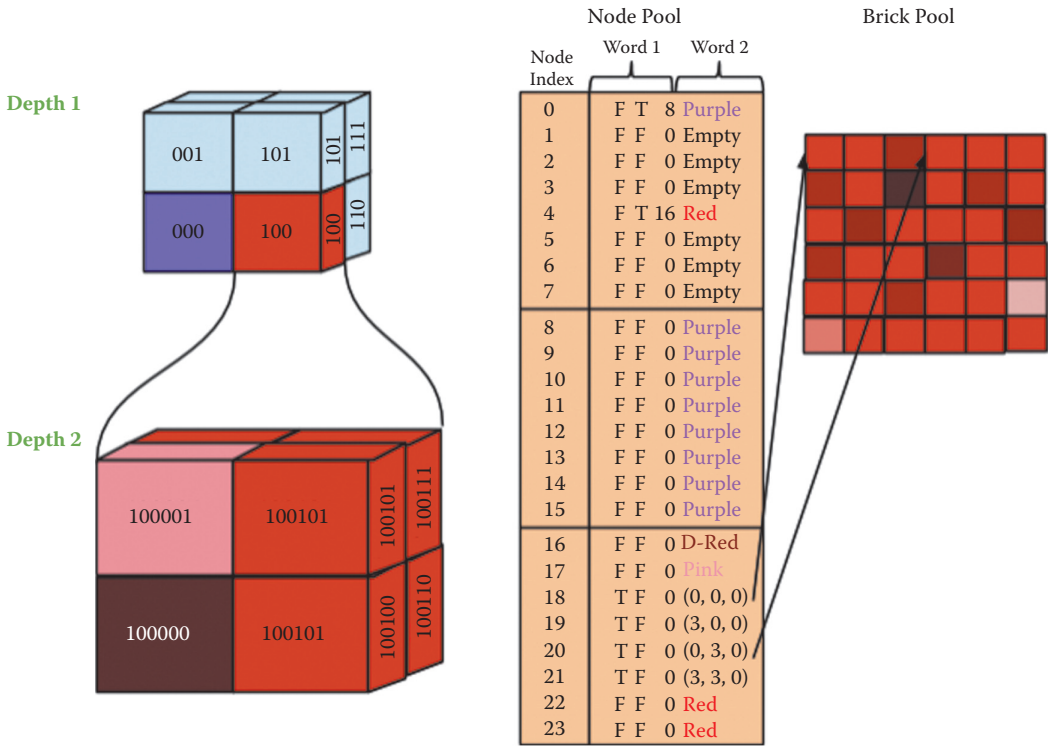

Figure 1.3. Sparse voxel octree data structure in linear GPU memory. It uses keys based on Morton codes to uniquely index nodes. The compact structure uses 64 bits per node. For hardware interpolation of values within the tree, node values can be backed by a brick in texture memory.

1.3.2 Updating the Map

Because our data structure is sparse, each new point cloud frame may contain points in parts of space that were previously unallocated in memory. For this reason, updating the map requires two steps: resizing the octree structure into newly observed space, and updating the color values within the tree with the new observations. Figure 1.4 shows the program flow for updating the map in more detail.

Update octree structure To expand our scene into unallocated space, we first must determine which new points correspond to unallocated nodes. We do this by computing the key for each point to determine its location in the octree. Fortunately, we can do this with only the constant octree parameters, its size and center location, without the need for any data within the octree. This makes the calculation completely data parallel over the incoming point cloud positions. The process of computing keys is in Listing 1.2.

```
struct char4 {
  char x, y, z, w;
};

struct Octree {
//The node data in GPU memory.
//Each node is 2 unsigned int's long.
unsigned int* node_pool;
//The number of nodes allocated in the node pool.
int size;
//The half length of each edge of the root node of the octree.
float edge_length;
//The 3D position of the center of the root node of the octree.
glm::vec3 center;
//The brick pool data in CUDA texture memory.
//Note: Our examples are limited to use of the node pool only.
cudaArray* brick_pool;
};

struct PointCloud {
//The 3D position of each point in the point cloud.
glm::vec3* positions;
//The corresponding RGBA color of each corresponding point.
char4* colors;
//The number of points in the cloud.
int size;
};
```

Listing 1.1. Data structures representing a sparse linear octree and colored point cloud data in GPU memory.

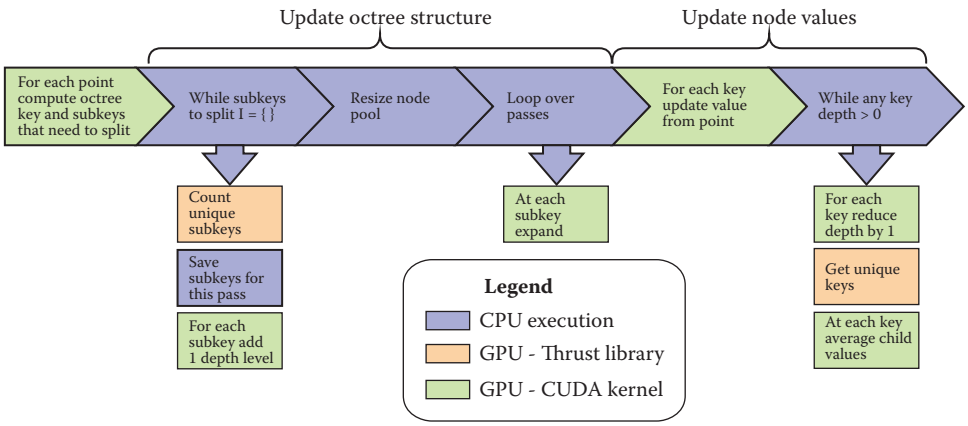

Figure 1.4. Program flow for updating the sparse octree from a point cloud frame. The process of updating the octree map from each incoming point cloud starts by counting how many new octree nodes must be created and resizing the node pool. Then, we can filter the updated color values through the tree.

```
typedef long long int octkey;

__device__ octkey computeKey(const glm::vec3& point,
  glm::vec3 center, const int tree_depth,
  float edge_length) {
  //Initialize the output value with a leading 1
  //to specify the depth.
  octkey key = 1;

  for (int i = 0; i < tree_depth; i++) {
    key = key << 3;

    //Determine in which octant the point lies.
    uint8_t x = point.x > center.x ? 1 : 0;
    uint8_t y = point.y > center.y ? 1 : 0;
    uint8_t z = point.z > center.z ? 1 : 0;

    //Update the code.
    key += (x + 2*y + 4*z);

    //Update the edge length.
    edge_length /= 2.0f;

    //Update the center.
    center.x += edge_length * (x ? 1 : -1);
    center.y += edge_length * (y ? 1 : -1);
    center.z += edge_length * (z ? 1 : -1);
  }
  return key;
}
```

Listing 1.2. CUDA device function to compute a key for a point. A kernel that parallelizes over points should call this.

The process of increasing the SVO size requires copying the data from GPU device to GPU device into a larger memory allocation. The SVO is represented by linear memory, so counting the number of new nodes is necessary to allocate sufficient continuous memory. Once we have the keys for all nodes that need to be accessed, we can use these keys to determine the subset that are not currently allocated in memory. This prepass loops through every tree depth, each time truncating all of the keys at the current depth and removing duplicate keys. We check the node for each key to determine whether its child nodes are allocated in memory. In each stage, keys that need to be allocated are stored in a list, and the length of this list $\times 8$ is the number of new nodes that need to be allocated, one for each child node.

With the set of unallocated keys in hand, we allocate a new set of continuous memory large enough for the new nodes, and we copy the old octree into this new location. Now, for each depth we parallelize over the keys in our collected set to initialize the new nodes. If GPU memory is available, it is advantageous to preallocate a large volume of memory to avoid this costly resizing process.

Update node values We use the same model as OctoMap, giving the probability that leaf node n is occupied given a series of sensor measurements $z_{1:t}$:

$$P(n|z_{1:t}) = \left[1 + \frac{1 - P(n|z_t)}{P(n|z_t)} \frac{1 - P(n|z_{1:t-1})}{P(n|z_{1:t-1})} \frac{P(n)}{1 - P(n)}\right]^{-1}.$$

This model conveniently reduces to addition of individual measurements when stored as a log-odds value. For convenience, we choose to use symmetric probability models where the probabilities of hit and miss are both equivalent. This reduces our log-odds calculation into simply keeping a running count of hits and misses.

To update the node values, we use the alpha channel of RGBA to encode a pseudo-log-odds of occupancy for each cell. When allocated, we initialize our cells to alpha = 127, which we interpret as probability 0.5 because it is the midpoint for an 8-bit unsigned integer. For a Kinect sensor, we use a probability of hit such that each observation adds 2 to the alpha value. This is for convenience since alpha is stored as an unsigned integer, and it seems to work well for the Kinect sensor model, saturating after 64 consistent hits or misses. The more often a point is observed within a portion of space, the more confident we are that the node is occupied. This helps to filter sensor noise in depth measurements by ensuring that we consistently receive point returns from a location before considering it to be occupied.

We also filter the color values received by the camera by using a running average, using the alpha channel as a weight function. Listing 1.3 shows the update and filtering process for each node. After the values are updated in the leaves of the octree, we can trickle them into the inner limbs of the tree by having each parent assume a value that averages their children.

1.3.3 Dynamic Scenes

When building a scene where all objects are static, it would be sufficient to update the map in only an additive fashion as discussed earlier. However, when objects are moving, it becomes necessary to have an update process that can remove parts of the map when they are observed to be unoccupied. Similar to OctoMap, we do this by processing the free space between the camera origin and each point in our point cloud. In each update, these nodes are observed to be free. Rather than adding an additional registered hit to these nodes, we register them as misses. With enough misses, these nodes will eventually return to being unoccupied.

Once these nodes are completely unoccupied, the memory for them is released. Rather than the expensive process of shifting all of the data in memory to fill in these holes, maintaining a list of free memory slots allows future tree expansions to fill data into them first.

```
__device__ int getFirstValueAndShiftDown(octkey& key) {
  int depth = depthFromKey(key);
  int value = getValueFromKey(key, depth-1);
  key -= ((8 + value) << 3 * (depth - 1));
  key += (1 << 3 * (depth - 1));
  return value;
}

__global__ void fillNodes(const octkey* keys, int numKeys,
  const char4* values, unsigned int* octree_data) {

  int index = blockIdx.x * blockDim.x + threadIdx.x;

  //Don't do anything if out of bounds.
  if (index >= numKeys) {
    return;
  }

  //Get the key for this thread.
  octkey key = keys[index];

  //Check for invalid key.
  if (key == 1) {
    return;
  }

  int node_idx = 0;
  int child_idx = 0;
  while (key != 1) {
    //Get the child number from the first three bits of the
    //Morton code.
    node_idx = child_idx + getFirstValueAndShiftDown(key);

    if (!octree_data[2 * node_idx] & 0x40000000) {
      return;
    }

    //The lowest 30 bits are the address of the child nodes.
    child_idx = octree_data[2 * node_idx] & 0x3FFFFFFF;
  }

  char4 new_value = values[index];
  unsigned int current_value = octree_data[2 * node_idx + 1];

  char4 current;
  short current_alpha = current_value >> 24;
  current.r = current_value & 0xFF;
  current.g = (current_value >> 8) & 0xFF;
  current.b = (current_value >> 16) & 0xFF;

  //Implement a pseudo low-pass filter with Laplace smoothing.
  float f1 = (1 - ((float)current_alpha/256.0f));
  float f2 = (float)current_alpha / 256.0f;
  new_value.r = new_value.r * f1 + current.r * f2;
  new_value.g = new_value.g * f1 + current.g * f2;
  new_value.b = new_value.b * f1 + current.b * f2;
  octree_data[2 * node_idx + 1] = ((int)new_value.r) +
    ((int)new_value.g << 8) + ((int)new_value.b << 16) +
    (min(255, current_alpha + 2) << 24);
}
```

Listing 1.3. CUDA kernel for updating values stored in octree nodes based on newly observed colors.

1.3.4 Managing Memory

The sparse octree used to represent a reconstructed 3D map will quickly grow too large to fit entirely in GPU memory. Reconstructing a typical office room at 1 cm resolution will often take as much as 6–8 GB. Use of a GPU with more memory will allow for larger scenes at higher resolutions, but there will always be applications where a physical memory increase is not practical to meet the requirements.

To handle this, we developed an out-of-core memory management framework for the octree. At first glance, this framework is a standard stack-based octree on the CPU. However, each node in the tree has an additional boolean flag indicating whether the node is at the root of a subtree that is located in linear GPU memory. It also holds a pointer to its location on the GPU as well as its size.

Next, these nodes can push/pull the data to and from the GPU. The push method uses recursion to convert the stack-based data into a linear array in CPU memory, then copies the memory to the GPU. It avoids the need to over-allocate or reallocate the size of the linear memory by first recursing through the node's children to determine the size of the subtree. The pull method copies the linear memory back to the CPU, then uses it to recursively generate it as a stack-based structure.

We use a *least recently used* (LRU) approach where all methods operating on the tree must provide an associated bounding box of the area that they will affect. First, this allows us to make sure that the entire affected volume is currently on the GPU before attempting to perform the operation. The octree will also keep a history of the N most recently used bounding boxes. When space needs to be freed, it will take the union of these stored bounding boxes and pull data that lies outside of this region back to the CPU.

1.4 Rendering Techniques

1.4.1 Extracting and Instancing Voxel Cubes

The brute-force method for rendering the SVO map is to extract the color values and 3D positions of each occupied leaf node. With these values, we can render a cube at each center position with a scale based on the depth in the SVO. (See Figure 1.5.)

Extracting the voxels requires two steps. First, in a prepass where each CUDA thread is assigned a Morton code, each voxel traverses into the SVO to determine whether the node with the corresponding code is occupied. We start with a set of keys at the minimum depth, iteratively create the 8 child keys for the occupied nodes, and remove the unoccupied node keys. Once we have determined the valid keys, we allocate space for our resulting data and extract it from the SVO into the buffer. We decode the Morton codes back into the 3D positions for each voxel.

Figure 1.5. Octree scene constructed from a live Kinect camera stream using CUDA. (a) The original raw camera image. (b) Voxel extraction and instanced rendering of an SVO map. (c) Voxel cone tracing of an SVO map. (d) Voxel cone tracing from a virtual camera view that does not match the physical view.

Once we have the position and color for each occupied voxel, we map it to an OpenGL *texture buffer object* (TBO), which is used by our vertex shader that instances a colored cube to represent the voxels (Listing 1.4).

1.4.2 Voxel Cone Tracing

Voxel cone tracing (VCT) is a physically based rendering technique similar to ray tracing [Crassin et al. 11]. It exploits the SVO data structure to avoid Monte Carlo integration of multiple rays to approximate the integral of the rendering equation. Instead, it approximates a cone by sampling values at higher levels of the SVO as the cone becomes wider. If all of the needed lighting information is incorporated into the octree, mip-mapping the values into the inner tree branches and texture interpolation performs the integration step inherently. (See Figure 1.6.)

We used voxel cone tracing to render our scene with CUDA. For each pixel, a CUDA thread traverses along a ray and samples a value from the SVO. The

```
#version 420

uniform mat4 u_mvpMatrix;
uniform mat3 u_normMatrix;
uniform float u_scale;

out vec3 fs_position;
out vec3 fs_normal;
out vec3 fs_color;

layout (location = 0) in vec4 vox_cent;
layout (location = 1) in vec4 vox_color;

layout (binding = 0) uniform samplerBuffer voxel_centers;
layout (binding = 1) uniform samplerBuffer voxel_colors;

const vec3 cube_vert[8] = vec3[8](
  vec3(-1.0, -1.0, 1.0),
  vec3(1.0, -1.0, 1.0),
  vec3(1.0, 1.0, 1.0),
  vec3(-1.0, 1.0, 1.0),
  vec3(-1.0, -1.0, -1.0),
  vec3(1.0, -1.0, -1.0),
  vec3(1.0, 1.0, -1.0),
  vec3(-1.0, 1.0, -1.0)
);

const int cube_ind[36] = int[36] (
  0, 1, 2, 2, 3, 0,
  3, 2, 6, 6, 7, 3,
  7, 6, 5, 5, 4, 7,
  4, 0, 3, 3, 7, 4,
  0, 1, 5, 5, 4, 0,
  1, 5, 6, 6, 2, 1
);

void main (void){
  gl_Position = u_mvpMatrix *
    vec4(cube_vert[cube_ind[gl_VertexID]]*u_scale +
    vec3(texelFetch(voxel_centers, gl_InstanceID)), 1.0);
  fs_position = gl_Position.xyz;
  fs_normal = u_normMatrix *
    normalize(cube_vert[cube_ind[gl_VertexID]]);
  fs_color = vec3(texelFetch(voxel_colors, gl_InstanceID));
}
```

Listing 1.4. GLSL vertex shader for instancing of colored voxel cubes using a TBO bound from CUDA.

octree depth sampled, d, for a distance, r, along the ray with a camera field of view, θ, the number of pixels in the camera image, n, and with an octree root node size, o, is given by

$$d = \left\lceil \log_2 \frac{o * n}{r \tan \theta} \right\rceil . \tag{1.1}$$

Each pixel continues to integrate its total color value using the alpha channel until it reaches its maximum of 255, or until the ray reaches a maximum length (usually 10 m).

Figure 1.6. Multiple renders of the same scene, both with voxel cone tracing. On the left, the maximum resolution is 1 cm, while on the right, it is capped at 16 cm.

1.5 Results

We tested the time required to expand, update, and filter an SVO scene with an updated point cloud frame from a Kinect sensor. We found that the time increased logarithmically with the number of allocated nodes in the SVO (Figure 1.7). The kernels that update the SVO execute serially in tree depth, but parallel over the nodes in each depth. The octree structure divides the nodes

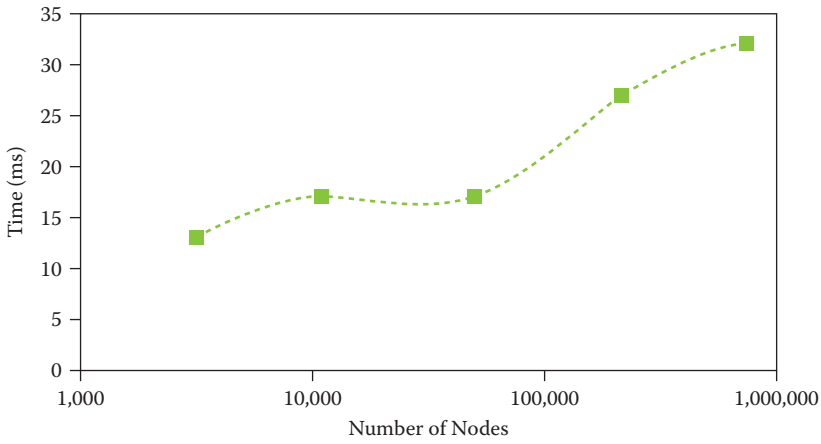

Figure 1.7. Evaluation of updating the SVO scene from a Kinect camera using an NVIDIA GTX 770 with 2 GB memory. The same scene is updated with multiple maximum depths. The edge length of the full SVO is 1.96 meters. We evaluate the update time and compare it with the change in the number of allocated nodes in the octree.

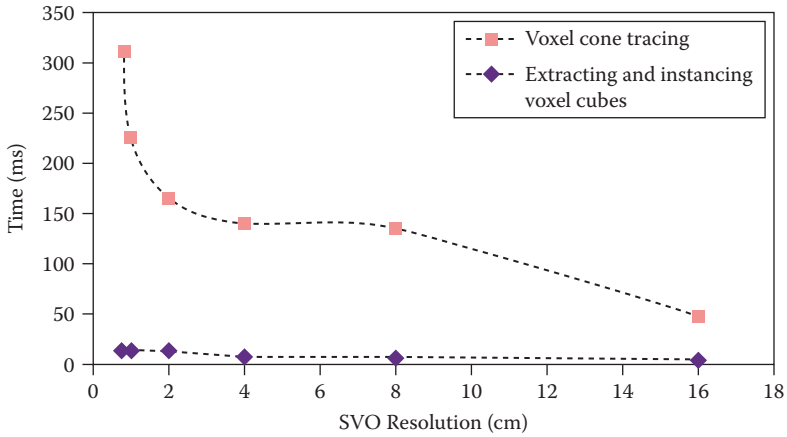

Figure 1.8. The SVO scene rendered with both voxel extraction and instancing and cone tracing (same scene as Figure 1.7). Voxel extraction and instancing achieves real-time performance at every resolution tested, but cone tracing slows down below real-time resolutions higher than 16 cm.

so that we can expect the depth to increase logarithmically with the number of nodes.

We compare the rendering time between both the voxel instancing and voxel cone tracing approaches with an identical scene at multiple levels of resolution. We found that the voxel instancing approach has steady real-time performance at all resolutions tested. Even at the lowest resolution, the voxel cone tracing technique was not real time. The runtime for VCT grows exponentially as the resolution increases (Figure 1.8).

1.6 Conclusion and Future Work

We have found that use of an SVO map allows for memory-efficient mapping. Camera noise is quickly filtered out within a few frames to create stable scenes. For debug views, voxel extraction and instanced rendering is useful for rendering values of the map at different levels of resolution. However, voxel cone tracing requires minimal additional computational cost and can render the scene at different views with similar quality to that of the original. (See Figure 1.9.)

There are similar mapping techniques implemented using conventional CPU computing, and we would like to benchmark the performance of our GPU mapping method against them on common data sets. We will also evaluate performance of complete AR pipelines (localization, mapping, rendering) with various hardware (GPUs, cameras) to determine the conditions where our techniques work best.

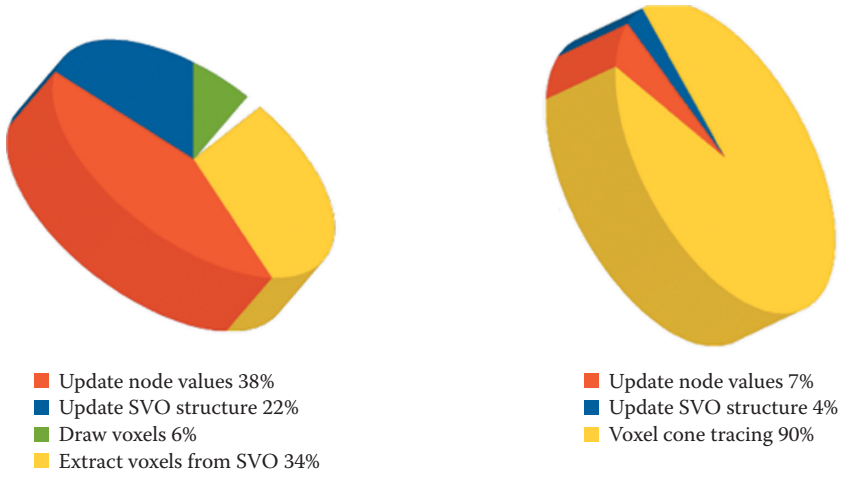

Figure 1.9. Using NVIDIA GeForce GTX 770 with 2 GB RAM, we measure the relative runtimes of mapping and rendering stages. In both cases, we map and render a $4 \times 4 \times 4$ meter volume at 2 cm resolution: Mapping and rendering with voxel instancing takes 32 ms (left) and with voxel cone tracing requires 184 ms (right).

We would like to explore use of intrinsic images in preprocessing the color values before adding them to the map. This would allow us to re-cast an artificial light into the scene without the rendering artifacts that we expect from improper shading. Rendering with a virtual light source would also blend virtual objects into the scene by casting shadows.

Also, today we are only able to add static virtual objects to our constructed scenes. It would be useful for dynamic virtual objects to move efficiently within the SVO.

1.7 Acknowledgment

We would like to thank Nick Armstrong-Crews for his valuable feedback in reviewing this chapter.

Bibliography

[Crassin et al. 09] Cyril Crassin, Fabrice Neyret, Sylvain Lefebvre, and Elmar Eisermann. "GigaVoxels: Ray-Guided Streaming for Efficient and Detailed Voxel Rendering." In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*, pp. 15–22. New York: ACM, 2009.

[Crassin et al. 11] Cyril Crassin, Fabrice Neyret, Miguel Sainz, Simon Green, and Elmar Eisemann. "Interactive Indirect Illumination Using Voxel Cone Tracing." *Computer Graphics Forum* 30:7 (2011), 1921–1930.

[Dryanovski et al. 13] Ivan Dryanovski, Roberto G. Valenti, and Jizhong Xiao. "Fast Visual Odometry and Mapping from RGB-D Data." In *IEEE International Conference on Robotics and Automation (ICRA)*, pp. 2305–2310. Washington, DC: IEEE Press, 2013.

[Hornung et al. 13] Armin Hornung, Kai M. Wurm, Maren Bennewitz, Cyril Stachniss, and Wolfram Burgard. "OctoMap: An Efficient Probabilistic 3D Mapping Framework Based on Octrees." *Autonomous Robots* 34:3 (2013), 189–206.

[Newcombe et al. 11] R.A. Newcombe, S. Izadi, O. Hilliges, D. Molyneaux, D. Kim, A. J. Davison, P. Kohli, J. Shotton, S. Hodges, and A. Fitzgibbon. "KinectFusion: Real-Time Dense Surface Mapping and Tracking." In *IEEE International Symposium on Mixed and Augmented Reality (ISMAR)*, pp. 127–136. Washington, DC: IEEE Press, 2011.

[Steinbrucker et al. 11] F. Steinbrucker, J. Sturm, and D. Cremers. "Real-Time Visual Odometry from Dense RGB-D Images." Paper presented at ICCV Workshop on Live Dense Reconstruction with Moving Cameras, Barcelona, Spain, November 12, 2011.

[Whelan et al. 12] T. Whelan, J. McDonald, M. Fallon M. Kaess, H. Johannsson, and J. Leonard. "Kintinuous: Spatially Extended KinectFusion." Paper presented at RSS Workshop on RGB-D: Advanced Reasoning with Depth Cameras, Sydney, Australia, July 9–13, 2012.