Fast and Simple Occlusion Culling

Wagner T. Corrêa – Princeton University – wtcorrea@cs.princeton.edu
James T. Klosowski – IBM Research – jklosow@us.ibm.com
Cláudio T. Silva – AT&T Labs-Research – csilva@research.att.com

Introduction

In many graphics applications, such as building walkthroughs and first-person games, the user moves around the interior of a virtual environment and the computer creates an image for each location of the user. For any given position, the user typically sees only a small fraction of the scene. Thus to speed up the image rendering, an application should avoid drawing the primitives in the environment that the user cannot see. There are several classes of algorithms to determine which primitives should be ignored, or culled. Back-face culling algorithms determine those primitives that face away from the user. View frustum culling determines the primitives that lie outside of the user's field of view. Occlusion culling determines the primitives that are occluded by other primitives.

While back-facing and view frustum culling algorithms are trivial, occlusion culling algorithms tend to be complex and usually require time consuming preprocessing steps. This gem describes two occlusion culling algorithms that are practical, effective, and require little preprocessing. The first one is the prioritized-layered projection (PLP) algorithm, which is an approximate algorithm that determines, for a given budget, a set of primitives that are likely to be visible. The second algorithm, cPLP, is a conservative version of PLP that guarantees to find all visible primitives.

The Visibility Problem

Given a scene composed of modeling primitives and a viewing frustum, we need to determine which primitive fragments are visible, i.e., connected to the eye-point by a line segment that meets the closure of no other primitive [Dobkin97]. Researchers have studied this problem extensively and many approaches to solve it exist [Cohen-Or01, Durand99]. In their survey on visibility algorithms, Cohen-Or et al. classify algorithms according to several criteria. Next, we briefly summarize those that are of most relevance to our gem:

From-point vs. from-region: Some algorithms compute visibility from the eyepoint only, while others compute visibility from a region in space. Since the user often stays for a while in the same region, the from-region algorithms amortize the cost of visibility computations over a number of frames.

Precomputed vs. online: Many algorithms require an off-line computation, while others work on the fly. For example, most from-region algorithms require a preprocessing step to divide the model in regions, and compute region visibility.

Object space vs. image space: Some algorithms compute visibility in object space, using the exact original 3D primitives. Others operate in image space, using only the discrete rasterization fragments of the primitives.

Conservative vs. approximate: Few visibility algorithms compute exact visibility. Most algorithms are conservative, and overestimate the set of visible primitives. Other algorithms compute approximate visibility, and do not guarantee finding all visible primitives.

In this gem, we describe two visibility algorithms that are simple solutions and work well in practice: the prioritized-layered projection algorithm, PLP, and its conservative version, cPLP.

The PLP Algorithm

PLP [Klosowski00] is an approximate, from-point, object-space visibility algorithm that requires very little preprocessing. PLP may be understood as a simple modification to the traditional hierarchical view frustum culling algorithm [Clark76]. The traditional algorithm recursively traverses the model hierarchy from the root node down to the leaf nodes. If a node is outside the view frustum, we ignore the node and its children. If the node is inside or intersects the view frustum, we recursively traverse its children. The traversal eventually visits all leaves within the view frustum. The PLP algorithm differs from the traditional one in several ways. First, instead of traversing the model hierarchy in a predefined order, PLP keeps the hierarchy leaf nodes in a priority queue called the *front*, and traverses the nodes from highest to lowest priority. When we visit a node (or *project* it, in PLP parlance), if it is visible, we add it to the *visible set*. Then, we remove it from the front, and add its *layer* of unvisited neighbors to the front (hence, the algorithm's name: prioritized-layered projection). Second, instead of traversing the entire hierarchy, PLP works on a *budget*, stopping the traversal after a certain number of primitives have been added to the visible set. Finally, PLP requires each node to know not only its children, but also all of its neighbors.

An implementation of PLP may be simple or sophisticated, depending on the heuristic to assign priorities to each node. Several heuristics precompute the initial *solidity* of a node, and accumulate the solidities along a traversal path. The node's accumulated solidity estimates how likely it is for the node to occlude an object behind it [Klosowski00]. In this gem, we use an extremely simple heuristic to assign priorities to the nodes. The node containing the eyepoint receives priority –1; its neighbors receive priority –2; their neighbors –3, and so on. Using this heuristic, the traversal proceeds in layers of nodes around the eyepoint. This is simple to implement, very fast, and quite accurate (we will show accuracy measurements when we present the run-time results). The only precomputation this heuristic requires is the construction of the hierarchy itself.

We use PLP as a front-end to the hardware's implementation of the Z-buffer algorithm [Foley90]. For a given budget, PLP gives us the set of primitives it considers most likely to maximize image quality. We simply pass these primitives to the graphics hardware.

Implementation

```
class Plp {
public:
    explicit Plp(const char *file_name);
    void start(const View &view);
    bool step();
    enum {DEFAULT_BUDGET = 10000};
    enum {NS_CLEAN, NS_PROJECTED, NS_ENQUEUED,
          NS_UNSEEN};
protected:
    void front_push(OctreeNode *node);
    OctreeNode *front_pop();
    void project(OctreeNode *node);
    void add_neighbors_to_front(OctreeNode *node);
    bool is_potentially_visible(OctreeNode *node);
    typedef set<OctreeNode *,
        OctreeNode::CompareSolidity> plp_front_t;
    plp_front_t _front;
    vector<OctreeNode *> _visible_set;
    unsigned _budget, _num_triangles_rendered;
    unsigned _time_stamp;
    Octree _octree;
    const View *_view;
    OctreeNode *_closest_leaf;
    ImageTileSet _tiles;
};

Plp::Plp(const char *file_name)
    : _budget(DEFAULT_BUDGET),
      _num_triangles_rendered(0), _time_stamp(0),
      _view(NULL), _closest_leaf(NULL)
{
    _octree.read(file_name);
}

void Plp::start(const View &view)
{
    _view = &view;
    _num_triangles_rendered = 0;
    _time_stamp = 0;
    reset_node_states();
    _front.clear();
    _visible_set.clear();
    _closest_leaf = _octree.find_closest_leaf(_view);
    if (_closest_leaf == NULL)
        return;
```

```
    _closest_leaf->set_solidity(0.0);
    _closest_leaf->set_layer(0);
    front_push(_closest_leaf);
}

bool Plp::step()
{
    if (_front.empty())
        return false;
    OctreeNode *node = front_pop();
    project(node);
    add_neighbors_to_front(node);
    return _num_triangles_rendered < _budget;
}

void Plp::front_push(OctreeNode *node)
{
    node->set_time_stamp(_time_stamp);
    _time_stamp++;
    _front.insert(node);
    set_node_state(node, NS_ENQUEUED);
}

// returns node most likely to be visible
OctreeNode *Plp::front_pop()
{
    OctreeNode *node = *(_front.begin());
    _front.erase(_front.begin());
    return node;
}

bool Plp::is_potentially_visible(OctreeNode *node)
{
    return _view->camera().sees_ignoring_near_plane(
        node->bounding_box());
}

void Plp::project(OctreeNode *node)
{
    set_node_state(node, NS_PROJECTED);
    if (is_potentially_visible(node)) {
        _visible_set.push_back(node);
        _num_triangles_rendered +=
            node->num_triangles();
    }
}
```

```
void Plp::add_neighbors_to_front(OctreeNode *node)
{
    const vector<OctreeNode *> &neighbors =
        node->neighbors();
    for (unsigned i = 0; i < neighbors.size(); i++) {
        OctreeNode *neighbor = neighbors[i];
        if (node_state(neighbor) != NS_CLEAN)
            continue;
        if (!is_potentially_visible(neighbor)) {
            set_node_state(neighbor, NS_UNSEEN);
            continue;
        }
        neighbor->set_layer(node->layer() + 1);
        neighbor->set_solidity(neighbor->layer());
        front_push(neighbor);
    }
}
```

The cPLP Algorithm

Although PLP is in practice quite accurate for most frames, it does *not* guarantee image quality, and some frames may show objectionable artifacts. To circumvent this potential problem, we use cPLP [Klosowski01], a conservative extension of PLP.

The main idea of cPLP is to use the visible set given by PLP as an initial guess, and keep adding nodes to the visible set until the front (the priority queue with nodes) is empty. This guarantees that the final visible set is conservative [Klosowski01]. There are many ways to implement cPLP, including exploiting new platform-dependent hardware extensions for visibility computation. The implementation we describe in this gem uses an item-buffer technique that is portable to any system that supports OpenGL.

The cPLP main loop consists of two steps. First, we determine the nodes in the front that are visible. We draw the bounding box of each node in the front using flat shading and a color equal to its identification number. We then read back the color buffer, and determine the nodes seen. Second, for each front node found to be visible, we project it (maybe adding it to the visible set), remove it from the front, and add its unvisited neighbors to the front. We iterate the main loop until the front is empty. The bottleneck of the item buffer-based implementation of cPLP is reading back the color buffer. To avoid reading the entire color buffer at each step, we break the screen into tiles. Tiles that are not modified in one step may be ignored in subsequent steps.

Implementation

```
void Plp::find_visible_front()
{
    // save masks
```

```
GLboolean cmask[4], dmask;
glGetBooleanv(GL_COLOR_WRITEMASK, cmask);
glGetBooleanv(GL_DEPTH_WRITEMASK, &dmask);
// empty front, and remember nodes to render
vector<OctreeNode *> nodes;
while (!_front.empty()) {
    OctreeNode *node = front_pop();
    set_node_state(node, NS_CLEAN);
    nodes.push_back(node);
}
// render front
glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
glDepthMask(GL_FALSE);
glClear(GL_COLOR_BUFFER_BIT);
for (unsigned i = 0; i < nodes.size(); i++) {
    OctreeNode *node = nodes[i];
     // 0 is background
    unsigned j = i + 1;
    GLubyte r = j & 0x000000FF;
    GLubyte g = j & 0x0000FF00;
    GLubyte b = j & 0x00FF0000;
    glColor3ub(r, g, b);
    node->render_solid_box();
}
// determine visible front
_tiles.read_active();
list<ImageTile *>::iterator li =
    _tiles.active_tiles().begin();
while (li != _tiles.active_tiles().end()) {
    ImageTile *tile = *li;
    Image &img = tile->image();
    GLubyte *p = img.pixels();
    unsigned n = img.num_pixels();
    bool tile_active = false;
    for (unsigned i = 0; i < n; i++) {
        GLubyte r = *p++;
        GLubyte b = *p++;
        GLubyte g = *p++;
        GLubyte a = *p++;
        unsigned j = r | g << 8 | b << 16;
        if (j == 0)
            continue;
        tile_active = true;
        j--;
        OctreeNode *node = nodes[j];
        if (node_state(node) != NS_ENQUEUED)
            front_push(node);
```

```
            }
            list<ImageTile *>::iterator t = li;
            li++;
            if (!tile_active)
                _tiles.deactivate(t);
        }
        // restore masks
        glColorMask(cmask[0], cmask[1], cmask[2],
            cmask[3]);
        glDepthMask(dmask);
    }

    void Plp::conservative_finish()
    {
        const Viewport &v = _view->viewport();
        _tiles.realloc(v.x_min(), v.y_min(), v.width(),
            v.height());
        _tiles.activate_all();
        // save state
        glPushAttrib(GL_ENABLE_BIT);
        GLboolean cmask[4], dmask;
        glGetBooleanv(GL_COLOR_WRITEMASK, cmask);
        glGetBooleanv(GL_DEPTH_WRITEMASK, &dmask);
        // compute z-buffer for approximate visible set
        glDisable(GL_LIGHTING);
        glClear(GL_DEPTH_BUFFER_BIT);
        glColorMask(GL_FALSE, GL_FALSE, GL_FALSE,
            GL_FALSE);
        glDepthMask(GL_TRUE);
        for (unsigned i = 0; i < _visible_set.size(); i++)
            _visible_set[i]->render_geometry();
        while (!_front.empty()) {
            find_visible_front();
            // empty front, and remember nodes to project
            vector<OctreeNode *> nodes;
            while (!_front.empty()) {
                OctreeNode *node = front_pop();
                nodes.push_back(node);
            }
            // project nodes, and determine new front
            for (unsigned i = 0; i < nodes.size(); i++)
            {
                OctreeNode *node = nodes[i];
                project(node);
                add_neighbors_to_front(node);
                node->render_geometry();
            }
```

```
        }
        // restore state
        glPopAttrib();
        glColorMask(cmask[0], cmask[1], cmask[2],
            cmask[3]);
        glDepthMask(dmask);
    }
```

Discussion

PLP and cPLP are attractive visibility algorithms for several reasons:

- PLP and cPLP are from-point algorithms, and they make no assumption about the model. In contrast, some from-region algorithms assume the model consists of axis-aligned rooms and portals [Teller91, Funkhouser93], which may be a big restriction.
- PLP and cPLP require little preprocessing. For most heuristics, the precomputation consists of creating the model hierarchy and computing simple summary statistics per node, such as the total number of primitives. This can be done quickly, even for a large model. On the other hand, other techniques [Teller91, Hong97, Zhang97] may require preprocessing times in the order of hours or days, even for relatively small models.
- Although occlusion culling algorithms, such as PLP, avoid rendering unseen geometry, they still may render small primitives that have little effect on the final image. As shown by El-Sana et al. [El-Sana01], PLP can be easily integrated with level-of-detail management.
- PLP is suitable for time-critical rendering. Even if we use the lowest levels of detail, the number of visible primitives in a given frame may overwhelm a low-end graphics card. The PLP budget gives the user a convenient way to compromise between accuracy and speed. The impact of slightly incorrect images on the user's perception of the walkthrough is often far less than the impact of low frame rates [Funkhouser96].

PLP is most useful when higher frame rates are more important than absolute accuracy, e.g., when the user is moving fast to get to a certain point. On the other hand, cPLP is necessary when artifacts are not acceptable, e.g., the user has reached its target and is closely examining its details. Ideally, an application should allow the user to switch back-and-forth between PLP and cPLP on the fly.

Experimental Results

To show what PLP and cPLP can do, we have run tests using the 13 million-triangle UNC power plant model, on a Pentium III 733MHz computer with Nvidia GeForce2 graphics. We collected statistics for both PLP and cPLP using a 500-frame path. Figure 1 shows a typical frame of this path, using a budget of 140,000 triangles per frame.

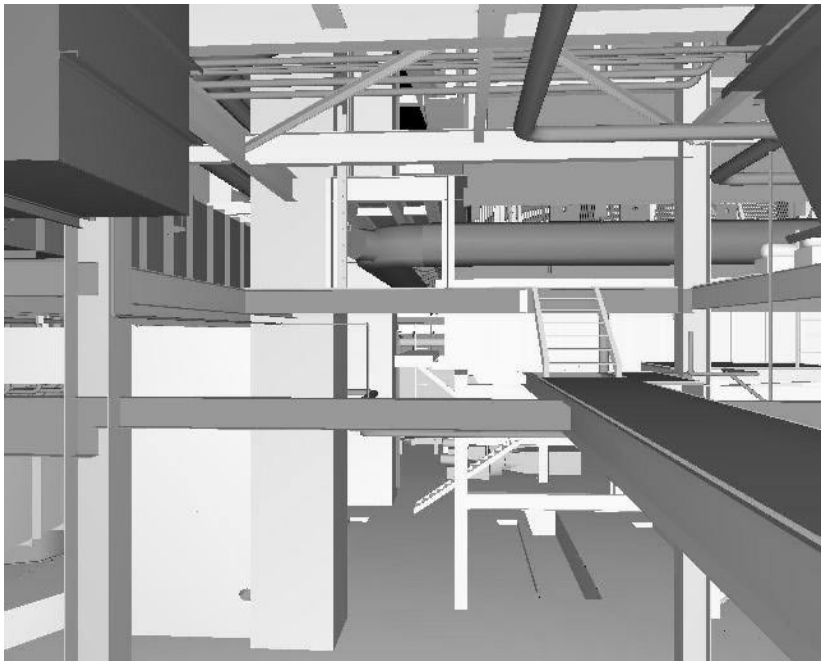

Figure 1. Walking through the UNC power plant [Walkthru01].

For PLP, the average frame rate was 10.1Hz, and for 75% of the times, the frame rate was above 9.3Hz. These rates give the user the illusion of a smooth walkthrough. For cPLP, the average frame rate was 2.1Hz and for 75% of the times, the frame rate was above 1.5Hz. Although the rates for cPLP are lower than the rates for PLP, the system is still usable, and produces images guaranteed to be 100% correct.

We measured the accuracy of PLP by counting the number of incorrect pixels in the images it generated versus the correct images. The average accuracy for PLP was 96.3%, and for 75% of the times, the accuracy was above 94.9%. Because of the layered traversal of the model hierarchy, the wrong pixels tend to be at regions far from the eyepoint. Sometimes the artifacts are noticeable, but they are usually tolerable, and have a small impact on the user experience. Recall that we achieved this level of accuracy with the embarrassingly simple heuristic of traversing the model hierarchy one layer at a time. We believe this accuracy can be even better with more sophisticated heuristics.

Conclusion

PLP and cPLP are practical solutions to the ubiquitous visibility problem. PLP allows the user to tradeoff speed and accuracy. Although there is no guarantee of image quality, in practice it is good enough to give the user a sense of smooth navigation. Whenever 100% accuracy is critical, the user may switch to cPLP, and still be able to walk through the model at slower frame rates.

There are several ways to improve upon what we present in this gem. First, we present only one simple heuristic for estimating the visibility of a node. More sophisticated

heuristics exist [El-Sana01], and there is still room for improvement. Second, these algorithms may be combined with level-of-detail management [El-Sana01]. Third, these algorithms may be used to drive caching schemes to handle models that are larger than the available main memory. Finally, these algorithms may be parallelized to exploit the power of a parallel machine or a cluster of PCs.

References

[Clark76] James H. Clark, "Hierarchical Geometric Models for Visible Surface Algorithms," *Communications of the ACM*, 19(10):547-554, October 1976.

[Cohen-Or01] Daniel Cohen-Or, Yiorgos Chrysanthou, Cláudio T. Silva, and Frédo Durand, "A Survey of Visibility for Walkthrough Applications," to appear in *IEEE Transactions on Visualization and Computer Graphics.*

[Dobkin97] David Dobkin and Seth Teller, *Handbook of Discrete and Computational Geometry*, chapter Computer Graphics, CRC Press, 1997.

[Durand99] Frédo Durand, *3D Visibility: Analytical study and Applications*, PhD thesis, Université Joseph Fourier, Grenoble, France, 1999.

[El-Sana01] Jihad El-Sana, Neta Sokolovsky, and Cláudio T. Silva, "Integrating Occlusion Culling with View-Dependent Rendering, Proceedings of IEEE Visualization 2001, pp. 371-378.

[Foley90] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes, *Computer Graphics: Principles and Practice*, Addison-Wesley, 2nd. edition, 1990.

[Funkhouser93] Thomas A. Funkhouser and Carlo H. Séquin, "Adaptive Display Algorithm for Interactive Frame Rates During Visualization of Complex Virtual Environments, Computer Graphics Proceedings (SIGGRAPH 1993), pp. 247-254.

[Funkhouser96] Thomas A. Funkhouser, "Database Management for Interactive Display of Large Architectural Models," Proceedings of Graphics Interface '96, pp. 1-8.

[Hong97] Lichan Hong, Shigeru Muraki, Arie E. Kaufman, Dirk Bartz, and Taosong He, "Virtual Voyage: Interactive Navigation in the Human Colon," Computer Graphics Proceedings (SIGGRAPH 1997), pp. 27-34.

[Klosowski00] James T. Klosowski and Cláudio T. Silva, "The Prioritized-Layered Projection Algorithm for Visible Set Estimation," in *IEEE Transactions on Visualization and Computer Graphics*, 6(2):108-123, April-June 2000.

[Klosowski01] James T. Klosowski and Cláudio T. Silva, "Efficient Conservative Visibility Culling Using the Prioritized-Layered Projection Algorithm," in *IEEE Transactions on Visualization and Computer Graphics*, 7(4):365-379, October-December 2001.

[Teller91] Seth Teller and Carlo H. Séquin, "Visibility Preprocessing for Interactive Walkthroughs," Computer Graphics Proceedings (SIGGRAPH 1991), pp. 61-69.

[Walkthru01] The Walkthru Project at UNC Chapel Hill, "Power Plant Model," available online at http://www.cs.unc.edu/~geom/Powerplant/.

[Zhang97] Hansong Zhang, Dinesh Manocha, Thomas Hudson, and Kenneth E. Hoff III, "Visibility Culling Using Hierarchical Occlusion Maps," Computer Graphics Proceedings (SIGGRAPH 1997), pp. 77-88.