Clipping a Mesh Against a Plane

David Eberly
Magic Software, Inc.
http://www.magic-software.com

Created: February 28, 2002

1 Introduction

Clipping is a standard operation in computer graphics that involves splitting an object by a plane and retaining that part of the object, if any, that is on one designated side of the plane. Typical applications include computing the portion of an object that is inside a viewing frustum for display purposes and constructing sets of intersection between two objects.

The plane is represented by the equation $\vec{N} \cdot \vec{X} - c = 0$ where $\vec{N}$ is a unit–length normal vector, c is a scalar, and $\vec{X}$ is any point on the plane. For any point $\vec{Y}$, we say that $\vec{Y}$ is on the *positive side of the plane* when $\vec{N} \cdot \vec{Y} - c > 0$, on the *negative side of the plane* when $\vec{N} \cdot \vec{Y} - c < 0$, or *on the plane* when $\vec{N} \cdot \vec{Y} - c = 0$. Similarly, $\vec{Y}$ is on the *nonnegative side of the plane* when $\vec{N} \cdot \vec{Y} - c \geq 0$ or on the *nonpositive side of the plane* when $\vec{N} \cdot \vec{Y} - c \leq 0$.

The simplest example is clipping a line segment by a plane as shown in Figure 1.

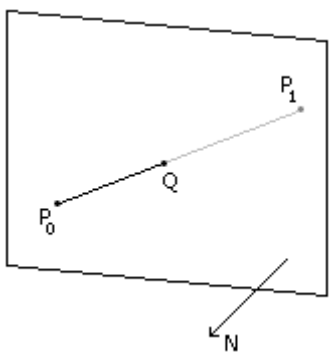

Figure 1. A line segment clipped by a plane.

The line segment has end points $\vec{P}_0$ and $\vec{P}_1$ and is denoted by $\langle \vec{P}_0, \vec{P}_1 \rangle$. The convention in this document is that clipping retains the portion of the object on the nonnegative side of the plane. The portion on the negative side is discarded. In Figure 1, the point $\vec{P}_0$ is on the positive side of the plane and the point $\vec{P}_1$ is on the negative side. The point of intersection of the line segment and the plane occurs at $\vec{Q}$. To be on the plane we need $\vec{N} \cdot \vec{Q} - c = 0$. To be on the line segment, we need $\vec{Q} = (1-t)\vec{P}_0 + t\vec{P}_1$ for some $t \in [0, 1]$.

Substituting this into the plane equation and solving for t leads to

$$t = \frac{\vec{N} \cdot \vec{P}_0}{\vec{N} \cdot (\vec{P}_0 - \vec{P}_1)} = \frac{\vec{N} \cdot \vec{P}_0 - c}{(\vec{N} \cdot \vec{P}_0 - c) - (\vec{N} \cdot \vec{P}_1 - c)} = \frac{d_0}{d_0 - d_1} \tag{1}$$

where $d_i = \vec{N} \cdot \vec{P}_i - c$ for $i = 0, 1$. The point of intersection is therefore $\vec{Q} = \vec{P}_0 + (d_0/(d_0 - d_1))(\vec{P}_1 - \vec{P}_0)$. The line segment that is retained is $\langle \vec{P}_0, \vec{Q} \rangle$. The line segment that is discarded is $\langle \vec{Q}, \vec{P}_1 \rangle$.

Generally, the clipping results of a line segment $\langle \vec{P}_0, \vec{P}_1 \rangle$ are as follows.

1. $d_0 \geq 0$ and $d_1 > 0$, or, $d_0 > 0$ and $d_1 \geq 0$: The entire segment is retained.
2. $d_0 \geq 0$ and $d_1 < 0$, or, $d_0 < 0$ and $d_1 \geq 0$: The entire segment is clipped.
3. $d_0 > 0$ and $d_1 < 0$, or, $d_0 < 0$ and $d_1 > 0$: The segment intersects the plane at a point $\vec{Q}$ that is interior to the segment. That is, the value of t in equation (1) satisfies $0 < t < 1$. If $d_0 > 0$, the retained segment is $\langle \vec{P}_0, \vec{Q} \rangle$. If $d_1 > 0$, the retained segment is $\langle \vec{Q}, \vec{P}_1 \rangle$.
4. $d_0 = 0$ and $d_1 = 0$. The segment lies entirely in the plane. Whether it is retained or discarded is dependent on the application. In the clipping algorithm of this document, such segments are discarded during the intermediate stage of the algorithm, but are re–added in the final stage.

The next example shows how to clip a triangle by a plane. Let $\vec{P}_0$, $\vec{P}_1$, and $\vec{P}_2$ be the triangle vertices. The clipping results are completely determined by the signs of $d_i = \vec{N} \cdot \vec{P}_i - c$ for $0 \leq i \leq 2$. Figure 2 shows two typical configurations.

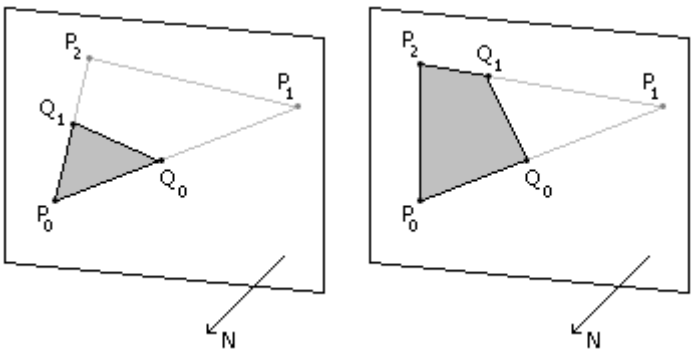

Figure 2. Triangles clipped by a plane.

In the left figure, the portion remaining after clipping is itself a triangle. In the right figure, the portion remaining is a convex quadrilateral. Some applications that manipulate only triangles as primitives will choose to split the quadrilateral into two triangles for any further processing. For example, if the triangle is being clipped relative to a view frustum defined by six planes, and the clipping operation for a single plane is designed to handle only triangles, then the quadrilateral must be decomposed into two triangles for clipping against the remaining planes.

In this document I show how to build a clipper that does not constrain itself to triangles, rather it manages the convex polygons that occur as a result of clipping. If the application that uses the clipper allows only

triangles, the resulting convex polygons from the clipping may be triangulated to satisfy those constraints. A key observation from Figure 2 is that you may compute the d_i first, then compare pairs of the d_i corresponding to edges of the triangle so that those edges may be clipped accordingly. This allows the edge processing to be a separate phase from the vertex processing. Less obvious is that the face processing, a face being a convex polygon, can be a separate phase from the edge processing. Thus, a clipper may be constructed that processes vertices first, edges second, and faces third.

2 Mesh Data Structures

The introduction motivated clipping of a line segment and of a triangle. The operations apply more generally to a mesh of convex polygons. The classical meshes use triangle faces, but the ideas presented here apply to faces that are convex polygons. The data structures used to represent the mesh form a *vertex–edge–face table*. The minimum information required in a mesh to support clipping is that each vertex knows its spatial location, each edge knows the two vertices that form its end points and knows all the faces that share the edge, and each face knows all the edges that bound it. Figure 3 shows the minimum data structures for the application mesh.

AVertex:	AEdge:	AFace:	AMesh:
Point point;	array(int) vertex;	set(int) edge;	array(AVertex) V;
	set(int) face;		array(AEdge) E;
			array(AFace) F;

Figure 3. The minimum information needed by an application mesh to support clipping.

The `AVertex` data structure can be slightly more general by allowing a level of indirection. The application can store an array of points and require the vertex to store an index into that array rather than storing the point itself. The array of integers in the `AEdge` data structure has length 2 since each edge has exactly two end points. The elements of this array are indices into the array of edges in the `Mesh` data structure. The set of faces in `AEdge` must be nonempty. The elements of this set are indices into the array of faces in the `AMesh` data structure. In many applications, a mesh is required to be *manifold* in the sense that an edge is shared by 1 or 2 faces. If the meshes are constrained to be manifold, then the set member of `AEdge` may be replaced by a length–2 array of integers. When an edge is shared by only one face, the first element of the array contains a nonnegative index into the face array of the mesh. The other element of the array stores an invalid index; by convention this can be -1. Finally, the set of edges in `AFace` must contain at least three elements since the smallest convex polygon for a face is a triangle. The elements of this set are indices into the array of edges in the `AMesh` data structure.

The clipping algorithm described here requires some additional information. It is convenient to allow the clipper to have data structures similar to those of the application but that allow storage for the additional items. Figure 4 shows the data structures.

CVertex:	CEdge:	CFace:	CMesh:
Point point;	array(int) vertex;	set(int) edge;	array(CVertex) V;
float distance;	set(int) face;	boolean visible;	array(CEdge) E;
int occurs;	boolean visible;		array(CFace) F;
boolean visible;			

Figure 4. The information needed by the clipper.

The `distance` member of `CVertex` is used to store the *signed distance* from the vertex to a plane. If the point is $\vec{P}$ and the plane has unit–length normal $\vec{N}$ and constant c, then the signed distance from the point to the plane is $d = \vec{N} \cdot \vec{P} - c$. The actual distance is $|d|$. Although I assumed earlier that the plane normal is a unit–length vector, that vector need not be unit length in the calculations. If it is not unit length, then d is no longer the signed distance. However, this is unimportant since all that is important for the clipping is the sign of d as mentioned earlier when enumerating all the possibilities for clipping a line segment.

The `occurs` member of `CVertex` provides support for computing the convex polygon resulting from the clipping. In Figure 2, the points of intersection with line segments and the plane are denoted $\vec{Q}_0$ and $\vec{Q}_1$. These points will be constructed by the clipper. The clipper will know immediately which subsegments are retained by considering the signed distances of the segment's original end points. The clipper also knows to retain edges on the positive side of the plane and to discard edges on the negative side. The intersection points of edges with the plane are added to the `CVertex` array of the `CMesh`. The vertex indices of the `CEdge` for a clipped edge are updated accordingly, and in place, so any `CFace` sharing the edge need not be updated when an intersection occurs. The `CFace` objects are notified about discarded edges. Each such edge is removed from the set of edges stored by the face. Because the edges are effectively processed in an arbitrary order, the clipper does not know during the edge processing phase that the segment $\langle \vec{Q}_0, \vec{Q}_1 \rangle$ belongs to the clipped face. The face processing phase has the responsibility for recognizing this. Using the right image in Figure 2 to illustrate, the original face is the convex polygon $\langle \vec{P}_0, \vec{P}_1, \vec{P}_2 \rangle$. I stress the fact that this is a *closed polyline*. After the edge processing, the face has been clipped to produce the *open polyline* $\langle \vec{Q}_1, \vec{P}_2, \vec{P}_0, \vec{Q}_0 \rangle$. The end points of this polyline must be located and an edge connecting them must be added to the `CMesh` and the appropriate index added to the edge set of the `CFace` object. The set of edges in the face is unordered, a fact that can make the search difficult. However, it is enough just to traverse the unordered set and let each vertex end point know how many times it has been visited. The points $\vec{P}_0$ and $\vec{P}_2$ are each seen 2 times; the points $\vec{Q}_0$ and $\vec{Q}_1$ are each seen 1 time. The `occurs` member is initially zero and is incremented each time the vertex is visited, thus leading to a final value of 1 or 2.

The `visible` member of `CVertex` is used to indicate which side of the plane the vertex occurs. The value is `true` if the point is on the nonnegative side of the plane, `false` if the point is on the negative side. This information is not necessary when clipping against a single plane. However, if the mesh is to be clipped against a set of planes, the visibility flag allows you to avoid processing vertices that were discarded by earlier planes.

The `visible` member of `CEdge` is also used to indicate which side of the plane the edge occurs. The value is `true` if one end point is on the positive side of the plane, the other end point on the nonnegative side. The value is `false`, otherwise, including the case when the edge is entirely on the plane. A discarded planar edge will be restored during the face processing phase when the polyline of the face is closed to form a convex polygon. The edge visibility member is necessary even when clipping only against one plane.

The `visible` member of `CFace` is used to indicate which side of the plane the face occurs. The value is `true` if at least one edge of the face is visible, `false` otherwise.

3 Initializing the Clipper Mesh

The clipper mesh data structures must be initialized from the application mesh. The pseudocode for this is shown below.

```
CMesh (AMesh appMesh)
{
    V.SetSize(appMesh.V.length);
    for (i = 0; i < V.length; i)
        V[i].point = appMesh.V[i].point;

    E.SetSize(appMesh.E.length);
    for (i = 0; i < E.length; i++)
    {
        E[i].vertex[0] = appMesh.E[i].vertex[0];
        E[i].vertex[1] = appMesh.E[i].vertex[1];
        for ( each j in appMesh.E[i].face )
            E[i].face.Insert(j);
    }

    F.SetSize(appMesh.F.length);
    for (i; i < F.length; i++)
    {
        for ( each j in appMesh.F[i].edge )
            F[i].edge.Insert(j);
    }
}
```

The vertices, edges, and faces of the clipper mesh are initialized as shown.

```
CVertex ()
{
    distance = 0;
    visible = true;
}

CEdge ()
{
    visible = true;
}

CFace ()
{
```

```
    visible = true;
}
```

The partial constructions are all that is required to start the processes. Other members are appropriately set when needed.

4 Clipping by a Plane

The basic structure of the clipping is as shown.

```
int CMesh.Clip (Plane clipplane)
{
    process vertices;  // compute distances to plane, set visibility flags

    if ( all vertices are on nonnegative side of plane )
        return +1;  // no clipping occurs
    if ( all vertices are on nonpositive side of plane )
        return -1;  // everything is clipped

    // mesh straddles the plane; some on positive side, some on negative side
    process edges;
    process faces;
    return 0;
}
```

Each of the processing phases is discussed below.

4.1 Vertex Processing

The vertex processing is given by the following simple code. The epsilon value is a small positive number and is used to avoid large amounts of edge clipping when vertices are nearly on the plane.

```
// compute signed distances from vertices to plane
int positive = 0, negative = 0;
for (int i = 0; i < V.length; i++)
{
    if ( V[i].visible )
    {
        V[i].distance = Dot(clipplane.N,V[i].point) - clipplane.c;
        if ( V[i].distance >= epsilon )
        {
            positive++;
        }
        else if ( V[i].distance <= -epsilon )
```

```
        {
            negative++;
            V[i].visible = false;
        }
        else
        {
            // point on the plane within floating point tolerance
            V[i].distance = 0;
        }
    }
}

if ( negative == 0 )
{
    // all vertices on nonnegative side, no clipping
    return +1;
}

if ( positive == 0 )
{
    // all vertices on nonpositive side, everything clipped
    return -1;
}
```

4.2 Edge Processing

The edge processing is also relatively simple and just compares the signed distances of the vertex end points to zero and takes the appropriate action with the edge.

```
for (int i = 0; i < E.length; i++)
{
    if ( E[i].visible )
    {
        float d0 = V[E[i].vertex[0]], d1 = V[E[i].vertex[1]];

        if ( d0 <= 0 && d1 <= 0 )
        {
            // edge is culled, remove edge from faces sharing it
            for ( each j in E[i].face )
            {
                F[j].edge.Remove(i);
                if ( F[j].edge.empty() )
                    F[j].visible = false;
            }

            E[i].visible = false;
            continue;
```

```
        }

        if ( d0 >= 0 && d1 >= 0 )
        {
            // edge is on nonnegative side, faces retain the edge
            continue;
        }

        // The edge is split by the plane.  Compute the point of intersection.
        // If the old edge is <V0,V1> and I is the intersection point, the new
        // edge is <V0,I> when d0 > 0 or <I,V1> when d1 > 0.

        float t = d0/(d0-d1);
        Point intersect = (1-t)*V[E[i].vertex[0]]+t*V[E[i].vertex[1]];
        index = V.length;
        V.Append(intersect);

        if ( d0 > 0 )
            E[i].vertex[1] = index;
        else
            E[i].vertex[0] = index;
    }
}
```

4.3 Face Processing

The face processing is the most complicated of the phases to implement.

```
// process the faces
for (int i = 0; i < F.length; i++)
{
    if ( F[i].visible )
    {
        // The edge is culled.  If the edge is exactly on the clip
        // plane, it is possible that a visible triangle shares it.
        // The edge will be re-added during the face loop.

        for ( each j in F[i].edge )
        {
            V[E[j].vertex[0]].occurs = 0;
            V[E[j].vertex[1]].occurs = 0;
        }

        int start, final;
        if ( GetOpenPolyline(F[i],start,final) )
        {
            // polyline is open, close it
```

```
            CEdge closeEdge;
            int eindex = E.length;
            E.Append(closeEdge);

            closeEdge.vertex[0] = start;
            closeEdge.vertex[1] = final;
            closeEdge.face.Insert(i);
            F[i].edge.Insert(eindex);
        }
    }
}
```

The detection of an open polyline and the subsequent determination of the vertex indices of the end points is given below.

```
boolean CMesh.GetOpenPolyline (CFace F, int& start, int& final)
{
    // Count the number of occurrences of each vertex in the polyline.  The
    // resulting 'occurs' values must be 1 or 2.
    for ( each j in F.edge )
    {
        V[E[j].vertex[0]].occurs++;
        V[E[j].vertex[1]].occurs++;
    }

    // determine if the polyline is open
    start = -1;
    final = -1;
    for ( each j in F.edge )
    {
        int i0 = E[j].vertex[0], i1 = E[j].vertex[1];
        if ( V[i0].occurs == 1 )
        {
            if ( start == -1 ) start = i0; else if ( final == -1 ) final = i0;
        }
        if ( V[i1].occurs == 1 )
        {
            if ( start == -1 ) start = i1; else if ( final == -1 ) final = i1;
        }
    }
    return start != -1;
}
```

5 Conversion Back to an Application Mesh

At first glance, the conversion from a clipper mesh back to an application mesh might appear straightforward. In fact you might think that it is similar to the initialization of the clipper mesh. This is not the case. First, the clipper has stored all the vertices, edges, and faces throughout the process of clipping against a set of planes. Many of these components are no longer visible because they have been clipped away. We do not want these components stored in the application mesh. Second, in many cases the application mesh might also store additional topological information about the relationships between the vertices, edges, and faces. The most common constraint on the application mesh is one of ordering of the face vertices and, by implication, the face edges. The clipper mesh does not maintain these additional relationships because they are not necessary for the clipping algorithm.

In this section I present an algorithm that handles the visibility issues by creating an array of visible vertices and an array of visible faces whose vertices are consistently ordered (all faces clockwise or all faces counterclockwise). The assumption is that a `AMesh` object has a construction mechanism to build all the mesh components given the array of points and the array of consistently ordered faces. In support of this, the `AFace` and `CFace` data structures are each given a new member to store the plane of the face. The normal vectors of the planes for the original application mesh are what are used to represent the ordering of the face vertices. The new member type is `Plane` and the member name is `plane`. The `CMesh` initialization code must be slightly modified so that its faces copy the application mesh face planes.

```
CMesh ()
{
    // previous code the same as before...

    F.SetSize(appMesh.F.length);
    for (i; i < F.length; i++)
    {
        F[i].plane = appMesh.F[i].plane;
        for ( each j in appMesh.F[i].edge )
            F[i].edge.Insert(j);
    }
}
```

The conversion code is as follows.

```
AMesh CMesh.Convert ()
{
    // create a lookup table for the visible vertices
    array(Point) point(V.length);
    array(int) vmap(V.length);
    vmap.SetAllElementsTo(-1);

    // copy the visible vertices into the table
    for (i = 0; i < V.length; i++)
    {
        if ( V[i].visible )
        {
```

```
            vmap[i] = point.length;
            point.Append(V[i].point);
        }
    }

    // Order the vertices for all the faces.  The output array has a
    // sequence of subarrays, each subarray having first element storing
    // the number of vertices in the face, the remaining elements storing
    // the vertex indices for that face in the correct order.  The indices
    // are relative to the V[] array.
    array(int) faces = GetOrderedFaces();

    // map the vertex indices to those of the new table
    for (i = 0; i < faces.length; /**/)
    {
        int numIndices = faces[i];
        i++;
        for (j = 0; j < numIndices; j++)
        {
            faces[i] = vmap[faces[i]];
            i++;
        }
    }

    return AMesh(point,faces);
}
```

The face ordering is accomplished by the following code.

```
array(int) CMesh.GetOrderedFaces ()
{
    array(int) faces;
    for (i = 0; i < F.length; i++)
    {
        if ( F[i].visible )
        {
            // Get the ordered vertices of the face.  The first and last
            // element of the array are the same since the polyline is
            // closed.
            array(int) vertices = GetOrderedVertices(F[i]);
            faces.Append(vertices.length-1);

            // The convention is that the vertices should be counterclockwise
            // ordered when viewed from the negative side of the plane of the
            // face.  If you need the opposite convention, switch the
            // inequality in the if-else statement.
            if ( Dot(F[i].plane.N,GetNormal(vertices)) > 0 )
            {
```

```
                // clockwise, need to swap
                for (j = vertices.length-2; j >= 0; j--)
                    faces.Append(vertices[j]);
            }
            else
            {
                // counterclockwise
                for (j = 0; j <= vertices.length-2; j++)
                    faces.Append(vertices[j]);
            }
        }
    }
    return faces;
}
```

The vertex ordering requires some type of sorting. The algorithm here uses a bubble–like sort, a reasonable choice if the faces have a small number of edges. If the number of edges is quite large in an application, you should consider adding a faster sort (probably switching between bubble sort and the faster sort based on number of edges).

```
void CMesh.GetOrderedVertices (CFace F)
{
    // copy edge indices into contiguous memory for sorting
    array(int) edges(F.edge.size);
    i = 0;
    for ( each j in F.edge )
        edges[i++] = j;

    // bubble sort to arrange edges in contiguous order
    for (i0 = 0, i1 = 1, choice = 1; i1 < edges.length-1; i0 = i1, i1++)
    {
        int current = E[edges[i0]].vertex[choice];
        for (j = i1; j < edges.length; j++)
        {
            Edge& rkETemp = E[edges[j]];
            if ( E[edges[j]].vertex[0] == current )
            {
                Swap(edges[i1],edges[j]);
                choice = 1;
                break;
            }
            if ( E[edges[j]].vertex[1] == current )
            {
                Swap(edges[i1],edges[j]);
                choice = 0;
                break;
            }
        }
```

```
    }

    array(int) vertices(edges.length+1);

    // add the first two vertices
    vertices.Append(E[edges[0]].vertex[0]);
    vertices.Append(E[edges[0]].vertex[1]);

    // add the remaining vertices
    for (i = 1; i < edge.length; i++)
    {
        Edge& rkE = m_akEdge[aiEOrdered[i]];
        if ( E[edges[i]].vertex[0] == vertices[i] )
            vertices[i+1] = E[edges[i]].vertex[1];
        else
            vertices[i+1] = E[edges[i]].vertex[0];
    }

    return vertices;
}
```

Finally, a numerically robust method for computing the normal from an ordered set of vertices is given below. The function relies on the fact that the first and last indices of the input array are the same.

```
Point CMesh.GetNormal (array(int) vertices)
{
    Point normal(0,0,0);
    for (i = 0; i <= vertices.length-2; i++)
        normal += Cross(V[vertices[i]],V[vertices[i+1]]);
    normal.Normalize();  // make the normal unit length
    return normal;
}
```

6 Support for Closing a Clipped Mesh

For purposes of display, once the mesh is clipped and converted back to the form the application requires, no other work must be done other than sending the faces through the rendering system. For purposes of intersection of closed meshes and planes (or other closed meshes), it is necessary to close the clipped mesh by adding faces whose boundaries are the polygons of intersection with the clipping plane. The face processing pseudocode listed earlier needs to be modified to support this.

6.1 Closing a Clipped Convex Polyhedron

When a convex polyhedron is split by a plane, the intersection of the polyhedron and the plane is a convex polygon. The portion of the original convex polyhedron on the nonnegative side of the plane has a hole in it,

but the convex polygon of intersection bounds a face that is used to plug the hole. Only a minor modification is required to the face processing code.

```
// The mesh straddles the plane.  A new convex polygonal face will be
// generated.  Add it now and insert edges when they are visited.
CFace closeFace;
closeFace.plane = clipplane;
int findex = F.length;
F.Append(closeFace);

for (int i = 0; i < F.length; i++)
{
    if ( F[i].visible )
    {
        // previous code the same as before...

        int start, final;
        if ( GetOpenPolyline(F[i],start,final) )
        {
            // polyline is open, close it
            CEdge closeEdge;
            int eindex = E.length;
            E.Append(closeEdge);

            closeEdge.vertex[0] = start;
            closeEdge.vertex[1] = final;
            closeEdge.face.Insert(i);
            F[i].edge.Insert(eindex);

            // the new lines of code
            closeEdge.face.Insert(findex);
            closeFace.edge.Insert(eindex);
        }
    }
}
```

Assuming the modifications have been added to the face processing code, the pseudocode for clipping a convex polyhedron (an `AMesh` object) against a plane is

```
ConvexPolyhedron ConvexPolyhedron.Clip (Plane plane)
{
    CMesh clipper(self);

    int side = kClipper.Clip(rkPlane);

    if ( side == +1 )
    {
        // polyhedron on nonnegative side of plane, nothing clipped
```

```
        return self;
    }

    if ( side == -1 )
    {
        // polyhedron on nonpositive side of plane, clip all
        return null;
    }

    // polyhedron split by plane, convert clip mesh to polyhedron
    return clipper.Convert();
}
```

The pseudocode for intersection of two convex polyhedra is

```
ConvexPolyhedron ConvexPolyhedron.Intersection (ConvexPolyhedron poly)
{
    CMesh clipper(self);

    for (i = 0; i < poly.F.length; i++)
    {
        if ( kClipper.Clip(poly.F[i].plane) == -1 )
        {
            // polyhedra do not intersect
            return null;
        }
    }

    // polyhedra do intersect, convert clip mesh to polyhedron
    return clipper.Convert();
}
```

6.2 Closing a Clipped Closed Polyhedron

If the polyhedron is a closed mesh, but not convex, the intersection of the polyhedron and plane results in a disjoint collection of polygons and/or line segments. The original face processing code also requires modification for this case.

```
// The mesh straddles the plane.  Just store any edges that are used to close
// the non-planar convex polygon faces and process them after this loop.
array(CEdge) edges;

for (i = 0; i < F.length; i++)
{
    if ( F[i].visible )
    {
        // previous code the same as before...
```

```
            int start, final;
            if ( GetOpenPolyline(F[i],start,final) )
            {
                // polyline is open, close it
                CEdge closeEdge;
                int eindex = E.length;
                E.Append(closeEdge);

                closeEdge.vertex[0] = start;
                closeEdge.vertex[1] = final;
                closeEdge.face.Insert(i);
                F[i].edge.Insert(eindex);

                // the new line of code
                edges.Append(eindex);
            }
        }
    }

    // extract the relevant polygon faces from the edge set
    array(CFace) faces = ExtractFaces(edges);

    int findex = F.length;
    for (i = 0; i < faces.length; i++, findex++)
    {
        // connect the face to each edge of the extracted face
        for ( each j in faces[i].edge )
            E[j].face.Insert(findex);

        // add the face to the clip mesh
        F.Append(faces[i]);
    }
```

The function `ExtractFaces` processed the edges on the clip plane to find all the connected components. Isolated line segments are ignored since they represent the polyhedron just touching the clip plane along that edge without crossing through the plane. Care must be taken to handle polygons such as a figure-eight where at least one vertex is shared by more than two edges. Such structures need to be further decomposed into simple polygons. An implementation of `ExtractFaces` will have the flavor of both functions `GetOpenPolyline` and `GetOrderedVertices`. Essentially you build an abstract undirected graph for the edges and find all connected components of that graph.