Dynamic and Implicit Trees for Graphics and Visualization on the GPU
Nathan Andryscoko and Xavier Tricoche
Purdue University, Department of Computer Science

Matrix Trees
We propose an octree and kd-tree data structure that uses matrices for the underlying data storage. Since we can derive all parent/child locations in memory, instead of explicitly storing pointers, our representation is easily represented on GPU architectures. For regular data structures (i.e. octrees) we also derive spatial information, which helps reduce GPU register pressure since this information does not need to be saved throughout any algorithms. We use a sparse matrix representation to minimize our footprint in memory, while maintaining our regular matrix structure. In this example quadtree, the sparse matrix data structure fills in any “holes” (dotted red lines).

Sparse Matrix Data Structure
Unlike previous pointerless tree representations, our data structure can adapt its sparsity to the tree’s shape. Though any sparse matrix data structure can be used, we recommend the use of the following block based one. In this data structure, we only allocate a block if data is contained within it. We use an auxiliary matrix to maintain references to our allocated blocks. This sparse matrix representation has O(1) access and insertion time, which means global memory accesses on the GPU is minimized.

Volume Ray Casting
Compared to previous traversal methods, our ray caster visits far fewer nodes. As a result, every single test resulted in faster render times, with a speed-up of nearly 300% in some cases. Additionally, we showed memory savings of nearly 50%.

Improved Traversal
Our regular tree structure allows for improved tree operations. First, we are able to spatially hash to any node. We can use this hash to quickly compute any neighbor location. For leaf finding, we can hash directly to the leaf level. If this node does not exist, we can do a binary traversal along the path to the root, giving us a O(log h) complexity, where h is the height of the tree. The previous state-of-the-art had a O(log N), where N is the number of nodes in the tree.

GPU Build Algorithm
The following algorithm is used to build our regular tree data structure on the GPU. The data type we are storing in the trees is triangles, but the algorithm is easily adjusted to other primitives. Except for the initial step (1), which has a minimal amount of iterations and is more quickly executed on GPU, each step is executed in parallel as its own GPU kernel. The scans and sort can efficiently be done using CUDPP. The graph below shows how much time is spent in each part of the algorithm. The build is both less theoretically complex and simpler to implement than the surface area heuristic trees built in ray tracing. Combined with faster render times, our trees outperform the previous state-of-the-art on the GPU for all ray tracing stages.

Ray Tracing
Using the matrix tree data structure results in faster build times and quicker rendering on the GPU than previous state-of-the-art methods. Not only do our trees require fewer node traversals, but we also perform fewer ray-primitive intersection tests. The image to the left demonstrates this property, where we compare a regular matrix kd-tree (left) to a surface area heuristic built kd-tree (right). Black indicates a low number of traversals/tests and white indicates high. Overall, we achieved a 50% speed increase for this scene, with other test scenes showing similar improvements.

Future work
We wish to represent very large data sets on the GPU. One such data set type is the unstructured meshes used in fluid dynamic simulations. In preliminary tests, we show promising results in representing this type of data using the matrix tree representation.