3D Domain Reconstruction for Graphics Hardware

Brian Budge

June 6, 2003

Abstract

Domain reconstruction from scattered data is a common and important problem. This final project explores the idea of using a kd-tree to aid in the interpolation required for reconstruction on graphics hardware. We will explain the algorithms for building and querying the kd-tree, as well as for performing Shepards order 2 interpolation.

1 Motivation

1.1 Domain Reconstruction

Domain reconstruction is an important tool for visualization. Many visualization algorithms, such as ray casting, rely on data to be on a uniform grid for efficient implementation (or for implementation at all). However, many data sets are constructed in a fashion that does not lend itself to uniformity.

For instance, consider taking the temperature in a room. There are hot spots next to windows, and cool spots next to air vents. Given this knowledge and a small set of sensors, it makes sense to place sensors strategically to obtain the most information possible. In other words, uniformly placing sensors throughout the room is a non-optimal solution, since we would be disregarding our knowledge of the room, and sacrificing the possible high dynamic range of temperature possible by placing sensors near windows and air vents.

So the solution then, is to scatter sensors such that the important areas are covered, but also there is some reasonable distribution throughout the rest of the room. This way we can capture the high range between windows and vents, but can still get an idea of what happens in the rest of the room.

Now, however, we have data that is difficult to visualize. In particular, efficient ray casting relies on having a rectilinear grid. So we must reconstruct the domain by some interpolation of the scattered data.

For more information see Franke and Nielson [4].

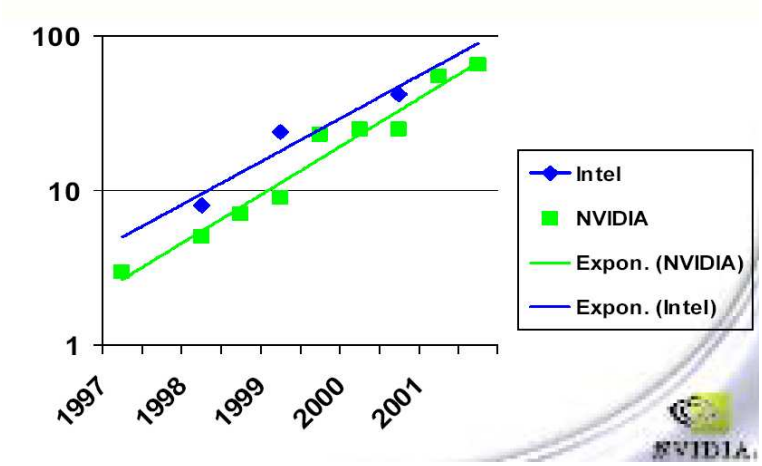

Figure 1: A log plot of millions of transistors vs years. The number of transistors is increasing faster on graphics hardware. Courtesy of NVIDIA

1.2 Kd-Trees

Kd-trees are interesting data structures which are particularly useful for multi-dimensional range searches and nearest neighbor lookups [8]. They can be used for nearest neighbor search in $O(\log(n))$ time, orthogonal range searching in $O(n^{1/2} + k)$ to find k elements [6], and can also do efficient k-nearest neighbor searches in fast time (bounded by the time to do orthogonal range search, but certainly far less - unable to find analysis). Fixed radius searches are also possible in $O(n^{1/2} + k)$, and are quite suitable for domain reconstruction purposes.

1.3 Graphics Hardware

Graphics hardware is an increasingly interesting and important computing phenomenon. The main reason for this is that graphics hardware seems to be following a Moore's Law cubed increase in speed over time (see figure 1).

Essentially, graphics hardware can achieve such large gains in speed due to its extremely parallel/pipelined nature. As time goes on, not only do the components themselves get faster, but more and more components are added to the hardware, which adds more parallelism. This is a synergistic combination, and allows for large speed increases between each new generation of hardware – typically every 6 months.

It makes sense, then, that we should try to harness this mini-supercomputer for our needs.

2 Algorithm and Analysis

Here we will present the complete algorithm, as well as an analysis of its running time at various stages of the algorithm. Due to graphics hardware's highly parallel nature, many factors of n will drop out of the analysis, where n is the number of input elements.

2.1 Sorting

The ability to sort, or at least partially sort, is required for building a kd-tree. Sorting on graphics hardware presents some interesting challenges. We wish to exploit the parallel nature of the hardware, so the algorithm must be able to touch many elements at the "same time" without the possibility of corrupting sorted elements. For example, running quicksort in parallel would be asking for trouble because elements previously rearranged by processor P_i might then be overwritten by processor P_j due to race conditions.

We analyze Odd-Even Sort and Bitonic Sort [5], both of which can be used to implement sorting networks [3].

2.1.1 Odd-Even Sort

Odd-Even Sort is essentially Bubble Sort for parallel networks. The idea behind this algorithm is that during any particular pass, if only even elements are tested with their right-hand neighbor, or only odd elements are tested with their right-hand neighbor, the same element will never be touched more than once during a pass.

So then the algorithm is as follows:

Algorithm 2.1: ODD-EVEN SORT($array, size$)

comment: Sort the array $array$ of length $size$

for $i \leftarrow 0$ **to** $size/2 - 1$
  comment: Single pass on evens
  $j \leftarrow 0$
  while $j \leq size - 1$
    if $array[j] > array[j+1]$
      $T \leftarrow array[j]$
      $array[j] \leftarrow array[j+1]$ $j \leftarrow j + 2$
      $array[j+1] \leftarrow T$
  comment: Single pass on odds
  $j \leftarrow 1$
  while $j \leq size - 1$
    if $array[j] > array[j+1]$
      $T \leftarrow array[j]$
      $array[j] \leftarrow array[j+1]$
      $array[j+1] \leftarrow T$
    $j \leftarrow j + 2$

This is easily extensible to hardware. We simply loop $n/2$ times, with each time through the loop performing 2 passes: A single pass to do possible exchanges on even and even +1 elements, and then another pass to do odd and odd +1 elements.

We can treat each pass as a constant time operation, since the hardware is so parallel, and each shader would perform a constant number of operations.

This means that this simple algorithm, when applied to graphics hardware, is $\Theta(n)$, which input size n. It should be noted, however, that passes can be expensive, as communication over the Advanced Graphics Port (AGP) bus is required.

2.1.2 Bitonic Sort

Bitonic Sort is not as easy to write in a non-recursive fashion as Odd-Even Sort. By its very nature, Bitonic Sort lends itself to a recursive implementation. Bitonic Sort relies on a *bitonic cleaner*, as well as a *bitonic merger* to perform the sorting. Bitonic cleaner makes $\Theta(n \log(n))$ compares to sort *a bitonic sequence*, and bitonic merger makes $\Theta(n)$ compares to merge. For a complete explanation of bitonic, refer to Cormen et al. [3]

The algorithm is better explained in words than by pseudocode. Essentially the important piece of the puzzle is that Bitonic cleaner requires its input to be bitonic in order to sort the sequence. Bitonic merger takes two sorted sequences, and merges them into one bitonic sequence. Note that any two elements are always bitonic. This gives rise to the following algorithm:

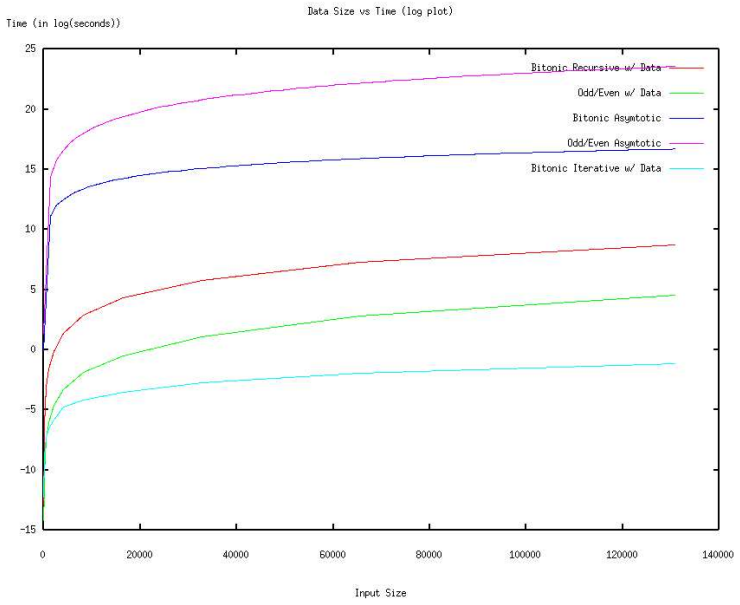

Figure 2: Theoretical vs implemented times for Bitonic and Odd-Even Sorts. Note that Odd-Even Sort is much faster than recursive Bitonic Sort, but that the opposite is true with regards to iterative Bitonic Sort. This is probably due to stack manipulation.

Sort each adjacent even/odd pair. Merge all of these pairs into groups of four. Now these groups of four are all bitonic. So we can use cleaner to sort all the groups of four. We again merge into eights, etc ...

At the end we end up with one large bitonic sequence, which when run through cleaner, will produce a sorted sequence.

This algorithm requires $\Theta(\log(n))$ passes of clean and merge.

In other words we have an algorithm where we loop $\log(n)$ times. Each time through the loop, we will perform $\log(n)$ passes to clean, and a single pass to merge.

When implementing this on graphics hardware, we can again assume that the entire input can be checked at the same time. This gives an algorithm that is $\Theta(\log^2(n))$.

2.1.3 Comparison

The ease of implementation of Odd-Even Sort certainly makes it a first choice for prototyping, but the *much* higher order of passes, makes it less feasible for a useful application. Unfortunately, requiring more passes of graphics hardware

is a bad trade off. For n in the small thousands, it is probably feasible to make the n passes, however, for n in the millions, it becomes less appealing.

Bitonic sorting has a higher overhead, but only slightly. For a full software implementation, the constants make Bitonic Sort less appropriate for small input. However, on graphics hardware, the constants involved with each pass already are likely to make a larger difference even for small input. More concisely, Bitonic Sort is likely to outperform Odd-Even Sort even for very small input.

Figure 2 shows implementations done fully in software. One might note that Odd-Even performs much better than Bitonic when Bitonic is written recursively. This is very deceiving, however, and might be attributed to the large amount of stack manipulation overhead required due to the doubly-doubly recursive nature of the algorithm.

Implementation of Bitonic Sort in an iterative environment leads to a much more (orders of magnitude) efficient algorithm. It should be noted that the version of Bitonic Sort discussed above requires a power of two inputs.

2.2 Kd-tree Build

The kd-tree build requires $\log(n)$ passes of either sorting routine in order to complete. However, the sorting happens on smaller and smaller subsets of the initial data. For instance, given an input $n = 2^k$ and use of Odd-Even sort, the first pass will require a sort of the entire data set, requiring n passes. The second pass will perform two separate sorts, requiring only $n/2$ passes.

This gives us:

$$\begin{aligned} & \sum_{i=1}^{\log_2(n)} \frac{n}{2^i} \\ = \; & 2n - 1 \\ = \; & \Theta(n) \end{aligned}$$

passes for Odd-Even Sort, each with constant time yielding a total of $\Theta(n)$.

For Bitonic Sort, the first build pass requires $\log^2(n)$ sort passes, the second requires $\log^2(n)/2$ sort passes, etc ...

and we have (again, for $n = 2^k$):

$$\begin{aligned} & \sum_{i=1}^{\log_2(n)} \log_2^2(\frac{n}{2^i}) \\ = \; & \sum_{i=1}^{\log_2(n)} \log_2^2(n) - \log_2^2(2^i) \\ = \; & \log_2^3(n) - \sum_{i=1}^{\log_2^2(n)} i \end{aligned}$$

$$
\begin{aligned}
&= \log_2^3(n) - \frac{\log_2(n)(\log_2(n)+1)}{2} \\
&= \frac{\log_2^3(n)}{2} - \frac{\log_2^2(n)}{2} \\
&= \log_2^3(n^{1/2}) - \log_2^2(n^{1/2}) \\
&= \Theta(\log^3(n^{1/2}))
\end{aligned}
$$

a result which is far more optimal than the Odd-Even Sort.

Essentially, each level is sorted, and partitioned so that the tree will be left balanced. In this way, we can store the tree into an array. (This is like storing a heap into an array) Each level uses a different dimension to sort (possibly repeating in a circular fashion). Graphics hardware should be able to easily handle up to four dimensions. For more information on kd-trees, see Bentley and Friedman [1].

2.3 Fixed Radius Search

Fixed radius search is an algorithm using a kd-tree which essentially returns all elements within an epsilon ball defined by a point and a radius. For instance, given integer points on the real line, and a query point of 0, with a radius of 2.5, the fixed radius search would yield $\{-2, -1, 0, 1, 2\}$.

The algorithm can be implemented to run in $O(n^{1/2} + k)$ by simply using the fact that orthogonal range queries run in $O(n^{1/2} + k)$. The only difference between the two algorithms is the distance metric used to determine if a point in our yield set or not (essentially, fixed radius uses the Euclidean metric, while orthogonal range uses the Manhattan metric).

The following algorithm is a simple adaptation from Christensen's method [2]. This algorithm is normally implemented as a recursive one, however, given restrictions of graphics hardware, it must be made iterative.

Algorithm 2.2: FIXED RADIUS SEARCH$(tree, size, position, radius)$

comment: Find all elements within radius distance to position

comment: closest stores the points found within the radius

comment: parent stores the parents

comment: dist1_2 stores the distances squared

$distanceSquared \leftarrow radius * radius$
$height \leftarrow \lceil \log_2(size) \rceil$
$index \leftarrow 1$
$level \leftarrow 0$
while *true*
  comment: move down through the subtrees until leaf is reached
  while $j \leq size - 1$
    $d \leftarrow distance\ in\ split\ direction\ from\ position\ to\ tree[index]$
    $dist1\_2[level] \leftarrow d * d$
    $index \leftarrow index * 2$
    if $d > 0$
      $index \leftarrow index + 1$
    $parent[level] \leftarrow index$
    $level \leftarrow level + 1$
  comment: see if the leaf is within the radius - if so update
  $tempDist2 \leftarrow distance\ squared\ from\ tree[index]\ to\ position$
  if $distanceSquared > tempDist2$
    $add\ tree[index]\ to\ closest$
  comment: move up tree until we reach a position we need to check
  $from \leftarrow -1$
  while $dist1\_2[level] >= distanceSquared || from \neq parent[level]$
    $from \leftarrow index$
    $index \leftarrow index/2$
    $level \leftarrow level - 1$
    if $index = 0$
      $return\ closest$
  comment: see if the non-leaf is within the radius - if so update
  $tempDist2 \leftarrow distance\ squared\ from\ tree[index]\ to\ position$
  if $distanceSquared > tempDist2$
    $add\ tree[index]\ to\ closest$
  $index \leftarrow parent[level]$ xor 1
  $level \leftarrow level + 1$

2.4 Shepards$_2$ Interpolation

Shepards interpolation was introduced in 1968 by Shepard [7]. It is used for interpolating between data that is not on any kind of grid, and is what we will be using for reconstructing our domain. The algorithm is very simple, and can easily be extended to k dimensions.

The variant we will be using is local Shepards$_2$ interpolation. Shepards$_2$ refers to a second order method, and local refers to the fact that we are only using local sites to interpolate.

The algorithm as we are implementing it is as follows:

For each point p that is being interpolated, gather points P which are within some radius r (via fixed radius search). Then the value at p is

$$value(p) = \frac{\sum_i^n value(P_i)/d(p,P_i)^2}{\sum_i^n 1/d(p,P_i)^2}$$

where d is a distance function, in our case the Euclidean distance, and where n is the number of values found from our fixed radius search.

The complexity of this function depends on the density of points in the domain. If the domain is sparsely populated, n will be very small on average, and if it is densely populated, n will tend to be large. The radius can be changed to get appropriate ranges of n.

2.5 Domain Reconstruction

So then the overall algorithm is:

1. Build kd-tree
2. Reconstruct Domain

Reconstructing the domain now is simply a matter of rendering an image where the pixel location of the image is an indication of the 1D, 2D, or 3D location in space (for instance, a 4096×4096 image could be used to reconstruct a $256 \times 256 \times 256$ domain).

The shader can not just be written as presented above. The problem with computing the fixed radius search before computing the Shepards interpolation is that we don't have temporary storage on common graphics hardware. However, we can compute the interpolation as we go as follows (assuming 3D domain).

Algorithm 2.3: GET SINGLE PIXEL VALUE($pixel, domainDim$)

comment: Figure out point in domain from pixel and domainDim,
comment: and also decide on a radius. The size variable should
comment: be a global constant
comment: Calculate the fixed radius search as in algorithm
comment: 2.2 except that the parts of the algorithm where
comment: we were adding elements to the **closest** set, we will
comment: instead add to the Shepard's calculation

$distanceSquared \leftarrow radius * radius$
$height \leftarrow \lceil \log_2(size) \rceil$
$index \leftarrow 1$
$level \leftarrow 0$
$numerator \leftarrow 0$
$denominator \leftarrow 0$
while *true*
  comment: move down through the subtrees until leaf is reached
  while $j \leq size - 1$
    $d \leftarrow distance\ in\ split\ direction\ from\ position\ to\ tree[index]$
    $dist1\_2[level] \leftarrow d * d$
    $index \leftarrow index * 2$
    if $d > 0$
      $index \leftarrow index + 1$
    $parent[level] \leftarrow index$
    $level \leftarrow level + 1$
  comment: see if the leaf is within the radius - if so update
  $tempDist2 \leftarrow distance\ squared\ from\ tree[index]\ to\ position$
  if $distanceSquared > tempDist2$
    $numerator \leftarrow numerator + tree[index].value/tempDist2$
    $denominator \leftarrow denominator + 1.0/tempDist2$
  comment: move up tree until we reach a position we need to check
  $from \leftarrow -1$
  while $dist1\_2[level] >= distanceSquared || from \neq parent[level]$
    $from \leftarrow index$
    $index \leftarrow index/2$
    $level \leftarrow level - 1$
    if $index = 0$
      $return\ numerator/denominator$
  comment: see if the non-leaf is within the radius - if so update
  $tempDist2 \leftarrow distance\ squared\ from\ tree[index]\ to\ position$
  if $distanceSquared > tempDist2$
    $numerator \leftarrow numerator + tree[index].value/tempDist2$
    $denominator \leftarrow denominator + 1.0/tempDist2$
  $index \leftarrow parent[level]$ xor 1
  $level \leftarrow level + 1$

This stage only requires a single pass to reconstruct the data. We assume again that our hardware is massively parallel, and so algorithm 2.3 has the same complexity as the Fixed Radius Search, algorithm 2.2, of $O(n^{1/2} + k)$, where k will depend on the density of elements in the domain.

3 Graphics Hardware

3.1 Implementation Requirements

Graphics hardware is not currently able to handle this algorithm as a whole. Both the build and reconstruction require conditionals and texture reads within the same shader, and the reconstruction also requires dynamic looping abilities.

Minimal changes to graphics hardware would either require texture reads within vertex programs, or dynamic conditional and looping within fragment programs. Additionally, it is likely that the reconstruction shading could go over on instruction count in the fragment program, since the current maximum number of instructions is 1024. If it were done in the vertex program, it is much more likely to finish, as maximum number of instructions is over 65,000.

It is likely that a card with these abilities will be on the market before the end of the year.

3.2 Projected Performance

For 131,000 scattered data points, the number of passes required to build the kd-tree is 4913. Since no large amounts of data need to be transported across the AGP bus, it is likely that this can be computed at well over 100 passes per second (the shaders are extremely simple - each pass through a shader requires two texture lookups, a compare, and a write). However, to be conservative we will say 100 passes per second. This yields a complete kd-tree build in less than 5 seconds.

The reconstruction pass will take much longer, as the program consists of hundreds of static instructions, and likely many thousands of dynamic instructions. It is likely that the radius could be chosen such that the average number of sites used in the Shepard's Interpolation is around 64. Additionally, the analysis projected on the order of $\sqrt{n}$ complexity, which would be about 360. So pessimistically, we could say that no instruction will get executed more than 500 times in a pass, and the total number of instructions run could be bounded by around 5,000. This number comes from saying that the code is around 200 static instructions, around half of those instructions will be run 500 times, with the other hundred being run only a small number of times. This is near 5,000.

Shader programs with 200 instructions can be run at about 60 frames per second, and so we could make the assumption (probably a fair one), that we can run only one twenty-fifth as fast ($200 \times 25 = 5,000$), which still yields over two reconstructions per second.

Given the Moore's Law cubed growth in speed, however, it is likely to run faster than that by the time a card that can perform these computations is released.

4 Conclusion

We have demonstrated a technique for performing domain reconstruction on near future commodity graphics hardware. The analyses show that it will be possible to make these calculations at fast rates, which will surely be better than traditional hardware given the speed increase trends of graphics hardware.

4.1 Source code and PDF file

The source can be found at `http://graphics.cs.ucdavis.edu/~bcbudge/finalProjectCode.tbz` and the PDF file (this file) can be found at `http://graphics.cs.ucdavis.edu/~bcbudge/finalProject.pdf`

References

[1] J. Bentley and J. Friedman, *Data structures for range searching*, ACM Computing Surveys, (1979), pp. 397–409.

[2] P. Christensen, *A practical guide to global illumination using photon mapping, coursenotes #43*, in SIGGRAPH coursenotes, 2002, pp. 93–121.

[3] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms, Second Edition*, 2001.

[4] R. Franke and G. Nielson, *Scattered data interpolation and applications: A tutorial and survey*, Geometric Modelling, Methods and Applications, (1991), pp. 131–160.

[5] D. Knuth, *The Art of Computer Programming, Volume 3 Sorting and Searching*, 1973.

[6] R. Pless, *Orthogonal range searching.* `http://www.cs.wustl.edu/~pless/506/l11w.html`, 2003.

[7] D. Shepard, *A two-dimensional interpolation function for irregularly-spaced data*, Proceedings of the 1968 23rd ACM national conference, (1968), pp. 517–524.

[8] S. Skiena, *Algorithm design manual.* `http://www2.toki.or.id/book/AlgDesignManual/BOOK/BOOK3/NODE134.HTM#kdtrees`, 1997.

â€¢ On the task of 3D reconstruction from a single image, we apply our point set generation network and signicantly outperform state of the art; â€¢ We systematically explore issues in the architecture and loss function design for point generation network; â€¢ We propose a principled formulation and solution to address the groundtruth ambiguity issue for the 3D reconstruction from single image task. 2. Related Work. 3D reconstruction from single images While most researches focus on multi-view geometry such as SFM and SLAM [11, 10], ideally, one expect that 3D can be reconstructed from the abundant Volume Graphics has extensive know-how in the area of visualization, interpretation and analysis of 3D data. Profit from our experience. Overview.Â With its fully integrated CT reconstruction, Volume Graphics offers a seamless connection to the comprehensive analysis and measurement functions of its software. Moreover, the software is independent of the hardware of the CT system and can process 2D data sets of different CT systems from different manufacturers â€” all within a single software environment, without interrupting the workflow. Without beam hardening correction: The incorrect display of gray values (2D slices) causes, amongst other issues, holes in the determined object surface (3D view). Online 3D reconstruction is gaining newfound interest due to the availability of real-time consumer depth cameras. The basic problem takes live overlapping depth maps as input and incrementally fuses these into a single 3D model. This is challenging particularly when real-time performance is desired without trading quality or scale.Â We show interactive reconstructions of a variety of scenes, reconstructing both ne-grained details and large scale environments. We illustrate how all parts of our pipeline from depth map pre-processing, camera pose estimation, depth map fusion, and surface rendering are performed at real-time rates on commodity graphics hardware.