# 6.837 Introduction to Computer Graphics
# Assignment 4: Grid Acceleration
# Due Wednesday October 15, 2003 at 11:59pm

This week, you will make your ray tracer faster using a spatial-acceleration data structure. You will implement grid acceleration, using fast ray marching. To convince yourself of the efficiency of your acceleration, you will analyze a set of statistics about your computation.

In order to test your grid structure before using it for acceleration, 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 very easily rasterize simple primitives in a grid; for example, to rasterize a sphere, simply test the distance between the center of a voxel and the sphere center.

After your grid modeling primitive is implemented and debugged, you will use it for acceleration. As a preprocess, you will insert all scene objects in the cells of the grid that they span. In order to test your object insertion code, you will render cells that contain one more more objects as opaque.

Then, you will modify your ray tracer to use the grid 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.

For this assignment, 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.

## 1 Ray Tracing Statistics

Use the provided `RayTracingStats` class to compute various statistics including the number of pixels, the number of rays cast, the number of ray/primitive intersections, 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:

- Call `RayTracingStats::Initialize(int width, int height, int num_x, int num_y, int num_z, const Vec3f &min, const Vec3f &max)` before beginning computation,
- Call `RayTracingStats::IncrementNumNonShadowRays()` for each non-shadow ray (a call to `RayTracer::TraceRay()`),
- Call `RayTracingStats::IncrementNumShadowRays()` for each shadow ray,
- Call `RayTracingStats::IncrementCellsTraversed()` for each cell traversed (a call to `MarchingInfo::nextCell()`),
- Call `RayTracingStats::IncrementNumIntersections()` for each ray/primitive intersection (not groups and transforms), and
- At the end of your main loop, print the various statistics by calling `RayTracingStats::PrintStatistics()`.

From these numbers we can compute the average number of rays per pixel, intersections per ray, grid cells per ray, rays per second, etc. Verify that the statistics are reasonable for simple test scenes. To verify that the number of rays cast is correct, add a `-no_shadows` command line argument to your program. Test this part of the assignment with examples from last week.

# 2 Grid

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 two `Vec3f`s describing the minimum and maximum coordinates of the grid, three integers describing the number of cells along the three axes, and a `Material*`.

```
Grid::Grid(Vec3f min, Vec3f max,
           int nx, int ny, int nz, Material *m);
```

An array of $nx \times ny \times nz$ `bool`s stores whether each voxel is opaque or transparent. Implement the `Grid::rasterizeSphere(Vec3f center, float radius)` method that sets the opaqueness of each voxel by testing whether its center is inside the sphere described by `center` and `radius`.

Test your sphere-rasterization routine on a small grid (e.g., 4x4) by printing the values of the array.

# 3 Fast Ray Marching with 3DDDA

To implement fast ray marching using 3DDDA, 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 the three axes ($dt_x, dt_y, dt_z$), and $sign_x, sign_y$, 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 grid. Write the appropriate accessors and modifiers.

The intersection of a ray with a `Grid` will use two helpers to initialize the march and to move to the next cell.

## 3.1 Initialization

First write:

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

This function sets the increments and the information relative to 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 routine with a very simple grid, for example a 4x4 grid from (-2,-2,-2) to (2,2,2). Use simple rays for which you can manually test the result. For example, initialize the cell march for a ray with origin (0.5, 0.5, 0.5), or for a ray with origin (-2.5, 0.5,0.5) and direction (1, 0, 0). Also test more general cases.

## 3.2 Marching step

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 ray marching code using the same strategy as for initialization. For example, use a ray with origin (-3, -2, 0.5) and directions such as (5, 2, 0). Note that the problem reduces to a 2D grid because the z component is 0. Manually compute the marching sequence, and 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).

### 3.3 Putting it all together

Finally, use these two helpers in your main grid-ray intersection routine. If the new cell is opaque, return the appropriate normal depending on which axis you advanced last. Test cases to visualize a simple sphere rasterization on your grid are provided.

## 4 Reducing the Number of Intersections

In order to use the grid as an acceleration structure, we must insert the scene primitives (but not groups and transforms) into the appropriate cells of the grid. First add two `Vec3f`s to your `Object3D` class to store the axis-aligned bounding box of each object. Add code to each subclass to compute these bounds (including `Group` and `Transform`). Verify that the computed scene bounding box is correct for various inputs.

Next modify the `Grid` class to also store pointers to the objects whose bounding box overlaps the cell. You may use the provided `Object3DVector` class to store an arbitrary number of objects per cell.

Finally, use the grid as a spatial acceleration for your ray caster. Implement two ray casting methods, `RayCast` (loops through all the objects in the scene as in previous assignments) and `RayCastFast` (uses the spatial acceleration data structure). Update your command line parsing so that the grid size may be specified at the command line: e.g., `-grid 5 5 5`. If no grid dimensions are provided, use the non-accelerated ray casting method.

Pay attention to objects overlapping multiple cells — don't incorrectly return intersections outside of the current cell.

## 5 What to Turn In

For each test scene, try different grid sizes and report the choice yielding the fastest time, and discuss the effect of this parameter on the running time.

Provide a `README.txt` file that discusses any problems you encountered, how long it took to complete the assignment, any extra credit work that you did and how we can test the new features.

## 6 Ideas for Extra Credit

Experiment with other acceleration data structures (recursive/nested grid, octree, non-nested grid, bounding volume hierarchy, etc.); Supersampling; Other distribution ray tracing effects; Flatten a scene graph which contains transformations by concatenating nested transformations; Compute a tight bounding box for each transformed triangle primitives by transforming the vertices; Test if the plane of the triangles intersects the grid cells; Volumetric rasterization

of other fun primitives; Use marking to prevent multiple intersections with a primitive that overlaps multiple cells; etc.

# 7 Other

`Object3DVector` (`object3dvector.h`)

Stores an arbitrary number of `Object3D` pointers. You may use STL (Standard Template Library) instead if you prefer.

**Ray Tracing Statistics** (`raytracing_stats.h` & `raytracing_stats.C`)

A class of static member variables to store and compute various ray tracing statistics.

**Parsing input files** (`scene_parser.h, and scene_parser.C`)

The `SceneParser` class has been extended to rasterize a sphere on a grid to help you debug your Grid class.

**New command line arguments**

`-no_shadows`
(don't cast any shadow rays)

`-grid 5 5 5`
(dimensions for the spatial acceleration data structure)

`-visualize_grid`
(render the grid cells opaquely)

`-visualize_grid_count`
(OPTIONAL: render the grid cells opaquely, and use color to indicate cells which contain more objects)