4.2.3 Merging Two Capsules

Source Code

Library
Containment

Filename
ContCapsule

Two capsules may be merged into a single capsule with the following algorithm. If one capsule contains the other, just use the containing capsule. Otherwise, let the capsules have radii $r_i > 0$, end points $\vec{P}_i$, and directions $\vec{D}_i$ for $i = 0, 1$. The center points of the line segments are $\vec{C}_i = \vec{P}_i + \vec{D}_i/2$. Unit-length directions are $\vec{U}_i = \vec{D}/|\vec{D}|$.

The line L containing the final capsule axis is computed below. The origin of the line is the average of the centers of the original capsules, $\vec{C} = (\vec{C}_0 + \vec{C}_1)/2$. The direction vector of the line is obtained by averaging the unit direction vectors of the input capsules. Before doing so, the condition $\vec{U}_0 \cdot \vec{U}_1 \geq 0$ should be satisfied. If it is not, replace $\vec{U}_1$ by $-\vec{U}_1$. The direction vector for the line is $\vec{U} = (\vec{U}_0 + \vec{U}_1)/|\vec{U}_0 + \vec{U}_1|$.

The final capsule radius r must be chosen sufficiently large so that the final capsule contains the original capsules. It is enough to consider the spherical ends of the original capsules. The final radius is

$$r = \max\{\text{dist}(\vec{P}_0, L) + r_0, \text{dist}(\vec{P}_0 + \vec{D}_0) + r_0, \text{dist}(\vec{P}_1, L) + r_1, \text{dist}(\vec{P}_1 + \vec{D}_1, L) + r_1\}.$$

Observe that $r \geq r_i$ for $i = 0, 1$.

The final capsule direction $\vec{D}$ will be a scalar multiple of line direction $\vec{U}$. Let $\vec{E}_0$ and $\vec{E}_1$ be the end points for the final capsule, so $\vec{P} = \vec{E}_0$ and $\vec{D} = \vec{E}_1 - \vec{E}_0$. The end points must be chosen so that the final capsule contains the end spheres of the original capsules. Let the projections of $\vec{P}_0$, $\vec{P}_0 + \vec{D}_0$, $\vec{P}_1$, and $\vec{P}_1 + \vec{D}_1$ onto $\vec{C} + t\vec{U}$ have parameters τ_0, τ_1, τ_2, and τ_3, respectively. Let the corresponding capsule radii be denoted ρ_i for $0 \leq i \leq 3$. Let $\vec{E}_j = \vec{C} + T_j\vec{D}$ for $j = 0, 1$. The T_j are determined by "supporting" spheres that are selected from the end point spheres of the original capsules. If $\vec{Q}$ is the center of such a supporting sphere of radius ρ for end point $\vec{E}_1$, then T_1 is the smallest root of the equation $|\vec{C} + T\vec{U} - \vec{Q}| + \rho = r$. Since $r \geq \rho$, the equation can be written as a quadratic

$$T^2 + 2\vec{U} \cdot (\vec{C} - \vec{Q})T + |\vec{C} - \vec{Q}|^2 - (r - \rho)^2 = 0.$$

This equation must have only real-valued solutions. Similarly, if the $\vec{Q}$ is the center of the supporting sphere corresponding to end point $\vec{E}_0$, then T_0 is the largest root of the quadratic. The quadratics are solved for all four end points of the original capsules, and the appropriate minimum and maximum roots are chosen for the final T_0 and T_1.

4.2.4 Merging Two Lozenges

Two lozenges may be merged into a single lozenge that contains them with the following algorithm. Let the lozenges have radii $r_i > 0$, origins $\vec{P}_i$, and edges $\vec{E}_{ji}$ for $i = 0, 1$ and $j = 0, 1$. The center points of the rectangles of the lozenge are $\vec{C}_i = \vec{P}_i + (\vec{E}_{0i} + \vec{E}_{1i})/2$. Unit-length edge vectors are $\vec{U}_{ji} = \vec{E}_{ji}/|\vec{E}_{ji}|$. Unit-length normal vectors are $\vec{N}_i = \vec{U}_{0i} \times \vec{U}_{1i}$.

The center point of the final lozenge is the average of the centers of the original lozenges, $\vec{C} = (\vec{C}_0 + \vec{C}_1)/2$.

The edge vectors are obtained by averaging the coordinate frames of the original lozenges using a quaternion representation. Let q_i be the unit quaternion that represents the rotation matrix $[\vec{U}_{0i}\ \vec{U}_{1i}\ \vec{N}_i]$. If $q_0 \cdot q_1 < 0$, replace q_1 by $-q_1$. The final lozenge coordinate frame is extracted from the rotation matrix $[\vec{U}_0\ \vec{U}_1\ \vec{N}]$ corresponding to the unit quaternion $q = (q_0 + q_1)/|q_0 + q_1|$.

The problem now is to compute r sufficiently large so that the final lozenge contains the original lozenges. Project the original lozenges onto the line containing $\vec{P}$ and having direction $\vec{N}$. Each projection has extreme points determined by the corners of the projected rectangle and the radius of the original lozenge. The radius r of the final lozenge is selected to be the length of the smallest interval that contains all the extreme points of projection. Observe that $r \geq r_i$ is necessary.

Project the rectangle vertices of original lozenges onto the plane containing $\vec{P}$ and having normal $\vec{N}$. Compute the oriented bounding rectangle in that plane where the axes correspond to $\vec{U}_i$. This rectangle is associated with the final lozenge and produces the edges $\vec{E}_i = L_i\vec{U}_i$ for some scalars $L_i > 0$. The origin point for the final lozenge is $\vec{P} = \vec{C} - \vec{E}_0/2 - \vec{E}_1/2$.

Source Code

Library
Containment

Filename
ContLozenge

4.2.5 Merging Two Cylinders

To keep the merging algorithm simple, the original two cylinders are treated as capsules: their representations are converted to those for capsules, end points are $\vec{P}_i$, directions are $\vec{D}_i$, and radii are r_i. The capsule merging algorithm is applied to obtain the cylinder radius r. Rather than fitting a capsule to the points $\vec{P}_i \pm r_i\vec{U}$ and $\vec{P}_i + \vec{D}_i \pm r_i\vec{U}$, the points are projected onto the line $\vec{P} + t\vec{D}$, where $\vec{P}$ is suitably chosen from one of the fitting algorithms. The smallest interval containing the projected points determines cylinder height h.

Source Code

Library
Containment

Filename
ContCylinder

4.2.6 Merging Two Ellipsoids

Computing a bounding ellipsoid for two other ellipsoids is done in a way similar to that of oriented boxes. The ellipsoid centers are averaged, and the quaternions representing the ellipsoid axes are averaged and then the average is normalized. The original ellipsoids are projected onto the newly constructed axes. On each axis, the smallest interval of the form $[-\sigma, \sigma]$ is computed to contain the intervals of projection. The σ-values determine the minor axis lengths for the final ellipsoid.

Source Code

Library
Containment

Filename
ContEllipsoid

4.2.7 Algorithm for Scene Graph Updating

The pseudocode for updating the spatial information in a scene graph is given below. Three abstract classifications are used: `Spatial`, `Geometry`, and `Node`. In an object-

Source Code

Library
Engine

Filename
Spatial
Geometry
Node

oriented implementation, the last two classes are both derived from `Spatial`. The `Spatial` class manages a link to a parent, local transforms, and a world transform. It represents leaf nodes in a tree. The `Node` class manages links to children. It represents internal nodes in the tree. The `Geometry` class represents leaf nodes that contain geometric data. It manages a model bounding volume.

The entry point into the update system for geometric state (GS) is

```
void Spatial::UpdateGS (float time, bool initiator)
{
    UpdateWorldData(time);
    UpdateWorldBound();
    if ( initiator )
        PropagateBoundToRoot();
}
```

The input parameter to the call is set to `true` by the node at which the update is initiated. This allows the calling node to propagate the world bounding volume update to the root of the scene graph.

The function `UpdateWorldData` is virtual and controls the downward pass that computes world transforms and updates time-varying quantities:

```
virtual void Spatial::UpdateWorldData (float time)
{
    // update dynamically changing render state
    for each render state controller rcontroller do
        rcontroller.Update(time);

    // update local transforms if managed by controllers
    for each transform controller tcontroller do
        tcontroller.Update(time);

    // Compute product of parent's world transform with this object's
    // local transform.  If no parent exists, the child's world
    // transform is just its local transform.

    if ( world transform not computed by a transform controller )
    {
        if ( parent exists )
        {
            worldScale = parent.worldScale*localScale;
            worldRotate = parent.worldRotate*localRotate;
            worldTranslate = parent.worldTranslate +
                parent.worldScale*(parent.worldRotate*localTranslate);
        }
```

```
        else
        {
            // node is the root of the scene graph
            worldScale = localScale;
            worldRotate = localRotate;
            worldTranslate = localTranslate;
        }
    }
}
```

The function `UpdateWorldBound` is also virtual and controls the upward pass and allows each node object to update its world bounding volume. Base class `Spatial` has no knowledge of geometric data and in particular does not manage a model bounding sphere, so the function is pure virtual and must be implemented both by `Geometry`, which knows how to transform a model bounding volume to a world bounding volume, and by `Node`, which knows how to merge world bounding volumes of its children.

Finally, the propagation of world bounding volumes is not virtual and is a simple recursive call:

```
void Spatial::PropagateBoundToRoot ()
{
    if ( parent exists )
    {
        parent.UpdateWorldBound();
        parent.PropagateBoundToRoot();
    }
}
```

The derived classes override the virtual functions. Class `Geometry` has nothing more to say about updating world data, but it must update the world bound,

```
virtual void Geometry::UpdateWorldBound ()
{
    worldBound = modelBound.TransformBy(worldRotate,
      worldTranslate,worldScale);
}
```

The model bound is assumed to be correct. If model data is changed, the application is required to update the model bound.

Class `Node` updates are as shown:

```
virtual void Node::UpdateWorldData (float time)
{
    Spatial::UpdateWorldData(time);
```

```
    for each child do
        child.UpdateGS(false); // child not initiator of
                               // original UpdateGS call
}

virtual void Node::UpdateWorldBound ()
{
    worldBound = firstChild.GetWorldBound();
    for each additional child do
        worldBound = Merge(worldBound,child.worldBound);
}
```

The downward pass is controlled by `UpdateWorldData`. The node first updates its world transforms by a call to the base class update of world transforms. The children of the node are each given a chance to update themselves, thus yielding a recursive chain of calls involving `UpdateGS` and `UpdateWorldData`. The update of world bounds is done incrementally. The world bound is set to the first child's world bound. As each remaining child is visited, the current world bound and the child world bound are merged into a single bound that contains both. Although this approach usually does not produce the tightest bound, it is much faster than methods that do attempt the tightest bound. For example, if bounding spheres are used, it is possible to compute the parent world bound as the minimum volume sphere containing any geometric data of the descendants. Such a computation is expensive and will severely affect the frame rate of the application. The trade-off is to obtain a reasonable world bounding volume for the parent that is inexpensive to compute.

Updating the set of current renderer states at the leaf nodes is also a recursive system just as `UpdateGS` is. Class `Geometry` maintains a set of such states; call that member `stateSet`. Each state can be attached to or detached from an object of this class. A state object itself has information that can be modified at run time. If the information is changed, then an update must occur starting at that node. The global renderer state set is maintained by the renderer, so any changes to renderer state by the objects must be communicated to the renderer. Class `Spatial` provides the virtual function foundation for the renderer state (RS) update:

```
void Spatial::UpdateRS (RenderState parentState)
{
    // update render states
    if ( parentState exists )
    {
        // parentState must remain intact to restore state after
        // recursion
        currentState = parentState;
        modify currentState with thisState;
    }
```

```
    else
    {
        // this object is initiator of UpdateRS, use default
        // renderer states
        currentState = defaultRenderState;
        PropagateStateFromRoot(currentState);
    }

    UpdateRenderState(currentState);
}
```

The initial call to `UpdateRS` is typically applied to a node in the tree that is not the root node. Any renderer state from predecessors of the initiating node must be accumulated before the downward recursive pass. The function `PropagateStateFromRoot` does this work:

```
    void Spatial::PropagateStateFromRoot (RenderState
                                          currentState)
    {
        // traverse to root to allow downward state propagation
        if ( parent exists )
            parent.PropagateStateFromRoot(currentState);

        // update parent state by current state
        modify currentState with thisState;
}
```

The call `UpdateRenderState` is pure virtual. Class `Geometry` implements this to update its renderer state at leaf nodes. Class `Node` implements this to perform the recursive traversal of the call on its children.

```
    void Geometry::UpdateRenderState (RenderState currentState)
    {
        modify thisState with currentState;
    }

    void Node::UpdateRenderState (RenderState currentState)
    {
        for each child do
            child.UpdateRS(currentState);
    }
}
```

Notice that `UpdateRS` and `UpdateRenderState` form a recursive chain just as `UpdateGS` and `UpdateWorldData` form a recursive chain.

4.3 Rendering a Scene Graph

The renderer manages a camera whose job it is to define the *view frustum,* the portion of the world to be viewed. The process of rendering the scene graph in the frustum at a given instant is typically referred to as the *camera click*. This process involves a traversal of the scene graph, and the graph is assumed to be current (as established by the necessary `UpdateGS()` and `UpdateRS()` calls at the relevant nodes).

Scene graph traversal includes object level culling as described earlier. If the world bounding volume for a node is outside the view frustum, then the subtree rooted at that node need not be traversed. If a subtree is not culled, then the traversal is recursive. The renderer states are collected during traversal until a leaf node of the scene graph is reached. At this point the renderer has all the state information it requires to be able to properly draw the geometry represented by the leaf node. The leaf node has the responsibility of providing the renderer with its geometric data such as vertices, triangle connectivity information, triangle normals (for back face culling), and surface attributes including vertex normals, colors, and texture coordinates.

Before the actual rendering of the leaf node object, it is useful to allow the object to perform any preparations that are necessary for proper display. For example, culling is based on world bounding volumes. The classes derived from `Geometry` have the liberty of keeping current the world bounding sphere via the `UpdateWorldBound` call. If an object is to be culled, then computing any expensive world data in the call to `UpdateWorldData` is wasteful. Instead, the `Geometry` classes could provide a Boolean flag indicating whether or not the world data is current. The call to `UpdateWorldData` updates world transforms, but additionally sets only the Boolean flag indicating the world data is not current. A prerendering function called *after* it is determined that an object is not to be culled can test the Boolean flag, find out the world data is not current, make the data current, then set the flag to indicate the data is current.

Another use of a prerendering function involves dynamic tessellation of an object. Chapter 10 discusses objects represented by a triangular mesh whose triangles are increased or reduced based on a continuous level-of-detail algorithm involving a preprocessed set of incremental mesh changes. The prerendering function can select the appropriate level of detail based on the current camera and view frustum. Chapter 8 discusses objects represented by curved surfaces. The prerendering function can dynamically tessellate the surfaces to the appropriate level of detail.

The complement of a prerendering function is a postrendering function that gives the object a chance to do any cleanup associated with prerendering and actual rendering.

4.3.1 Culling by Spheres

The test for intersection of bounding volume with view frustum is performed in world space since the world bounding information is kept current by the object and the world view frustum information is kept current by the camera. Let the world

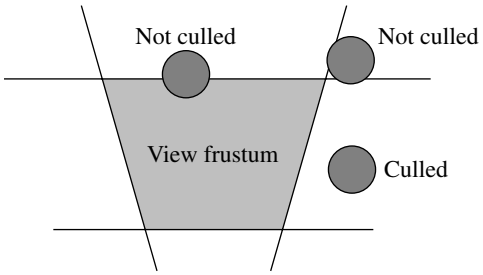

Figure 4.2 Examples of culled and unculled objects.

Source Code

Library
Intersection

Filename
IntrPlnSphr

bounding sphere have center $\vec{C}$ and radius r. Let a view frustum plane be specified by $\vec{N} \cdot \vec{X} = d$, where $\vec{N}$ is a unit-length vector that points to the interior of the frustum. The bounding sphere does not intersect the frustum when the distance from $\vec{C}$ to the plane is larger than the sphere radius. An object is completely culled if its bounding sphere satisfies

$$\vec{N} \cdot \vec{C} - d < -r \tag{4.8}$$

for one of the frustum planes. The left-hand side of the inequality is the signed distance from $\vec{C}$ to the plane. The right-hand side is negative and indicates that to be culled, $\vec{C}$ must be on the outside of the frustum plane and must be at least the sphere radius units away from the plane. The test requires 3 multiplications and 3 additions. The pseudocode is

```
bool CullSpherePlane (Sphere sphere, Plane plane)
{
    return Dot(plane.N,sphere.C) - plane.d < -sphere.r;
}
```

It is possible for a bounding sphere to be outside the frustum even if all six culling tests fail. Figure 4.2 shows examples of an object that is culled by the tests. It also shows examples of objects that are not culled, one object whose bounding sphere intersects the frustum and one object whose bounding sphere does not intersect the frustum. In either case, the object must be further processed in the clipping pipeline. Alternatively, the exact distance from bounding sphere to frustum can be computed at greater expense than the distances from sphere to planes.

Better-fitting bounding volumes can lead to rejection of an object when the bounding sphere does not, thereby leading to savings in CPU cycles. However, the

application must keep the bounding volume current as the object moves about the world. For each change in a rigid object's orientation, the bounding volume must be rotated accordingly. This leads to a trade-off between more time to update bounding volume and less time to process objects because they are more accurately culled.

The following sections describe the culling algorithms for oriented boxes, capsules, lozenges, cylinders, and ellipsoids. In each section the frustum plane is $\vec{N} \cdot \vec{X} = d$ with unit-length normal pointing to frustum interior.

4.3.2 Culling by Oriented Boxes

An oriented bounding box is outside the frustum plane if all its vertices are outside the plane. The obvious algorithm of testing if all eight vertices are on the "negative side" of the plane requires eight comparisons of the form $\vec{N} \cdot \vec{V} < d$. The vertices are of the form

Source Code

Library
Intersection

Filename
IntrPlnBox3

$$\vec{V} = \vec{C} + \sigma_0 a_0 \vec{A}_0 + \sigma_1 a_1 \vec{A}_1 + \sigma_2 a_2 \vec{A}_2,$$

where $|\sigma_i| = 1$ for all i (eight possible choices, two for each σ_i). Each test requires computing signed distances

$$\vec{N} \cdot \vec{V} - d = (\vec{N} \cdot \vec{C} - d) + \sigma_0 a_0 \vec{N} \cdot \vec{A}_0 + \sigma_1 a_1 \vec{N} \cdot \vec{A}_1 + \sigma_2 a_2 \vec{N} \cdot \vec{A}_2.$$

The 4 dot products are computed once, each dot product using 3 multiplications and 2 additions. Each test requires an additional 3 multiplications and 4 additions (the multiplications by σ_i are not counted). The eight tests therefore require 36 multiplications and 40 additions.

A faster test is to project the box and plane onto the line $\vec{C} + s\vec{N}$. The symmetry provided by the box definition yields an interval of projection $[\vec{C} - r\vec{N}, \vec{C} + r\vec{N}]$. The interval is centered at $\vec{C}$ and has radius

$$r = a_0 |\vec{N} \cdot \vec{A}_0| + a_1 |\vec{N} \cdot \vec{A}_1| + a_2 |\vec{N} \cdot \vec{A}_2|.$$

The frustum plane projects to a single point

$$\vec{P} = \vec{C} + (d - \vec{N} \cdot \vec{C})\vec{N}.$$

The box is outside the plane as long as the projected interval is outside, in which case $\vec{N} \cdot \vec{C} - d < -r$. The test is identical to that of sphere-versus-plane, except that r is known for the sphere but must be calculated for each test of an oriented bounding box. The test requires 4 dot products, 3 multiplications, and 3 additions for a total operation count of 15 multiplications and 11 additions. The pseudocode is

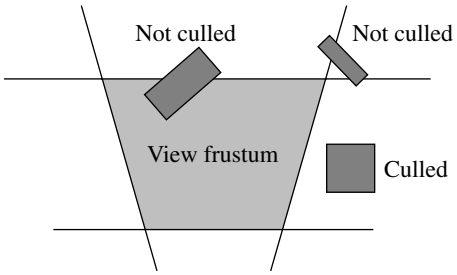

Figure 4.3 Examples of culled and unculled objects.

```
bool CullBoxPlane (Box box, Plane plane)
{
    r = box.a0*|Dot(plane.N,box.A0)| +
        box.a1*|Dot(plane.N,box.A1)| +
        box.a2*|Dot(plane.N,box.A2)|;

    return Dot(plane.N,box.C) - plane.d < -r;
}
```

As with the sphere, it is possible for an oriented bounding box not to be culled when tested against each frustum plane one at a time, even though the box is outside the view frustum. Figure 4.3 illustrates such a situation.

4.3.3 Culling by Capsules

Source Code

Library
Intersection

Filename
IntrPlnCap

A capsule consists of a radius $r > 0$ and a parameterized line segment $\vec{P} + t\vec{D}$, where $\vec{D} \neq \vec{0}$ and $t \in [0, 1]$. The signed distances from plane to end points are $\delta_0 = \vec{N} \cdot \vec{P} - d$ and $\delta_1 = \vec{N} \cdot (\vec{P} + \vec{D}) - d$. If either $\delta_0 \geq 0$ or $\delta_1 \geq 0$, then the capsule is not culled since it is either intersecting the frustum plane or on the frustum side of the plane. Otherwise, both signed distances are negative. If $\vec{N} \cdot \vec{D} \leq 0$, then end point $\vec{P}$ is closer in signed distance to the frustum plane than is the other end point $\vec{P} + \vec{D}$. The distance between $\vec{P}$ and the plane is computed and compared to the capsule radius. If $\vec{N} \cdot \vec{P} - d \leq -r$, then the capsule is outside the frustum plane and it is culled; otherwise it is not culled. If $\vec{N} \cdot \vec{D} > 0$, then $\vec{P} + \vec{D}$ is closer in signed distance to the frustum plane than is $\vec{P}$. If $\vec{N} \cdot (\vec{P} + \vec{D}) - d \leq -r$, then the capsule is culled; otherwise it is not culled. The pseudocode for the culling algorithm is given below. The Boolean result is `true` if and only if the capsule is culled.

```
bool CullCapsulePlane (Capsule capsule, Plane plane)
{
    sd0 = Dot(plane.N,capsule.P) - plane.d;
    if ( sd0 < 0 )
    {
        sd1 = sd0 + Dot(plane.N,capsule.D);
        if ( sd1 < 0 )
        {
            if ( sd0 <= sd1 )
            {
                // P0 closest to plane
                return sd0 <= -capsule.r;
            }
            else
            {
                // P1 closest to plane
                return sd1 <= -capsule.r;
            }
        }
    }

    return false;
}
```

4.3.4 Culling by Lozenges

SOURCE CODE

LIBRARY
Intersection

FILENAME
IntrPlnLoz

A lozenge consists of a radius $r > 0$ and a parameterized rectangle $\vec{P} + s\vec{E}_0 + t\vec{E}_1$, where $\vec{E}_0 \neq \vec{0}$, $\vec{E}_1 \neq \vec{0}$, $\vec{E}_0 \cdot \vec{E}_1 = 0$, and $(s, t) \in [0, 1]^2$. The four rectangle corners are $\vec{P}_{00} = \vec{P}$, $\vec{P}_{10} = \vec{P} + \vec{E}_0$, $\vec{P}_{01} = \vec{P} + \vec{E}_1$, and $\vec{P}_{11} = \vec{P} + \vec{E}_0 + \vec{E}_1$. The signed distances are $\delta_{ij} = \vec{N} \cdot \vec{P}_{ij} - d$. If any of the signed distances are nonnegative, then the lozenge either intersects the plane or is on the frustum side of the plane and it is not culled. Otherwise, all four signed distances are negative. The rectangle corner closest to the frustum plane is determined, and its distance to the plane is compared to the lozenge radius to determine if there is an intersection. The pseudocode for the culling algorithm is

```
bool CullLozengePlane (Lozenge lozenge, Plane P)
{
    sd00 = Dot(plane.N,lozenge.P) - plane.d;
    if ( sd00 < 0 )
    {
        dotNE0 = Dot(plane.N,lozenge.E0);
        sd10 = sd00 + dotNE0;
```

```
            if ( sd10 < 0 )
            {
                dotNE1 = Dot(plane.N,lozenge.E1);
                sd01 = sd00 + dotNE1;
                if ( sd01 < 0 )
                {
                    sd11 = sd10 + dotNE1;
                    if ( sd11 < 0 )
                    {
                        // all rectangle corners on negative side
                        // of plane
                        if ( sd00 <= sd10 )
                        {
                            if ( sd00 <= sd01 )
                            {
                                // P00 closest to plane
                                return sd00 <= -lozenge.r;
                            }
                            else
                            {
                                // P01 closest to plane
                                return sd01 <= -lozenge.r;
                            }
                        }
                        else
                        {
                            if ( sd10 <= sd11 )
                            {
                                // P10 closest to plane
                                return sd10 <= -lozenge.r;
                            }
                            else
                            {
                                // P11 closest to plane
                                return sd11 <= -lozenge.r;
                            }
                        }
                    }
                }
            }

            return false;
        }
```

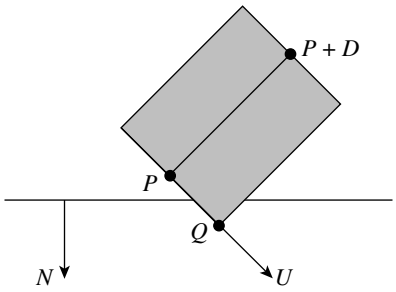

Figure 4.4 Projection of cylinder and frustum plane, no-cull case.

4.3.5 Culling by Cylinders

Source Code

Library
Intersection

Filename
IntrPlnCyln

A cylinder consists of a radius $r > 0$, a height $h \in [0, \infty]$, and a parameterized line segment $\vec{C} + t\vec{W}$, where $|\vec{W}| = 1$ and $t \in [-h/2, h/2]$. Figure 4.4 shows a typical no-cull situation. Let the plane be $\vec{N} \cdot \vec{X} = d$, where $|\vec{N}| = 1$. Let $\vec{U}$, $\vec{V}$, and $\vec{W}$ form an orthonormal set of vectors. Any cylinder point $\vec{X}$ can be written as $\vec{X} = \vec{C} + y_0\vec{U} + y_1\vec{V} + y_2\vec{W}$, where $y_0^2 + y_1^2 = r^2$ and $|y_2| <= h/2$. Let $y_0 = r\cos(A)$ and $y_1 = r\sin(A)$. Substitute $\vec{X}$ in the plane equation to get

$$-(\vec{N} \cdot \vec{W})y_2 = (\vec{N} \cdot \vec{C} - d) + (\vec{N} \cdot \vec{U})r\cos(A) + (\vec{N} \cdot \vec{V})r\sin(A).$$

If $\vec{N} \cdot \vec{W} = 0$, then the plane is parallel to the axis of the cylinder. The two intersect if and only if the distance from $\vec{C}$ to the plane satisfies

$$|\vec{N} \cdot \vec{C} - d| \le r.$$

In this situation the cylinder is culled when $\vec{N} \cdot \vec{C} - d \le -r$.

If $\vec{N} \cdot \vec{W} \neq 0$, then y_2 is a function of A. The minimum and maximum values can be found by the methods of calculus. The extreme values are

$$\frac{d - \vec{N} \cdot \vec{C} \pm \sqrt{1 - (\vec{N} \cdot \vec{W})^2}}{\vec{N} \cdot \vec{W}}.$$

The plane and cylinder intersect if and only if

$$\min(y_2) \le h/2 \quad \text{and} \quad \max(y_2) \ge -h/2.$$

In this situation the cylinder is culled when the previous tests show no intersection and $\vec{N} \cdot \vec{C} - d \leq -r$. The pseudocode is

```
bool CullCylinderPlane (Cylinder cylinder, Plane plane)
{
    sd0 = Dot(plane.N,cylinder.P) - plane.d;
    if ( sd0 < 0 )
    {
        dotND = Dot(plane.N,cylinder.D)
        sd1 = sd0 + dotND;
        if ( sd1 < 0 )
        {
            dotDD = Dot(cylinder.D,cylinder.D);
            r2 = cylinder.r*cylinder.r;
            if ( sd0 <= sd1 )
            {
                // P0 closest to plane
                return dotDD*sd0*sd0 >= r2*(dotDD-dotND*dotND);
            }
            else
            {
                // P1 closest to plane
                return dotDD*sd1*sd1 >= r2*(dotDD-dotND*dotND);
            }
        }
    }

    return false;
}
```

The quantities $\vec{D} \cdot \vec{D}$ and r^2 can be precomputed and stored by the cylinder as a way of reducing execution time for the intersection test.

4.3.6 Culling by Ellipsoids

Source Code

Library
Intersection

Filename
IntrPlnElp3

An ellipsoid is represented by the quadratic equation $Q(\vec{X}) = (\vec{X} - \vec{C})^{\mathrm{T}} M (\vec{X} - \vec{C}) = 1$, where $\vec{C}$ is the center of the ellipsoid, where M is a positive definite matrix, and where $\vec{X}$ is any point on the ellipsoid. An ellipsoid is outside a frustum plane whenever the projection of the ellipsoid onto the line $\vec{C} + s\vec{N}$ is outside the frustum plane. The projected interval is $[-r, r]$. Figure 4.5 shows a typical no-cull situation. The ellipsoid is culled whenever

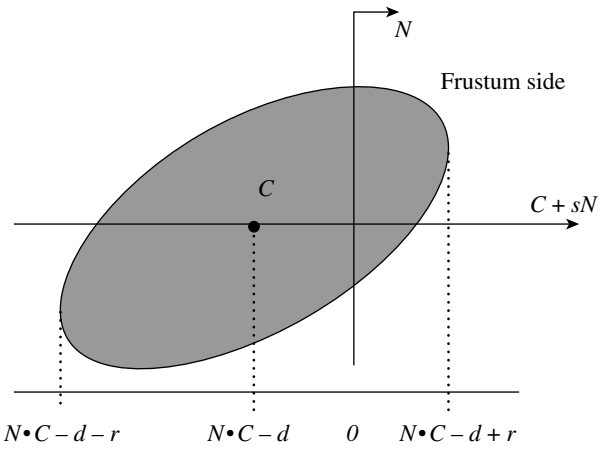

Figure 4.5 Projection of ellipsoid and frustum plane, no-cull case.

$\vec{N} \cdot \vec{C} - d \le -r.$

The construction of r is as follows. The points $\vec{X}$ that project to the end points of the interval must occur where the normals to the ellipsoid are parallel to $\vec{N}$. The gradient of $Q(\vec{X})$ is a normal direction for the point, $\vec{\nabla} Q = 2M(\vec{X} - \vec{C})$. Thus, $\vec{X}$ must be a solution to $M(\vec{X} - \vec{C}) = \lambda \vec{N}$ for some scalar λ. Inverting M and multiplying yields $\vec{X} - \vec{C} = \lambda M^{-1}\vec{N}$. Replacing this in the quadratic equation yields $1 = \lambda^2 (M^{-1}\vec{N})^{\mathrm{T}} M (M^{-1}\vec{N}) = \lambda^2 \vec{N}^{\mathrm{T}} M^{-1} \vec{N}$. Finally, $r = \vec{N} \cdot (\vec{X} - \vec{C}) = \lambda \vec{N}^{\mathrm{T}} M^{-1} \vec{N}$, so $r = \sqrt{\vec{N}^{\mathrm{T}} M^{-1} \vec{N}}$. The pseudocode is

```
bool CullEllipsoidPlane (Ellipsoid ellipsoid, Plane plane)
{
    sd0 = Dot(plane.N,ellipsoid.C) - plane.d;
    if ( sd0 < 0 )
    {
        r2 = Dot(plane.N,ellipsoid.Minverse*plane.N);
        return sd0*sd0 >= r2;
    }

    return false;
}
```

4.3.7 Algorithm for Scene Graph Rendering

An abstract class `Renderer` has a method that is the entry point for drawing a scene graph:

Source Code

Library
Engine

Filename
Renderer
Spatial
Geometry
Node
TriMesh

```
void Renderer::Draw (Spatial scene)
{
    scene.OnDraw(thisRenderer);
}
```

Its sole job is to start the scene graph traversal and pass the renderer for camera access and for accumulating render state. The method is virtual so that any derived class renderer can perform any setup before, and any cleanup after, the scene graph is drawn.

The class `Spatial` implements

```
void Spatial::OnDraw (Renderer renderer)
{
    if ( forceCulling )
        return;

    savePlaneState = renderer.planeState;

    if ( !renderer.Cull(worldBound) )
        Draw(renderer);

    renderer.planeState = savePlaneState;
}
```

The class `Spatial` provides a Boolean flag to allow the application to force culling of an object. If the object is not forced to be culled, then comparison of the world bounding volume to the camera frustum planes is done next. As mentioned in Section 3.4, if the bounding volumes are properly nested, once a bounding volume is inside a frustum plane there is no need to test bounding volumes of descendants against that plane. In this case the plane is said to be *inactive*. The renderer keeps track of which planes are active and inactive (the plane state). The current object must save the current plane state since the state might change during the recursive pass and the old state must be restored.

The member function `Draw` of class `Spatial` is also a pure virtual function. Class `Geometry` manages the leaf node renderer state and uses the `Draw` function to tell the renderer about the state it should use for drawing that leaf node. Class `Node` again provides for the recursive propagation to its children.

```
void Geometry::Draw (Renderer renderer)
{
    renderer.SetState(thisState);
}

void Node::Draw (Renderer renderer)
{
    for each child do
        child.OnDraw(renderer);
}
```

Notice the pattern of recursive chains provided by classes `Spatial` and `Node`. In this case `Draw` and `OnDraw` form the recursive chain.

Finally, for a specific class derived from `Geometry` that has actual data, the renderer must implement how to draw that data. For example, if `TriMesh` is derived from `Geometry` and manages a triangle mesh with vertices, normals, colors, and texture coordinates, the class must implement the virtual function as

```
void TriMesh::Draw (Renderer renderer)
{
    Geometry::Draw(renderer);
    renderer.Draw(this);
}
```

The call to the base class `Draw` tells the renderer to use the current rendering state at the leaf node. The next call allows the renderer to do its specific work with the triangle mesh. The `Draw` call in the renderer is a pure virtual function. If class `SoftRender` is derived from `Renderer` and represents software rendering, then the entire geometric pipeline of transformation, clipping, projection, and rasterizing is encapsulated in `Draw` for `SoftRender`. On the other hand, if class `HardRender` is derived from `Renderer` and represents a hardware-accelerated renderer, then `Draw` probably does very little work and can feed the hardware card directly.