RAY-TRACING WITH UNIFORM SPATIAL DIVISION: PRACTICE

PROGRAMAÇÃO 3D
MEIC/IST

PROF. JOÃO MADEIRAS PEREIRA
Bibliography

- Chapter 22, “Ray-Tracing From The Ground Up”; Kevin Suffern, including images

- Articles:
Ray-Tracing

- Flexible, accurate, high-quality rendering
- Slow
- Simplest ray tracer:
  - Test every ray against every object in the scene
  - N objects, M rays $\rightarrow O(N \times M)$
- Using an acceleration scheme:
  - Acceleration scheme = sub-linear complexity of N
  - Grids and hierarchies
  - N objects, M rays $\rightarrow O(\log(N) \times M)$
  - $\log(N)$ is a theoretical estimate, in reality it depends on the Scene
  - Speedups of over 100x for complex scenes are possible
Uniform Grid

- Uniform Spatial Division
- Ray steps through the grid and is tested against objects in the grid cells along the path of the ray
- Can avoid testing the vast majority of the objects for each ray
- Grid building and traversal overheads can negate savings...
Uniform Grid: Problems

- Grid does not adapt to empty space and local complexity
  - Works best for uniformly distributed objects (seldom happens in reality)
  - Typical scenes have areas of complex geometry with empty space between them
- Empty space:
  - Time is wasted tracing the ray through empty grid cells
- Local complexity:
  - Too many objects in each grid cell
  - Could increase grid resolution, but that makes the empty space problem worse
- Difficult to choose optimal grid resolution that minimizes rendering time: tradeoff between these two problems
- Despite this, a grid is still much better than nothing
Uniform Grid working

- The ray only passes through certain cells;
- The cells are tested in depth-first order; i.e., in the order as \( t \) increases;
- It stops as the ray hits the nearest object.
Grid Construction

- Add the objects;
- Compute the bounding box;
- Set up the cells

- A scene with N objects will have $O(N^{1/3})$ cells in each direction;
Grid’s Bounding Box
Computing the Grid’s BB

```c
Point3D
Grid::min_coordinates(void) {
    BBox bbox;
    Point3D p0(kHugeValue);

    int num_objects = objects.size();

    for (int j = 0; j < num_objects; j++) {
        bbox = objects[j]->get_bounding_box();

        if (bbox.x0 < p0.x)
            p0.x = bbox.x0;
        if (bbox.y0 < p0.y)
            p0.y = bbox.y0;
        if (bbox.z0 < p0.z)
            p0.z = bbox.z0;
    }

    p0.x -= kEpsilon; p0.y -= kEpsilon; p0.z -= kEpsilon;

    return (p0);
}
```
Grid’s Cells Set Up

- Let \((w_x, w_y, w_z)\) the dimensions of the Grid in WCS; \(N\) is the number of objects; \(m\) is a multiplying factor that allows to vary the number of cells
- \(s = (\text{Grid Volume} / N)^{1/3} = (w_x \times w_y \times w_z / N)^{1/3}\)
- \(N_x = \text{trunc} (m \times w_x / s) + 1\)
- \(N_y = \text{trunc} (m \times w_y / s) + 1\)
- \(N_z = \text{trunc} (m \times w_z / s) + 1\)
- Total # of cells is \(N_x \times N_y \times N_z = m^3 \times N\)
- Grids seem to have a sweet spot when there are about 8-10 times more cells than objects
- Experiment different number for \(m\). Start with \(m=2\)
- Cells stored as 1D array of length \(N_x \times N_y \times N_z\)
- Index of cell \((i_x \ i_y \ i_z)\) is
  \[
  \text{Index} = i_x + N_x \times i_y + N_x \times N_y \times i_z
  \]
Cells and Objects

<table>
<thead>
<tr>
<th>Cell</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>empty</td>
</tr>
<tr>
<td>b</td>
<td>4</td>
</tr>
<tr>
<td>c</td>
<td>4</td>
</tr>
<tr>
<td>d</td>
<td>1 and 2</td>
</tr>
<tr>
<td>e</td>
<td>1, 2, 3, 4, 5, but only 1, 2, and 3 are actually in the cell</td>
</tr>
<tr>
<td>f</td>
<td>4 and 5</td>
</tr>
<tr>
<td>g</td>
<td>1, but object 1 is not in the cell</td>
</tr>
<tr>
<td>h</td>
<td>1 and 5</td>
</tr>
<tr>
<td>i</td>
<td>5</td>
</tr>
</tbody>
</table>
Cells and Objects

- To discover which cells an object belong, first compute the cells that contain the min and max coord of the object’s BB.
Calculating the cells’ indices

- Integers \((i_x, i_y, i_z) \in [0, n_x - 1] \times [0, n_y - 1] \times [0, n_z - 1]\)
- \(f(p_x) = (p_x - p_{ox}) / (p_{1x} - p_{ox}) \in [0.0, 1.0]\)
- \(i_x = \lceil N_x \times f(p_x) \rceil \in [0, N_x] ->\) integer-valued step function
- Problem in the max right boundary
- Use of function: \( \text{float clamp( float x, float min, float max)}\)
  - \(i_x = \text{clamp}(N_x \times f(p_x), 0, N_x - 1)\)
Cells and Objects

For (each obj from objects_array)

/* Compute indices of both cells that contain min and max coord of obj bbox */
int ixmin = clamp((obj_bbox.x0 - p0x) * Nx / (p1x - p0x), 0, Nx - 1);
int i ymin = clamp((obj_bbox.y0 - p0y) * Ny / (p1y - p0y), 0, Ny - 1);
int izmin = clamp((obj_bbox.z0 - p0z) * Nz / (p1z - p0z), 0, Nz - 1);
int ixmax = clamp((obj_bbox.x1 - p0x) * Nx / (p1x - p0x), 0, Nx - 1);
int iy max = clamp((obj_bbox.y1 - p0y) * Ny / (p1y - p0y), 0, Ny - 1);
int izmax = clamp((obj_bbox.z1 - p0z) * Nz / (p1z - p0z), 0, Nz - 1);

/* insert obj to the overlaped cells */
for(int i = i zmin; i <= izmax; i++)
    for(int i = iymin; i <= iy max; i++)
        for(int i = ixmin; i <= ix max; i++)
            index = i * Nx + i * Ny + i * Nz
            cells[index] -> add_object (obj)
Grid Traversal

- Ray- Bounding Box hit function must return:
  - the min and max t values in x, y and z directions (see next slide);
  - the 2 hit points in the BB (kay-kajya (KK) algorithm)
- It must handle inside or outside rays;

if the ray misses the Grid’s BB
  return false; // to and t1 crossover in the KK alg.

/* Calculate Starting Cell */
if the ray starts inside the Grid
  find the cell that contain the ray origin;
else
  find the cell where the ray hits the Grid from the outside;
traverse the Grid()
tmin and tmax from hit ray-BBox

\[ t_{\text{min}} = \frac{V_{\text{min}} - V_0}{V_d} \]
\[ t_{\text{max}} = \frac{V_{\text{max}} - V_0}{V_d} \]

If \( t_{\text{min}} > t_{\text{max}} \) swap \( t_{\text{min}} \) with \( t_{\text{max}} \)
Starting Cell

- $o_x, o_y, o_z$ is ray origin; $x_0..., x_1..., x_n$ are the grid’s BB vertices

```c
int ix, iy, iz;

if (bbox.inside(ray.o)) {
   ix = clamp((ox - x0) * nx / (x1 - x0), 0, nx - 1);
   iy = clamp((oy - y0) * ny / (y1 - y0), 0, ny - 1);
   iz = clamp((oz - z0) * nz / (z1 - z0), 0, nz - 1);
} else {
   Point3D p = ray.o + t0 * ray.d;
   ix = clamp((p.x - x0) * nx / (x1 - x0), 0, nx - 1);
   iy = clamp((p.y - y0) * ny / (y1 - y0), 0, ny - 1);
   iz = clamp((p.z - z0) * nz / (z1 - z0), 0, nz - 1);
}
```
Finding the nearest intersection

- Based on the 3D Digital Differential Analyser (3-DDDA)
Stepping the ray through the Grid

- Amanatides and Woo observation: although the ray-cell hits are unequally spaced along the ray, the hits on sets of parallel planes in the $x$, $y$ or $z$-direction are very even.

\[
dt_x = \frac{(t_{x\text{max}} - t_{x\text{min}})}{N_x}
\]
\[
dt_y = \frac{(t_{y\text{max}} - t_{y\text{min}})}{N_y}
\]
\[
dt_z = \frac{(t_{z\text{max}} - t_{z\text{min}})}{N_z}
\]
Stepping the ray through the Grid

- To find out next cell: is one across in x-direction or one up in y-direction?

```plaintext
compute txnext and tynext for the initial cell;
if txnext < tynext
  ix += 1;
txnext += dtx;
else
  iy += 1;
tynext += dty
```
Set Up Grid Traversal

// ray parameter increments per cell in the x, y, and z directions

double dtx = (tx_max - tx_min) / nx;
double dty = (ty_max - ty_min) / ny;
double dtz = (tz_max - tz_min) / nz;

double tx_next, ty_next, tz_next;
int ix_step, iy_step, iz_step;
int ix_stop, iy_stop, iz_stop;

// A ray has direction (dx, dy). Possible cases for direction xx':

if (dx > 0) {
    tx_next = tx_min + (ix + 1) * dtx;
    ix_step = +1;
    ix_stop = nx;
} else {
    tx_next = tx_min + (nx - ix) * dtx;
    ix_step = -1;
    ix_stop = -1;
}

if (dx == 0.0) {
    tx_next = kHugeValue; // WHY?
    ix_step = -1; // just to initialize. Never used
    ix_stop = -1;
}
while (true) {
    GeometricObject* object_ptr = cells[ix + nx * iy + nx * ny * iz];
    if (tx_next < ty_next && tx_next < tz_next) {
        if (object_ptr && object_ptr->hit(ray, t, sr) && t < tx_next) {
            material_ptr = object_ptr->get_material();
            return (true);
        }
    }
    tx_next += dtx;
    ix += i_step;
    if (ix == ix_stop)
        return (false);
}
else {
    if (ty_next < tz_next) {
        if (object_ptr && object_ptr->hit(ray, t, sr) && t < ty_next) {
            material_ptr = object_ptr->get_material();
            return (true);
        }
    }
    ty_next += dty;
    iy += iy_step;
    if (iy == iy_stop)
        return (false);
}
else {
    if (object_ptr && object_ptr->hit(ray, t, sr) && t < tz_next) {
        material_ptr = object_ptr->get_material();
        return (true);
    }
}
    tz_next += dtz;
    iz += iz_step;
    if (iz == iz_stop)
        return (false);
Mailboxes (cache)

- Grid Traversal problem due to multiple intersections: intersection may not be occurring in the current cell.
- Avoid it by labeling the rays with a unique number, and storing (caching) this number during an intersection test, in the primitive