Project #1: Height field

B03902124 黃信元

I. Heightfield-ray intersection algorithm

I have tried 3 different algorithms for intersection. Two of the algorithms are built on top of the first. The first algorithm is the simple 2D-DDA. I will briefly talk about what I have done. First is to transform the ray into object space and check intersection with the bounding box of the heightfield. Although pbrt had checked intersection with the **BBox** in world space, this is more compact thus can gain a lot of speed-up (7.5s to 3.0s for **landsea-big** using single core). Then I did the traversal on the 2D-grid. In 3D-DDA, when the ray hit a grid cell, it check intersection with all the primitives in the cell. Although we only need to check intersection with two triangles, this is still quite time consuming. A simple speed up is to create a bounding box for each cell, and check intersection with the cell bounding box before going into two-triangles intersection test. Overhead of creating bounding box is very low since it takes only a few scan through the heightfield data. But we can further lower the computational cost by lowering the bounding box intersection time. This can be done by making use of **NextCrossingT** (**txNxt, tyNxt, txCur, tyCur** in my program) which are already calculated when traversing grid. Therefore, we only need one loop in **BBox::IntersectP**. Overall we can have speed up from 3.0s to 2.4s for **landsea-big** using single core.

I also tried some complicated data structure to speed up the intersection. Since KD-tree has been shown to be a very good data structure, I tried integrating KD-tree with 2D-DDA. (the codes are in **heightfield2-KD.***) I wrote a 2D KD-tree with each leaf node being a sub-grid. For traversal, I follow the procedure in pbrt for KD-tree and in each leaf node I use 2D-DDA for intersection. Although being very complex in coding, it does not have any speed-up but slightly slows down the intersection test. (2.4s to 2.5s in **landsea-big**) I did some further research in counting the grids traversed in the simple 2D-DDA. I found that for **landsea-big**, the averaged number of cells traversed by rays that intersect the heightfield bounding box (otherwise will simply return **false**), is only 9.75 including both sea and land. This is very small since there are about 1000000 grids. To gain speed up from KD-tree, each leaf node must be as small as something like 4 * 4. But then, the tree would be too big thus having a lot of overheads. (Another statistical fact is that in the simple 2D-DDA, the average number of entering two-triangles intersection test per box-hitting ray is only 1.01. This explains the speed up of using bounding box per cell)

The third method I have tried is a variant of 2D-DDA. Instead of walking a cell at a time, it feels faster to jump more cells when you can. Therefore I create a bounding box for each overlapping 3*3 cells (16 points). Each time the ray does not intersect with the bigger **BBox**, we can jump right out of the bigger cell (it will always contract >= 3 jumps into 1 jump, thus saving >= 2 jumps). Otherwise we follow the usual mechanism of 2D-DDA, i.e. one grid at a time with single-cell **BBox** speed-up. The codes are in **heightfield2-3J.***. This mechanism indeed lower the averaged number of cells traversed from 9.75 to 4.29. But similar to KD-tree speed-up, it still yields slightly longer training time (2.4s to 2.5s for **landsea-big**). This may due to the overhead of creating the bigger bounding boxes and the overhead of checking the intersection with this bigger bounding box. Also, jumping through more cells take more computations than moving through one, which takes only one **float** addition and an **int** increment. However when jumping through several cells, we have to move both x and y, which would take more **float** addition and even needs a **float** multiplication.

Although I have tried a few different approaches, the simple 2D-DDA with a small modification yields the best performance. Later on, I will consider only the first approach.

II. Smooth shading algorithm

This part is more standard, for each vertex on the grid, I calculate the average **dpdu** and **dpdv** based on the six adjacent triangles (there are less adjacent triangles for boundary points). Then in **GetShadingGeometry**, for a point on the triangle, it's **dpdu** and **dpdv** are redefined as the weighted average of **dpdu** and **dpdv** on the three triangle vertices. The weighting is the point's barycentric coordinate on the triangle. Then the normal of the point is calculated based on it's

dpdu and **dpdv**. At first I interpolate only the normal, although doing so the picture is not much different than now, the surface is a little bit less smooth.

III. Performance Evaluation

The operating system for experiments is Mac OS. The processor is 4 GHz Intel Core i7 with memory 16GB 1600MHz DDR3. Cores detected by pbrt is 8 cores. The result for single-cores (--**ncores 1**) and 8-cores (default) are both shown, denoted as (1) and (8). Real time is shown.

	landsea-0	landsea-1	landsea-2	landsea-b	texture	hftest
Default HF (1)	1.202s	1.301s	1.140s	6.573s	0.766s	0.177s
My HF (1)	1.145s	1.180s	1.088s	2.406s	0.682s	0.171s
My HF w/ P (1)	1.152s	1.190s	1.098s	2.450s	0.690s	0.175s
Default HF (8)	0.457s	0.487s	0.459s	4.884s	0.235s	0.059s
My HF (8)	0.445s	0.449s	0.442s	0.907s	0.208s	0.059s
My HF w/ P (8)	0.444s	0.450s	0.444s	0.935s	0.211s	0.059s

IV. All Results (6 + 12)

PBRT Default | My heightfield | My heightfield w/ Phong