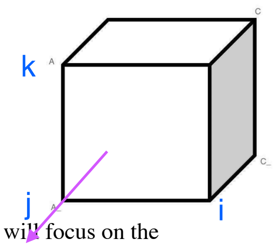


6.837 Intro to Computer Graphics Assignment 5: Voxel Rendering

In this assignment and the next, you will make your ray tracer faster using a spatial-acceleration data structure. This week we will focus on the implementation of a grid data structure and on fast ray grid intersection. Next week you will use the grid to accelerate your ray tracer.

To test your grid structure, you will implement the grid as a modeling primitive. Volumetric modeling can be implemented by affecting a binary opaqueness value for each grid cell. This is the equivalent of the discrete pixel representation of 2D images. Each volume element (or voxel) will be rendered as a solid cube. You can easily rasterize simple primitives in a grid (similar to pixel rasterization). For example, to rasterize a sphere, simply compare the distance between the center of a voxel and the sphere center to the sphere radius. You will use the grid to store the objects of your scene. In order to test your object insertion code, you will render cells that contain one or more objects as opaque and use color to visualize the density.

Initially, you may assume that no transformations are used. This way you may effectively ignore the group hierarchy and insert all primitives by scanning the scene in a depth-first manner. For the later test cases you will need to correctly transform the bounding boxes of each primitive before rasterizing it to the grid.

Tasks

implement it in addobject

- Extend your `Object3D` class to include a pointer to a `BoundingBox` instance. Each subclass should compute its conservative bounding box in its constructor, as appropriate. As discussed in class, the bounding box of a group is simply the union of the bounding boxes of the contained objects. The conservative bounding box of a transformed object is computed by transforming the 8 corners of the original bounding box and taking their axis-aligned bounding box. Add the accessor method `BoundingBox* Object3D::getBoundingBox()`.

  - boundingbox.h
  - boundingbox.C

  To avoid creating an infinite bounding box, don't try to compute the bounding box of a `Plane` (the `BoundingBox` of this primitive should be NULL). Don't include this infinite primitive when computing the bounding box of a group. Test your bounding box code by printing out the intermediate and scene bounding boxes for various test cases from previous assignments. Make sure your code handles scenes whose bounding box *does not* include the origin.

- Derive from `Object3D` a `Grid` class for an axis-aligned uniform grid. Initially a `Grid` will simply store whether each cell (voxel) is occupied so it can be rendered opaque or transparent. The constructor takes a pointer to the `BoundingBox` of the scene, and three integers describing the number of cells along the three axes.

  ```
  Grid::Grid(BoundingBox *bb, int nx, int ny, int nz);
  ```

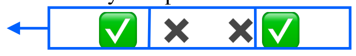
  Internal to your grid class, an array of n_x x n_y x n_z boolean values stores whether each voxel is opaque or transparent. Later we'll replace this array of booleans with an array of arrays of `Objects3D`.

- Extend your `Object3D` class with a new method:

  ```
  virtual void insertIntoGrid(Grid *g, Matrix *m);
  ```

  Note that this function is *not* pure virtual. Initially this function will do nothing. Override this function for spheres by implementing the corresponding method for the `Sphere` class. This method sets the opaqueness of each voxel which *may* intersect the sphere. You can do this by comparing the center of voxel to the sphere center and radius. Your computation should be conservative: you must label cells which might overlap the sphere, but it's also ok to label cells opaque which do not overlap the sphere. You may ignore the 2nd argument (`Matrix *m`) for this part, it will be used later in the assignment.

- Extend your `RayTracer` class to have a pointer to a `Grid` instance. Add support for two new command line arguments: `-grid <nx> <ny> <nz>`, to specify the size of the grid; and `-visualize_grid`, which indicates that the grid occupancy should be drawn instead of the scene objects. If the `-grid` command line option is not used, the grid will be `NULL`. Within the `RayTracer` constructor you'll initialize the grid and insert all of the objects in your scene into that grid. You'll need to write `Group::insertIntoGrid`, which like all group methods simply calls `insertIntoGrid` for all contained objects. Now you can test your sphere-rasterization routine with the first examples by simply printing the values of the grid's boolean array.



- Implement `Grid::paint()` to previsualize the occupancy of your grid. You may simply render 6 quads for each occupied grid cell, or to be more efficient, only draw the faces between occupied and unoccupied cells. Initially test your code with a very small grid (e.g. 1x1x1), on simple scenes. Make sure you have specified the correct normals.

  or transparent

- Implement ray marching through the grid so that you can also raytrace a visualization of grid occupancy. *Note: By using 3DDDA we can do this much more efficiently than computing a ray-box intersection with each voxel!* You will only use ray-box intersection to find the initial intersection of the ray with the scene bounding box.

  You will need a place to store the information for the current ray and the current grid cell. Implement a `MarchingInfo` class that stores the current value of *tmin*; the grid indices *i*, *j*, and *k* for the current grid cell; the next values of intersection along each axis (*t_next_x*, *t_next_y*, and *t_next_z*); the marching increments along each axis (*d_tx*, *d_ty*, *d_tz*), and the sign of the direction vector components (*sign_x*, *sign_y*, and *sign_z*). To render the occupied grid cells for visualization you will also need to store the surface normal of the cell face which was crossed to enter the current cell. Write the appropriate accessors and modifiers.

- The intersection of a `Ray` with a `Grid` will use two helper functions to initialize the march and to move to the next cell. First write:

  ```
  void Grid::initializeRayMarch(MarchingInfo &mi, const Ray &r, float tmin) const;
  ```

  This function computes the marching increments and the information for the first cell traversed by the ray. Make sure to treat all three intersection cases: when the origin is inside the grid, when it is outside and the ray hits the grid, and when it is outside and it misses the grid. Test your `initializeRayMarch` routine by manually casting simple axis-aligned rays into a low-resolution grid. Also test more general cases. Make sure to test ray origins which are inside and outside the scene bounding box. Next, implement:

```
    void MarchingInfo::nextCell();
```

This update routine choose the smallest of the next t values (*t_next_x*, *t_next_y*, and *t_next_z*), and updates the corresponding cell index. Test your `nextCell` ray marching code using the same strategy as for initialization. Manually compute the marching sequence corresponding to a particular ray and then print the steps taken by your code. Try other origins and directions to make sure that your code works for all orientations (in particular, test both positive and negative components of the direction).

- To help you debug your ray marching code, we've added additional previsualization options to the `RayTree` class introduced in the last assignment. As before, a single ray is traced by pressing the '`t`' key. Then the '`g`' key will toggle between (1) visualizing all occupied grid cells, (2) visualizing the cells traversed while walking along the ray, and (3) the faces crossed to enter each cell along the ray.

    - rayTree.h
    - rayTree.C
    - glCanvas.h
    - glCanvas.C

  `GLCanvas::initialize` method has been modified to take in two additional parameters, the `Grid`, and a boolean indicating whether to visualize the grid or not. Insert the following commands within your `Grid` intersection routines as appropriate:

  ```
  void RayTree::AddHitCellFace(Vec3f a, Vec3f b, Vec3f c, Vec3f d, Vec3f normal, Material *m);
  void RayTree::AddEnteredFace(Vec3f a, Vec3f b, Vec3f c, Vec3f d, Vec3f normal, Material *m);
  ```

  To specify an entire cell, call `AddHitCellFace` 6 times. In the examples below, a color gradient has been used to show the order in which the cells are traversed (white, purple, ... orange, red). This gradient is optional, but can be helpful in debugging. To see the ray intersections more clearly, you may wish to turn off the transparent ray rendering in `RayTree::paint()`

- Now, use the two helpers functions in your ray tracer. The `Grid::initializeRayMarch` routine should be called at the beginning of the `Grid::intersect routine`. Once the `MarchingInfo` data structure is initialized, `MarchingInfo::nextCell` should be used to walk through the grid and check for opaque voxels. If the next cell is opaque/occupied, return the appropriate normal depending on which axis you advanced last. Verify that your raytraced image looks the same as the OpenGL previsualization.

- Rasterize all scene primitives (but not groups and transforms) into the appropriate cells of the grid. You'll need to modify the `Grid` class to store pointers to the objects whose bounding box overlaps the cell instead of simply boolean occupancy values. We've provided a dynamically resizable vector class, `Object3DVector`, to store an arbitrary number of `Object3D` pointers. You may use STL (Standard Template Library) instead if you prefer.

    - object3dvector.h

  In the examples below, a color gradient has been used to visualize the number of primitives which overlap each grid cell. Cells colored white have just 1 primitive, purple have 2 primitives, ... and cells colored red have many more. You should implement a similar visualization, but you may use a different color scheme.

- Finally, extend your implementation to correctly handle transformations. We could explicitly "flatten" the object hierarchy to create a new scene graph with a single transformation above each primitive. But it's also easy to just push all the transformations down to the primitive objects. Now we'll use the 2nd argument (`Matrix *m`) to `Object3D::insertIntoGrid` (and it's derived methods). Initially `m = NULL`. When you encounter a transformation node in the hierarchy you will concatenate it to the previous transformation (if any). Make sure you apply the transformations in the correct order.

  You are not required to handle transformations in your `Sphere::insertIntoGrid` method. If `m ≠ NULL`, it's ok to fall back on the `Object3D::insertIntoGrid` implementation. To explicitly call a parent class method, pre-pend the method with the class name:

  ```
      this->Object3D::insertIntoGrid(g,m);
  ```

  NOTE: We won't test or grade your transformation code until the next assignment, so it's ok if this isn't fully implemented or debugged yet. But we highly recommend that you try to do so now. Test scenes 10, 11, & 12 contain transformations.

Ideas for Extra Credit

- Implement a special case for transformations of triangle primitives to get a tighter bounding box
- Test if the plane of the triangles intersects the grid cells (less useful for small triangles)
- Volumetric rasterization of other fun implicit objects

Input Files

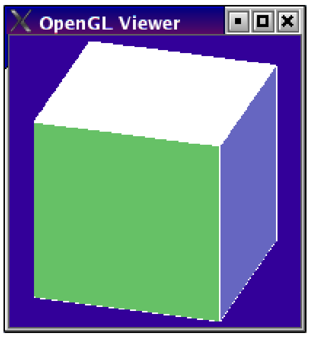
- scene5_01_sphere.txt
- scene5_02_spheres.txt
- scene5_03_offcenter_spheres.txt
- scene5_04_plane_test.txt
- scene5_05_sphere_triangles.txt
- scene5_06_bunny_mesh_200.txt
- scene5_07_bunny_mesh_1k.txt
- scene5_08_bunny_mesh_5k.txt
- scene5_09_bunny_mesh_40k.txt
- scene5_10_scale_translate.txt
- scene5_11_rotated_triangles.txt
- scene5_12_nested_transformations.txt

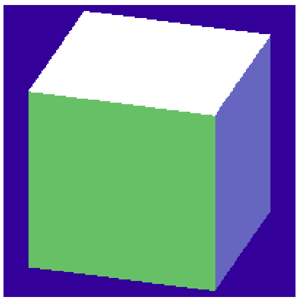
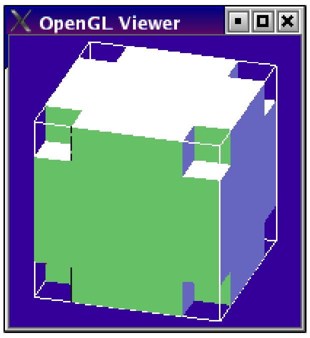
New Triangle Models

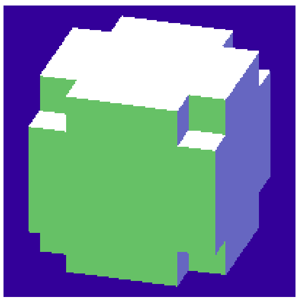
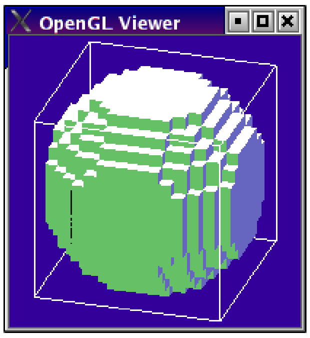
- bunny_5k.obj
- bunny_40k.obj

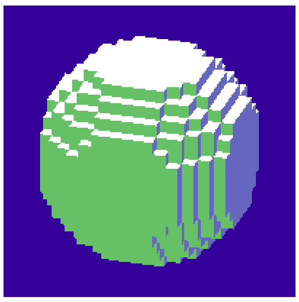
Sample Results

```
raytracer -input scene5_01_sphere.txt -size 200 200 -output output5_01a.tga -gui -grid 1 1 1 -visualize_grid
raytracer -input scene5_01_sphere.txt -size 200 200 -output output5_01b.tga -gui -grid 5 5 5 -visualize_grid
raytracer -input scene5_01_sphere.txt -size 200 200 -output output5_01c.tga -gui -grid 15 15 15 -visualize_grid
```

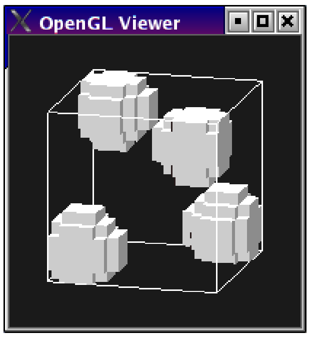


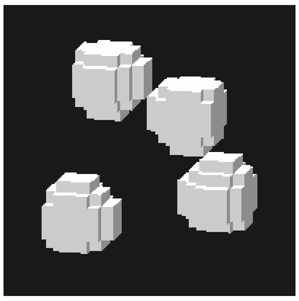
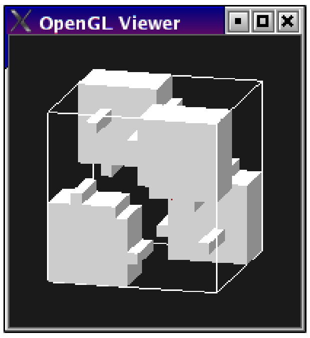



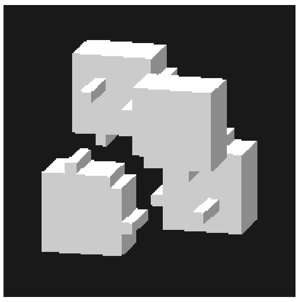





```
raytracer -input scene5_02_spheres.txt -size 200 200 -output output5_02a.tga -gui -grid 15 15 15 -visualize_grid
raytracer -input scene5_02_spheres.txt -size 200 200 -output output5_02b.tga -gui -grid 15 15 3 -visualize_grid
```

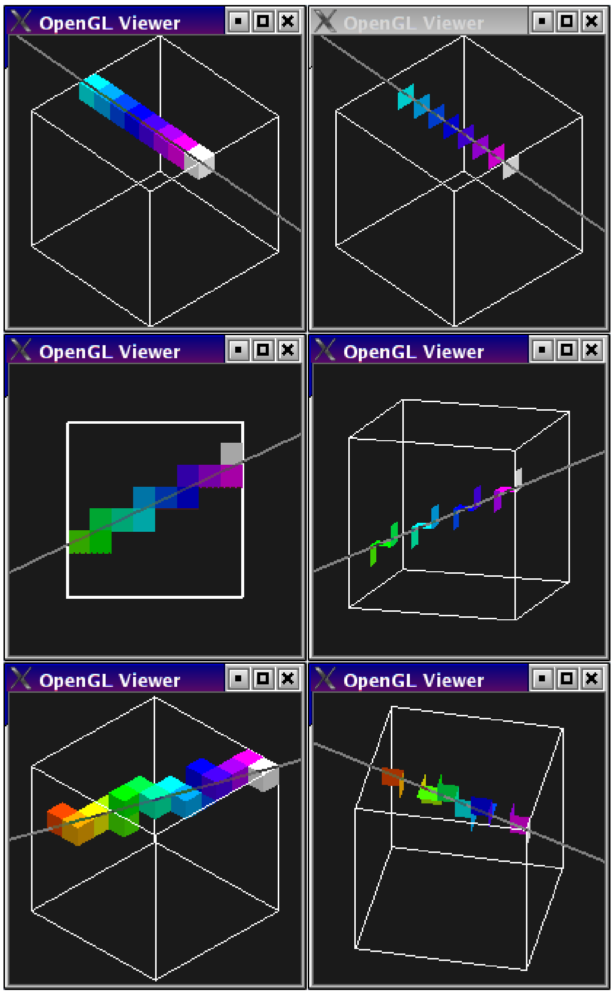


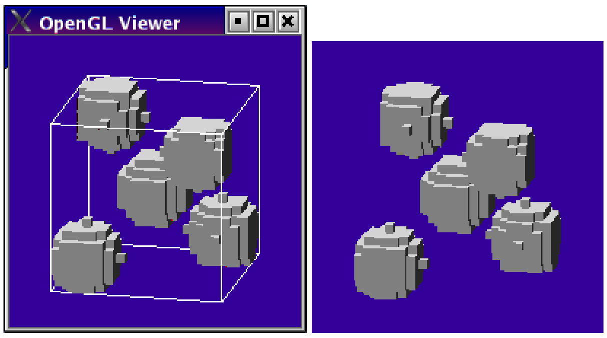


```
raytracer -input scene5_02_spheres.txt -size 200 200 -gui -grid 8 8 8 -visualize_grid
```

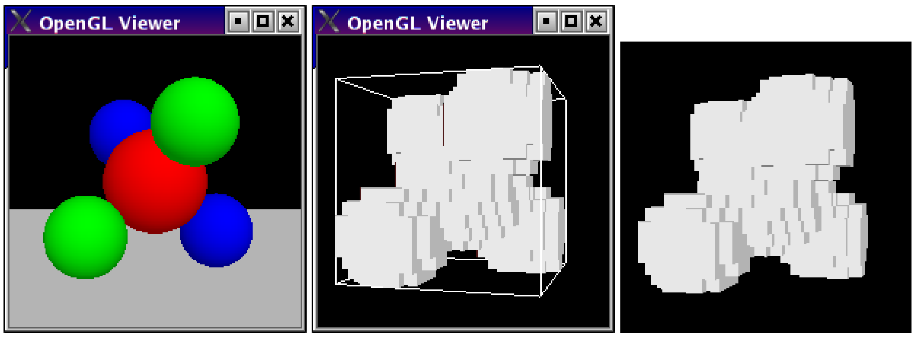
Visualize Hit Cells *Visualize Entered Faces*



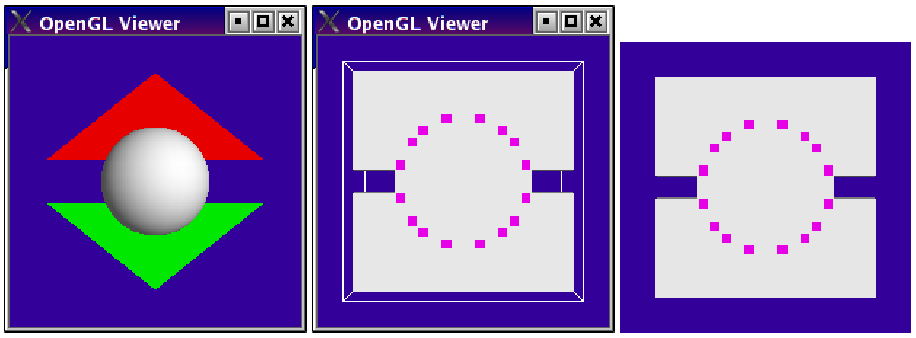
```
raytracer -input scene5_03_offcenter_spheres.txt -size 200 200 -output output5_03.tga -gui -grid 20 20 20 -visualize_grid
```



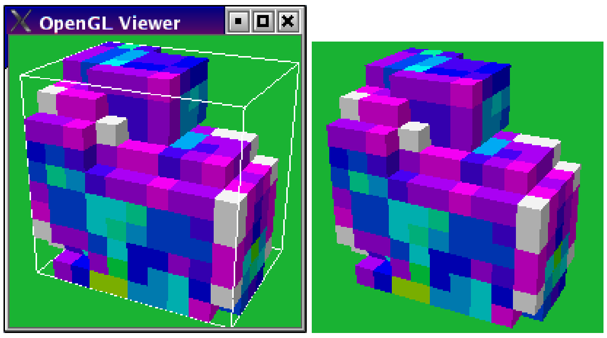
```
raytracer -input scene5_04_plane_test.txt -size 200 200 -gui -tessellation 30 15 -gouraud
raytracer -input scene5_04_plane_test.txt -size 200 200 -output output5_04.tga -gui -grid 15 15 15 -visualize_grid
```

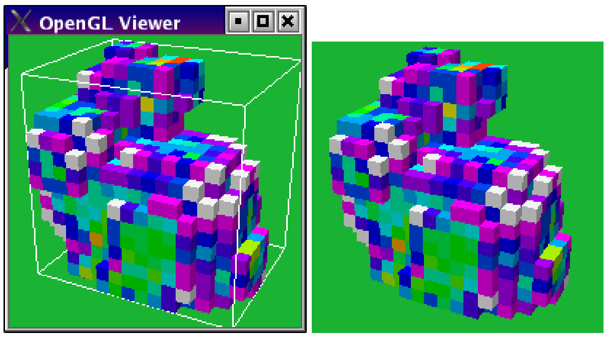


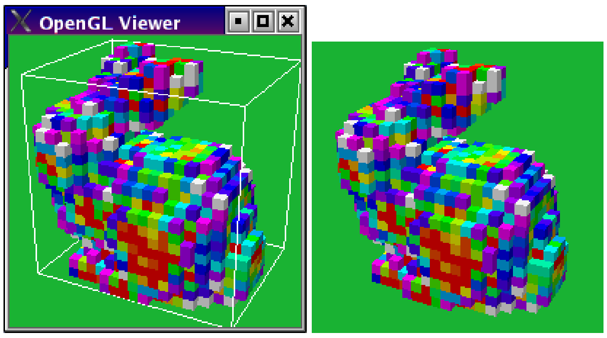
```
raytracer -input scene5_05_sphere_triangles.txt -size 200 200 -gui -tessellation 30 15 -gouraud
raytracer -input scene5_05_sphere_triangles.txt -size 200 200 -output output5_05.tga -gui -grid 20 20 10 -visualize_grid
```

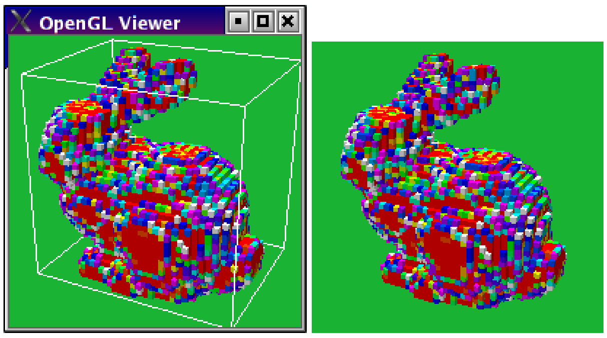


```
raytracer -input scene5_06_bunny_mesh_200.txt -size 200 200 -output output5_06.tga -gui -grid 10 10 7 -visualize_grid
raytracer -input scene5_07_bunny_mesh_1k.txt -size 200 200 -output output5_07.tga -gui -grid 15 15 12 -visualize_grid
raytracer -input scene5_08_bunny_mesh_5k.txt -size 200 200 -output output5_08.tga -gui -grid 20 20 15 -visualize_grid
raytracer -input scene5_09_bunny_mesh_40k.txt -size 200 200 -output output5_09.tga -gui -grid 40 40 33 -visualize_grid
```

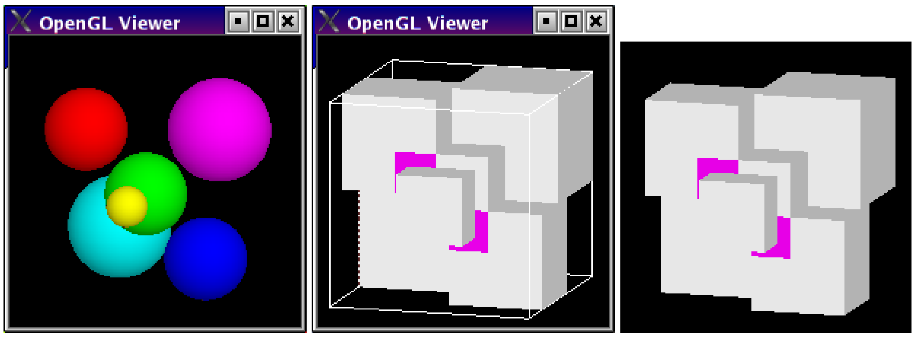






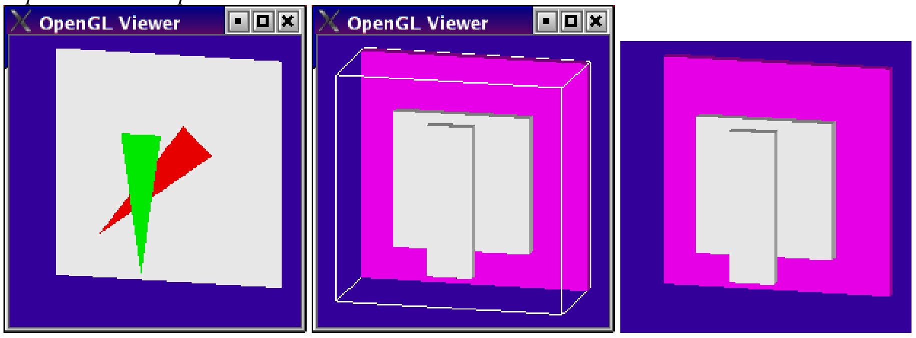


```
raytracer -input scene5_10_scale_translate.txt -size 200 200 -gui -tessellation 30 15 -gouraud
raytracer -input scene5_10_scale_translate.txt -size 200 200 -output output5_10.tga -gui -grid 15 15 15 -visualize_grid
```



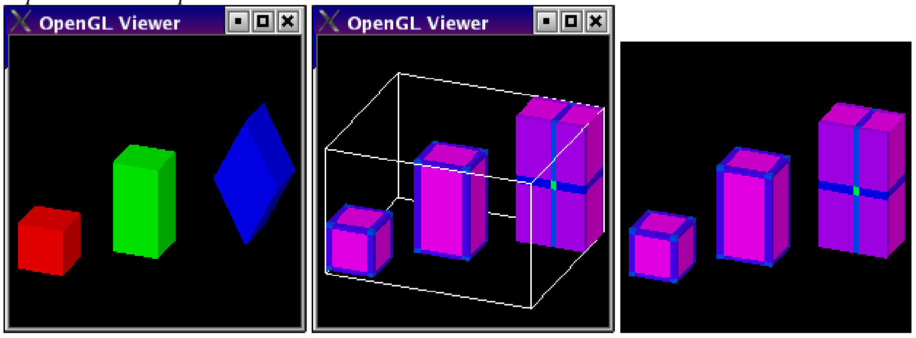
```
raytracer -input scene5_11_rotated_triangles.txt -size 200 200 -gui
raytracer -input scene5_11_rotated_triangles.txt -size 200 200 -output output5_11.tga -gui -grid 15 15 9 -visualize_grid
```

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 scene5_12_nested_transformations.txt -size 200 200 -gui
raytracer -input scene5_12_nested_transformations.txt -size 200 200 -output output5_12.tga -gui -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.



See the main Assignments Page for submission information.