HPSC Lab 12

Luna McBride

December 8 2023

1 Ray Tracing Description

This lab traces rays to their destination to see whether or not they have been blocked. The point of the lab is to be able to describe ray tracing in a controlled environment while also attempting to make the code general purpose for all possible mesh alignments. The exact mesh used is a cylinder inside another cylinder like a wire in a cable. The rays are emitting in all directions from singular faces in the external mesh pointing inward, emphasizing how the rays are only affecting their internal environment. This ray tracing code also only checks for what a specified cell "sees" for lack of a better word, so reflection and amount of energy coming through is not accounted for. In this sense, this makes the expected output the shadow created against the other wall from a cell emitting energy against the internal cylinder rather than the ray tracing expected in the modern video gaming scene. Rays are still being traced however, so it counts.

The code starts by checking each ray emitted against each face of the internal mesh. This is described by the following equations:

$$d = \frac{(\mathbf{p_0} - \mathbf{l_0}) \cdot \mathbf{n}}{\mathbf{l} \cdot \mathbf{n}}.$$

Figure 1: The Equation for d (from https://en.wikipedia.org/wiki/Line%E2%80%93plane_intersection)

$$\mathbf{p} = \mathbf{l_0} + \mathbf{l}\, d$$

Figure 2: The point equation for intersection

Figure 2 shows the equation for the point where the ray hits the specified face. It is simply a point on the ray (l_0, l standing for line) plus the product of the scalar value d and the ray (l) itself. Figure 1 shows how the d scalar value is calculated, starting with a point on the plane (or face of the mesh in this case, p_0) minus the reference point on the ray (l_0). The dot product of the value is then taken with the normal vector from the plane (n). All of this is then divided by the dot product of the ray and the normal vector (l*n). If l*n is 0, that means the ray does not hit this face on the mesh. Otherwise, it hits at exactly one point due to the rays being linear and not bouncing off of anything. There is also a use case in this equation for if the ray is entirely in the face ($(p_0 - l_0)$ * n = 0), but this is not possible in this use case.

In the case where the specified ray does not hit any of the faces on the internal cylinder, the loop checking this finishes and the corresponding face on the other side of the external mesh is added to the list of faces hit by the rays. Otherwise, there are checks in place to see which face of the internal cylinder specifically is blocking the ray. Note that the nature of this code does not need the internal component to be a cylinder; the two components could be a frozen pizza in an oven and still fit this use case. The cylinder just makes a more clear solution by blocking all of the rays all the way up and down.

For each cell of the blocking cylinder, each of the corners are checked to make sure the point calculated earlier is within the specified cell. This can be calculated using the two vectors that make up the specified corner and a third vector pointing to the point. The cross product between the two wall vectors and the point vector can be taken to get an idea of the direction from said wall to the point. The cross products finds a vector that is at right angles to the vectors passed in with a magnitude based on the direction between them. This magnitude is what is used here, as taking the dot product of these two resulting vectors can then show what direction from the original walls the point exists in. If this is a value greater than or equal to zero, then the point is within these walls. If the value is greater than or equal to zero for all of the corners, the point is within this cell. In this case, the cell ID is stored and can then be layered over the internal cylinder to show where rays are being blocked. The image showing this will be displayed in the next section, as there is a problem inherent to how it is being handled that needs to be shared first.

The code displaying these methods is not entirely fool-proof, however. These methods have a difficult time tracking if the ray hits a corner specifically, thus creating an issue of the point being in multiple planes. The code only checks one plane and returns, thus giving a list of blocked face IDs that is not entirely complete for all possible rays. The current code also checks for n*l vectors less than a tolerance rather than seeing if the value is around zero like expected. This is fine for a model that wraps around itself like this, but the code would have problems if the intersection equation produced a negative value.

Then comes a problem inherent to how the code is being handled. The code does not trace the rays themselves and find the first item hit like would be expected when tracing the path of a ray, but rather loops through the different cells of the internal mesh until it finds the first face that was hit by the ray in the order it checks the faces. This results in the following blocked mesh:

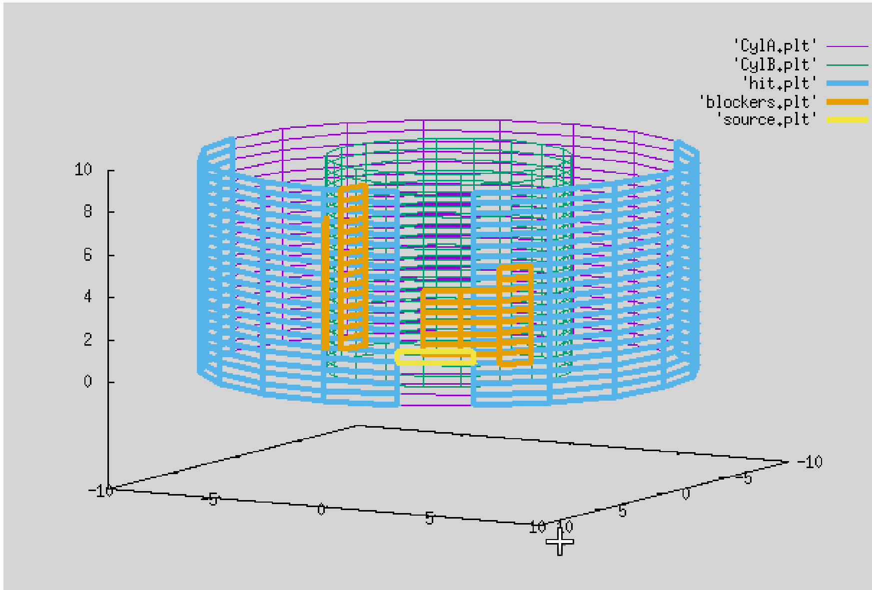

Figure 3: The Blocked Mesh

These cells do indeed block the ray, but they are not the first in the ray's path. The second half of the cylinder blocks just fine because that is the direction the faces are calculated by. It also makes sense that fewer faces on the blocking mesh would be hit because the rays do not have the chance to spread out before being blocked. It is just the blocking on the other side that creates the issue.

This is the biggest issue with the code overall. This is a downfall of the specific approach rather than something that can be fixed quickly, however. The simple fixes, like moving the mesh based on where the source is or changing the calculation's starting point to based on the source, would not work for cases where multiple cells along the external mesh are firing from different angles. The code would need to be changed to actually following the ray rather than seeing where it hits in order to fix this problem. This lab does not allow enough time to implement and test a fix of this sort however, so it will be for the best to acknowledge this as a major problem and keep going.

2 Results

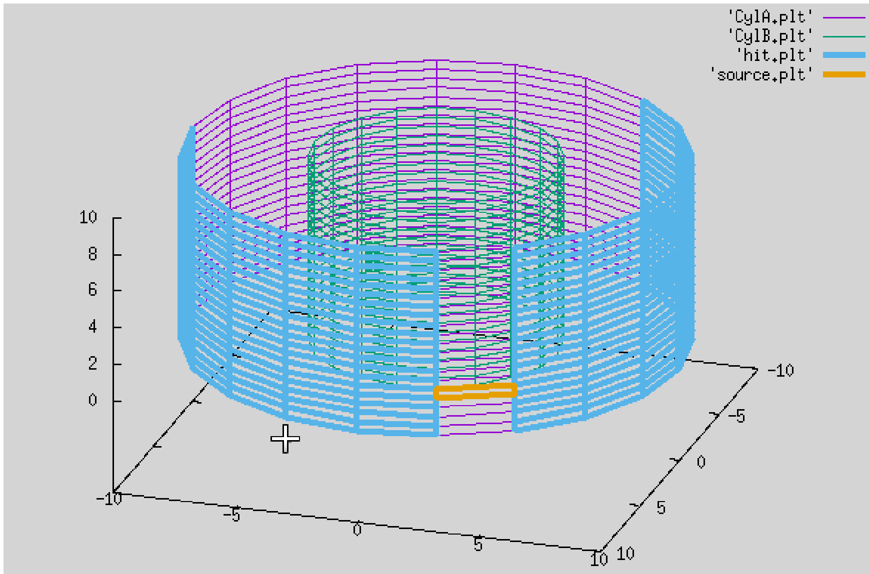

Figure 4: The Resulting Mesh

The resulting mesh has exactly the same hits and misses as the example. Both have 7 columns of misses at the back away from the source acting as the shadow of the internal cylinder, as well as all of the cells in the same axial column as misses. The missed column around the source is caused by the rays being completely parallel to the source, thus being unable to receive the rays due to none being completely flat. Compare that to the two columns surrounding this one, which are curved slightly following the cylinder's shape. This results in them receiving rays, no matter how little they get. This would not be the case if the surface was not curved in this manner.

The code specifically does not handle partial existence of rays either, but this is an idea inherent to the meshes in general. This is visible here more clearly than in previous labs, as the mesh forms shapes. Take this result for instance, which was generated by making each of the meshes 40 by 40 rather than 20 by 20:

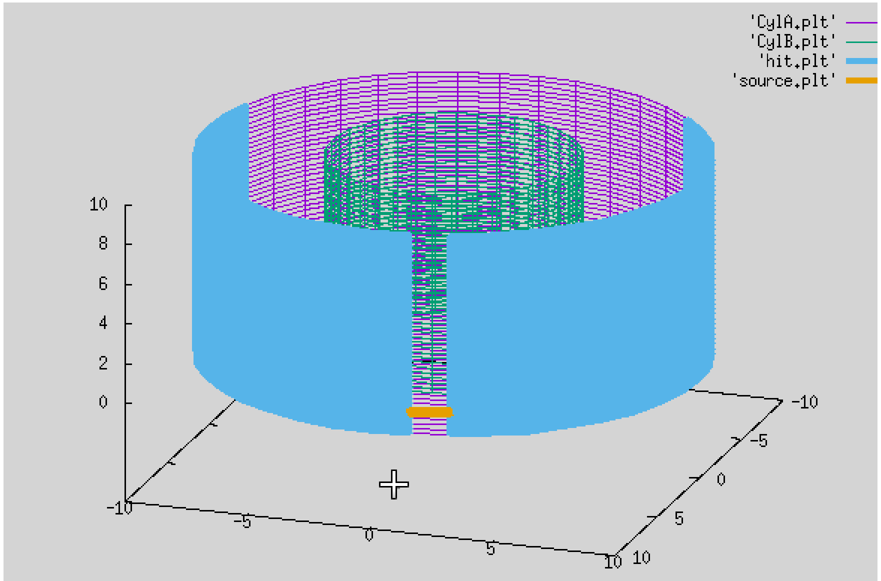

Figure 5: Double the Resulting Mesh

The results become more fine-grained by upping the number of cells rather than trying to calculate partial cells. It is made clear in this case, where the shadow is narrower than in the previous results. This is actually a more accurate shadow than the other one. It is made of only 13 faces rather than the 14 that would be expected from doubling the original 7 faces if the the shadow was an entirely accurate depiction. This not only proves that the result scales as expected, but shows off a fundamental concept of using larger numbers of basic faces that are easy to work with rather than trying to break down a larger shape into pieces that do not have the same fundamental mathematical basis. This is the same idea used in video game models, resulting in models and geometry that are millions of basic polygons rather than a few complex shapes.

3 Self Evaluation

This lab proved a bit easier due to the visual concepts involved. I had the majority of the base code done before lab even started. It became a case of one small error ruining everything, however. I could not get it to work throughout lab. My partner and I even compared code with the main portions all being the same conceptually. The mesh was also so weird in that state, as even the majority of the non-blocked cells came up as blocked. The problem ended up being that I put the summation variable in dot as an integer rather than a double. Three days was spent going through the code that was already essentially done because summ was an int and not a double. At least the experimentation showed off the blocking issue on the internal cylinder, as it was interesting to see how the approach itself could be wrong rather than the code.

4 Appendix: C++ Source Code

4.1 geom.h

```
//   ==
//   ||
//   ||
//   ||   Geometric Utilities
//   ||
//   ||
//   ==

//   ==
//   ||
//   ||   cross: Computes the cross product of v and w, returning the result in c
//   ||
//   ==

void cross( double *v , double *w , double *c )
{
  c[0] = (v[1]*w[2]) - (v[2]*w[1]);
  c[1] = (v[0]*w[2]) - (v[2]*w[0]);
  c[2] = (v[0]*w[1]) - (v[1]*w[0]);
  return;
}
//   ==
//   ||
//   ||   dot: Computes the dot product of v and w
//   ||
//   ==



double dot( double *v , double *w )
{
  double summ = 0;
  kLOOP summ = summ + v[k]*w[k];

  return summ;
}


//   ==
//   ||
//   ||   Intersection of a line with a plane.
//   ||
//   ||   n    = vector, normal to the plane
//   ||   p0   = a point in the plane
//   ||   L    = a vector pointing along the line
//   ||   L0   = a point on the line
//   ||
//   ||
//   ||   References
//   ||
//   ||   [1] https://en.wikipedia.org/wiki/Line%E2%80%93plane_intersection
```

```
//  ==

int Intersect_LinePlane( double *L0 , double *L , double *p0 , double *n , double *xyzInt )
{

  // Dot the line's vector with the plane's normal:

  double dotProduct = dot(L, n);

  if ( fabs(dotProduct) < 1.e-10 ) return -1;

  // Compute distance from point  L0 to the plane

  double p0L0[3]; kLOOP p0L0[k] = p0[k] - L0[k];

  double d = dot( p0L0 , n ) / dot(L,n);

  // Compute the intersection point

  xyzInt[0] = L0[0] + (d*L[0]);
  xyzInt[1] = L0[1] + (d*L[1]);
  xyzInt[2] = L0[2] + (d*L[2]);

  return 1;
}



//  ==
//  ||
//  ||  insideCorner: Given a corner of a quad in points A, B, and C, determine
//  ||                if point P is inside that corner.
//  ||
//  ||
//  ||        C        (1) Form vector pointing from A to B: vAB
//  ||         +       (2) Form vector pointing from C to A: vCA
//  ||         |       (3) Form vector pointing from A to P: vAP
//  ||         |       (4) P is inside the C-A-B corner if the cross products
//  ||         |           of these two vectors are aligned, i.e., their dot > 0:
//  ||         |           cA = vAB x vAP  and cB = vCA x vAP
//  ||         |
//  ||        A+------------------+ B
//  ||
//  ||
//  ||            * P
//  ==

bool insideCorner( double *P , double *A, double *B , double *C )
{
  double vAB[3];
  double vCA[3];
  double vAP[3];

  kLOOP
```

```
    {
      vAB[k] =  B[k] - A[k];
      vCA[k] =  A[k] - C[k];
      vAP[k] =  P[k] - A[k];
    }

  double cB[3] ; cross(vAB, vAP, cB );
  double cC[3] ; cross(vCA, vAP, cC );

  if ( dot(cB,cC) >= 0 )
    {
      return true;
    }

  return false;
}

//  ==
//  ||
//  ||  insideQuad: Given a quadrilateral as points in x,y,z arrays
//  ||              determine if point P is inside the quadrilateral.
//  ||
//  ||              It is inside the quadrilateral if it is inside
//  ||              each of its four corners.
//  ||
//  ==

int insideQuad( double *Pt , double *Q0 , double *Q1 , double *Q2 , double *Q3 )
{
  if ( insideCorner ( Pt , Q0 , Q1 , Q3 ) and insideCorner ( Pt , Q1 , Q0 , Q2 )
    and insideCorner ( Pt , Q2 , Q1 , Q3 ) and insideCorner ( Pt , Q3 , Q2 , Q0 ) )
    {
    return 1;
  }

  return 0;
}

//  ==
//  ||
//  ||  LineHitsFace: Returns 1 if the line specified by a point L0 and vector L
//  ||                intersect a face specified by four points Q0 - Q3.
//  ||
//  ==

int LineHitsFace( double *L0 , double *L , double *Q0 , double *Q1 , double *Q2 ,
    double *Q3 , double *n)
{
  // (1) Compute intersection point of line with the plane containing the face

  double xyzInt[3];
  int intersects = Intersect_LinePlane( L0 , L , Q0 , n , xyzInt );

  if ( ! intersects ) return 0;

```

```
  // (2) See if the intersection point (xyzInt) is inside the face's quadrilateral

  return insideQuad( xyzInt , Q0 , Q1, Q2, Q3 );


}
```

4.2 ovenWalls.cpp

```
//  ===============================================================================
//  ||                                                                           ||
//  ||              ovenWalls                                                    ||
//  ||              ------------------------------------------------------      ||
//  ||              T H E R M A L   R A D I A T I O N                            ||
//  ||                                                                           ||
//  ||              D E M O N S T R A T I O N   C O D E                          ||
//  ||              ------------------------------------------------------      ||
//  ||                                                                           ||
//  ||       Developed by: Scott R. Runnels, Ph.D.                               ||
//  ||                     University of Colorado Boulder                        ||
//  ||                                                                           ||
//  ||                For: CU Boulder CSCI 4576/5576 and associated labs         ||
//  ||                                                                           ||
//  ||           Copyright 2020 Scott Runnels                                    ||
//  ||                                                                           ||
//  ||                     Not for distribution or use outside of the            ||
//  ||                     this course.                                          ||
//  ||                                                                           ||
//  ===============================================================================

#include "ovenWalls.h"

class Cylinder
{
```

```
public:

  int node;
  int nCella;       // Number of cells in the axial direction
  int nCellc;       // Number of cells in the circular direction
  int nReala;       // Number of nodes in the axial direction
  int nRealc;       // Number of nodes in the circular direction
  int nField;       // Number of nodes
  double dtheta;    // Angular spacing of nodes, in degrees
  double dz;        // Axial spacing of nodes
  double radius;    // Cylinder radius
  double length;    // Cylinder length
  int    **face;    // Faces:  face[f][i] gives the nodes of face f
  int    nFaces;    // Number of faces
  double **coord;   // Vertices
  double **normal;  // Face normals
  double **center;  // Face centers

  Cylinder(){}

  Cylinder( int _nCella , int _nCellc , double _radius , double _length , int Out1In2 )
  {
    // Store user inputs

    nCella = _nCella;
    nCellc = _nCellc;

    nReala = nCella + 1 ;
    nRealc = nCellc     ;

    nField = nReala * nRealc;
    nFaces = nCella * nCellc;

    radius = _radius;
    length = _length;

    dtheta = 360.   / nCellc;
    dz     = length / nCella;

    // Form mesh

    coord  = Array2D_double(nField + 1 , 3);
    face   = Array2D_int   (nFaces + 1 , 5);
    normal = Array2D_double(nFaces + 1 , 3);
    center = Array2D_double(nFaces + 1 , 3);

    int faceCount = 0;

    for ( int j = 1 ; j <= nRealc ; ++j )
    for ( int i = 1 ; i <= nReala ; ++i )
      {
   int p = pid(i,j);

   double theta = dtheta * j;
```

```
    double zval   = dz      * (i-1);

    coord[p][0] = radius * cos(theta*3.1415/180.);
    coord[p][1] = radius * sin(theta*3.1415/180.);
    coord[p][2] = zval;

    // Form face, i.e., for this face number (which happens to be p)
    // collect the pids of the 4 nodes that comprise this face.

    int point[5];
    point[1] = p    ;
    point[2] = p + 1;
    point[3] = point[2] + nReala;
    point[4] = point[1] + nReala;

    // Correct point for when we have wrapped completely around the circle

    if ( j == nRealc )
      {
        point[3] = pid(i+1,1);
        point[4] = pid(i,1);
      }

    // Store the point values in this face, p (but not for the last point in an i-row

    if ( i < nReala)
      {
        ++faceCount;
        for ( int i = 1 ; i <= 4 ; ++i ) face[faceCount][i] = point[i];
      }
       }

     // Compute normals for each face

     for ( int f = 1 ; f <= nFaces ; ++f )
       {
    double xav = 0. , yav = 0. , zav = 0.;
    for ( int k = 1 ; k <= 4 ; ++k )
      {
        xav += coord[ face[f][k] ][0]  ;
        yav += coord[ face[f][k] ][1]  ;
        zav += coord[ face[f][k] ][2]  ;
      }

    xav /= 4.;
    yav /= 4.;
    zav /= 4.;

    double mag = sqrt((xav*xav + yav*yav));

    normal[f][0] = xav/mag;
    normal[f][1] = yav/mag;
    normal[f][2] = 0.;
    center[f][0] = xav;
```

```
   center[f][1] = yav;
   center[f][2] = zav;

   if ( Out1In2 == 2 ) for ( int k = 0 ; k < 3 ; ++k ) normal[f][k] *= -1.;

      }

  }



  #include "plotter.h"
  int pid(int i , int j ) { return ( i + (j-1) * nReala); }

};

#include "geom.h"


//  ==
//  ||
//  ||
//  ||  RayIsBlocked
//  ||
//  ||
//  ==

int RayIsBlocked( Cylinder &cSource       ,   // Mesh from which ray is emanating
        Cylinder &cBlocker     ,   // Mesh that may block that ray
        int       sourceFaceID ,   // Emanating face
        double   *Ray0         ,   // Starting point of ray
        double   *Ray          ,   // Vector pointing along ray
        VI &facesBlocking      )   // List of faces that are blocking
{
  double Q0[3], Q1[3], Q2[3], Q3[3];
  double blockerN[3];

  int blockerFaceID = 1;
  int blocked = 0;

  while ( ! blocked  && ++blockerFaceID <= cBlocker.nFaces )
    {

      // Vertices of potential blocker

      kLOOP Q0[k] = cBlocker.coord[ cBlocker.face[blockerFaceID][1] ][k];
      kLOOP Q1[k] = cBlocker.coord[ cBlocker.face[blockerFaceID][2] ][k];
      kLOOP Q2[k] = cBlocker.coord[ cBlocker.face[blockerFaceID][3] ][k];
      kLOOP Q3[k] = cBlocker.coord[ cBlocker.face[blockerFaceID][4] ][k];

      // Normal of potential blocker

      kLOOP blockerN[k] = cBlocker.normal[blockerFaceID][k];   // ...use cBlocker.normal vector

```

```
      //  Look for intersection

   blocked = LineHitsFace(Ray0, Ray, Q0, Q1, Q2, Q3, blockerN);

    }

  if ( blocked > 0 ) {
    facesBlocking.push_back(blockerFaceID);
  }
  return blocked;

}


//  ==
//  ||
//  ||
//  ||  Main Program
//  ||
//  ||
//  ==

int main(int argc, char *argv[])
{

  printf("\n");
  printf("\n");
  printf("Ray Tracing Demo Code\n");
  printf("\n");
  printf("\n");

  double length = 10.;
  double radius = 10.0;
  //             nCell   nCell                                      Facing In
  //             Axial   Angular     Radius         Length          Or Out
  //            ------  --------    ----------     ----------       -------
  Cylinder CylA(   20   ,  20      ,   radius     ,   length         ,   2      );
  Cylinder CylB(   20   ,  20      ,   radius/2.  ,   length         ,   1      );

  CylA.plot("CylA");
  CylB.plot("CylB");

  // Set up parameters for ray tracing

  VI facesBlocking;
  VI facesHit;

  // R A Y   T R A C E   F O R   S O U R C E   F A C E  1 to target face 100


  double Ray0[3], Ray[3];
  int sourceFaceID = 5;
  int targetFaceID = 51;

  CylA.plotFace("source",sourceFaceID);
```

```

  // (1) Point on source face

  kLOOP Ray0[k] = CylA.center[ sourceFaceID ] [k];

  // (2) Ray to target

  for ( targetFaceID = 1 ; targetFaceID <= CylA.nFaces ; ++targetFaceID )
    {
      kLOOP Ray [k] = CylA.center[ targetFaceID ] [k] - Ray0[k];

      // (3) Plot ray

      CylA.plotPointVec("ray", Ray0 , Ray , 1.000 );

      // (4) Find blockers of ray

      int blocked = 0;

      if ( dot(CylA.normal[sourceFaceID],CylA.normal[targetFaceID]) < .9999  )
   {
     blocked = RayIsBlocked( CylA , CylB , sourceFaceID , Ray0 , Ray ,
    facesBlocking );

     if ( blocked <= 0 ) facesHit.push_back(targetFaceID);
   }
    }

  // Plot hits and blockers

  CylA.plotFacesInList("hit",facesHit);
  CylB.plotFacesInList("blockers",facesBlocking);

  return 0;

}
```