Report for Project #1

DEADLINE: 10/27/2016, 14:00
INSTRUCTOR: Yung-Yu Chuang

王冠鈞 b03902027

About `Heightfield2` Class Implementation

The entire implementation basically consists of two parts: (1) the intersection functions and (2) the smooth shading. Basically both of them are done, but due to my underestimation of regarding the difficulty of this project, I just couldn't make the results more refined before deadline, and some of the rendered images just sucks. If I have a more refined result within a week after the deadline, I'd probably upload that and this version can be omitted. Anyway, I'll start talking about the implementation.

(1) *The Intersection functions:* `Heightfield2::Intersection()` *et al.* This function is a major point in the implementation. It is used to find whether there is a valid intersection between a ray and the heightfield. My implementation contains two parts, the first one is a 2D-DDA algorithm, and the second one is a ray-triangle intersection function. Since for every adjacent four points $(x, y, z_1), (x + 1, y, z_2), (x, y + 1, z_3), (x + 1, y + 1, z_4)$, it can form two triangles by drawing a diagonal line from (x, y, z_1) to $(x + 1, y + 1, z_4)$. Then we can regard a heightfield as lots of triangles and we just have to examine whether the ray intersects with any of these triangles, and the 2D-DDA algorithm just helps to filter the possible triangles. The following paragraphs talk about the two algorithms mentioned respectively.

When the `Intersection()` function is called, it will first perform a DDA algorithm. First, it will test whether the ray intersects with the bounding box. If does, It will start from the first intersection of the box and the ray, and until the ray leaves the box or an intersection with the heightfield is found, it examines what (x, y) coordinates the ray goes through and examines every triangle within those coordinates. In this process, it will call `triIntersection()` or other similar ones, which is specialized with checking the triangle-ray intersection.

After entering the above-mentioned function, it will check whether it intersects with a specific triangle. Since such algorithm has already been implemented in somewhere in `pbrt-v2` (`shapes/trianglemesh.cpp`: `Triangle::Intersect()`), I just used it with some minor changes. In this algorithm, it uses barycentric coordinate to check intersection points, by solving some linear equations, and checks whether there is a valid solution for the parameters b_1, b_2. If so, then it continues computing its `DifferentialGeometry` data and others, and then returns.

(2) *The Smooth Shading.* Without smooth shading, the results will just be filled with triangles, which looks unnatural. Since there is also a shading algorithm already implemented in `pbrt-v2` (`shapes/trianglemesh.cpp`: `Triangle::GetShadingGeometry()`), I also applied this by making minor changes on this function. Also, since this algorithm requires the normal data of every reference point on the heightfield, I let the class generate an array of normal data when the class is constructed. For every reference point in the heightfield, there are either 1, 2, 3 or 6 triangles intersect with that point. The normal of that point is defined to be the sum of all the triangle normals beside the point. After such data are constructed, the algorithm can just use them to calculate some interpolations and write the results into the `DifferentialGeometry` class, which will affect the shading effects.

Comparison with class HeightField

Though applying some sort of acceleration such as DDA and bounding box may expect its faster performance, it just runs slower then the older heightfield with reasons unknown. The following are the times taken when generating the images below:[1] [2] [3]

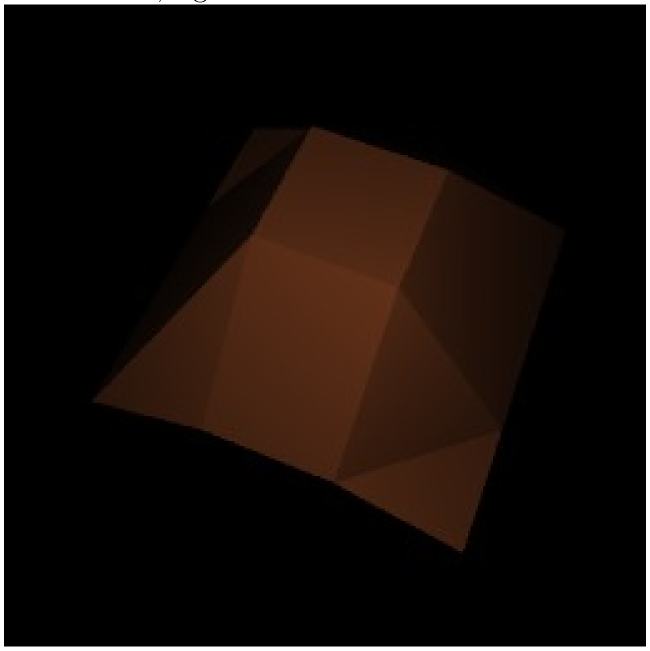
	`Heightfield`			`Heightfield2` (no shading)			`Heightfield2` (with shading)		
time	pbrt	real	user	pbrt	real	user	pbrt	real	user
`hftest`	0.1	0.654	0.600	0.1	0.349	0.680	0.1	0.240	0.637
`landsea-0`	0.4	1.023	3.860	9.8	10.331	1:35.047	8.5	8.986	1:37.013
`landsea-1`	0.4	0.919	4.153	15.9	16.374	2:47.423	15.6	16.058	3:03.577
`landsea-2`	0.3	0.882	3.507	4.9	5.412	54.200	5.2	5.760	56.527
`texture`	0.2	0.465	2.487	3.1	3.290	33.157	2.9	3.082	33.870
`landsea-big`	0.9	7.346	13.603	324.9	5:25.969	62:49.546	332.0	5:33.372	63:56.953

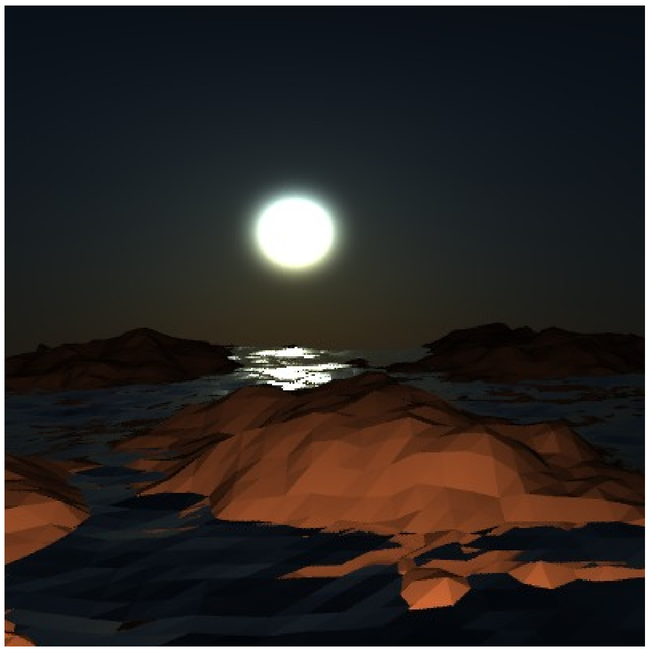
In general, the entire implementation is not successful (see the images below), and I've no enough time resources to thoroughly cope with the problems, both of which make me considerably frustrated.

Images

These are the oringinal images:[4]

left: `hftest`, right: `landsea-0`





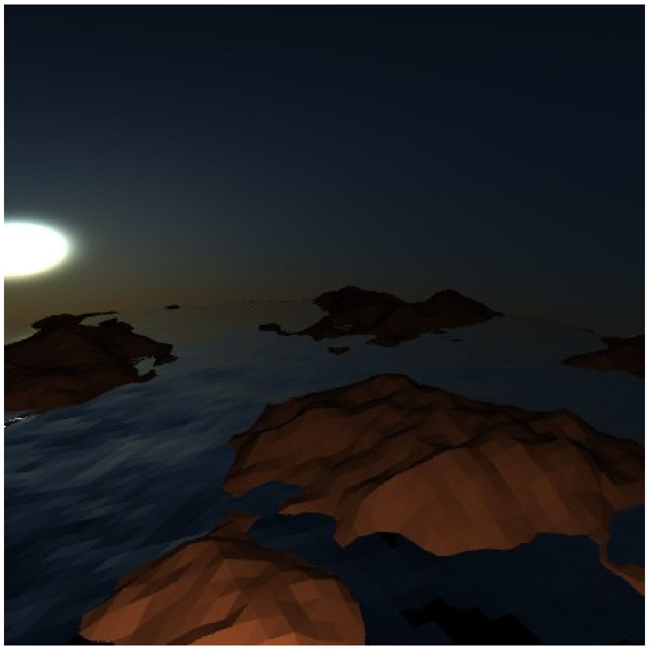
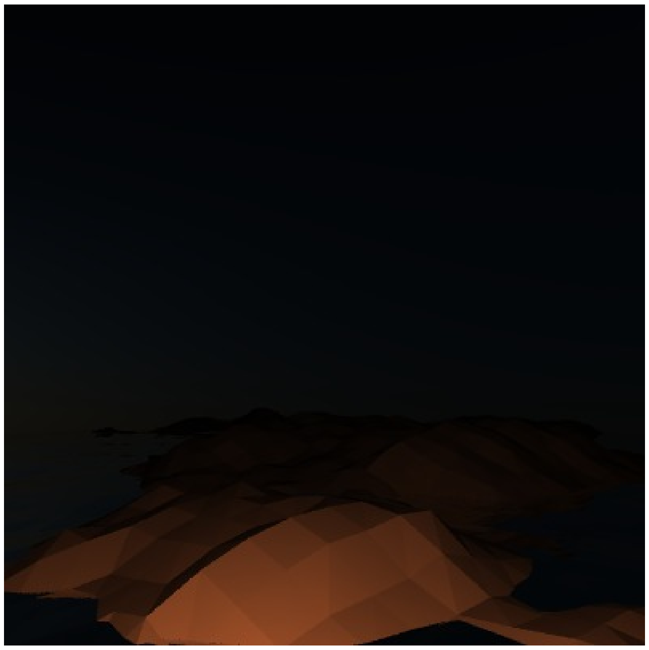
left: `landsea-1`, right: `landsea-2`

[1]`pbrt` time: Time given in the rendering progress bar.

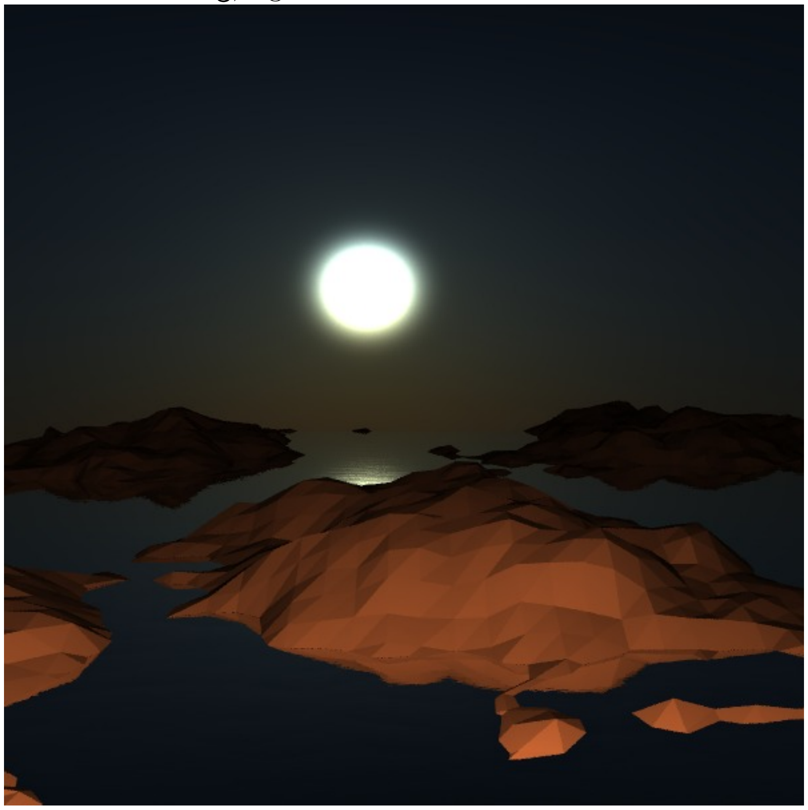
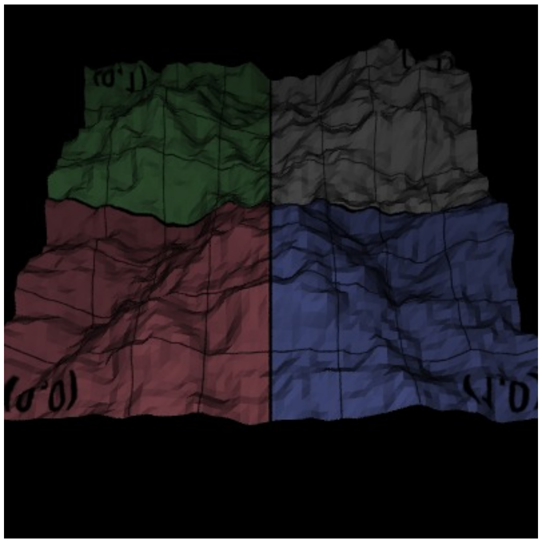
[2]real time: The real-world time, given in command `time`.

[3]user time: The CPU time in user mode; the sum of times of all CPU cores. Also given in command `time`.

[4]In the following, some of the images may look darker. Perhaps it's because I used two different `.exr` converters to convert them into `.jpg`.

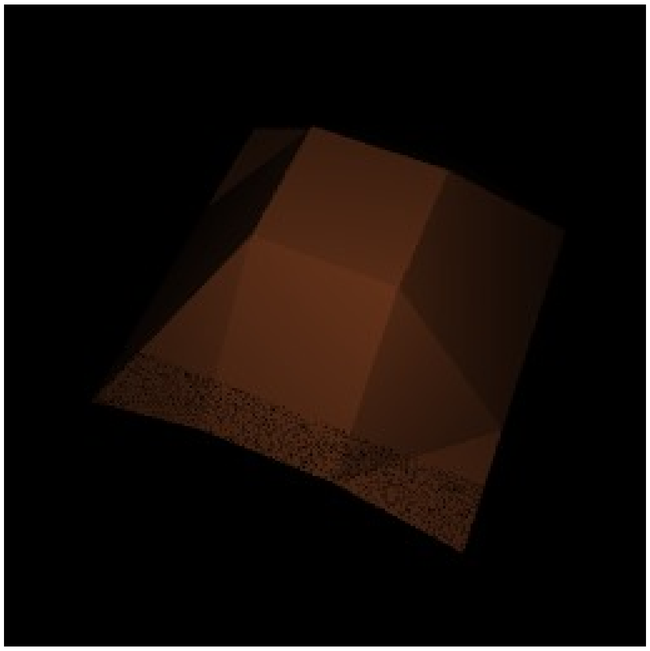
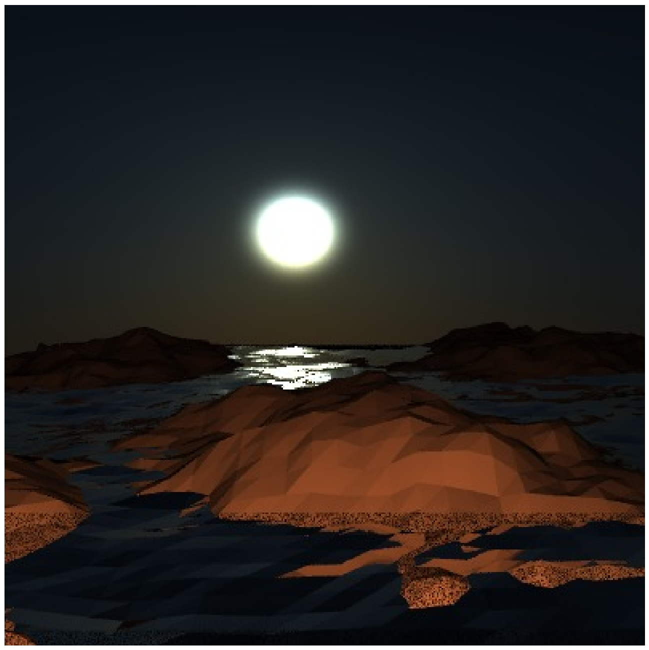



left: `landsea-big`, right: `texture`

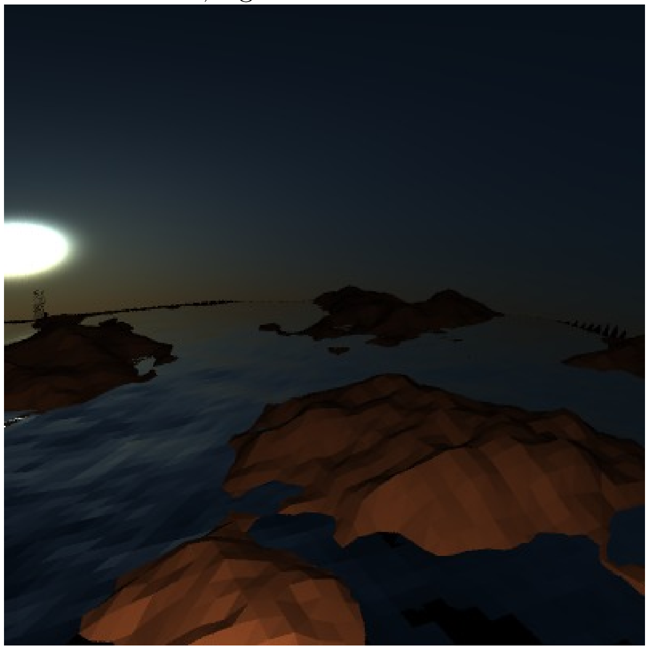
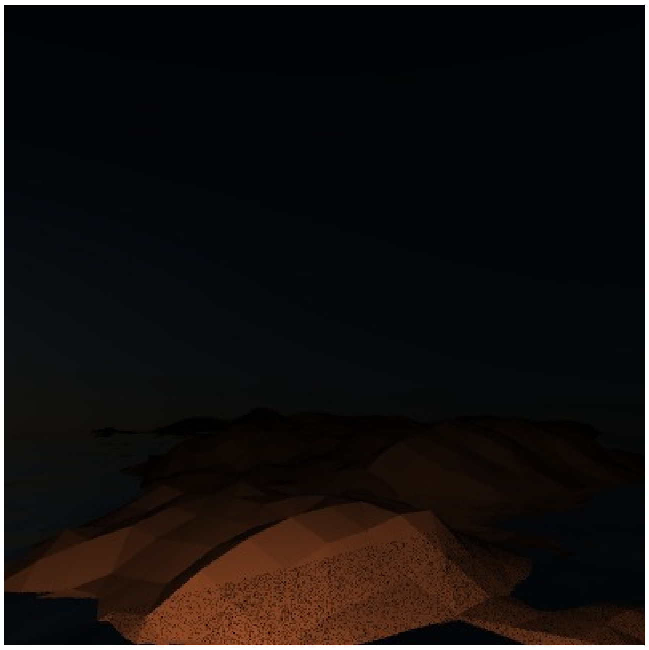



These are the `heightfield2` images without shading. The results has some minor flaws in some specific areas:

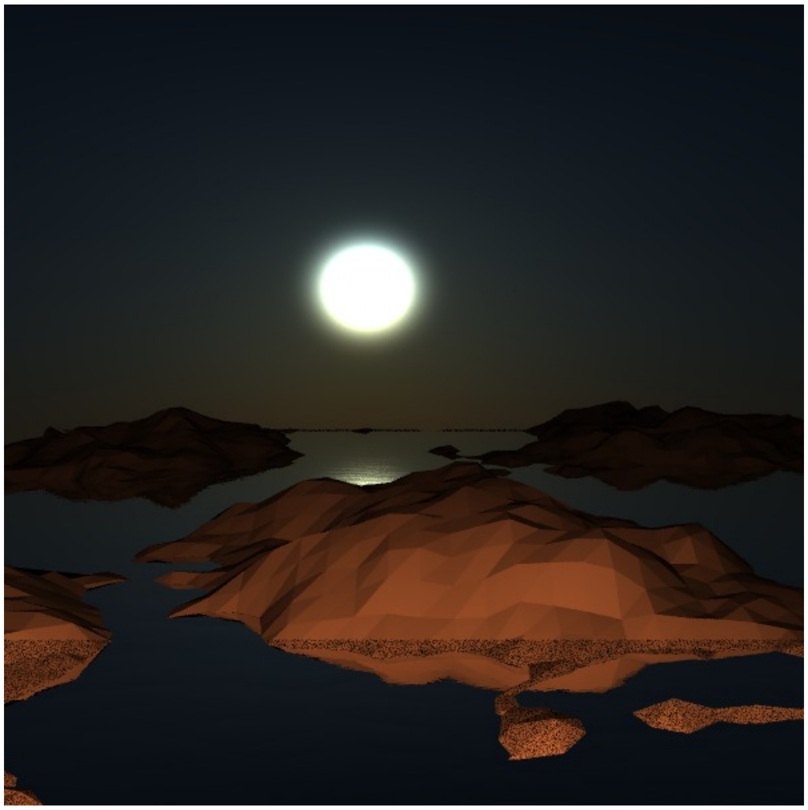
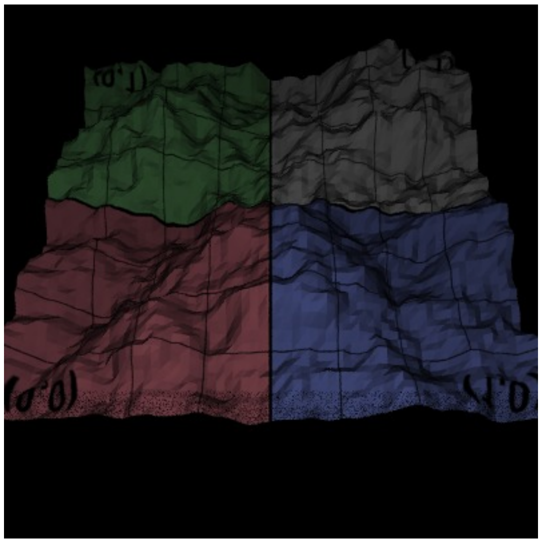
left: `hftest`, right: `landsea-0`

left: `landsea-1`, right: `landsea-2`

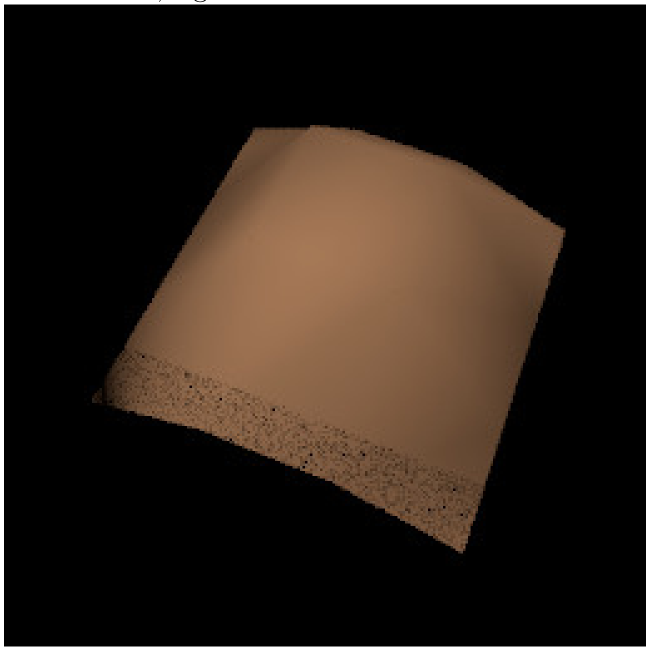
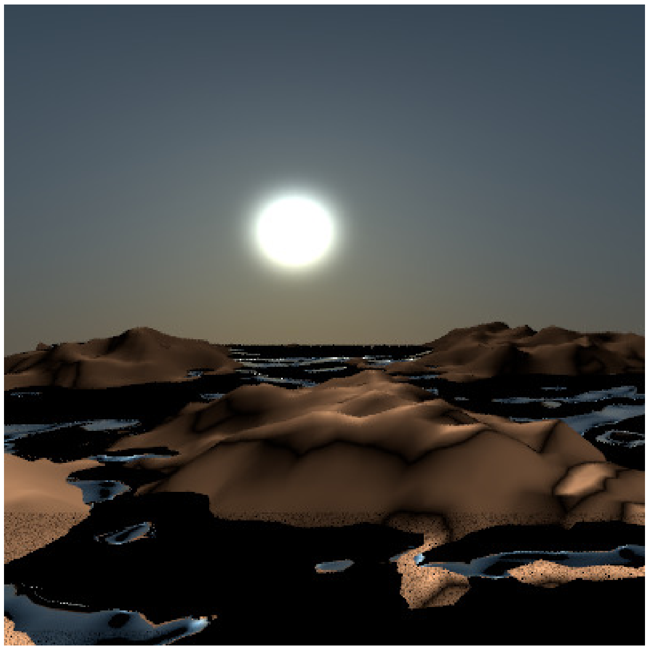



left: `landsea-big`, right: `texture`

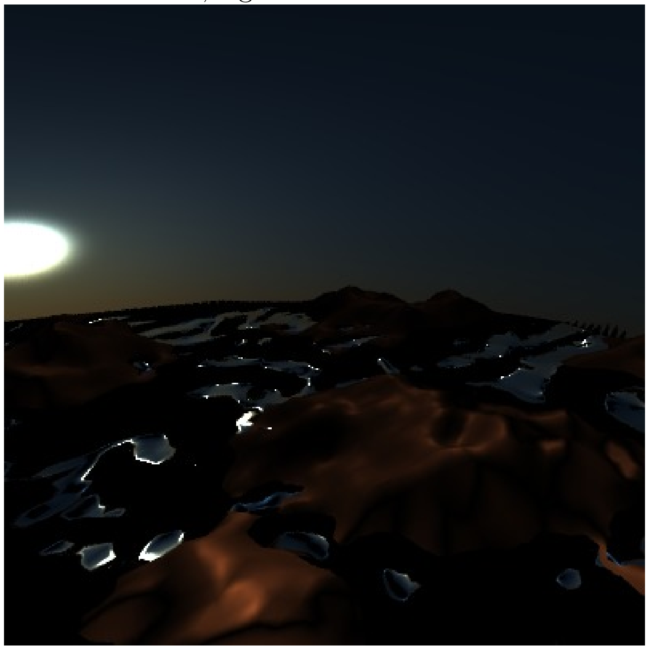
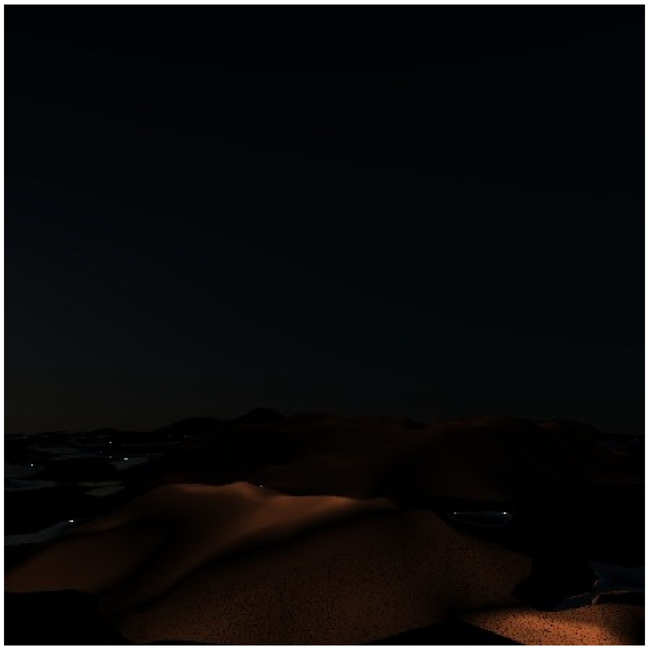



These are the `heightfield2` images with shading. The results has some frustrating flaws on the ill-rendered seas. I've found that when a heightfield applies `mirror` as its texture, it will end up like that, and it is the major bug / problem in the current version. If god wills, I'll figure out the reasons that contribute to the problem:

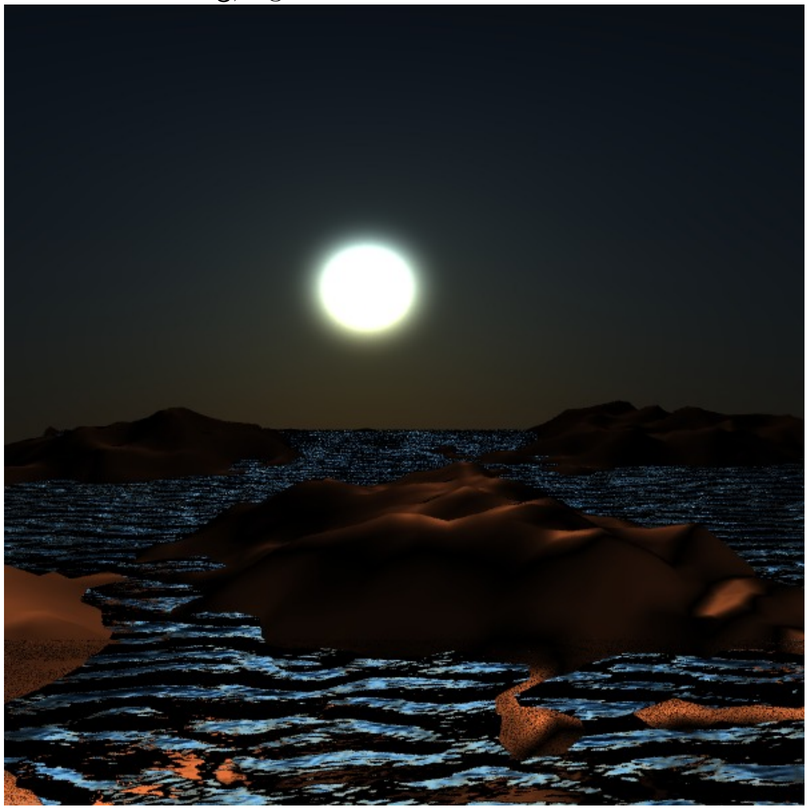
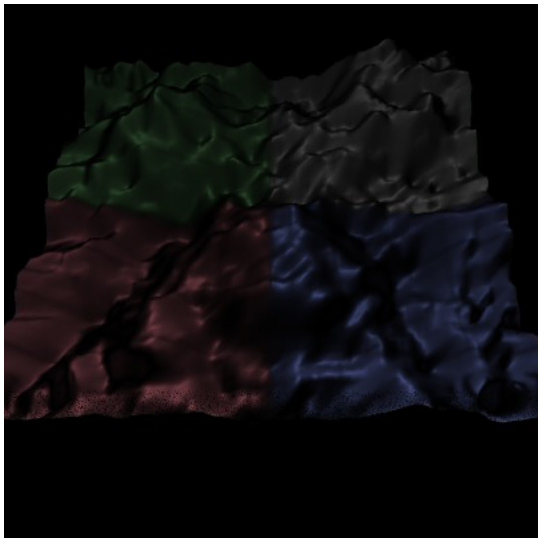
left: `hftest`, right: `landsea-0`

left: `landsea-1`, right: `landsea-2`

left: `landsea-big`, right: `texture`

Running Environment

The result images were rendered on the CSIE Workstation server. I used linux9 for execution. linux9 has a CPU of 2.53 GHz clock rate along with 16 cores, and 99 GB of memory. For the execution test, I used 12 of the cores (`-ncores 12`).

Other Notes

During this project, I encountered a new type of output: image file, which cannot be easily debugged. Thus, I used a special debugging tool, `vdb`[5], which allows users to draw lines or points to make sure that the geometric objects behave correctly.

[5] `https://github.com/zdevito/vdb`