

Representing and Processing Screen Space in Augmented Reality

Blaine Bell

blaine@cs.columbia.edu

Steven Feiner

feiner@cs.columbia.edu

Columbia University

450 Computer Science Building

1214 Amsterdam Avenue

New York, NY 10027

(212) 939-7000

fax: (212) 666-0140

Keywords: View management, view management pipeline, collaborative augmented reality, visible-surface-determination, binary space partitioning tree, z-buffer, object-buffer, interval tree, splitting planes, space management, label placement, image-plane representation, screen space, world-in-miniature

REPRESENTING AND PROCESSING SCREEN SPACE IN AUGMENTED REALITY

View management involves computing what a user sees within a 3D environment when it is projected onto a display screen. Doing this interactively to create effectively laid out user interfaces for augmented reality, virtual reality, or any other kind of 3D user interface, requires representing and processing screen space. In this chapter, we describe how to compute a 2D screen-space representation that corresponds to the visible portions of the projections of 3D objects on the screen. We describe in detail two visible-surface-determination algorithms that are used to generate these representations: one based on a Binary-Space Partitioning tree, and one based on a hardware accelerated z-buffer and object buffer. We compare the performance and accuracy tradeoffs of these algorithms, and present examples of how to use our representation to satisfy visibility constraints that avoid unwanted occlusions, making it possible to label and annotate objects in 3D environments.

INTRODUCTION

Augmented Reality (AR) makes it possible to concurrently visualize both the real world and overlaid virtual information. While designing a conventional user interface requires deciding what information should be presented, and how and where it should be shown, designing an AR user interface thus requires addressing another crucial problem: the superimposed virtual information can occlude things that we would otherwise see in the real world. Therefore, to avoid obscuring important objects, it is necessary to determine the position and size of virtual objects within the user interface relative to what is seen in the real world.

View management (Bell et al., 2001) refers to the layout decisions that determine spatial relationships among objects in a 3D user interface. For an application to perform view management, it must compute the visibility of objects of interest in a 3D environment, as seen from a selected 3D viewpoint, taking into account visibility constraints. It also must represent and process the visibility information once it has been projected onto the 2D screen space representing the user's view.

Figure 1 shows the order in which computations are done in our view management pipeline (Bell et al., 2005). To precisely maintain the specified visual constraints, this pipeline should get executed for each frame rendered. However, since some of these computations can take a considerable amount of time to compute, asynchronous execution of the pipeline, as in the Decoupled Simulation Model (Shaw et al., 1993), can preserve interactive rendering frame rates at the expense of view management

accuracy. (These tradeoffs might be preferable depending on the user's preference or task.)

In this chapter, we focus on how to compute the visibility information on which view management is based, as performed in step 3 of Figure 1. This occurs after the viewing specification has been determined (step 2), and the 3D projection has been defined. The resulting visibility information will be used further down the pipeline to satisfy view management constraints, such as avoiding occlusion or determining placements for annotations that are close to the visible portions of associated objects.

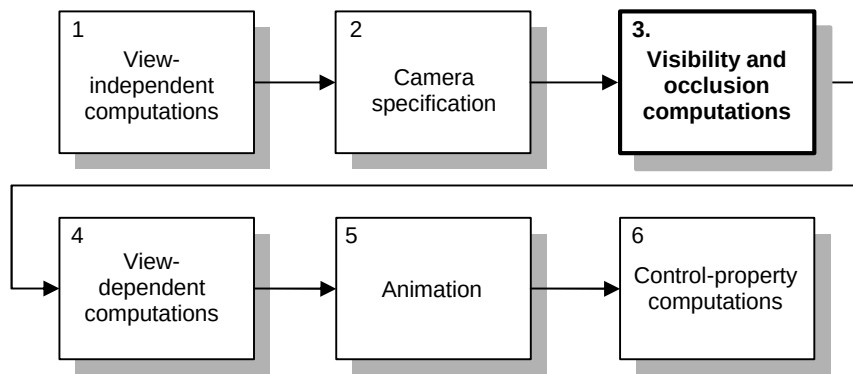


Figure 1. View management computational pipeline: Screen space processing occurs in step 3.

Figure 2 demonstrates how we apply view management to a collaborative augmented reality environment. It is photographed through an optical see-through head-worn display, in which the user directly views the real world through optics that also reflect graphics rendered on miniature LCDs. One user is sitting across from another, discussing a virtual model of the Columbia campus located between them. In response to a request from the user whose view is displayed, all virtual buildings have been annotated with their names. Each label is scaled within a user-selectable range and positioned automatically. A label is preferentially placed within a visible portion of its building's projection. If there is no space to accommodate a label legibly within that projection, it is instead placed near the projection, but not overlapping other buildings or annotations, and is connected to its building by an arrow. Additional annotations in the scene include a meeting agenda (shown at the left), a virtual copy of a building that one of the users has created, and popup information related to that building (both shown at the right). The projections of these

annotations avoid other objects, including the visible user's head, to maintain a clear line of sight between users.



Figure 2. View management in an augmented reality meeting scenario (imaged through a tracked, optical see-through, head-worn display).

In this example, the system needs to compute visibility for the virtual 3D environment, which consists of the campus buildings, and the real world, which consists of the other user's body, the conference table, and additional objects that cannot be seen in this

view. Determining visibility and occlusion requires the use of a visible-surface-determination algorithm. However, none of the visible-surface-determination algorithms that are conventionally used in 3D computer graphics will directly produce the representations needed to solve view management constraints. To satisfy these constraints, which affect the placement of labels and other annotations, we use a visible-surface-determination algorithm in combination with a 2D spatial representation to represent the visible portions of the projections of 3D objects on the view plane. Each representation of a 2D spatial region consists of a list of rectangles that are axis-aligned, are possibly overlapping, and are the largest such rectangles that lie wholly inside that region.

We have used two different visible-surface-determination algorithms: one based on a Binary Space Partitioning (BSP) tree (Fuchs et al., 1980), and one based on a hardware-accelerated z-buffer (Catmull, 1974) and object buffer (Atherton, 1981). In the

rest of this chapter, we discuss the 2D spatial representation, we describe how to generate the desired 2D spatial representations using the BSP tree and the z-buffer, and we compare their performance and accuracy tradeoffs for different kinds of 3D applications. We then present examples of how to use these visibility representations to label and annotate objects in 3D environments.

RELATED WORK

Visible-surface-determination algorithms have long been a research topic in computer graphics (Sutherland et al., 1974). However, standard visible-surface-determination algorithms do not create a 2D screen-space representation that is suitable for view management. For example, the most common algorithm used for visible-surface determination is the z-buffer or depth-buffer algorithm (Catmull, 1974), in which the frame buffer is paired with a z-buffer of the same width and height as the frame buffer. Each frame buffer pixel thus has an associated z-value (depth value) in the z-buffer that represents the depth of the closest object rendered for that pixel. This makes it possible to account for scene visibility by iterating through the scene polygons and rendering each pixel only if its z-value is closer than the z-value already associated with that pixel. If a unique value is rendered into the frame buffer for each object, then the frame buffer becomes an *object buffer* (Atherton, 1981) that can be used to determine the

object visible at each pixel. However, since the results produce a discrete representation on the image plane with pixel granularity, we describe how additional processing can generate a representation that better supports view management.

List-priority visible-surface-determination algorithms rely on explicit sorting of polygons in a view-dependent (back-to-front or front-to-back) order, splitting polygons to eliminate visibility conflicts. For example, the Binary Space Partitioning (BSP) tree algorithm (Fuchs et al., 1980; Schumaker et al., 1969; Paterson & Yao, 1990; Thibault & Naylor, 1987) splits polygons during a preprocessing step in which a tree data structure is constructed. The visibility order for an arbitrary projection of a scene is determined by a view-dependent traversal of the tree, which needs to be modified only when the scene geometry changes. These algorithms assume that some additional representation is provided to accumulate the image-plane representation that results from rendering the polygons in the proper order. In this chapter, we

also describe how a sorted list of polygons in visibility order can be used to generate an appropriate 2D image-plane representation. (Similarly, the visibility skeleton (Durand et al., 1997)—a graph data structure that encodes all possible visibility relationships in a polygonal scene—could also be used to derive an image-plane representation for view management.)

There are also other ways of representing what is visible on the image plane. Area-subdivision algorithms partition the 2D image plane into regions of common visibility. For example, Warnock's algorithm (Warnock, 1969) recursively subdivides the image plane into quadrants, creating a representation that would later be known as a quad tree (Samet, 1990). Unfortunately, this representation partitions the image plane into non-overlapping pieces. Consequently, it makes it relatively expensive to determine placements that are larger than any of the individual non-overlapping pieces, which is necessary for effective view management.

View management has also been accomplished using other types of representations and techniques. In one approach (Ali et al., 2005), a scene object buffer is directly evaluated and annotations are laid out using a force-directed approach (Fruchterman & Reingold, 1991). Azuma and Furmanski (Azuma, 2003) perform interactive label placement by executing a radial search around a point to be labeled to find a free position, in the spirit of earlier work on static map label placement (Christensen et al., 1995). While these approaches perform view management without using our separate 2D image-plane representation, it is more difficult for them to avoid overlap completely.

2D REPRESENTATION

The 2D visibility representation that we use to solve view management constraints consists of a list of axis-aligned rectangles for each visible region. Each of these rectangles within the lists is as large as it can be without extending past the boundaries of the region. Furthermore, there are no redundant

rectangles (i.e., no rectangle is enclosed by another in the list), and the rectangles exhaustively cover the entire region. Once this representation is calculated, many view-management constraints can be solved by searching the 2D visible space of the objects on the view plane. For example, if any axis-aligned rectangle were to be drawn within the region, then at least one of these largest space rectangles will completely enclose the drawn rectangle. This makes it very easy for programmers to determine whether a desired rectangle can be drawn within the region by sequentially comparing it with each rectangle in the region's representation.

Figure 3 shows the rectangles that represent the empty space (i.e., the area where objects do not exist) within a rectangular area when there are no objects (part a), one object (part b), and two objects (part c). We have developed algorithms for adding, deleting, and moving objects within an arbitrary 2D space, which maintain the representation so that applications can query the empty space at any time (Bell & Feiner, 2000). Since these lists of rectangles

represent an arbitrary space, they can also be used to represent the visible space of an object on the 2D view plane.

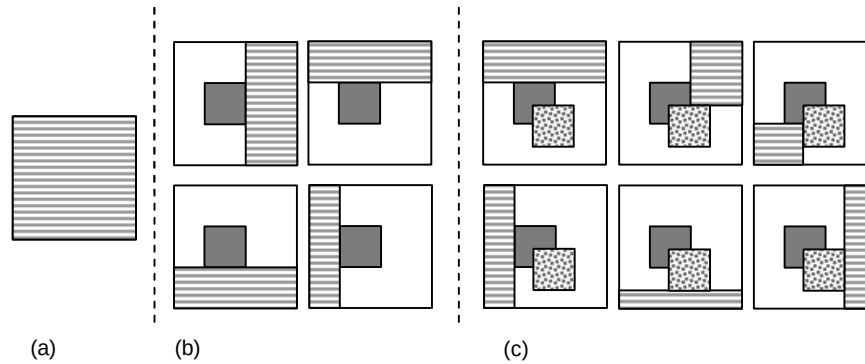


Figure 3 2D empty-space representation consists of axis-aligned largest rectangles (horizontal stripes). (a) No scene objects. (b) One scene object (solid) is added. (c) Second scene object (dotted) is added.

COMPUTING VISIBILITY USING A BINARY SPACE PARTITIONING (BSP) TREE

We can perform visible-surface determination by using the BSP tree algorithm (Fuchs et al., 1980) to efficiently sort the upright extents of the objects' projections to produce the visibility order for an arbitrary projection of a scene. A BSP tree is a binary tree whose nodes typically represent actual polygons (or polygon fragments) in the scene. Because we need to determine the

visibility order for objects, we use BSP tree planes to separate objects rather than individual polygons (Schumaker et al., 1969; Paterson & Yao, 1990). We choose these planes using the heuristics of Thibault and Naylor (Thibault & Naylor, 1987). Although BSP trees are often used for purely static scenes, dynamic objects can be handled efficiently by adding these objects to the tree last and removing and adding them each time they move (Chrysanthou & Slater, 1992).

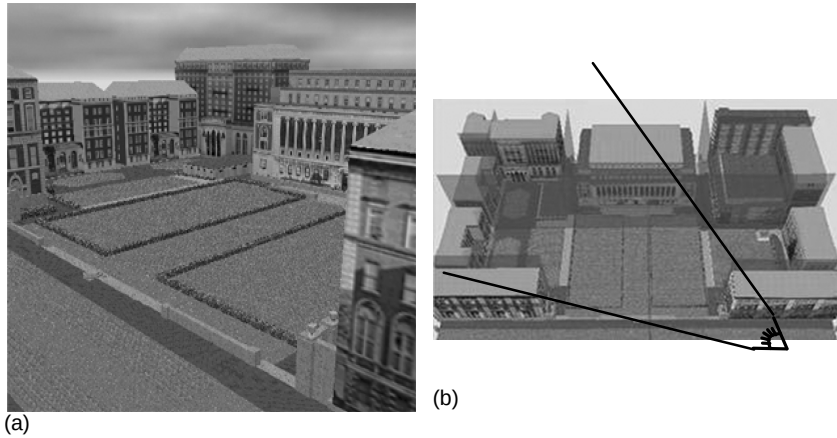


Figure 4. (a) 3D view of our campus. (b) Bird's eye view of the campus scene, which includes semi-transparent splitting planes for the BSP tree. Eye icon and lines indicate the user's view frustum in (a) with respect to the model.

Determining the Visible Portions of Each Object

When visibility computations are needed, we determine the visible portions of the scene objects by using our 2D space representation. As objects are processed in a front-to-back order using the BSP tree, the space representation is populated with *full-space* rectangles that represent the 2D projections of the 3D scene objects. A complementary empty space representation encodes the 2D screen space of the view plane that is not populated by any previously processed object. By evaluating the empty space representation before an object is processed, the visible (i.e., unoccluded) space for that object is determined to be the empty space within the object's 2D projection (i.e., the space that has no projections of previously processed objects). We obtain the largest visible-space representation for each object by clipping the largest empty-space rectangles that overlap the object's projected area and eliminating redundant rectangles that are wholly contained in any others. This visible space representation provides

an efficient way to find suitable locations for annotations that must be placed relative to the visible area of an object's projection.

Figure 4(a) shows an example of a 3D view of our campus, along with a bird's eye view of the scene (Figure 4b) that includes semi-transparent splitting planes for the BSP tree and an eye icon (▲) with lines that indicate the user's view frustum. For each node obtained from the BSP tree in front-to-back order, the 2D upright extent of its projection is computed by projecting each vertex and adding the encompassing axis-aligned bounding rectangle to the space representation. Each row from top to bottom in Figure 5 shows how each node's projection is processed in order to compute each node's visible space, from front-to-back relative to the user's viewpoint. Initially, the 2D spatial representation of the view plane has no full-space rectangles and consists of one largest space rectangle that represents the entire rectangular-shaped view plane (Figure 5a, upper right).

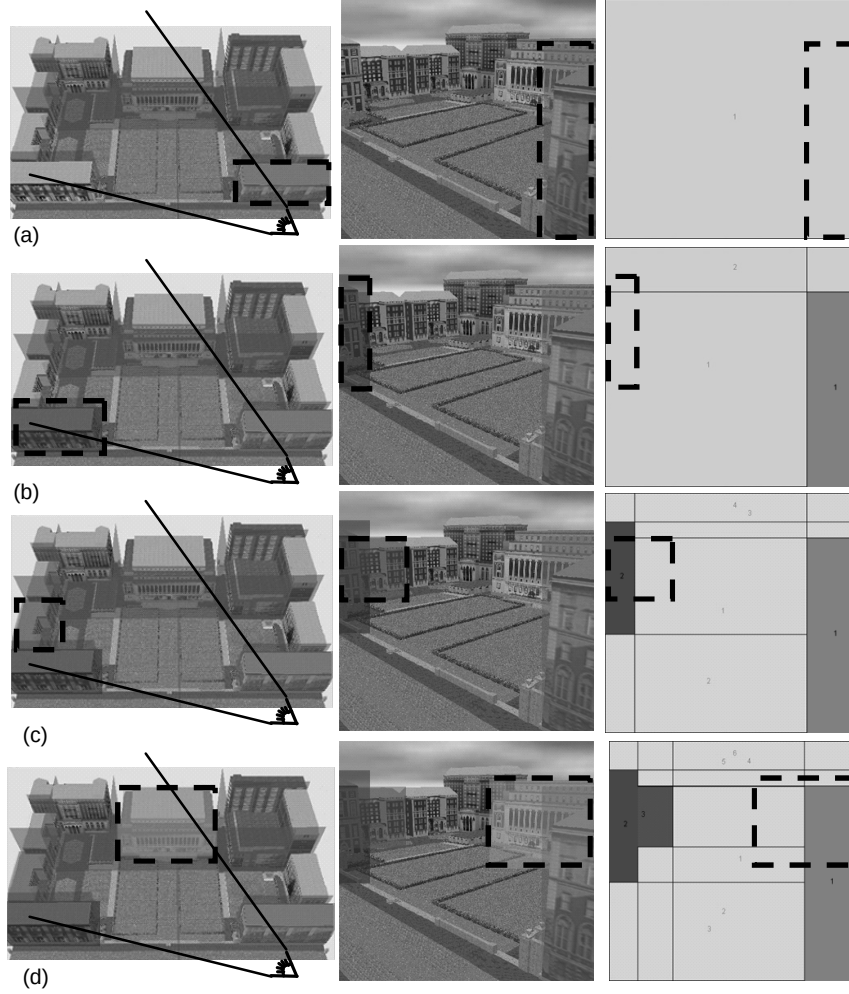


Figure 5. Computing screen space of each object from front-to-back relative to the user's viewpoint. Each row, from top to bottom, shows the processing of an additional object. The current BSP tree object being processed is shaded in the bird's eye view (left column), its projection is shown in the same shade on the 2D view plane (middle column), and its projection is outlined with dashes in the 2D spatial representation (right column). The spatial representation is used to determine visible space for the object by computing the empty space within the BSP tree node's projection. After a projection has been processed, it is added as full-space to the 2D spatial representation.

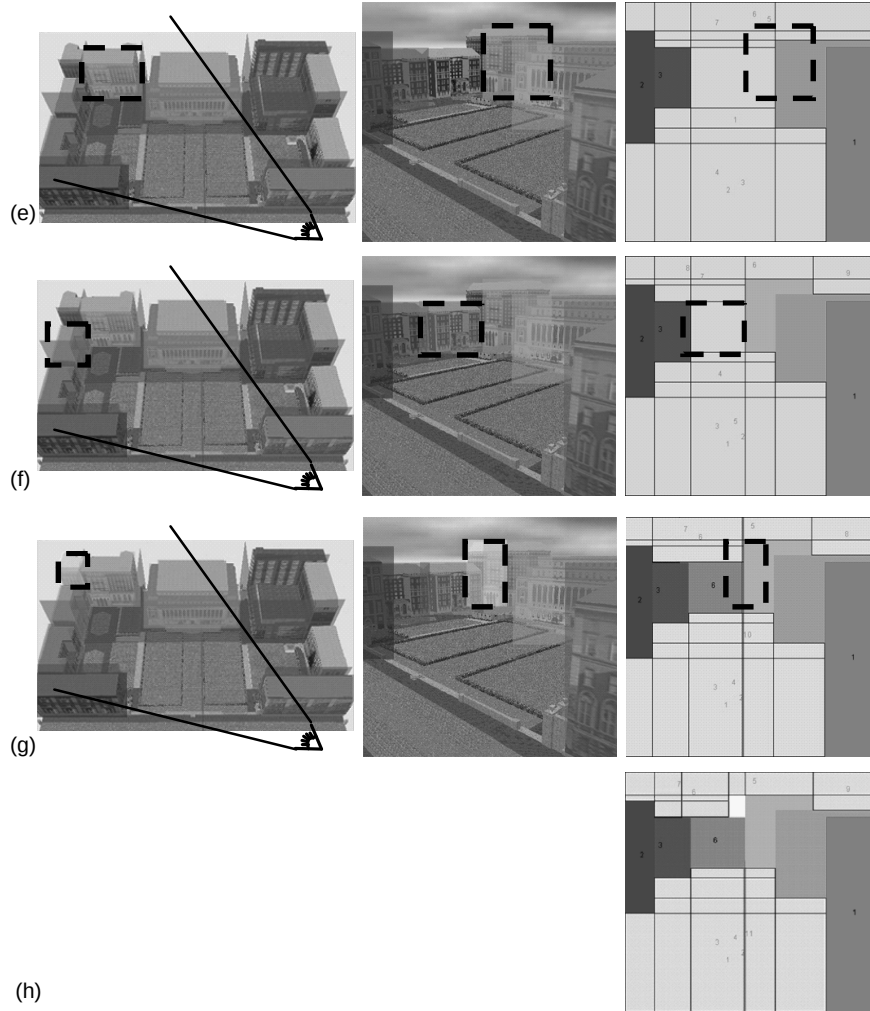


Figure 5 (continued). Computing screen space for the remaining objects from front-to-back relative to the user's view point. (h) Screen space representation after the last object is processed.

When processing each node, the upright extent of its 2D projection is intersected with the members of the current list of largest empty-space rectangles, as shown in Figure 6. This can be performed efficiently because we maintain the largest empty-space rectangles in a 2D interval tree (Samet, 1990), allowing an efficient window query to determine the members that intersect the extent. (The intersection operation itself is simple because all rectangles are axis-aligned.) The intersection yields a set of rectangles (not necessarily contiguous), some of which may be wholly contained within others; these subset rectangles are eliminated. The result is a set of largest rectangles whose union is the visible portion of the node (Figure 6c). In our example, the visible space of the first two nodes is computed as their entire projections (Figure 5a–b). This is because there are no previously processed projections that overlap each node’s projected area (i.e., the dashed outline in the middle column), which both lie completely within the 2D empty space (Figure 5, right column).

Each subsequent object processed (Figure 5c–g), has a projection that overlaps the previously processed projections, and thus, its visible space includes only a portion of the projected area. Consider the third node processed (Figure 5c): the detail presented in Figure 7(a) shows the upright extent of its 2D projection as a dashed outline overlaid onto the 3D projection (left) and overlaid onto the 2D space representation (right). The empty space is evaluated within the dashed outline, as shown in Figure 7(a), and results in the one dark grey rectangle (right) that represents the node’s visible space. The fourth node’s projection (light grey in Figure 5d) is evaluated within the empty space and the visible space is determined to be the two dark grey rectangles shown on the right in Figure 7(b). The rest of the nodes are processed in the same way (shown in Figure 5e–g), and the resulting empty space representation (Figure 5h) can be used, in view-dependent computations, for objects that must avoid the projections of all of the objects in the scene.

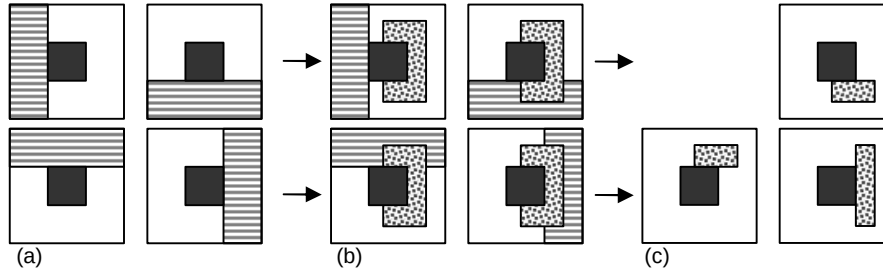


Figure 6. Adding a new object to the view-plane representation. (a) original object (solid), shown separately with each of the four original largest empty spaces (striped) (b) New object extent (dotted) is added. (c) Resulting visible largest spaces of new object extent (dotted).

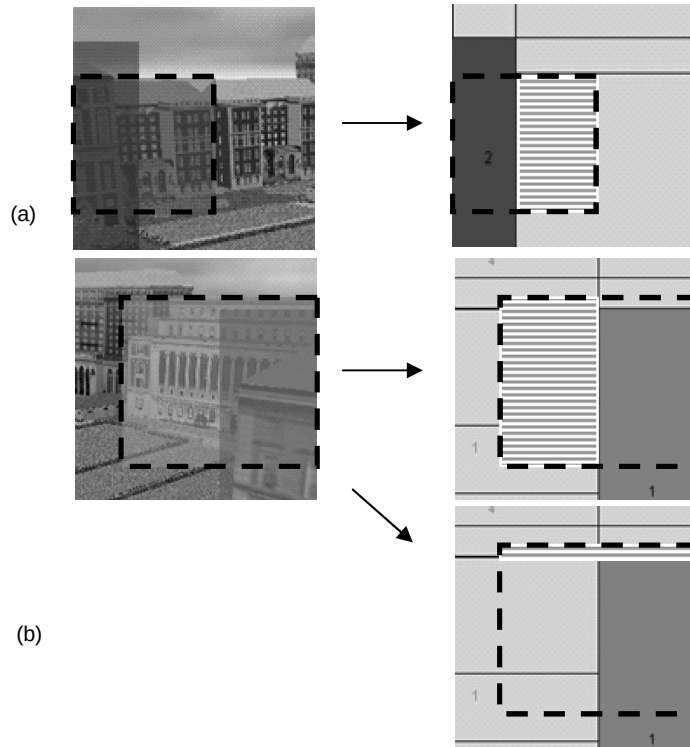


Figure 7. Processing of projections (dashed outline) of visible BSP tree nodes (from Figure 5c–d). Projections are overlaid onto the 3D projection on the left of each subfigure. The empty space is evaluated within the area of these projections to result in the largest visible space rectangles (striped), shown on the right, that represent the visible spaces for the nodes. (a) Object added in Figure 5(c). (b) Object added in Figure 5(d).

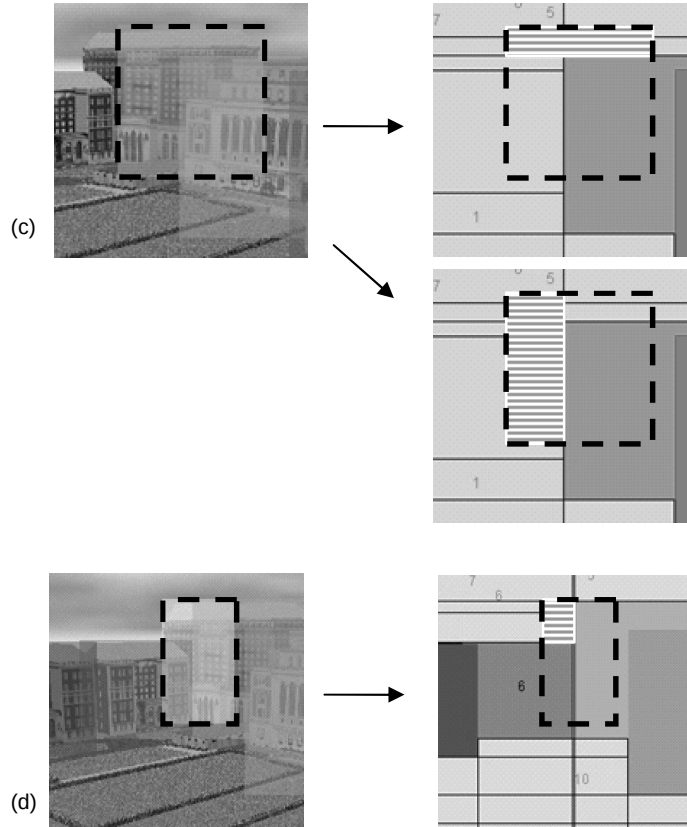


Figure 7 (continued). (c) Object added in Figure 5(e). (d) Object added in Figure 5(g).

Some objects are represented by a single BSP tree node, while others are split across multiple nodes. Consider Figure 8, where objects are placed so that it is not possible to create a plane that will partition the scene without splitting one or more objects. In

this case, we continue to use the heuristics of Thibault and Naylor (Thibault & Naylor, 1987) to minimize the amount of splitting in the BSP tree. However, there are certain objects, specifically concave objects, which often do not get split well using these heuristics. This can result in large screen space approximations for concave objects, including empty space, which can inappropriately increase the area allocated to them on the view plane. To address this, we create splitting planes manually to assure that the BSP tree has multiple parts for some of the objects. This makes it possible to support concave objects, which could alternatively be supported automatically by using convex decomposition algorithms (Mangan & Whitaker, 1999; Lien & Amato, 2004).

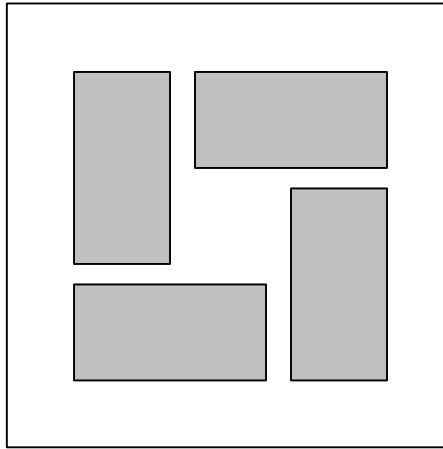


Figure 8. Example of an object layout (in 2D or in 3D as a plan view of a scene) that requires a BSP plane to split at least one of the objects. There is no plane that will partition the group of objects without cutting an object in parts. This requires us to split an object in the object BSP tree into multiple parts that are used to compute 2D screen space individually. These visible parts are then combined together to form the visible space of an entire object.

Figure 9 shows an example of an object split into multiple parts by a BSP tree plane, along with the approximation of each part's projection transparently overlaid. For each part of an object, the largest visible rectangles are computed, as described previously in this section, and are shown for these specific parts in Figure 7(c–d). Once the visible rectangles have been computed for each part of an object, they must now be combined to obtain the visible

space rectangles for the entire object. To do this, each part of the same object is coalesced into an accumulated list of visible rectangles, which represents the screen space for the entire object. In the next section, we describe how the visible space rectangles of two parts of an object are coalesced. This algorithm can be used sequentially on multiple parts to obtain one unified visible space representation for an entire object.

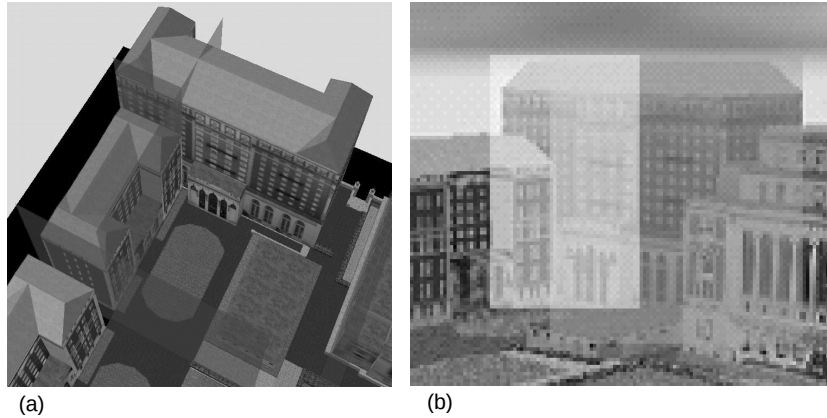


Figure 9. The last building added to the representation in Figure 5(f–g) has two BSP tree object parts. (a) Building in the top part of the image is split by a plane. (b) Projections of both parts are used to determine the visible space for this object. The lighter rectangle represents the projection of the left building part, and the overlapping darker rectangle represents the projection of the right building part.

Coalescing Lists of Largest Visible Rectangles

To coalesce two lists of largest visible rectangles, we merge each visible rectangle from one list with the visible rectangles of the other list. This operation is similar to the space management deletion algorithm (Bell & Feiner, 2000), in which the largest empty-space rectangles that are revealed when a covering rectangle is deleted are combined with all adjacent largest empty-space rectangles. Unlike the deletion algorithm, we combine lists of largest visible rectangles incrementally, which means that some pairs of rectangles being processed may overlap. To handle these cases, the function that combines two largest visible rectangles must support overlap, creating up to two new largest visible rectangles, as shown in Figure 10. Thus, for each rectangle in one list, we find all ways in which it can combine with the rectangles in the other list, and remove those that are enclosed by any other, to determine the largest visible rectangles in the union of the two lists. Figure 11 shows an example of coalescing the two projected

BSP tree object parts from Figure 9. Each part consists of the visible space rectangles that result from processing the projections shown in Figure 7(c–d). The horizontal striped rectangle from the left part is combined with the two vertically striped rectangles from the right part, which results in the three circled rectangles that represent the largest visible spaces for this entire object.

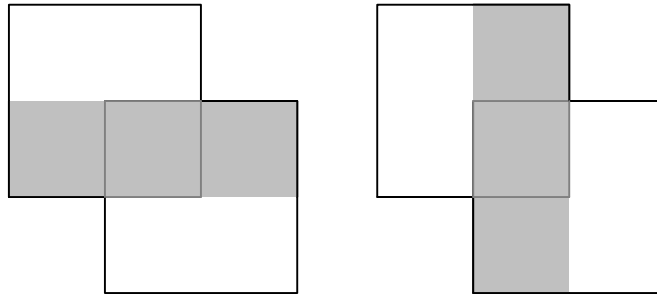


Figure 10. Two overlapping largest rectangles (white boxes) create an additional largest rectangle (grey boxes) in each dimension in which their combined extent is greater than that of either original rectangle.

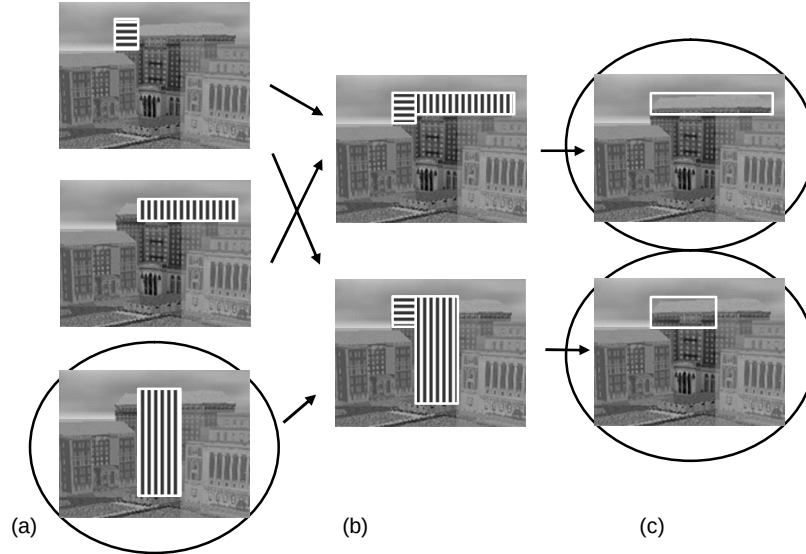


Figure 11. Coalescing lists of visible space rectangles of the two BSP tree parts (from Figure 9) of the center building, which is partially occluded by the buildings highlighted on the left and right. (a) Initial visible spaces of both parts include the horizontal striped rectangle from one BSP tree part and two vertically striped rectangles from the other BSP tree part (resulting from computations in Figure 7c–d). (b) The rectangle (horizontal striped) from one part’s visible representation is combined with the two adjacent rectangles (vertically striped) of the other part. (c) Results of the two combine operations. The three circled rectangles represent the largest visible spaces of the building. The two original rectangles in (a) that are not circled are not included in the largest visible spaces because they are wholly contained within at least one resulting circled rectangle.

The ability to coalesce two sets of rectangles offers the opportunity to combine the visibility results of multiple objects. When laying out a new object, a rectangle can then be selected that lies within one object. Alternatively, the coalescing algorithm can

be used to merge the largest rectangles of multiple objects to create a meta-object that corresponds to the union of those objects. This enables the system to lay out a new object whose projection spans the space of multiple objects. Objects can either be coalesced as they are added to the representation or after the representation is built. If both the original objects and the coalesced objects are desired, the coalesced objects can be created in a second space manager representation.

Since full space objects are represented the same way as empty space, both can be coalesced; for example, to treat certain objects as empty space (e.g., the grass, pavement, and sky shown in the campus view of Figure 4). Note that coalescing full space with empty space is *not* the equivalent of simply not processing the objects when constructing the representation: some objects might obscure parts of others. For example, in our model, the grass and pavement obscure underground infrastructure that would be treated as visible if the grass and pavement were not added.

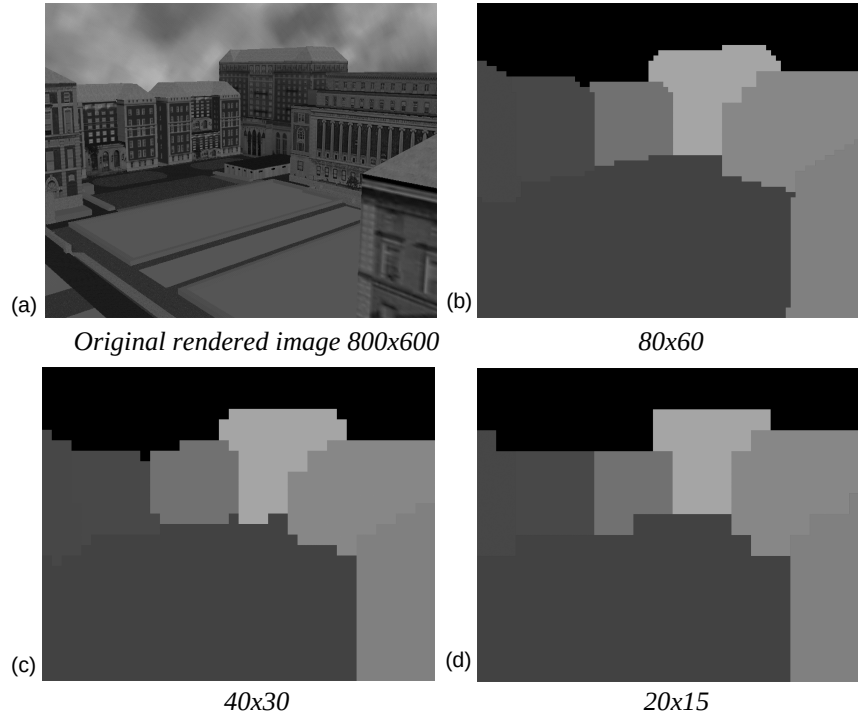


Figure 12. Object buffer rendering. (a) Frame buffer at original resolution. (b–d) Object buffers at different lower resolutions.

COMPUTING VISIBILITY USING A Z-BUFFER

Alternatively, we can compute the 2D spatial representation of the visible regions of objects in a 3D scene using a z-buffer algorithm (Catmull, 1974). By rendering each object in a distinct color with

the current viewing specification, we can use the frame buffer as an *object buffer* (Atherton, 1981), in which each pixel's color encodes the identity of the object visible at that pixel. Figure 12 shows an example of a rendered image (a, upper left) with three object buffers (b–d), each with a different resolution. While higher resolutions provide more accuracy, we need to process these object buffers to convert them into our 2D representation; therefore, performance decreases as the number of pixels to be processed increases.

Note that we must be careful to avoid the problems that can occur when two adjacent polygons in the same object fail to share a vertex on a shared edge, potentially causing missed pixels (Foley et al., 1996). If there are unshared vertices of the polygons on each side of an edge of an object, the pixels close to the edge's projection might not all render as the same object. This can result in a few mis-rendered pixels, which, when converted into our representation, will cause a large area to be improperly split into a

set of smaller ones. To prevent this, we add the necessary extra vertices to polygons in our models.

We use a scan line-order sweep algorithm to convert the resulting pixel representation to the spatial representation that is needed to satisfy our constraints (illustrated in Figure 13, with pseudo-code shown in Figures 14–15). Since it is likely that adjacent pixels will have the same value, we evaluate each pixel, starting at the top left corner of the buffer, moving across the width of the image (in Figure 14, the *for* loops in lines 7–29), with the variables *x* and *y* to keep track of positions in the object buffer, and the variables *pixelX* and *pixelY* to keep track of positions in the rendered image). We also must maintain the current object value (*currentObjectValue*) and the last position at which the pixel value has changed (*previousObjectValue*). Once a change is detected (e.g., Figure 13a, and shown in pseudo-code in Figure 14, lines 17–19) or the end of the scan line is reached (Figure 14, lines 22–28), a rectangle is created from the last to the current position

(with unit height) and is combined with the current object's visible-space representation. The current object and last position are then reset, and the algorithm continues. The function `processRectangle`, which is used Figure 14 and shown in Figure 15, does the necessary processing, once a new rectangle has been detected.

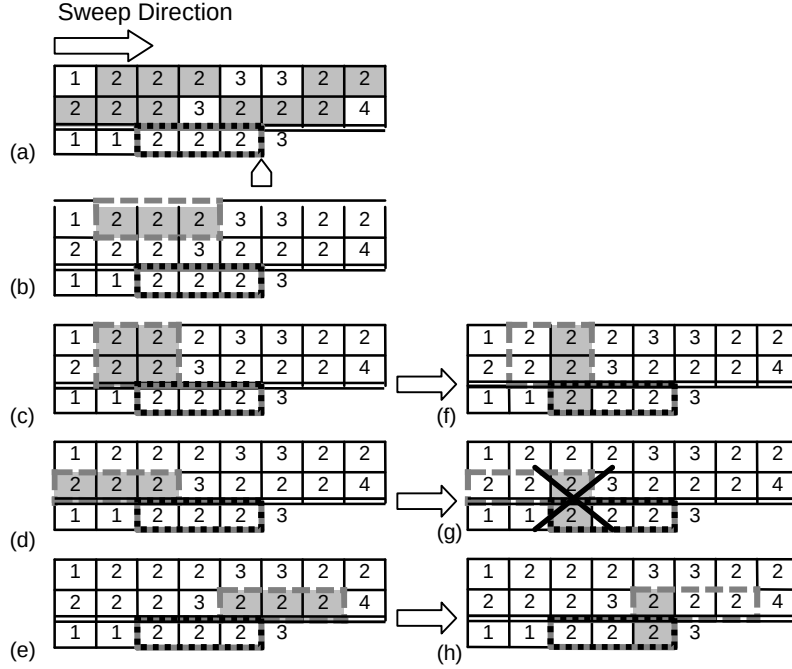


Figure 13. Z-buffer visibility sweep algorithm. At this stage of processing, a new rectangle (outlined in dotted black and grey) is being added to the visible space for the given object labeled "2" in the object buffer representation shown in each subfigure. (a) Visible space for object "2" highlighted in grey. Marker shows the end of the span of pixels for object "2" which initiates the processing of the new rectangle. (b-e) Four highlighted grey rectangles on the left are processed and, if adjacent, are combined with the new rectangle. (f-h) The results of these combinations consist of the three highlighted grey rectangles on the right. Rectangle in (g) is a subset of the rectangle in (f) and is therefore discarded (indicated by an "X"). After the entire z-buffer is processed, the accumulated final visible space representation for each visible object consists of a list of largest space rectangles.

Since rectangles on the current scan line are combined with rectangles from the previous scan line, a one-dimensional interval

tree of all rectangles from the previous scan line for each object is kept for fast lookup (`objectXIT` is an array of 1D interval trees, initialized in Figure 14, line 2). Any rectangle from the last scan line that is not a result of a combine operation is considered part of the final result and does not need further processing. The set of largest visible space rectangles for each visible region consists of all resulting rectangles, including the rectangles that have been combined on the last scan line. These resulting rectangles are kept in variable `objectVPrep`, which is an array of 2D interval trees, initialized in Figure 14, line 1.

Figure 13 shows the z-buffer visibility sweep algorithm at the stage of processing a new rectangle (outlined in dotted black and grey) within a small portion of the object buffer, where the numbers associate each square pixel to a related object in the scene. Each pixel is evaluated in the sweep direction (left to right on rows from top to bottom) to find new rectangles for objects, which are found when the next pixel's associated object is

different from the previous pixel's associated object or the end of the row is reached. In this example, the new rectangle in Figure 13(a) is processed because the next pixel to the right of the pointer is labeled "3". To process the new rectangle, it is added to the visible space associated with the same object labeled "2", which is highlighted in grey in Figure 13(a). The visible space for each object in the scene is accumulated as a list of largest empty space rectangles throughout the sweep process.

To add the new rectangle, it must be compared with each rectangle in the same object's visible-space representation that overlaps it in the sweep direction. In this example, the comparisons are executed on the four solid grey rectangles shown on the left in Figure 13(b–e). The grey highlighted rectangle in Figure 13(b) is not adjacent to the current row (solid double line) and is therefore added to the visible space of the object, since it is no longer needed for processing. (Note that this rectangle could have been detected as no longer needed for processing at the end of the previous line, at

the expense of extra processing at the end of each row.) The grey highlighted rectangles in Figure 13(c–e) are adjacent to the new rectangle, and thus are combined with the new rectangle in the vertical direction to create the three new grey rectangles highlighted in Figure 13(f–h), respectively. The new combined rectangle in Figure 13(g) is a subset of the rectangle highlighted in Figure 13(f) and is therefore discarded (indicated by an “X”, and removed in Figure 15, line 11). After the rectangles in Figure 13(c–d) are processed to produce the proposed rectangles in Figure 13(f–g), they can no longer be adjacent to any more processed space and therefore are added to the visible space of the object (shown in the pseudo-code in Figure 15, lines 13–16). All other computed rectangles in Figure 13(e, f, h) and the new rectangle (outlined in dotted black and grey) are kept for this object in the 1D interval tree for further processing (Figure 15, line 22).

```

0  computeVPRectanglesForEachObjectFromZBuffer
    (double offscreenToScreenRatio){
1  IntervalTree2D objectVPRep [ ] = new IntervalTree2D[N];
2  IntervalTree1D objectXIT[ ] = new IntervalTree1D[N];
3  ImageBuffer buffer = createandRenderObjectBuffer();
4  int width = buffer.getWidth(), height = buffer.getHeight();
5  int startX, lastX, previousObjectValue, currentObjectValue;
6  double pixelX = 0, pixelY = 0;
7  for (y = 0, pixelX = 0; y < height; y++, pixelY += offscreenToScreenRatio){
8      previousObjectValue = -1;
9      for (x=0; x < (width-1); x++, pixelX += offscreenToScreenRatio){
10         currentObjectValue = buffer.getPixel(x,y);
11         if (previousObjectValue== -1){ /* No object in previous pixel */
12             startX = lastX = Math.round(pixelX);
13             previousObjectValue = currentObjectValue;
14         } else if (previousObjectValue==currentObjectValue){
15             /* pixel is same as previous, increment last and continue */
16             lastX = Math.round(pixelX + offscreenToScreenRatio);
17         } else {
18             /* change detected, combine region to all adjacent spaces for object */
19             processRectangle( objectVPRep[currentObjectValue],
20                             objectXIT[currentObjectValue], currentObjectValue,
21                             startX, placeX, placeY, offscreenToScreenRatio);
22             startX = lastX = Math.round(pixelX);
23             previousObjectValue = currentObjectValue;
24         }
25     }
26     /* for the last pixel in each row */
27     currentObjectValue = buffer.getPixel(x,y);
28     if (previousObjectValue != currentObjectValue){
29         processRectangle( objectVPRep[previousObjectValue],
30                         objectXIT[previousObjectValue], previousObjectValue, startX,
31                         Math.round(pixelX), pixelY, offscreenToScreenRatio);
32         processRectangle( objectVPRep[currentObjectValue],
33                         objectXIT[currentObjectValue], currentObjectValue,
34                         Math.round(pixelX), Math.round(pixelX + offscreenToScreenRatio),
35                         pixelY, offscreenToScreenRatio);
36     } else {
37         processRectangle(objectVPRep[previousObjectValue],
38                         objectXIT[previousObjectValue], previousObjectValue, startX,
39                         Math.round(pixelX + offscreenToScreenRatio), pixelY,
40                         offscreenToScreenRatio);
41     }
42 }
43 }
44 for each object O from 0 to N          /* all resulting spaces in interval
45     objectVPRep[O].addAll(objectXIT[O])  tree are spaces for objects */
46 }

```

Figure 14. Pseudocode for the sweep algorithm to convert the discrete object-buffer representation to our 2D space representation. Iterating through each row to detect rectangles to process.

```

0  processRectangle(IntervalTree2D objectVP, IntervalTree1D objectXIT, int objectValue,
    int startX, int placeX, double placeY, double offscreenToScreenRatio){
1      Rectangle newRect = new Rectangle(startX,
    Math.round(placeY - offscreenToScreenRatio),
    placeY, Math.round(placeY));
2      List allNewPotentialRects = new List();
3      allNewPotentialRects.add(newRect);
4      List allOverlappingInXRects = objectXIT.getOverlappingWithoutEdges(newRect);
5      for each rectangle R in allOverlappingInXRects {
6          if (isAdjacentInY(newRect, R)){
7              Rectangle combinedRect = consensusInY(newRect, R);
8              if (R isEnclosedBy combinedRect){
9                  objectXIT.remove(R);
10             }
11             remove any Rectangle in allNewPotentialRects
                enclosed by combinedRect;
12             add combinedRect to allNewPotentialRects if it is
                not enclosed by any Rectangle in allNewPotentialRects;
13             if (R.maxX < placeX){
14                 objectVP.add(R);
15                 objectXIT.remove(R);
16             }
17         } else {
18             objectVP.add(R);
19             objectXIT.remove(R);
20         }
21     }
22     add all Rectangles in allNewPotentialRects to objectXIT;
23 }

```

Figure 15. Pseudo-code for the sweep algorithm to convert the discrete object-buffer representation to our 2D space representation. Processing a new rectangle.

VISIBILITY COMPUTATION PERFORMANCE

The two methods we have described, using a BSP tree and using a z-buffer, have considerably different performance behaviors and results. Depending on the type of 3D application, different methods and algorithms should be used to query the resulting 2D

spatial representations. In this section, we compare these two methods by discussing some measurements of their performance and accuracy. Next, we discuss the benefits and drawbacks of each method to help clarify the applications for which they are most appropriate, and then provide some tips we have found useful for querying the resulting spatial representations.

Figures 16–18 show an evaluation of the performance, results, and accuracy of the two visibility algorithms on a Pentium 4 Mobile 2GHz CPU with 1GB of RAM and an NVidia Quadro 4GoGL. In these examples, the camera is moving around our 3D campus model, which consists of 32 buildings. The camera automatically follows a predetermined path around the campus so that approximately 14.5 buildings on average are visible in every frame rendered. While these results were generated from only one example, representing a single way of controlling the view, they provide us with useful data for discussing the important differences between the algorithms.

Figure 16 shows the performance when computing the visibility for every frame rendered in the evaluation. In this case, the BSP tree performed better than even an extremely low resolution z-buffer (10×7 pixels, shown in the far right column). Since the z-buffer algorithm processes each individual pixel, the resolution has a direct impact on the performance, while the resolution has no effect on the performance of the BSP tree algorithm itself

Performance is correlated with the number of 2D rectangles that are produced in the resulting representation, as shown in Figure 17. As the buffer resolution decreases, so does the number of rectangles in the resulting representation. Although the BSP tree algorithm results in more rectangles than the low-resolution z-buffer, it still performs better. This is because the BSP tree algorithm uses the empty space result, which represents the 2D space that does not contain the visible projections of any objects and requires no further processing. If this region, which we use for outside labels and annotations, is calculated using the z-buffer

algorithm, each unrendered pixel needs to be processed, just like every other region in the frame buffer.

The results of these visibility algorithms are approximations. To measure their accuracy, we compare the visibility in the full-size rendered image (by computing an object buffer of the same size) and compare it with the results of each of the algorithms. Figure 18 shows these comparison results as the percentage of pixels computed by the algorithms that matched the actual visibility determined in the full-size object buffer. The z-buffer accuracy far exceeds the BSP tree projection accuracy, even when used at a very low resolution. In the BSP tree algorithm, the objects are approximated by a low polygonal count to minimize the number of vertices projected and the projections are also approximated as upright 2D rectangles. Incorrect approximation regions typically arise whenever the projections do not project as upright 2D rectangles, and can be addressed by approximating these projections using more rectangles, as shown in Figure 19. Using

more rectangles increases the accuracy of the representation, but also increases processing time.

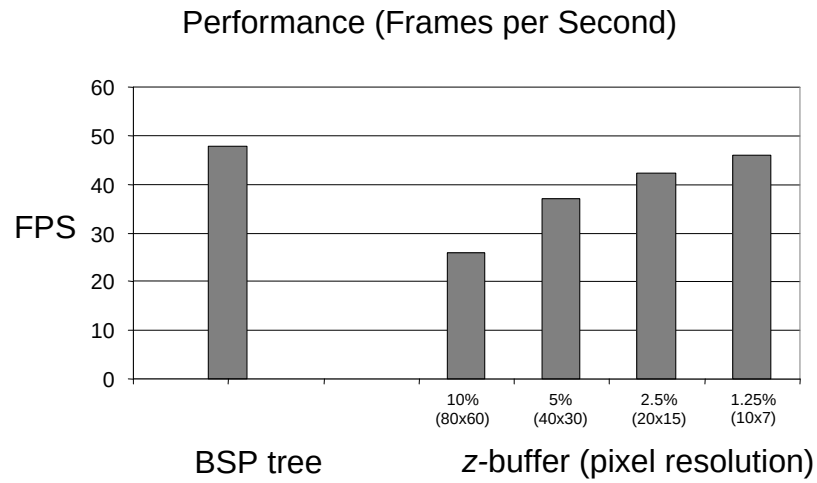


Figure 16. Performance comparison computing visibility using the two methods. The z-buffer is rendered with different resolutions. Fraction of full resolution is shown for each z-buffer resolution.

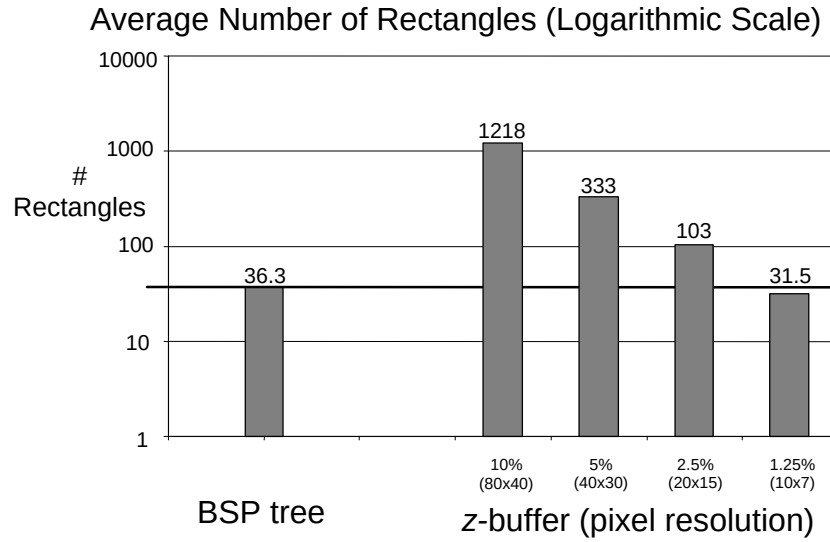


Figure 17. The number of 2D upright rectangles in the resulting 2D representation for the two methods. The z-buffer algorithm is rendered with different resolutions. Fraction of full resolution is shown for each z-buffer resolution.

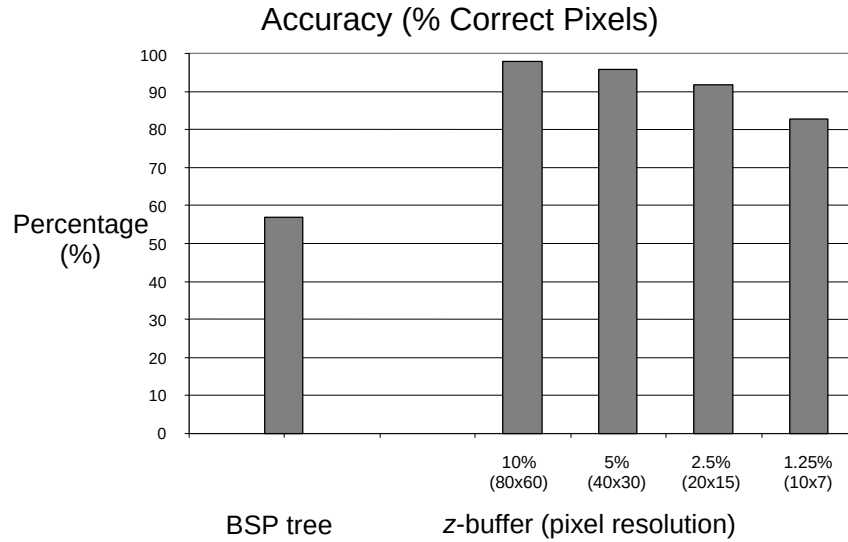


Figure 18. Difference measured between the visibility of objects determined by the algorithms and the actual visibility of objects at final rendered resolution, computed as percentage of pixels with correct visibility. Percentage of full resolution is shown for each z-buffer resolution.

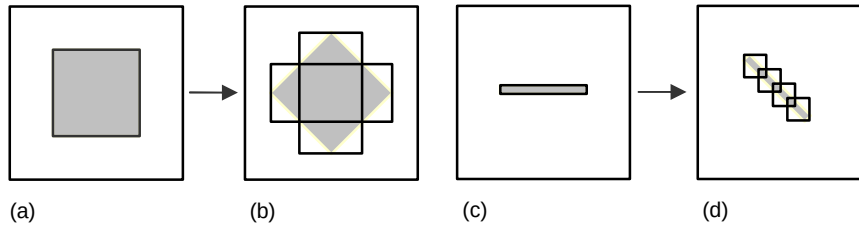


Figure 19. 2D upright rectangular approximations. (a,c) Rectangular projections that do not need better approximations. (b,d) Multiple rectangles (black outlines) can be used to create a better approximation of an object's projection to increase accuracy.

USING REPRESENTATIONS FOR LABELING AND ANNOTATING

The resulting 2D representation supports a variety of layout strategies. Each object is represented by the largest rectangles contained within the visible parts of the object's projection. Thus, an appropriate position and size can be selected for a new object within the visible portion of the projection of any object or group of objects. This makes it possible to perform the equivalent of area feature labeling (Roessel, 1989) in a 3D environment in which objects can partially occlude each other; in this situation, we determine which parts of an object are visible, and select that one that would be best to contain a label. For example, Figure 20 shows an indoor augmented reality application that automatically places labels within the projections of their associated objects. If there is not room within the spatial representation, then an object's label is placed outside but close to the projection of its object.

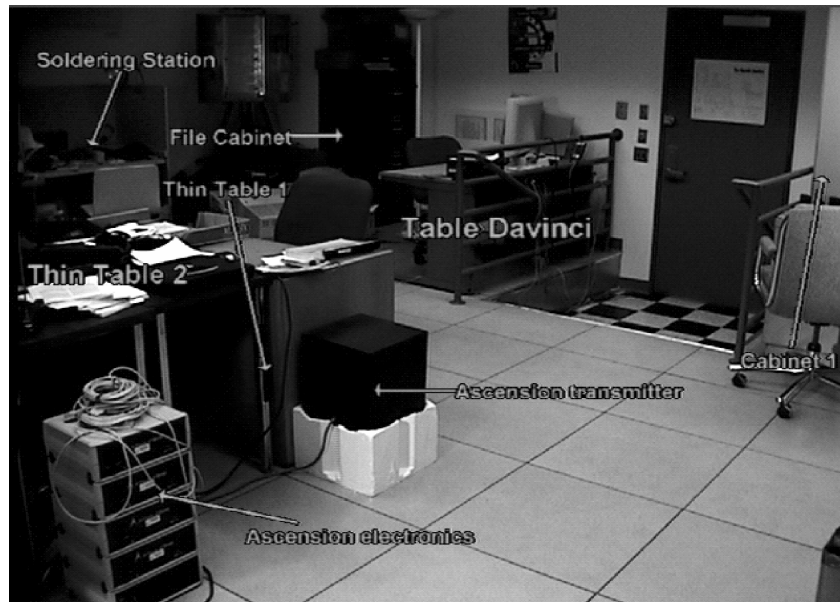


Figure 20. Indoor augmented reality environment with labels automatically placed.

For computing visibility interactively, the benefits of the BSP tree algorithm for static scene geometry far exceed the benefits of using the z-buffer. The frame rate (Figure 16) is most important, since, for typical 3D environments, the BSP tree algorithm produces a small number of rectangles. Alternatively, using the z-buffer with a low resolution has some negative effects, as well. Although the accuracy (Figure 18) remains fairly high, when the resolution is extremely low (e.g., 10×7 pixels), each z-buffer pixel

represents a large number of frame-buffer pixels (e.g., in an 800×600 image, each pixel in a 10×7 z-buffer represents an 80×86 pixel area in the full-resolution frame-buffer). At this coarse z-buffer resolution, it is difficult to detect smaller changes and object movement; for example, if an object moves 50 pixels, it is possible that the visibility representation will not change at all. This produces problems; for example, an arrow that points out an object may not move when it is supposed to move, because its placement is determined by the visibility representation. Furthermore, objects may experience large discontinuities in size and motion between frames in an animation. In contrast, since the BSP tree algorithm computes the projections in floating point coordinates, even slight object motion changes the visibility representation, so that the layouts change accordingly.

Nevertheless, there are interactive applications in which using a z-buffer can be worthwhile. For example, in the outdoor mobile augmented reality application shown in Figure 21, the user is

walking around, looking at buildings and restaurants. Here, the head-worn display has a small field of view, the objects are quite large, and the users are sufficiently close to the objects that many project onto large areas of the screen. In these cases, a high-resolution visibility solution is not needed, and a relatively low-resolution z-buffer can be used.

The z-buffer is also useful for highly dynamic scenes in which many objects are moving relative to each other. Assuming that the entire scene is rendered for each frame, there is no rendering overhead. In contrast, the BSP tree data structure needs to be updated whenever an object is moved to properly determine the front-to-back order for the objects in the scene (Chrysanthou & Slater, 1992).

A high-resolution z-buffer is well suited for labeling and annotating a static image. If computational performance is not an issue, the z-buffer produces a representation with high accuracy (Figure 18). At the expense of increased processing time, view

management is performed more accurately and the labels and other annotations are laid out more precisely.



Figure 21. Outdoor augmented reality environment. (a) Backpack system. (b) Information about the restaurant is seen in the user's view through a head-worn display.

CONCLUSIONS AND FUTURE WORK

We have described two methods for computing visibility and how we use them to compute screen space. Converting the output of these visibility algorithms into our 2D screen-space representation requires a considerable amount of processing, especially since there is no hardware acceleration. For interactive graphics, the

amount of processing is extremely important, since it has a direct relationship to the rendering frame rate. If the frame rate is too low, then the application may not be usable. Therefore, it is important for the programmer to choose the appropriate algorithm based on the needs of the application.

We are extending our view management research to address new ways of placing labels and other annotations, while also affecting how visibility is computed. Instead of placing labels and other annotations relative to the visible projections of entire objects, we wish to place them relative to specific faces of objects, possibly oriented so that each is parallel to its specified face. This requires that visibility information be computed for each of the involved faces.

Performance and scalability are the most challenging issues for computing visibility in a 3D environment. As the number of objects increase, more processing is needed. We plan to examine in more detail how our algorithms scale as a function of the

number of objects. We also intend to apply standard computer graphics techniques based on bounding-volume hierarchies (Rubin & Whitted, 1980; Kay & Kajiya, 1986) to help support view management in large scale environments.

There are situations in which two or more 3D environments must be processed. For example, Figure 22 (Bell et al., 2002) shows a virtual “world-in-miniature” (Stoakley et al., 1995) rendered in front of the real world. In this case, none of the objects in the virtual world-in-miniature requires visibility computations (other than those used for rendering) since there are no associated annotations to lay out. However, the virtual objects in the world-in-miniature block objects in the real world, and thus must be taken into account when computing labels and annotations for the real world. Instead of traversing the BSP tree for the virtual world-in-miniature and projecting each virtual object, we can use an approximation of all their projections. This would improve the performance of our algorithms, since the 2D spatial representation

would have fewer rectangles than if all world-in-miniature objects had been processed.

These optimizations can potentially be applied in many parts of a 3D application, depending on the situation. Therefore, we have begun to develop a rule-based system that a view management component could use to determine when and how to use these techniques to minimize performance cost.



Figure 22. Augmented reality user interface that incorporates a world-in-miniature to give the user a better understanding of their environment.

ACKNOWLEDGEMENTS

This work was supported in part by ONR Contract N00014-04-1-0005, DAFAFRL Contract FA8650-05-2-6647, NSF Grant 0121239, NLM Contract R01 LM06593-01, and gifts from Intel, Microsoft, and Mitsubishi Electric Research Labs. Blaine Bell

was supported during part of this work by an IBM Graduate Research Fellowship. Ryuji Yamamoto created the campus model shown in the figures.

REFERENCES

- Ali, K., Hartmann, K., & Strothotte, T. (2005). Label layout for interactive 3D illustrations. *Journal of WSCG*, 13, 1–8.
- Atherton, P. R. (1981). *A method of interactive visualization of CAD surface models on a color video display*. Proc. SIGGRAPH '81. 279–287.
- Azuma, R. T. (2003). *Evaluating label placement for augmented reality view management*. Proc. Second IEEE and ACM International Symposium on Mixed and Augmented Reality. 66–75.
- Bell, B., & Feiner, S. (2000). *Dynamic Space Management for User Interfaces*. Proc. ACM Symp. on User Interface Software and Technology 2000, San Diego, CA. 239–248.
- Bell, B., Feiner, S., & Höllerer, T. (2001). *View Management for Virtual and Augmented Reality*. Proc. ACM Symp. on User Interface Software and Technology 2001, Orlando, FL. 101–110.
- Bell, B., Feiner, S., & Höllerer, T. (2005). Maintaining Visibility Constraints for View Management in 3D User Interfaces. In O. Stock & M. Zancanaro (Eds.), *Multimodal Intelligent Information Presentation* (pp. 255–277): Kluwers.
- Bell, B., Höllerer, T., & Feiner, S. (2002). *An Annotated Situation-Awareness Aid for Augmented Reality*. Proc. ACM Symp. on User Interface Software and Technology 2002, Paris, France. 213–216.
- Catmull, E. (1974). *A Subdivision Algorithm for Computer Display of Curved Surfaces*. Ph. D Thesis, University of Utah, Salt Lake City, UT.
- Christensen, J., Marks, J., & Shieber, S. (1995). An Empirical Study of Algorithms for Point-Feature Label Placement. *ACM Transactions on Graphics*, 14(3), 203–232.
- Chrysanthou, Y., & Slater, M. (1992). Computing dynamic changes to BSP trees. In *Computer Graphics Forum (Proc. Eurographics '92)* (Vol. 11, pp. 321–332).
- Durand, F., Drettakis, G., & Puech, C. (1997). The visibility skeleton: a powerful and efficient multi-purpose global visibility tool, *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*: ACM Press/Addison-Wesley Publishing Co.
- Foley, J., van Dam, A., Feiner, S., & Hughes, J. (1996). *Computer Graphics: Principles and Practice, 2nd Ed.* in C. Reading, MA: Addison-Wesley.
- Fruchterman, T. M. J., & Reingold, E. M. (1991). Graph drawing by force-directed placement. *Softw. Pract. Exper.*, 21(11), 1129–1164.
- Fuchs, H., Kedem, Z., & Naylor, B. (1980). *On visible surface generation by a priori tree structures*. Proc. SIGGRAPH, Seattle, WA. 124–133.
- Kay, T. L., & Kajiya, J. T. (1986). Raytracing complex scenes. *Computer Graphics (SIGGRAPH '86 Proceedings)*, 20(4), 269–278.

- Lien, J.-M., & Amato, N. M. (2004). Approximate convex decomposition. In *Proceedings of the Twentieth Annual Symposium on Computational Geometry* (pp. 457–458). Brooklyn, New York, USA: ACM Press.
- Mangan, A. P., & Whitaker, R. T. (1999). Partitioning 3D surface meshes using watershed segmentation. *IEEE Transactions on Visualization and Computer Graphics*, 5(4), 308–321.
- Paterson, M. S., & Yao, F. F. (1990). Optimal binary space partitions for orthogonal objects. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms* (pp. 100–106). San Francisco, California, United States: Society for Industrial and Applied Mathematics.
- Roessel, J. (1989). An Algorithm for Locating Candidate Labeling Boxes within a Polygon. *The American Cartographer*, 16(3), 201–209.
- Rubin, S. M., & Whitted, T. (1980). A 3-dimensional representation for fast rendering of complex scenes, *Proceedings of the 7th Annual Conference on Computer Graphics and Interactive Techniques*. Seattle, Washington, United States: ACM Press.
- Samet, H. (1990). *The Design and Analysis of Spatial Data Structures*. Reading, MA: Addison-Wesley.
- Schumaker, R. A., Brand, P., Gilliland, M., & Sharp, W. (1969). *Study for Applying Computer-Generated Images to Visual Simulation* (AFHRL-TR-69-14): U.S. Air Force Human Resources Laboratory.
- Shaw, C., Green, M., Liang, J., & Sun, Y. (1993). Decoupled simulation in virtual reality with the MR toolkit. *ACM Transactions on Information Systems (TOIS)*, 11(3), 287–317.
- Stoakley, R., Conway, M., & Pausch, R. (1995). *Virtual Reality on a WIM: Interactive Worlds in Miniature*. Proc. CHI '95, Denver, CO. 265–272.
- Sutherland, I. E., Sproull, R. F., & Schumacker, R. A. (1974). A Characterization of Ten Hidden-Surface Algorithms. *ACM Comput. Surv.*, 6(1), 1–55.
- Thibault, W., C., & Naylor, B., F. (1987). *Set operations on polyhedra using binary space partitioning trees*. Proc. SIGGRAPH '87. 153–162.
- Warnock, J. E. (1969). *A hidden surface algorithm for computer generated halftone pictures*. Ph.D Thesis, University of Utah.