Exploiting coherence in 2 1/2 D visibility computation

Dieter Schmalstieg and Robert F. Tobler
Vienna University of Technology, Austria
(dieter | rft)@cg.tuwien.ac.at http://www.cg.tuwien.ac.at/

In interactive 3-D graphics used for visualization, virtual reality and computer games, the complexity of the scenes (the amount of geometric primitives passed to the display algorithm) frequently exceeds the capacity of the rendering engine and compromises interactive frame rates.

Visibility computation is used to reduce the amount of geometry that must be passed to the rendering engine. The model is subdivided into subvolumes called "cells", and the visibility algorithm computes the set of cells that can at least partly be seen from the given viewpoint Airey [Aire90] called this the "potentially visible set" (PVS). The geometry associated with these cells (walls, interior objects) is then displayed, and conventional hidden surface removal (e.g. a Z-buffer) is employed to resolve the actual visibility. Variants of such algorithms have been discussed in [Tell93, Gree93, Lueb95, Nayl95, Yage95].

If visibility can be completely determined from the floor plan, 3-D visibility is effectively reduced to a 2-D problem. This imposes restrictions on the model (e.g. bridges are impossible), but nevertheless allows interactive display of a large variety of interesting models such as building interiors and dungeons. Since a three-dimensional image is produced from an extended two-dimensional data structure, we call such algorithms *2 1/2D visibility algorithms.*

At the low end, computer games such as DOOM [ID94] make use of this restricted type of environment that can be rendered with little effort. Image generation for such a model is done by intersecting rays with the scene. When a wall is hit, a vertical span is drawn. Thereby, one pixel column is drawn in turn, thereby avoiding the need for a more complex general 3-D rendering algorithm.

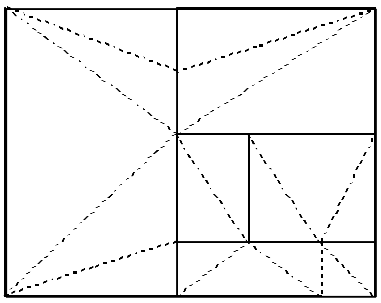

Figure 1: A triangulated floor plan

Here we propose an algorithm that exploits coherence to accelerate visibility computation in 2 1/2D and can be used for on-the-fly computation of potentially visible sets, and also for computing the exact visible portion of the scene. The algorithm computes the equivalent of a viewing ray swept horizontally from left to right, which is a one-dimensional sequence of discrete visibility changes. The intervals between the visibility changes are spatially coherent (a single wall is visible). This information can be used for incremental operations such as drawing, texturing etc.

The input to the algorithm is a mesh of triangular cells (Figure 1), where each edge can or cannot be a wall. Such a data structure can easily be obtained by triangulation from a floor plan or blue print.

Adjacency information of the cells allows the PVS to be computed by visiting all potentially visible cells using a recursive algorithm. The algorithm starts by considering the edges of the cell containing the viewpoint (the initial *current cell*). If one of these edges is a wall, it can be displayed immediately. Otherwise, we step over this *current edge* from the current cell to the *adjacent cell.*

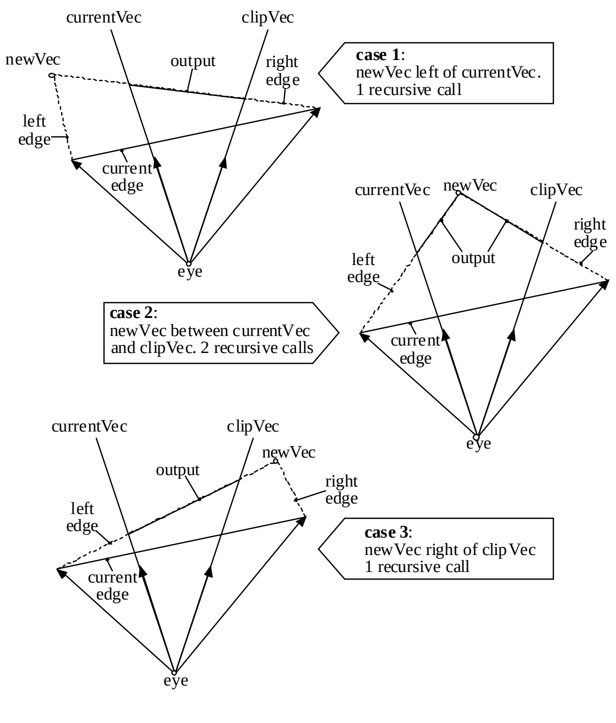
currentVec
clipVec
newVec
output
right edge
left edge
current edge
eye
case 1: newVec left of currentVec. 1 recursive call
case 2: newVec between currentVec and clipVec. 2 recursive calls
case 3: newVec right of clipVec 1 recursive call

Figure 2: The algorithm distinguishes three cases

The adjacent cell has one "new" vertex that is not part of the current edge. The position of this vertex within the sector defined by viewpoint and current edge is relevant for the operation of the algorithm. Three topological cases are distinguished (Figure 2) and lead to different recursive calls. All relevant points are examined by their relative position as seen from the eye.

To examine these relations, the following variables are needed:

- `currentVec`: the vector representing the "viewing ray" that is monotonically swept over the scene
- `currentEdge`: the edge shared by current cell and adjacent cell
- `clipVec`: the vector up to which the current edge is visible
- `newVec`: the vector from the eye to the vertex of the adjacent cell not contained in the current edge.
- `leftEdge`: the left edge of the adjacent cell containing the newVec.
- `rightEdge`: the left edge of the adjacent cell containing the newVec.

The relative position of these vectors with respect to each other can easily and quickly be computed with a dot product (`isLeft` operator).

In the following, we sketch an implementation that computes the PVS. For brevity, we have abandoned those portions of the implementation that deal with the manipulation of the triangle mesh.

```
Vec2D currentVec; //global

void
PVS(Edge currentEdge, Vec2D clipVec)
{
  Vec2D newVec;
  if(isWall(currentEdge))
  {
    //output: 1. current cell in PVS,
    //        2. current edge visible
    //     from currentVec to clipVec
    currentVec = clipVec;
    return;
  }
  newVec = getNewVec(currentEdge);

  if(isLeft(newVec,clipVec))
    PVS(rightEdge,clipVec); //case 1,2
  if(isLeft(currentVec,newVec))
    PVS(leftEdge,clipVec); //case 2,3
}
```

Exact visibility computation. In the presented form, the algorithm outputs all cells that are potentially visible, i.e. where at least one point in the cells extent can be seen from the given viewpoint. However, the exact fraction of the current cell that is visible can easily be computed by intersecting the cell with the sector between `currentVec` and `clipVec`.

Incremental computation of vertical spans. The same idea can be applied if the purpose is to produce vertical spans for a simple display algorithm. In this case, `currentVec` is incrementally advanced in constant steps corresponding to the distance between pixel rows on the screen until it is right of `clipVec`. Depth computation, shading and texturing operations can be carried out incrementally in the same way.

The major advantage of the proposed algorithm over raycasting is its use of coherence as can be seen in the code fragment above: The whole visible fraction of a wall is considered at a time, allowing faster incremental drawing operations and avoiding the need to re-compute ray/scene intersection for every pixel column.

Extending the algorithm to three dimensions. In 3-D, the model space can be represented as a mesh of tetrahedrons or cubes whose faces may be either walls or portals to neighboring cells. Again starting with the cell that contains the eyepoint, all walls of this cell are drawn, and then the algorithm steps on to the adjacent cells. The major problem with this approach is that clipping regions can become complex, and that a single monotonic sweep like in 2-D is no longer possible (compare [Lueb95]).

References

[Aire90] J. M. Airey, J. H. Rohlf, F. Brooks Jr.: Towards Image Realism with Interactive Update Rates in Complex Virtual Building Enviroments. Computer Graphics, Vol. 24, No. 2, pp. 41 (1990)

[Gree93] N. Greene, M. Kass, G. Miller: Hierarchical Z-Buffer Visibility. Proceedings of SIGGRAPH'93, pp. 231-237 (1993)

[ID94] ID Software: DOOM. Computer game (1994)

[Lueb95] D. Luebke, C. Georges: Portals and Mirrors: Simple, Fast Evaluation of Potentially Visible Sets. Proceedings SIGGRAPH Symposium on Interactive 3D Graphics, pp. 105-106 (1995)

[Nayl95] B. Naylor: Interactive playing with large synthetic environments. SIGGRAPH Symposium on Interactive 3D Graphics (1995)

[Tell93] S. Teller: Visibility Computations in Densely Occluded Polyhedral Environments. PhD Thesis, Princeton University (1993)

[Yage95] Yagel R., Ray W.: Visibility Computation for Efficient Walkthrough of Complex Environments. Presence, Vol. 5, No. 1, pp. 45-60 (1995)