6.837 Intro to Computer Graphics
Assignment 6: Grid Acceleration & Solid Textures

In this assignment you will modify your ray tracer to use the grid from the previous assignment for fast ray casting. You will use your ray marching code and intersect all the objects stored in each traversed cell. You must pay attention to intersections outside the cell and implement early rejection to stop marching when you have found an appropriate intersection. To measure the performance improvement, you will analyze various statistics about your raytracer.

Finally, you will add new Material effects where the color of the material varies spatially using procedural solid texturing. This will allow you to render checkerboard planes and truly satisfy the historical rules of ray-tracing. Solid textures are a simple way to obtain material variation over an object. Different material properties (color, specular coefficients, etc.) vary as a function of spatial location.

Tasks

- Use the provided static `RayTracingStats` class to compute various statistics including the number of pixels, the number of rays cast, the number of ray/primitive intersection operations, the number of cells in the grid, the number of grid cells traversed with the ray marching technique, and the total running time. Add the following timing and counter increment functions provided in the `RayTracerStatistics` class to your code:

  - raytracing_stats.h
  - raytracing_stats.C

  Before beginning computation, call:

  ```
  RayTracingStats::Initialize(int _width, int _height, BoundingBox *_bbox,
                              int nx, int ny, int nz);
  ```

  For each non-shadow ray cast (a call to `RayTracer::TraceRay()`), call:

  ```
  RayTracingStats::IncrementNumNonShadowRays();
  ```

  For each shadow ray cast, call:

  ```
  RayTracingStats::IncrementNumShadowRays();
  ```

  For each ray/primitive intersection operation (each call to `intersect` for non-group and non-transform objects), call:

  ```
  RayTracingStats::IncrementNumIntersections();
  ```

  For each cell traversed (a call to `MarchingInfo::nextCell()`), call:

  ```
  RayTracingStats::IncrementNumGridCellsTraversed();
  ```

  From these numbers we can compute the average number of rays per pixel, primitive intersection calls per ray, grid cells per ray, rays per second, etc. Add support for an additional command line argument, `-stats`. If this option is specified, at the end of your rendering loop, print the various statistics by calling:

  ```
  RayTracingStats::PrintStatistics();
  ```

  Verify that the statistics are reasonable for simple test scenes without the grid, with and without shadows. Test this part of the assignment with examples from past assignments. Note that the number of rays cast and objects intersected can be predicted from the image resolution and number of objects in the scene.

- Now use the grid as a spatial acceleration for your ray caster. Implement two ray casting methods, `RayCast` and `RayCastFast`. If no grid is specified at the command line, the grid will be NULL and you should use the non-accelerated ray casting method (`RayCast`) which simply loops through all the objects in the scene as in previous assignments. If a grid is specified at the command line (`-grid` *`nx ny nz`*) and `-visualize_grid` is *not* specified, use the accelerated ray casting method (`RayCastFast`), which marches along the ray through the grid and only tests the ray for intersection with objects in the current cell.

  Make sure that you accelerate all your ray intersection routines, in particular do not forget shadow rays. Pay attention to objects that overlap multiple cells --- don't incorrectly return intersections outside of the current cell. Implementation of the marking strategy mentioned in lecture to avoid duplicate ray-primitive intersections for objects that overlap multiple cells is optional (extra credit).

- Since infinite primitives (e.g., planes) should not be stored in the grid, you'll have to handle them separately. You'll probably want to add an extra `Object3DVector` to the `Grid` class to store these primitives. You will always compute the intersections with these primitives and compare them to the closest in-grid intersection.

- Finish your implementation of the grid rasterization for scene hierarchies that include transformations (flattening the transformation hierarchy). You'll need to store the accumulated transformation matrix with each primitive. You can do this by creating a new `Transformation` that wraps around the primitive `Object3D` and store that in the Grid.

- For each test scene below, render with and without shadows and other recursive rays, with and without the grid, and with different grid sizes. Convince yourself that the resulting statistics make sense and summarize your findings in your `README.txt`. You might even discover some bugs or inefficiencies in your code! Don't copy-paste the statistics from each test case into your README.txt, *summarize the results in words*. Put some effort into your analysis, we will be grading it. Discuss how the various command line or scene parameters affect the running time and suggest how they could be automatically chosen. Also suggest how particular optimizations might affect the running time or memory used by your implementation. If you've implemented any bounding-box or grid-related optimizations (for extra credit) you should analyze the performance impact.

- Next we'll have fun with procedural solid materials. Derive a class `Checkerboard` from `Material`. SceneParser has been extended to parse the new material types.

  - scene_parser.h
  - scene_parser.C

  The `CheckerBoard` class will store pointers to two `Materials`, and a pointer to a `Matrix` which describes how the world space coordinates are mapped to the 3D solid texture space. Your solid texture implementation will describe a unit-length axis-aligned checkerboard between the two materials and the matrix will control the size and orientation of checkerboard cells. The prototype for the constructor is:

  ```
  CheckerBoard(Matrix *m, Material *mat1, Material *mat2);
  ```

  Handle this new material type in the interactive viewer by implementing `CheckerBoard::glSetMaterial()`. Because OpenGL does not implement general procedural texturing, you will simply call the corresponding `glSetMaterial()` method of the first material.

Implement the `CheckerBoard::shade` routine for ray tracing. You simply need to delegate the work to the `shade` function of the appropriate `Material` of the checkerboard. Using a float-to-integer conversion function `floor` and the function `odd`, devise a boolean function that corresponds to a 3D checkerboard. As a hint, start with the 1D case, just alternated segments of unit length, then generalize to 2 and 3 dimensions. Remember to multiply the intersection point by the matrix to properly transform the materials.

- Next you'll build materials using Perlin Noise, which lets you to add controllable irregularities to your procedural textures. Here's what Ken Perlin says about his invention (http://www.noisemachine.com/talk1/):

  > *Noise appears random, but isn't really. If it were really random, then you'd get a different result every time you call it. Instead, it's "pseudo-random" - it gives the appearance of randomness. Its appearance is similar to what you'd get if you took a big block of random values and blurred it (ie: convolved with a gaussian kernel). Although that would be quite expensive to compute.*

  So we'll use his more efficient (and recently improved) implemenation of noise (http://mrl.nyu.edu/~perlin/noise/) translated to C++:

  - perlin_noise.h
  - perlin_noise.C

  First derive the class `Noise` from `Material` which computes the function:

  ```
  N(x,y,z) = noise(x,y,z) + noise(2*x,2*y,2*z)/2 + noise(4*x,4*y,4*z)/4 + ...
  ```

  for the specified number of octaves. With just the first octave, the Noise function is very smooth and blobby. Including additional octaves of higher frequencies adds finer details to the texture. The value of `N(x,y,z)` will be a floating point number that you should clamp and use to interpolate between the two contained `Materials`. Here's the constructor for `Noise`:

  ```
  Noise(Matrix *m, Material *mat1, Material *mat2, int octaves);
  ```

  UPDATE (11/2): You'll need to interpolate the values returned by your `Material::Shade` computation. In addition, to correctly handle the ambient term and the reflective and transmissive components of materials, you'll need to modify the prototype of some `Material` accessor functions to also take the world space coordinate as an argument.

- Ken Perlin's original paper and his online notes have many cool examples of procedural textures that can be built from the noise function. You will implement a simple `Marble` material, also a subclass of `Material`. This shader uses the `sin` function to get bands of color that represent the veins of marble. These bands are perturbed by the noise function as follows:

  ```
  M(x,y,z) = sin (frequency * x + amplitude * N(x,y,z))
  ```

  Try different parameter settings to understand the variety of appearances you can get. Remember that this is not a physical simulation of marble, but a procedural texture attempting to imitate our observations. The constructor for `Marble` is:

  ```
  Marble(Matrix *m, Material *mat1, Material *mat2, int octaves, float frequency, float amplitude);
  ```

- For extra credit you can implement a simple `Wood` shader. There are thousands of different types of wood, there is no right answer for how to implement this texture. We've provided the code to parse some useful parameters -- feel free to modify the parameters as you choose. The code we've provided expects this constructor:

  ```
  Wood(Matrix *m, Material *mat1, Material *mat2, int octaves, float frequency, float amplitude);
  ```

  Procedural materials can be used for fun recursive material definitions. You can have a checkerboard where cells have different refraction and reflection characteristics, or a nested checkerboard containing different wood materials. The notion of recursive shaders is central to production rendering.

Additional References

- Robert Cook, "Shade Trees," Computer Graphics, vo]. 18, no. 3, July 1984.
- Darwin R. Peachey, "Solid Texturing and Complex Surfaces", SIGGRAPH vol. 19, no. 3, 1985, pp. 279-286.
- Ken Perlin, "An Image Synthesizer", SIGGRAPH vol. 19, no. 3, 1985, pp. 287-296.
- Ken Musgrave: http://www.kenmusgrave.com/
- Justin Legakis: Marble Applet

Ideas for Extra Credit

- Use marking to prevent duplicate intersections with a primitive that overlaps multiple cells,
- Experiment with other acceleration data structures (recursive/nested grid, octree, bounding volume hierarchy, etc.),
- Implement distribution ray tracing effects,
- Implement the `Wood` material class (test scenes 16 & 18),
- Create a new scene with complex geometry, interesting lighting and/or fun procedural solid textures,
- 2D Texture mapping (you'll need to specify a parameterization),
- Phong normal interpolation, bump mapping, etc.

Input Files

- scene6_01_sphere.txt
- scene6_02_sphere_triangles.txt
- scene6_03_sphere_plane.txt
- scene6_04_bunny_mesh_200.txt
- scene6_05_bunny_mesh_1k.txt
- scene6_06_bunny_mesh_5k.txt
- scene6_07_bunny_mesh_40k.txt
- scene6_08_scale_translate.txt
- scene6_09_rotated_triangles.txt
- scene6_10_nested_transformations.txt
- scene6_11_mirrored_floor.txt
- scene6_12_faceted_gem.txt
- scene6_13_checkerboard.txt
- scene6_14_glass_sphere.txt
- scene6_15_marble_cubes.txt
- scene6_16_wood_cubes.txt
- scene6_17_marble_vase.txt

- scene6_18_6.837_logo.txt

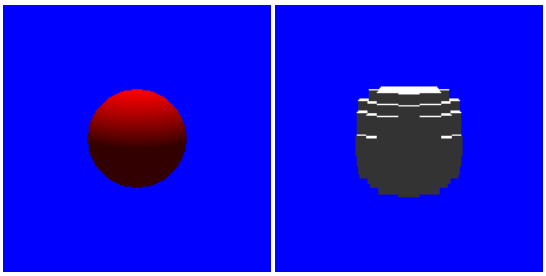
New Triangle Models

- vase.obj
- 6.837.obj

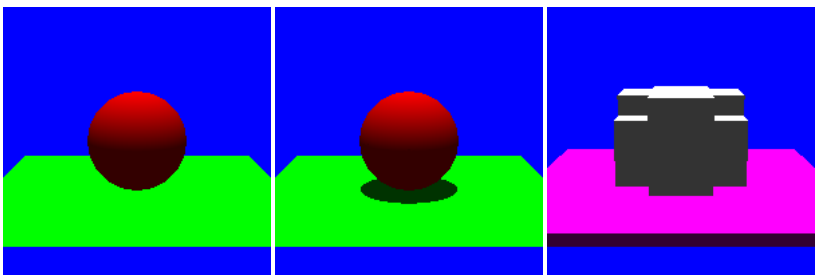
Sample Results

Analyze your raytracing statistics for each example and make sure they make sense. In some cases, using the grid may slow you down. Also, the total number of ray-primitive intersection operations may actually *increase* when the grid is added (unless you've implemented marking to prevent duplicate intersections between the ray and the same primitive).

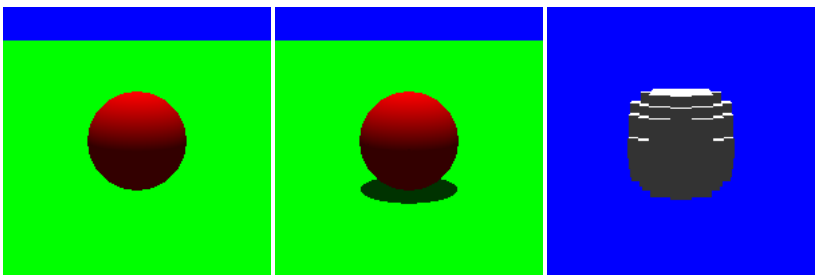
```
raytracer -input scene6_01_sphere.txt -output output6_01a.tga -size 200 200 -stats
raytracer -input scene6_01_sphere.txt -output output6_01b.tga -size 200 200 -grid 10 10 10 -stats
raytracer -input scene6_01_sphere.txt -output output6_01c.tga -size 200 200 -grid 10 10 10 -visualize_grid
```



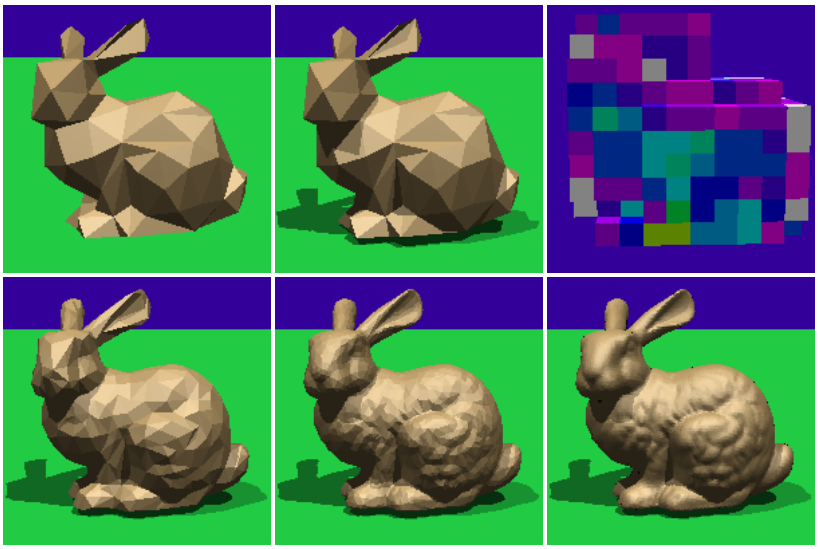
```
raytracer -input scene6_02_sphere_triangles.txt -output output6_02a.tga -size 200 200 -stats
raytracer -input scene6_02_sphere_triangles.txt -output output6_02b.tga -size 200 200 -grid 10 10 10 -stats
raytracer -input scene6_02_sphere_triangles.txt -output output6_02c.tga -size 200 200 -stats -shadows
raytracer -input scene6_02_sphere_triangles.txt -output output6_02d.tga -size 200 200 -grid 10 10 10 -stats -shadows
raytracer -input scene6_02_sphere_triangles.txt -output output6_02e.tga -size 200 200 -grid 10 10 10 -visualize_grid
```



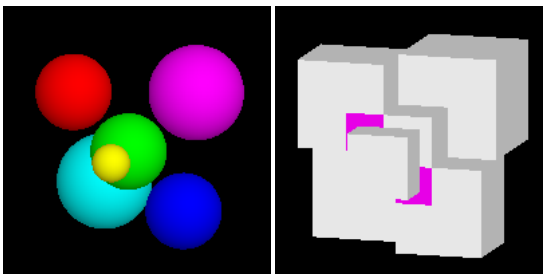
```
raytracer -input scene6_03_sphere_plane.txt -output output6_03a.tga -size 200 200 -stats
raytracer -input scene6_03_sphere_plane.txt -output output6_03b.tga -size 200 200 -grid 10 10 10 -stats
raytracer -input scene6_03_sphere_plane.txt -output output6_03c.tga -size 200 200 -stats -shadows
raytracer -input scene6_03_sphere_plane.txt -output output6_03d.tga -size 200 200 -grid 10 10 10 -stats -shadows
raytracer -input scene6_03_sphere_plane.txt -output output6_03e.tga -size 200 200 -grid 10 10 10 -visualize_grid
```



```
raytracer -input scene6_04_bunny_mesh_200.txt -output output6_04a.tga -size 200 200 -stats
raytracer -input scene6_04_bunny_mesh_200.txt -output output6_04b.tga -size 200 200 -grid 10 10 7 -stats
raytracer -input scene6_04_bunny_mesh_200.txt -output output6_04c.tga -size 200 200 -stats -shadows
raytracer -input scene6_04_bunny_mesh_200.txt -output output6_04d.tga -size 200 200 -grid 10 10 7 -stats -shadows
raytracer -input scene6_04_bunny_mesh_200.txt -output output6_04e.tga -size 200 200 -grid 10 10 7 -visualize_grid
raytracer -input scene6_05_bunny_mesh_1k.txt  -output output6_05.tga -size 200 200 -grid 15 15 12 -stats -shadows
raytracer -input scene6_06_bunny_mesh_5k.txt  -output output6_06.tga -size 200 200 -grid 20 20 15 -stats -shadows
raytracer -input scene6_07_bunny_mesh_40k.txt -output output6_07.tga -size 200 200 -grid 40 40 33 -stats -shadows
```

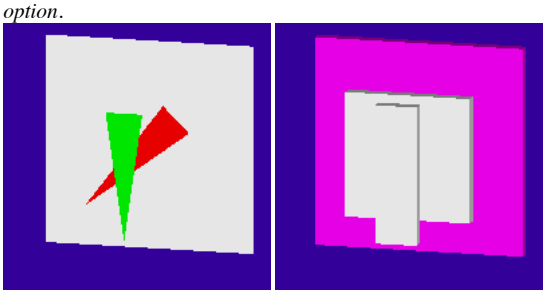


```
raytracer -input scene6_08_scale_translate.txt -size 200 200 -output output6_08a.tga
raytracer -input scene6_08_scale_translate.txt -size 200 200 -output output6_08b.tga -grid 15 15 15
raytracer -input scene6_08_scale_translate.txt -size 200 200 -output output6_08c.tga -grid 15 15 15 -visualize_grid
```



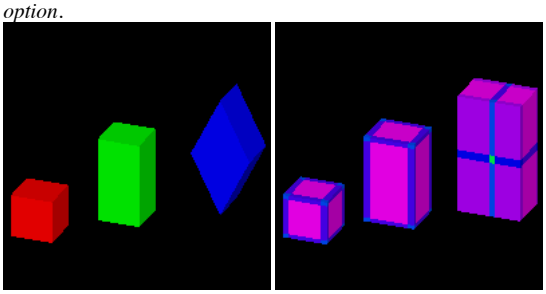
```
raytracer -input scene6_09_rotated_triangles.txt -size 200 200 -output output6_09a.tga
raytracer -input scene6_09_rotated_triangles.txt -size 200 200 -output output6_09b.tga -grid 15 15 9
raytracer -input scene6_09_rotated_triangles.txt -size 200 200 -output output6_09c.tga -grid 15 15 9 -visualize_grid
```

Note: the grid voxelization of the green triangle uses an optional special case for transformed triangles and will look different if you have not implemented this option.

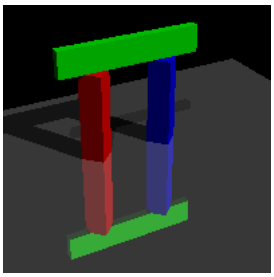
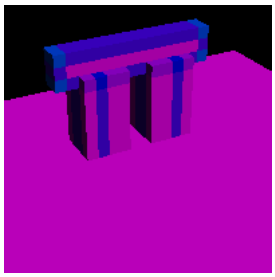


```
raytracer -input scene6_10_nested_transformations.txt -size 200 200 -output output6_10a.tga
raytracer -input scene6_10_nested_transformations.txt -size 200 200 -output output6_10b.tga -grid 30 30 30
raytracer -input scene6_10_nested_transformations.txt -size 200 200 -output output6_10c.tga -grid 30 30 30 -visualize_grid
```

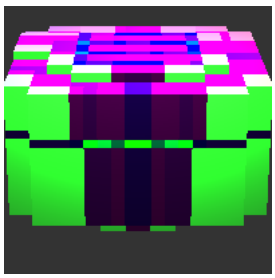
Note: the grid voxelization of the blue rhombus uses an optional special case for transformed triangles and will look different if you have not implemented this option.



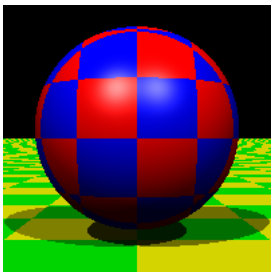
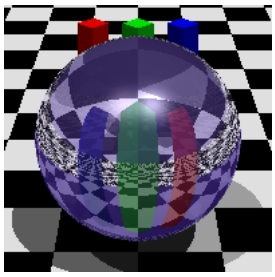
```
raytracer -input scene6_11_mirrored_floor.txt -size 200 200 -output output6_11a.tga -shadows -bounces 1 -weight 0.01 -stats
raytracer -input scene6_11_mirrored_floor.txt -size 200 200 -output output6_11b.tga -shadows -bounces 1 -weight 0.01 -grid 40 10 40 -stats
raytracer -input scene6_11_mirrored_floor.txt -size 200 200 -output output6_11c.tga -grid 40 10 40 -visualize_grid
```

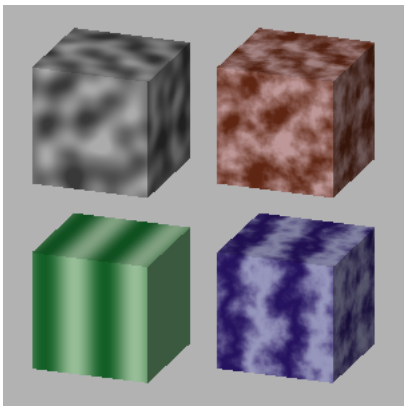
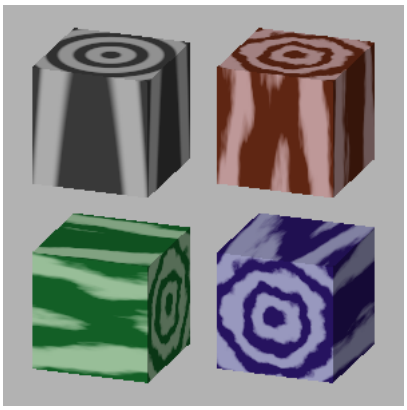
```
raytracer -input scene6_12_faceted_gem.txt -size 200 200 -output output6_12a.tga -shadows -shade_back -bounces 5 -weight 0.01 -stats
raytracer -input scene6_12_faceted_gem.txt -size 200 200 -output output6_12b.tga -shadows -shade_back -bounces 5 -weight 0.01 -grid 20 20
raytracer -input scene6_12_faceted_gem.txt -size 200 200 -output output6_12c.tga -grid 20 20 20 -visualize_grid
```

```
raytracer -input scene6_13_checkerboard.txt -size 200 200 -output output6_13.tga -shadows
raytracer -input scene6_14_glass_sphere.txt -size 200 200 -output output6_14.tga -shadows -shade_back -bounces 5 -weight 0.01 -grid 20 20
```

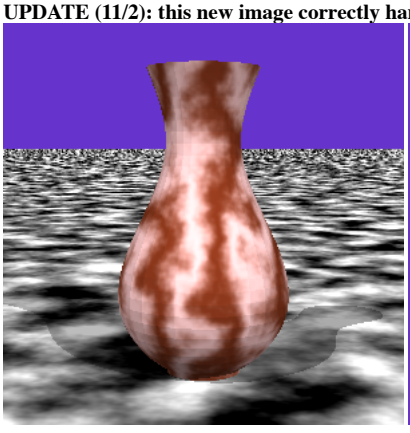
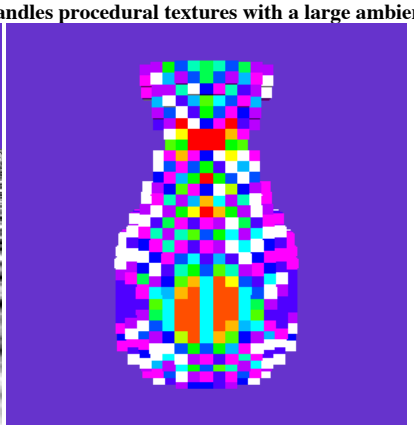



```
raytracer -input scene6_15_marble_cubes.txt -size 300 300 -output output6_15.tga
raytracer -input scene6_16_wood_cubes.txt -size 300 300 -output output6_16.tga
```

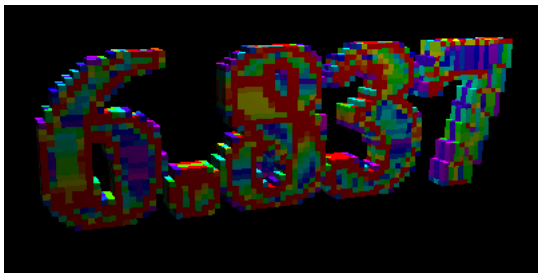



```
raytracer -input scene6_17_marble_vase.txt -size 300 300 -output output6_17a.tga -grid 15 30 15 -bounces 1 -shadows
raytracer -input scene6_17_marble_vase.txt -size 300 300 -output output6_17b.tga -grid 15 30 15 -visualize_grid
```

UPDATE (11/2): this new image correctly handles procedural textures with a large ambient light contribution.

```
raytracer -input scene6_18_6.837_logo.txt -size 400 200 -output output6_18a.tga -shadows -shade_back -bounces 5 -weight 0.01 -grid 80 30 3
raytracer -input scene6_18_6.837_logo.txt -size 400 200 -output output6_18b.tga -grid 80 30 3 -visualize_grid
```

See the main Assignments Page for submission information.
