Anmol Singh (2013021)
Sambhav Satija (2013085)

# Report : Project Milestone - 2

GPU Computing, Winter 2016

Milestone 2: Working raytracer with uniform grid

## Milestone 2 execution:

The major things we managed to accomplish or tried are:

**Support for loading triangulated meshes.**
After trials with simple geometric shapes like spheres, we decided to add support to load and render triangulated meshes of arbitrary objects. We found a good repository of 3D models at http://www.cs.cmu.edu/~kmcrane/Projects/ModelRepository/.
We implemented parsing for .obj files and processed them to create a usable list of all triangles in the model.

**Texture mapping.**
After tracing the silhouettes of a sample model ("Spot" from the above link), we attempted to implement texture mapping using the provided texture.

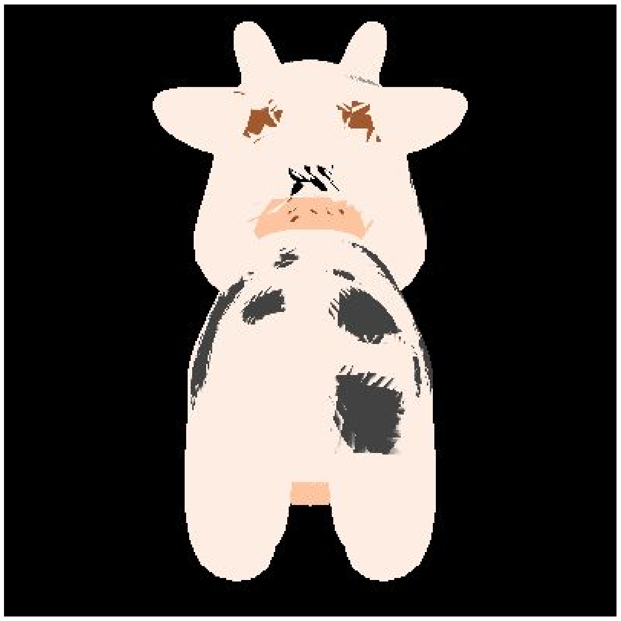

Fig.1 This image tried very hard to be a cow.

We tried to map the textures to the points of intersections by interpolating the texture using the barycentric coordinates of the point of intersection from the vertices of the intersected triangle. However, this resulted in incorrect mapping(Fig 1.), after which we shifted to Blinn Phong shading to get visual results.

**Shading.**
We implemented the Blinn-Phong shading.
The Blinn-Phong model adds specular highlights to the model apart from the diffuse parameter given by Lambertian shading. This is given by:

$$L = k_a I_a + k_d I \max(0, \mathbf{n} \cdot \mathbf{l}) + k_s I \max(0, \mathbf{n} \cdot \mathbf{h})^{\alpha}$$

**GPU Implementation.**
Everything described above was implemented as a serial C++ program first. Finally, we shifted our program to CUDA C/C++.
We implemented a basic parallel algorithm, where each thread runs the trace function for one pixel. We used a 2D block configuration. The speedup achieved enabled us to run the ray tracer for larger images.

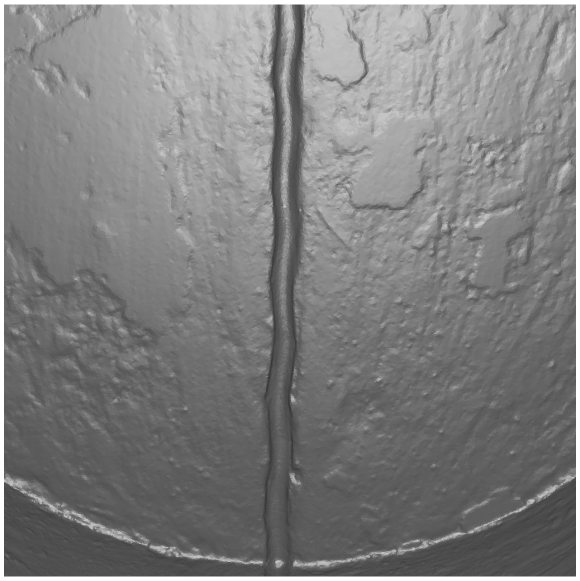

Fig 2a. “Nefertiti”

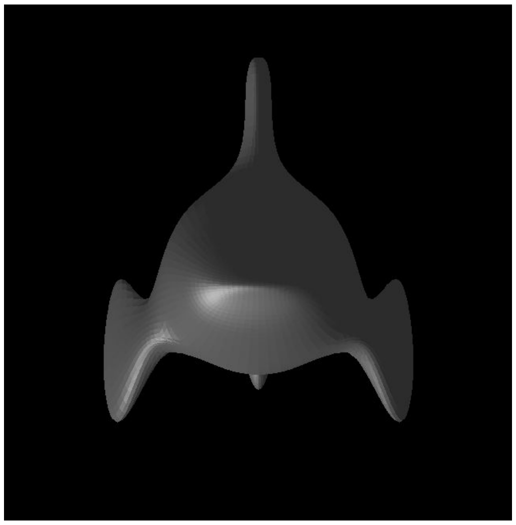
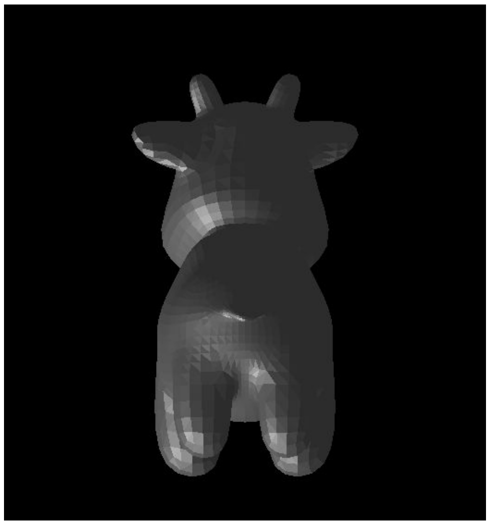

Fig 2b. “Blub”, Fig 2c. “Spot”

**Timing data (in ms):**

| Model | Triangles | Image size | CPU time (kernel equivalent) | GPU time (kernel) | GPU time (kernel + memory transfer) | Speed-up |
|---|---|---|---|---|---|---|
| Spot | 5856 | 1024x1024 | 150833.781 | 1168.107 | 1173.545 | 128.53 |

**Kernel analysis:**

nvprof results for “Spot (1024x1024)”

```
==9956== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 99.63%  1.16912s         1  1.16912s  1.16912s  1.16912s  trace_kernel(float*, Triangle*,
Vec3<float>*)
  0.37%  4.2978ms         1  4.2978ms  4.2978ms  4.2978ms  [CUDA memcpy DtoH]
  0.00%  44.001us         2  22.000us  1.1840us  42.817us  [CUDA memcpy HtoD
```

The bottleneck here is quite clearly the kernel. To optimize it, in future milestones, we will be implementing acceleration data structures. Since we expect non uniform work across our threads, we could also attempt implementing load balancing.