RC 21973 (98761) 26 February 2001 **Computer Science**

IBM Research Report

Efficient Conservative Visibility Culling Using the Prioritized-Layered Projection Algorithm

James T. Klosowski
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Claudio T. Silva
AT&T Labs-Research
180 Park Ave.
P.O. Box 971
Florham Park, NJ 07932

Research Division
Almaden - Austin - Beijing - Haifa - T. J. Watson - Tokyo - Zurich

Efficient Conservative Visibility Culling Using The Prioritized-Layered Projection Algorithm

James T. Klosowski* Cláudio T. Silva†

Abstract

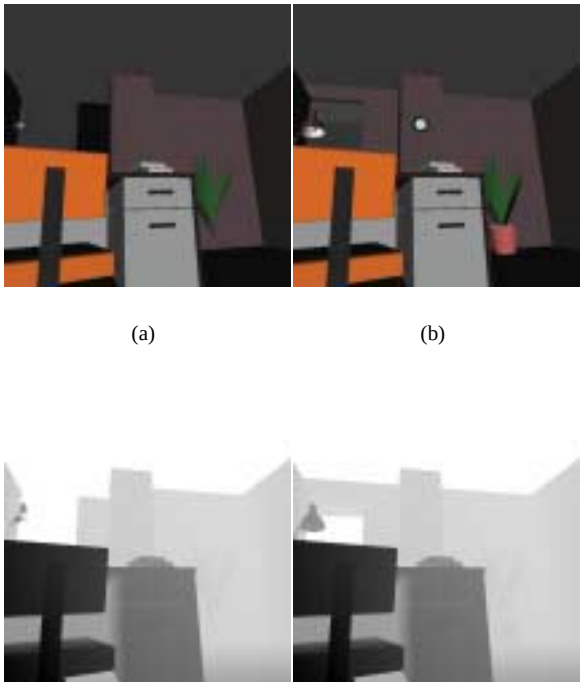
We propose a novel conservative visibility culling technique based on the Prioritized-Layered Projection (PLP) algorithm. PLP is a time-critical rendering technique that computes, for a given viewpoint, a partially correct image by rendering only a subset of the geometric primitives, those that PLP determines to be most likely visible. Our new algorithm builds on PLP and provides an efficient way of finding the remaining visible primitives. We do this by adding a second phase to PLP which uses image-space techniques for determining the visibility status of the remaining geometry. Another contribution of our work is to show how to efficiently implement such image-space visibility queries using currently available OpenGL hardware and extensions. We report on the implementation of our techniques on several graphics architectures, analyze their complexity, and discuss a possible hardware extension that has the potential to further increase performance.

Index Terms Conservative visibility, occlusion culling, interactive rendering

1 Introduction

Interactive rendering of very large data sets is a fundamental problem in computer graphics. Although graphics processing power is increasing every day, its performance has not been able to keep up with the rapid increase in data set complexity. To address this shortcoming, techniques are being developed to reduce the amount of geometry that is required to be rendered, while still preserving image accuracy.

Occlusion culling is one such technique whose goal is to determine which geometry is hidden from the viewer by other geometry. Such occluded geometry need not be processed by the graphics hardware since it will not contribute to the final image produced on the screen. Occlusion culling, also known as visibility culling[1], is especially effective for scenes with high depth complexity, due to the large amount of occlusion that occurs. In such situations, much geometry can often be eliminated from the rendering process. Occlusion culling techniques are usually conservative, producing images that are identical to those that would result from rendering all of the geometry. However, they can also be approximate techniques that produce images that are mostly correct, in exchange for even greater levels of interactivity. The approximate approaches are more effective when only a few pixels are rendered incorrectly, limiting any artifacts that are perceivable to the viewer.



(a) (b) (c) (d)

Figure 1: Office model: (a) This image was computed using PLP and is missing several triangles. (b) The correct image showing all the visible triangles rendered with cPLP. (c) The current z-buffer, rendered as luminance, for the image in (a). Black/white represents near/far objects. (d) Final z-buffer for the correct image in (b).

The Prioritized-Layered Projection (PLP) algorithm, introduced by Klosowski and Silva [16, 17], is one such example of an approximate occlusion culling technique. Rather than performing (expensive) conservative visibility determinations, PLP is an aggressive culling algorithm that estimates the visible primitives for a given viewpoint, and only renders those primitives that it determines to be most likely visible, up to a user-specified budget. Consequently, PLP is suitable for generating partially correct images for use in a time-critical rendering system. To illustrate this approach, consider the images of the office model shown in Fig. 1. The image generated by

*IBM T. J. Watson Research Center, PO Box 704, Yorktown Heights, NY 10598; jklosow@us.ibm.com.

†AT&T Labs-Research, 180 Park Ave., PO Box 971, Florham Park, NJ 07932; csilva@research.att.com.

[1]Visibility culling is also used in a more general context to refer to all algorithms that cull geometry based on visibility, such as back-face culling, view frustum culling, and occlusion culling.

PLP for this viewpoint is shown in Fig. 1(a), while the correctly rendered image is in Fig. 1(b). We can see that the image rendered by PLP is fairly accurate, although portions of the model are missing, including the plant stand, clock, door jam, and parts of the desk lamp.

PLP works by initially creating a partition of the space occupied by the geometric primitives. Each cell in the partition is then assigned, during the rendering loop, a probabilistic value indicating how likely it is that the cell is visible, given the current viewpoint, view direction, and geometry in the neighboring cells. The intuitive idea behind the algorithm is that a cell containing much geometry is likely to occlude the cells behind it. At each point of the algorithm, PLP maintains a priority queue, also called the *front*, which determines which cell is most likely to be visible and therefore projected next by the algorithm. As cells are projected, the geometry associated with those cells is rendered, until the algorithm runs out of time or reaches its limit of rendered primitives. At the same time, the neighboring cells of the rendered cell are inserted into the front with appropriate probabilistic values. It is by scheduling the projection of cells as they are inserted in the front that PLP is able to perform effective visibility estimation.

In [16, 17], PLP was shown to be effective at finding visible primitives for reasonably small budgets. For example, for a city model containing 500K triangles, PLP was able to find (on average) 90% of the visible triangles while rendering only 10% of the total geometry. This number alone does not guarantee the quality of the resulting images, since the missing 10% of the visible triangles could occupy a very large percentage of the screen or may be centrally located so that the incorrect pixels are very evident to the viewer. To address this concern, the authors reported the number of incorrect pixels generated by the PLP algorithm. In the worst case, for the same model and viewpoints discussed above, PLP only generated 4% of the pixels incorrectly. These two statistics support the claim that PLP is effective in finding visible geometry.

As mentioned previously, approximate occlusion culling techniques will sacrifice image accuracy for greater rendering interactivity. While this tradeoff may be acceptable in some applications (especially those that demand time-critical rendering), there are many others (such as manufacturing, medical, and scientific visualization applications) that cannot tolerate such artifacts. The users of these applications require that all of the images generated to be completely accurate. To address the requirements of these applications, we describe an efficient conservative occlusion culling algorithm based upon PLP. Essentially, our new algorithm works by filling in the holes in the image where PLP made the mistake of not rendering the complete set of visible geometry.

An interesting fact is that after rendering PLP's estimation of the visible set, as shown in Fig. 1(a), most of the z-buffer gets initialized to some non-default value, as illustrated by Fig. 1(c). This figure corresponds to the z-buffer rendered as luminance, where black represents near objects, and white represents far objects. If we were to render the cells in the front (see Fig. 3), the visible cells would protrude through the rendered geometry. The techniques we present in this paper are based on incrementally computing which cells in PLP's front are occluded (that is, can not be "seen" through the current z-buffer), and eliminating them from the front until the front is empty. When this condition holds, we know we have the correct image (1(b)) and z-buffer (1(d)).

The use of (two-dimensional) depth information to avoid rendering occluded geometry is not a new idea. The Hierarchical Z-Buffer technique of Greene et al. [14] is probably the best known example of a technique that effectively uses such information. However, even before this seminal paper, Kubota Pacific already had hardware support on their graphics subsystem for visibility queries based on the current status of the depth buffer. In Section 5, we will put our new techniques into context with respect to the relevant related work.

Figure 2: Illustration of the accuracy of PLP: For the same viewpoint and model as shown in Fig. 1, the visible geometry that PLP rendered is shown in white, and the visible geometry that PLP did not render is shown in red.

The main contributions of our work are:

- We propose cPLP, an efficient interactive rendering algorithm that works as an extension to the PLP algorithm by adding a second phase which uses image-space visibility queries.
- We show how to efficiently implement such image-space visibility queries using available OpenGL hardware and extensions. Our implementation techniques can potentially be used in conjunction with other algorithms.
- We discuss the performance and limitations of current graphics hardware, and we propose a simple hardware extension that could provide further performance improvements.

The remainder of our paper has been organized as follows. In Section 2, after a brief overview of PLP and some aspects of its implementation, we detail our new cPLP algorithm. We present several techniques for the implementation of our image-space visibility queries using available OpenGL hardware and extensions in Section 3. We also propose a simple hardware extension to further improve rendering performance. In Section 4 we report on the overall performance of the various techniques on several graphics architectures. In Section 5, we provide a brief overview of the previous work on occlusion culling, followed by a more thorough comparison of our current algorithm with the most relevant prior techniques. Finally, we end the presentation with some concluding remarks.

2 The Conservative PLP Algorithm

The conservative PLP algorithm (cPLP) is an extension to PLP which efficiently uses image-space visibility queries to develop a conservative occlusion culling algorithm on top of PLP's time-critical framework. In this section, we briefly review the original PLP algorithm and then present our cPLP algorithm. Our image-space visibility queries, a crucial part of the implementation of cPLP, are discussed in Section 3.

2.1 Overview of PLP

Prioritized-Layered Projection is a technique for fast rendering of high depth complexity scenes. It works by estimating the visible primitives in a scene from a given viewpoint incrementally. At the heart of the PLP algorithm is a space-traversal algorithm, which prioritizes the projection of the geometric primitives in such a way as to delay rendering primitives that have a small likelihood of being visible. Instead of explicitly overestimating the visible set of primitives, as is done in conservative techniques, the algorithm works on a budget. For each viewpoint, the viewer can provide a maximum number of primitives to be rendered and the algorithm will deliver what it considers to be the set of primitives which maximizes the image quality, based upon a visibility estimation metric. PLP consists of an efficient preprocessing step followed by a time-critical rendering algorithm as the data is being visualized.

PLP partitions the space that contains the original input geometry into convex cells. During this one-time preprocessing, the collection of cells is generated in such a way as to roughly keep a uniform density of primitives per cell. This sampling leads to large cells in unpopulated areas and small cells in densely occupied areas. Originally, the spatial partitioning used was a Delaunay Triangulation [16]; however, an octree has recently been shown in [17] to be a more effective data structure, both in terms of efficiency and ease of use. Since an octree is actually a hierarchy of spatial nodes as opposed to a disjoint partition, we only utilize the set of all leaf nodes of the octree, since these do provide such a partition.

Using the number of geometric primitives contained in a given cell, a *solidity* value ρ is defined, which represents the intrinsic occlusion that this cell will generate. During the space traversal algorithm, solidity values are accumulated by cells based upon the current viewing parameters (viewpoint and view direction), as well as the normal of the face shared by neighboring cells. Using these accumulated values, the traversal algorithm prioritizes which cells are most likely to be visible and therefore should be projected. For a complete treatment of these calculations, please refer to [16, 17].

Starting from the initial cell which contains the viewpoint, PLP attempts to carve cells out of the tessellation. It does this by always projecting the cell in the front F (*the front* is the collection of cells that are immediate candidates for projection) that is least likely to be occluded according to its solidity value. For each new viewpoint, the front is initially empty, and we insert the cell containing (or closest to) the viewpoint. This cell is then immediately projected (since it is the only candidate currently in the front) and as its neighboring cells are inserted into the front, their accumulated solidity values are estimated to reflect their position during the traversal. At the next iteration, the cell in the front most likely to be visible is projected, and its neighboring cells are inserted into the front with appropriate solidity values. If a cell has already been inserted into the front, its solidity values are updated accordingly. Every time a cell in the front is projected, all of the geometry assigned to it is (scheduled to be) rendered.

2.2 The cPLP Algorithm

As previously mentioned, the cPLP algorithm is built on top of PLP. The basic idea is to first run PLP to render an initial, approximate image. As a side effect of rendering this image, two further structures will be generated that we can exploit in cPLP: (i) the depth buffer corresponding to the approximate image, and (ii) PLP's priority queue (front), which corresponds to the cells of the spatial partition that would be rendered next by PLP if it had more time. In cPLP, we will iteratively use the depth buffer to effectively cull the cells in the front until all of the visible geometry has been rendered. The general idea can be summarized as follows:

(1) Run PLP using a small budget of geometric primitives.

This step generates a partially correct image with "holes" (regions of incorrect pixels), the corresponding depth buffer, and the priority queue (front) containing the cells that would be projected next.

(2) While the front is not empty, perform the following steps:

(2a) Given the current front, determine which cells are occluded, using image-space visibility queries, and remove them from the front.

(2b) Continue running PLP, so that each cell in the current front gets projected, since we know that they are all visible.

During this phase, new cells that neighbor the projected cells are inserted into the front as before, although they not candidates for projection during this iteration. We terminate this iteration after each of the original cells (i.e. those in the front after step (2a)) have been projected.

As cells are rendered in step (2b), the holes (and the depth buffer) get filled in, until the image is complete. A nice feature of cPLP is that we know we are done exactly when the front is empty.

One advantage of the formulation given above is that cPLP is able to perform several visibility queries during each iteration. At the same time, the main complication in implementing cPLP comes from the visibility queries in step (2a). This is further discussed in Section 3.

2.3 Challenges

There are primarily three obstacles that cPLP must overcome to be a conservative, interactive rendering algorithm. It must start with a good estimation of the correct image, determine which regions of the estimation are incorrect, and find the remaining visible geometry. Of course, to be truly interactive, each of the solutions to these challenges must be performed very efficiently. This can be done thanks to the way PLP was designed. We discuss each of these issues below.

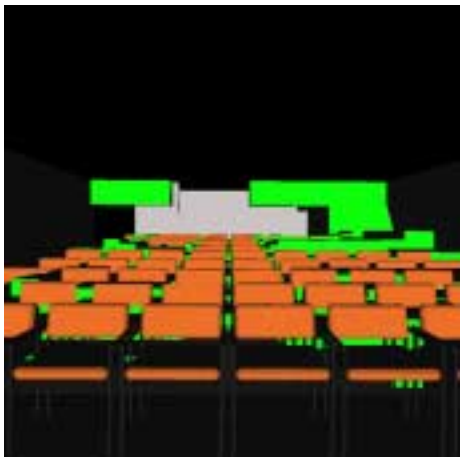


Figure 3: The current front is highlighted in green. By determining where the front is still visible, it is possible to localize the remaining work to be done by our rendering algorithm.

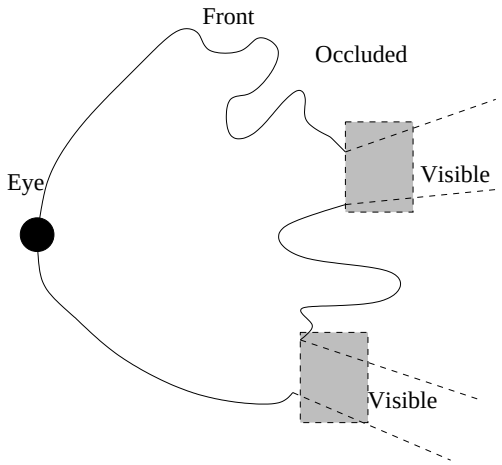


Figure 4: This figure illustrates the technique used in finding the remaining visible cells in cPLP. These cells are found by limiting the remaining work done by the algorithm to only the visible regions.

Estimating the image As demonstrated in [16, 17], PLP is very effective in finding the visible polygons and correctly rendering the vast majority of pixels, even when using relatively small budgets. To illustrate this point, Figs. 1(a) and 1(b) show images of an office model for the PLP and cPLP algorithms. PLP was fairly successful in finding most of the visible geometry for this viewpoint. To better visualize the accuracy of PLP, Fig. 2 highlights the visible geometry that PLP rendered in white, and the visible geometry that PLP did not render in red. By taking full advantage of the accuracy of PLP, our conservative algorithm can quickly obtain a good estimation of the correct image.

This feature can also be used to potentially speed-up other occlusion culling techniques (such as those in [25, 33]), which rely on using the z-buffer values to cull geometry.

Finding the holes As PLP projects cells (and renders the geometry inside these cells), it maintains the collection of cells that are immediate candidates for projection in a priority queue, called the front. Clearly, as the primitives in the scene are rendered, parts of the front get occluded by the rendered geometry. In Fig. 3, we illustrate this exact effect. If no "green" (the color that we used for the front) were present, the image would be correct. In general, the image will be completed, and rendering can be stopped, after all of the cells in the front are occluded by the rendered primitives. Thus, to find the holes in the estimated image, we need only consider the cells in the front.

Filling the holes The final piece that we need to build cPLP is how to complete the rendering once we know what parts of the front are still visible. For this, it is easier to first consider the current occluded part of the front. Basically, we can think of the occluded front as a single occluder (see Fig. 4) that has a few holes (corresponding to the green patches in Fig. 3). Thinking analogously to the work of Luebke and Georges [19], the holes can be thought of as "portals", or reduced viewing frusta, through which all of the remaining visible geometry can be seen. An equivalent formulation is to incrementally determine what cells belong to these smaller view frusta by using an efficient visibility query (discussed below).

3 Implementing Visibility Queries

As previously discussed, to extend PLP into a conservative algorithm, we need to efficiently determine which cells in the front are visible. The visibility queries will take place in image-space and will utilize the current depth buffer. In this section, we first describe three techniques for implementing these queries using available OpenGL hardware and extensions. These include using a hardware feature available on some graphics architectures (such as some Hewlett-Packard (HP) and Silicon Graphics (SGI) graphics adapters), an item-buffer technique that requires only the capability of reading back the color buffer, and an alternative approach that uses an extension of OpenGL 1.2. Then, we discuss some further optimization techniques. Finally, we end this section by proposing a new hardware extension that has the potential to speed up visibility queries even further.

3.1 Counting Fragments After Depth Test

One technique for performing the visibility queries of cPLP is to use the HP occlusion culling extension, which is implemented in their fx series of graphics accelerators. This proprietary feature, which actually seems quite similar to the capabilities of the Kubota Pacific Titan 3000 reported by Greene et al. [14], makes it possible to determine the visibility of objects as compared to the current values in the z-buffer. The idea is to add a feedback loop in the hardware which is able to check if changes *would* have been made to the z-buffer when scan-converting geometric primitives. The actual hardware feature as implemented on the fx series graphics accelerators is explained in further detail in [25, 26]. Though not well-known, several other vendors provide the same functionality. Basically, by simply adding instrumentation capabilities to the hardware which are able to count the fragments which

pass the depth test, any architecture can be efficiently augmented with such occlusion culling capabilities. This is the case for the SGI Visual Workstation series which have defined an extension called `GL_SGIX_depth_pass_instrument` [27, pages 72–75]. Several new graphics boards, such as the SGI InfiniteReality 3 and the Diamond FireGL have such functionality. Even low-cost PC cards such as the 3Dfx Voodoo graphics boards have had similar functionality in their Glide library (basically by supporting queries into the hardware registers). Since the functionality proposed by the different vendors is similar, in the rest of this paper, we concentrate on the HP implementation of such occlusion culling tests.

One possible use of this hardware feature is to avoid rendering a very complex object by first checking if it is potentially visible. This can be done by checking whether a bounding volume *bv*, usually the bounding box of the object, is visible and only rendering the actual object if *bv* is visible. This can be done using the following fragment of C++ code:

```
glEnable(GL_OCCLUSION_TEST_HP);
glDepthMask(GL_FALSE);
glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);
DrawBoundingBoxOfObject();
bool isVisible;
glGetBooleanv(GL_OCCLUSION_RESULT_HP, &isVisible);
glDisable(GL_OCCLUSION_TEST_HP);
glDepthMask(GL_TRUE);
glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
if (isVisible)
  DrawGeometryofObject();
```

This capability is exactly what is required by our cPLP visibility queries. Given the current z-buffer, we need to determine what cells in the front are visible. It is a simple task to use the HP hardware to query the visibility status of each cell.

The HP occlusion culling feature is implemented in several of their graphics accelerators, for example, the fx6 boards. Although performing our visibility queries using the HP hardware is very easy, the HP occlusion culling test is not cheap. In an HP white paper [26], it is estimated that performing an occlusion query with a bounding box of an object on the fx6 is equivalent to rendering about 190 25-pixel triangles. Our own experiments on an HP Kayak with an fx6 estimates the cost of each query being higher. Depending upon the size of the bounding box, it could require anywhere between 0.1 milliseconds (ms) to 1 ms. This indicates that a naive approach to visibility culling, where objects are constantly checked for being occluded, might actually hurt performance, and not achieve the full potential of the graphics board. In fact, it is possible to slow down the fx6 considerably if one is unlucky enough to project the polygons in a back-to-front order, since none of the bounding boxes would be occluded. In their most recent offerings, HP has improved their occlusion culling features. The fx5 and fx10 accelerators can perform several occlusion culling queries in parallel [9]. Also, HP reports that their OpenGL implementation has been changed to use the occlusion culling features automatically whenever feasible. For example, prior to rendering a large display list, their software would actually perform an occlusion query before rendering all of the geometry.

Utilizing the HP occlusion culling feature has proven to be the simplest and most efficient of our three techniques for performing the visibility queries needed by cPLP. Unfortunately, at this time, this hardware feature is not widely available in other graphics boards (for instance, neither of market leaders Nvidia or ATI support this feature). Because of this, we next describe a simple item-buffer technique, whose only requirement is the capability to read back the color buffer. In Section 3.6, we propose a simple extension of the OpenGL functionality which extends the fragment-counting idea, by adding components of the techniques described next.

3.2 An Item Buffer Technique

It is possible to implement visibility queries similar to the ones provided by the HP occlusion test on generic OpenGL hardware. The basic idea is to use the color buffer to determine the visibility of geometric primitives. For example, if one would like to determine if a given primitive is visible, one could clear the color buffer, disable changes to the z-buffer (but not the actual z test), and then render the (bounding box of the) primitive with a well-known color. If that color appears during a scan of the color buffer, we know that some portion of the primitive passed the z test, which means the (bounding box of the) primitive is actually visible.

There are two main costs associated with the item-buffer technique: transferring the color buffer from the graphics adapter to the host computer's main memory and the time it takes the CPU to scan the color buffer. The transfer cost can be substantial in comparison to the scanning cost (see Table 2). Consequently, it is much more efficient to do many visibility queries at once. By coloring each of the cells in the front with a different color, it is possible to perform many queries at the same time.

An unwanted side effect of checking multiple cells is that a cell, *C*, in the front may be occluded by other cells in the front, as opposed to the current z-buffer which contains depth information for the *previously rendered geometry*. This is a problem because although cell *C* is occluded by the other cells in the front, the geometry contained within cell *C* may not be occluded by the geometry within the other cells. A multi-pass algorithm is therefore required to guarantee that a cell is properly marked as occluded. Initially, all cells in the front are marked as "potentially visible". We also disable writing to the z-buffer, so that it remains accurate with respect to the geometry previously rendered by PLP. To retain the color buffer information for this geometry, we save the initial image generated by PLP during step (1) (see Sections 2.2 and 3.5). Each pass of the algorithm then clears the color buffer and renders the boundary of each of the cells in the front that is potentially visible using a distinct color. We then transfer and scan the color buffer to determine which cells are actually visible and mark them. Iterating in this fashion, we can determine exactly which cells are visible with respect to the previously rendered geometry. The remaining cells are determined to be occluded by the previously rendered geometry and need not be considered further. The multi-pass algorithm terminates once the color buffer scan indicates that none of the rendered cells, for the current pass, were determined to be visible. That is, the color buffer is completely empty of all colors. Note that potentially visible cells will need to be rendered multiple times, however, once a cell is found to be visible in one pass, it is marked appropriately and not rendered again. Pseudo-code for the item-buffer technique is included below.

```
glDepthMask(GL_FALSE);
for each cell c in front {
  markCellPotentiallyVisible(c);
}
bool done = false;
while (!done) {
  glClear(GL_COLOR_BUFFER_BIT);
  for each cell c in front {
    if (potentiallyVisible(c))
      renderCell(c);
  }
  glReadPixels(0, 0, width, height,
    GL_RGBA, GL_UNSIGNED_BYTE, visible_colors);
  int cnt = 0;
  for each cell c that appears in visible_colors {
    markCellVisible(c);
    cnt++;
  }
  if (cnt == 0)
    done = true;
}
```

3.3 The OpenGL Histogram Extension

The item-buffer technique just proposed performs a lot of data movement between the graphics accelerator's memory and the host computer's main memory. On most architectures, this is still a very expensive operation, since the data must flow through some shared bus with all of the other components in the computer. We propose a different technique which uses intrinsic OpenGL operations to perform all the computations on the graphics accelerators, and only move a very small amount of data back to the host CPU.

Our new technique shares some similarity to the previous item-buffer technique. For instance, it also needs to render the potentially visible cells multiple times, until no visible cell is found. However, the new method uses OpenGL's histogramming facility, available in the ARB_imaging extension of OpenGL 1.2, to actually compute the visible cells (see [1]). After rendering the potentially visible cells in this case, rather than transferring the color buffer to the host's CPU and scanning it for the visible cells, we simply enable the histogramming facility and transfer the color buffer into texture memory (still on the graphics accelerator). During this transfer, OpenGL will compute the number of times a particular color appears. A short array with the accumulated values can then be fetched by the host CPU with a single call. A fragment of our C++ code illustrates this approach.

```
glEnable(GL_TEXTURE_2D);
glEnable(GL_HISTOGRAM_EXT);
glHistogramEXT(GL_HISTOGRAM_EXT,  256,
    GL_LUMINANCE, GL_TRUE );
glCopyTexSubImage2D(GL_TEXTURE_2D, 0,
    0, 0, WIDTH, HEIGHT, WIDTH, HEIGHT);
GLuint histogram_values[256];
glGetHistogramEXT(GL_HISTOGRAM_EXT,  GL_FALSE,
    GL_LUMINANCE, GL_UNSIGNED_INT,
    histogram_values);
glResetHistogramEXT ( GL_HISTOGRAM_EXT );
glDisable(GL_TEXTURE_2D);
glDisable(GL_HISTOGRAM_EXT);
```

After this code is executed, the array histogram_values contains the number of times each color (here uniquely identified by an integer between 0 to 255) appeared. With this technique, the graphics board does all the work, and only transfers the results to the host CPU. The same termination criterion exists for this multi-pass algorithm as for the item-buffer technique, although we can more easily test for this condition in this case. For instance, if histogram_values[0] is equal to WIDTH $\times$ HEIGHT, meaning all pixels are the same (background) color, then no cells are visible and we terminate the algorithm.

3.4 Improving Visibility Query Performance

It is possible to improve the performance of our visibility query techniques by implementing several optimizations. The previous two techniques need to perform operations that touch all the pixels in the image, possibly multiple times. To avoid computations in areas of the screen that have already been completely covered, we have implemented a simple tiling scheme that greatly reduces the amount of transfers and scans required. The basic idea is to simply divide the screen into fixed tiles. For a 512x512 pixel image, we could break the screen up into 64 tiles, each containing a block of 64x64 pixels. During the multi-pass algorithm, we need to keep track of the active tiles, those that in the previous iteration contained visible primitives. After each iteration, tiles get completed, and the number of tiles which need to be rendered to and scanned decreases.

Another simple optimization for the item-buffer technique was to minimize the number of color channels to transfer to the host computer's main memory. For example, if we have r bits to represent the red color component on our machine, and we have fewer than 2^r cells to check in the front, we can uniquely color these cells using only the red color component. Consequently, we would only need to transfer and scan the GL_RED component for each pixel in the image, as opposed to transferring and scanning the entire GL_RGBA component.

We have implemented and are currently using these two optimizations. A non-conservative optimization for our techniques would be to compute visibility in a lower resolution than the actual rendering window [33]. Although a quite effective optimization, this might lead to undesirable artifacts. This is one of the reasons we do not use it in our system.

3.5 Integration with cPLP

The techniques presented so far essentially solve step (2a) of cPLP. Both the item-buffer technique as well as the histogramming technique need to have access to the color buffer of the machine being used for its computations. For each pass, they require that the color buffer be cleared, which conflicts with the image computation which is performed in steps (1) and (2b). Naively, it would be necessary to save the complete color buffer (or at least the active tiles) before each call to step (2a) and restore it before the call to step (2b).

Instead, since we expect that after step (1) most of the visible triangles have been rendered, we simply save the image step (1) generated, and ignore the changes to the color buffer from then on (we re-rendered the extra geometry in the end to recover the correct image). The important thing is to correctly account for the z-buffer changes that are triggered by the rendering of the geometry inside the cells. To do this, before step (2b), we change the masks on the z-buffer so that it gets updated as geometry is rendered in (2b). When the front becomes empty, we know the z-buffer was completed. At that point, we perform a single image restore (with the image we

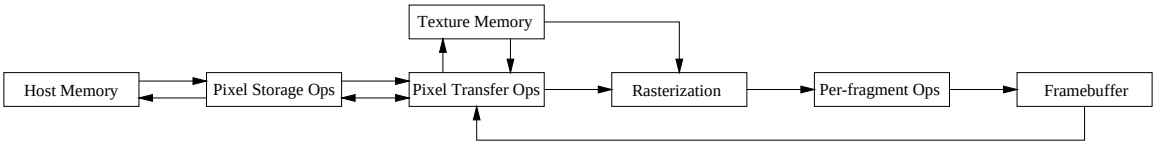


Figure 5: OpenGL imaging pipeline

saved in step (1)), and we re-render all the geometry that was found to be visible since that point.

Fig. 10 provides an overview of our cPLP algorithm as described. For a sample view of an office model, snapshots were taken at several iterations (step 2) of our algorithm. Figs. 10(a)-(c) illustrate the current color buffer and front (in blue) at each iteration. The remaining visible geometry will come from within the visible front cells. (d)-(f) illustrate the tiles of the screen that have been completed and therefore do not need to be scanned during subsequent iterations. (c) and (f) correspond to the final (correct) image, since all of the tiles have been completely covered. Note that in (b), the front cells, which are barely visible, are in the upper left corner and near the two desks in the middle of the screen. As expected, the tiles that represent these areas are not marked as completed.

3.6 Extending the OpenGL Histogram

Here we propose a modification to OpenGL that has the potential to greatly improve performance. In particular, it would make it possible to avoid the costly multi-pass visibility computations that we are currently forced to use, and it can be seen as a generalization of the HP occlusion culling test.

OpenGL background Before we go into details, it helps to understand a bit more on how OpenGL works. The graphics pipeline is the term used for the path a particular primitive takes in the graphics hardware from the time the user defines it in 3D to the time it actually contributes to the color of a particular pixel on the screen. At a very high level, a primitive must undergo several operations before it is drawn on the screen. A flowchart of the operations is shown in Fig. 5.

The user has several options for specifying vertices that are grouped into primitives, e.g., triangles or quads. Primitives go through several stages (not shown), and eventually, get to the rasterization phase. It is at rasterization that the colors and other properties of each pixel are computed. During rasterization, primitives get broken into what we usually refer to as "fragments". Modern graphics architectures have several per-fragment operations that can be performed on each fragment as they are generated. As fragments are computed, they are further processed, and the hardware incrementally fills the framebuffer with an image.

Per-Fragment Histogramming The OpenGL histogramming facility, part of the pixel transfer operations shown in Fig. 5, operates on images, which can potentially come from the framebuffer. The OpenGL histogram works by counting the number of times a color appears in a given image.

The reason we need to perform multiple passes to determine when cells are visible at this time is that we are using the color buffer to find which of the primitives passed the z-test. With the standard pipeline, we only get the "top layer" of visible cells, since one of the per-fragment operations that occurs before a pixel is written to the color buffer is the depth-test. If a per-fragment histogramming facility is added to the pipeline and it could be used to perform the same exact operation on *fragments* (which pass the z-test), it would be possible to count how many fragments of a given primitive passed the z-test. If this number is zero, the primitive would be occluded, otherwise, the histogram value would not only tell us that it is visible, but actually provide an upper bound on the number of its pixels that are visible. With the proposed change in the OpenGL pipeline, we would still be able to perform several queries at the same time, but we would not be required to perform multiple passes over the framebuffer.

The per-fragment histogramming functionality we are proposing is a clean way to extend the (already useful) techniques based on counting the number of fragments which pass the z-test (such as the HP occlusion culling test), so that it is able to handle multiple and more general tests with better performance. We would like to point out that the hardware cost (in component cost or chip area) would likely be non-trivial, since high-performance graphics hardware is highly parallel (for instance, Nvidia's GeForce can compute four fragments simultaneously), and the extra hardware for the per-fragment histogramming would have to be replicated for each fragment generator. Of course, this is already the case for several other extensions, including the existing fragment counting hardware. We believe the actual cost (in time) of our augmented test would be similar to the cost of a single HP test, while we would be able to perform several tests concurrently.

4 Experimental Results

We performed a series of experiments to determine the effectiveness of our new cPLP algorithm. We report results for each of the three implementations of our visibility queries presented in Section 3, as well as several alternatives for benchmarking:

cPLP-HP: cPLP, using the HP occlusion culling extension,

cPLP-IB: cPLP, using the item-buffer technique,

cPLP-HG: cPLP, using the OpenGL histogram extension,

cPLP-EXT: cPLP, using our hardware extension proposed in Section 3.6,

PLP: the original PLP,

VF-BF: view frustum and back-face culling only,

HP: using the HP hardware to perform the visibility queries without the benefit of running PLP to preload the color and depth buffers.

Machine	CPU(s)	Graphics	RAM
SGI Octane	1 X R12000, 300MHz	MXE	512MB
SGI Onyx	12 X R10000, 195MHz	Infinite Reality	2GB
HP Kayak	2 X Pentium II, 450MHz	fx6	384MB

Table 1: The configurations of the machines used in our experiments. The number of processors *P* per machine is listed in the CPU(s) column, in the form: P X cpu-type, cpu-speed.

Test model The primary model that we report results on is shown in Fig. 9(a) and consists of three copies, placed side by side, of the third floor of the Berkeley SODA Hall. Arranging the copies in such a way helps us better understand how the different occlusion culling techniques function in a high depth complexity environment, since they have their greatest opportunity where there is significant occlusion. Each room in the model has various pieces of furniture and in total, the three replicas contain over one million triangles.

We generated a 500-frame path that travels right-to-left, starting from the upper right corner of Fig. 9(a). In Fig. 9(b)–(e), we show a few representative frames of the path. The number of visible polygons in each frame varies considerably, especially when moving from room to room.

Machine architectures Our experiments were performed on a three different architectures: an SGI Octane, an SGI Onyx, and an HP Kayak. The configurations of the machines are listed in Table 1.

Preprocessing As discussed in Section 2, the preprocessing step of cPLP, which is identical to the preprocessing step of the original PLP algorithm, is very efficient. The preprocessing includes reading the input geometry from a file, building the octree, determining which geometry each cell contains, and computing the initial solidity values. The total preprocessing times for the one million triangle model mentioned above was 76 seconds, 128 seconds, and 90 seconds, for the Octane, Onyx, and Kayak, respectively. While these times are actually quite modest, we have an additional opportunity to reduce the preprocessing requirement. For portability purposes, we are currently using an ASCII format to store the model. For each of the three machines being used, at least half (42, 64, and 56 seconds, respectively) of the preprocessing time listed above was spent simply reading in the model. If we were to store the model in a compact binary format, the input portion of the preprocessing would likely be reduced considerably. The octree construction, geometry assignment, and initial solidity computation only required 34, 64, and 34 seconds, respectively, on each of the three machines, and could likely be reduced by carefully optimizing our code. For the experiments reported here, we subdivided the octree until each leaf contained fewer than 5000 triangles. This resulted in 1429 octree leaf cells being created.

Rendering results We present our main rendering results for the various cPLP implementations in Fig. 6. The vertical axis represents the average rendering time for each of the 500 steps in the path generated for the test model. The horizontal axis represents the initial budget used by PLP to render what it determined to be the most likely visible geometry, thereby preloading the color and depth buffers.

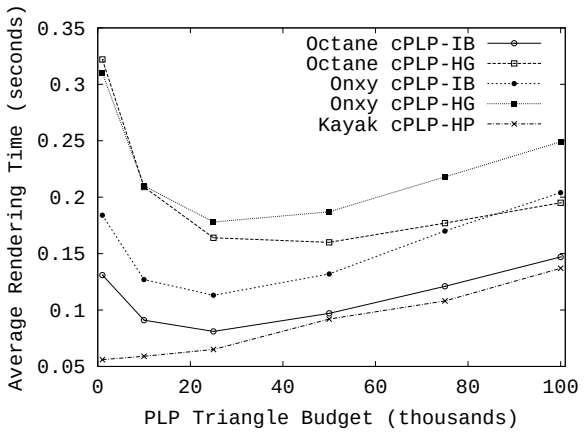


Figure 6: Average rendering times per frame for the implementations of the cPLP algorithm. The PLP budget, reported in thousands of triangles, determines the number of triangles initially rendered to fill-in the depth buffer.

If we compare the item-buffer and histogram techniques, we see that the item-buffer is considerably faster on each of the SGI machines. All of these runs[2] tended to reach their minimum values for an initial PLP budget of 25K triangles, or roughly 2.5% of the total number in the model. For this budget, the rendering times for the item-buffer technique on the Octane and Onyx were 0.081 and 0.113 seconds on average per frame. This is equivalent to rendering 12.35 and 8.85 frames per second, respectively. In comparison, the histogram approach took 0.164 and 0.178 seconds on average per frame, or the equivalent of 6.10 and 5.62 rendered frames per second.

We did not run cPLP-HG on the Kayak since the OpenGL histogram extension is not available on that machine. Also, the cPLP-IB technique on the Kayak was very slow, requiring 0.864 seconds on average per frame. We explain why this is the case when we discuss the costs of the primitive operations for each of the techniques below. The HP hardware occlusion culling extension was clearly not available on the SGIs, and so we can only report on this technique on the Kayak.

cPLP-HP was the most efficient algorithm but we were a little surprised by the fact that it increased in running time as we increased the PLP budget. We anticipated that we would see a parabolic curve similar to the runs on the two SGI machines. Initially, we considered that running PLP followed by our cPLP-HP visibility queries was not benefitting us at all on the Kayak. To test this hypothesis, we implemented another technique, HP, that used the hardware occlusion culling extension without the benefit of running PLP first to preload the depth buffer. Given the set of leaves in our octree, we first discarded those nodes that were outside the view frustum, and then sorted the remaining nodes according to their distance from the viewpoint. We then performed visibility queries for the nodes in this order. On average, the HP technique required 0.157 seconds per frame, which is considerably slower than our cPLP-HP algorithm.

While sorting the nodes according to distance appeared to

[2]The only exception being the Octane cPLP-HG method, which reached a minimum at a PLP budget of 50K triangles.

be a good technique, it clearly cannot capture any occlusion information as did cPLP. In addition, this HP technique does not have a mechanism for determining which nodes are still visible and which sections of the screen are yet incomplete. Consequently, this method cannot easily determine when it is finished, and therefore must perform many more visibility queries than the cPLP-HP technique. One could think of modifying this HP approach so that the queries are performed in a hierarchical fashion since we have the octree constructed anyway. However, while in some cases this could reduce the overall rendering time, in many others the times will increase due to the increase in the number of visibility queries. We shall discuss shortly the times required for the HP visibility queries. Thus, although the benefit gained from PLP was not exactly as we anticipated, it still plays a crucial role in achieving interactive rendering times.

To quantify how well our conservative culling algorithm is working, we implemented a simple rendering algorithm, VF-BF, that performed only view frustum and back-face culling. These traditional culling approaches were also used within cPLP. The VF-BF algorithm is considerably slower than all of the cPLP implementations. For example, on the Octane, VF-BF took 0.975 seconds to render each frame on average. Thus, our cPLP-IB and cPLP-HG methods render frames 12 and 6 times faster than the VF-BF technique. Our cPLP-HP method provides even better comparisons. Such improvements in rendering speeds, which were similar on all of the architectures, are crucial for any application requiring interactivity.

Of the time spent by our cPLP approaches, a good portion of that time was actually spent running the initial PLP algorithm. For example, on the Octane, out of the 0.081 seconds it takes to render a frame on average, 0.064 seconds were occupied by the initial PLP algorithm, and 0.017 seconds used by the iterative visibility queries to complete the rendered image. For the item-buffer and histogram techniques, the average number of iterative visibility queries per frame ranged from 4.7 iterations, for an initial PLP budget of only 1000 triangles, to 1.5 iterations, for an initial budget of 100000 triangles.

Primitive Operation Costs To better understand the rendering times reported in Fig. 6, we analyzed the cost of performing the underlying primitive operations for each of the methods. By looking at these results, we can offer additional insight into why each of the methods works as well, or as poorly, as it does.

For the cPLP-HP technique, the visibility queries involve enabling the HP culling extension, rendering a cell, and reading back the flag to indicate whether the z-buffer would have changed if we had actually rendered the cell. We timed the visibility queries on the HP Kayak and found that the time ranged between 100 microseconds (μs) and 1000μs. In addition to these costs, the HP visibility query can also interrupt the rendering pipeline, thereby reducing the overall throughput. Consequently, it is imperative when using these queries to do so with some caution. It is especially advantageous when you are very likely to find significant occlusion. Otherwise, many queries may be wasted and the overall rendering performance will be reduced.

The primitive operation for the item-buffer technique is the transferring of the color buffer from the graphics accelerators memory to the main memory of the host computer. This is done in OpenGL using a single call to glReadPixels. The other main cost associated with this technique is the time it takes the CPU to scan the color buffer to determine which cells have actually contributed to the image. We report these numbers for each of our machines in Table 2. It is immediately apparent why the cPLP-IB technique on the Kayak is so slow. The transfer and scan times are considerably slower (for the 512x512 image) than on the SGIs. Another interesting observation, which also helps justify our tiling optimization in Section 3.4, is the substantial increase in time that is required to transfer and scan a 512x512 pixel image, as opposed to only a 64x64 pixel (sub)image.

Machine	SGI Octane		SGI Onyx		HP Kayak	
Image Size	64^2	512^2	64^2	512^2	64^2	512^2
Transfer	217	4483	564	7733	375	11250
Scan	30	2300	20	1000	47	3430
Total	247	6783	584	8733	422	14680

Table 2: Times for the primitive operations of the item-buffer technique. An image size of 64^2 refers to an image that is 64x64 pixels in size. The transfer time is the dominant cost of this method. All times are reported in microseconds.

For those machines that support the OpenGL histogram extension, the underlying operations include copying an image, or sub-image in the case of our tiles, from the framebuffer to texture memory. We have timed this operation with the histogram extension enabled to see how much time is required for the copy with the histogram calculations. The histogram calculation also includes the time to retrieve and scan the histogram results. On the Octane it takes 800μs for a 64x64 pixel image, and 34000μs for a 512x512 image. On the Onyx, it takes 690μs for a 64x64 pixel image, and 13500μs for a 512x512 image. (We should note that it is quite difficult to perform such measurements, but we have done our best to report accurate results.) We were surprised by the amount of time required to copy the image to texture memory and perform the histogram computations. Our initial belief was that by using the actual hardware to perform our visibility queries, our rendering times would decrease. Unfortunately, this is not the case at this point in time. While the Onyx appears to be more advanced than the (newer) Octane in its histogramming features, neither machine performs well enough to be faster than the item-buffer techniques.

Depth Complexity To further test our cPLP algorithms, we considered another model with extremely high depth complexity. Fig. 7 shows an interior view of a skyscraper model which consists of over one million triangles. The model, courtesy of Ned Greene, consists of 54 copies of a module, each with almost 20K triangles.

The purpose of this experiment was to determine the depth complexity of this model when rendering it using the various techniques. By depth complexity, we refer here to the average number of times a z-test is performed for each pixel in the image. If our cPLP techniques are effective at determining occlusion, our methods should reduce the depth complexity considerably in comparison to a standard rendering algorithm. Using one such technique, VF-BF, we determined the depth complexity of this model (for this viewpoint) to be 26.70 on average, for all of the pixels in the image. Using cPLP, we were able to reduce this value to only 7.97. We emphasize that these numbers refer to the number of z-tests per pixel, as opposed to the number of z-tests that pass (i.e., resulting in the pixel's color being overwritten by a fragment that is closer

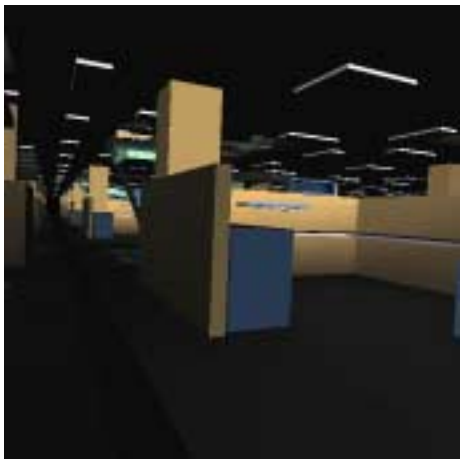


Figure 7: Interior view of a skyscraper model. cPLP reduced the depth complexity of this rendered image from 26 to 8.

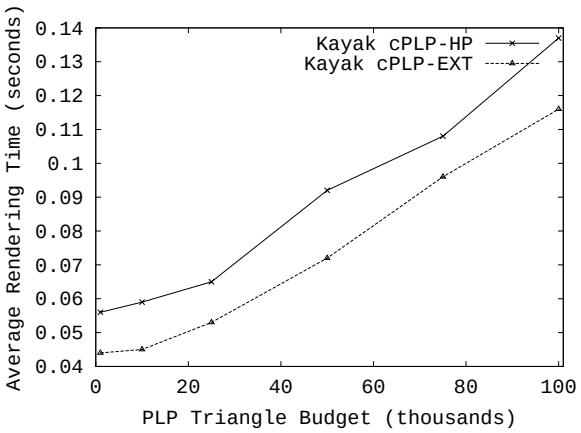


Figure 8: Average rendering times per frame for cPLP-HP and our proposed hardware extension method cPLP-EXT. The PLP budget, reported in thousands of triangles, determines the number of triangles initially rendered to fill-in the depth buffer.

to the viewer), which has been reported in other approaches. We opted for this number since the number of z-tests more accurately reflect the work that is done during the rendering algorithm.

cPLP-EXT Since we do not actually have hardware which implements our proposed extension, here we extrapolate on its performance based on the results we have, assuming we were to add such an extension to the HP Kayak fx6. Using cPLP-IB, it is possible to determine the number of tests that can be performed in parallel for each triangle budget in Fig. 6. Assuming our extension is properly implemented, we believe it should take no more time than the fragment counting technique already available on several architectures. While measuring on HP machines, we found that in the worst case, an occlusion test costs 1 ms. But since we have to bring more data from the graphics hardware for our extension, we will assume that each query is twice as expensive, or 2 ms, to account for the extra data transfer. (Since only extremely small arrays of 256 values are being transfered, we don't actually believe it would have such an impact.)

Table 3 summarizes our findings. Basically, we are computing the time for cPLP-EXT as a sum of the initial PLP cost (initialize its per-frame data structures, such as zeroing the solidity of each cell; and rendering the first batch of triangles for all frames), plus the total number of parallel EXT tests (which we assume take 2 ms each), plus the time to rendering the extra triangles (at a rate of approximately 1 million triangles/sec) which are found as visibility tests are performed.

With these assumptions, we can see that our frame rates get considerably better (see Fig. 8), and we could potentially achieve a frame rate of 23 Hz (versus 18 Hz for cPLP-HP; an improvement of 28%) if we had a hardware implementation of our extension. We would like to point out that the advantage would be even greater if the cost of initializing PLP's per-frame data structures was made lower. Our current PLP implementation uses an STL set, which is not particularly optimized for linear traversals which are necessary during initialization. If necessary, it would be possible to optimize this code further.

5 Related Work

There has been a substantial amount of recent work on occlusion culling (see, for instance, [5, 6, 8, 11, 18, 23, 24, 30, 31]). The purpose of this section is not to do an extensive review of all occlusion culling algorithms. For that, we refer the interested reader to the recent surveys by Cohen-Or et al. [7] and Durand [10]. Instead, we focus on reviewing work that is more closely related to our own, so that we can indicate the similarities and differences with our current work.

Closely related to our work are techniques that use two-dimensional depth information to avoid rendering occluded geometry. An early example of this is a technique by Meagher [20] that stores the scene in an octree, and the framebuffer in a quadtree. Meagher renders the octree in a strict front-to-back order, while keeping track of which parts of the quadtree get filled, in order to avoid touching parts of the octree that can not be seen. Naylor [22] proposes another version of this idea, where instead of using an octree and a quadtree, he uses two binary-space partitioning trees [12], one in 3D, the other in 2D, to efficiently keep both the scene and the image respectively. The 3D BSP can be used to traverse the scene in a strict front-to-back order, and the 2D BSP is used to keep the areas of the screen which get filled. Our current approaches differ from these methods in that they do not render in a strict front-to-back order (which was shown to be less effective), but rather allow PLP to determine the order in which to visit (and render) the cells.

The Hierarchical Z-Buffer (HZB) technique of Greene et al. [14] is probably the best known example of a technique that efficiently uses depth information for occlusion culling. Their technique is related to Meagher [20] in that it also uses an octree for managing the scene, which is rendered in front-to-back order. Another similarity is that they also use a quadtree, but not for the actual framebuffer (as in [20]). Instead, they use the quadtree to store the z-buffer values, which allow for fast rejection of occluded geometry. The HZB technique also explores temporal coherency by initializing the depth buffer with the contents of the visible geometry in the previous frame.

PLP Budget (triangles)	PLP Time (s)	# EXT Tests	Avg. Extra Triangles	Average Time (s)	Frame Rate (Hz)
1,000	0.019	4.688	16844	0.044	22.7
10,000	0.028	3.376	10978	0.045	22.2
25,000	0.043	2.426	5641	0.053	18.9
50,000	0.066	1.908	2796	0.072	13.9
75,000	0.091	1.630	1770	0.096	10.4
100,000	0.112	1.372	1247	0.116	8.6

Table 3: Performance of cPLP-EXT on a "hypothetical" HP Kayak fx6. All times are reported in seconds. The average extra triangles are the number of triangles that get rendered in addition to the PLP budget. See text for further details.

The Hierarchical Z-Buffer has several similarities to cPLP. Their use of the visible geometry from the previous frame for the purpose of estimating the visible geometry is similar to our approach, although in our case, we use the visibility estimation properties of PLP to estimate the current frame. One advantage of doing it this way is that (as we have shown earlier) the front intrinsically tells us where to continue rendering to fill-up the z-buffer. HZB has no such information; it renders the remaining geometry in front-to-back order. The fact that we employ a spatial partitioning instead of a hierarchy in object-space is only a minor difference. Depending upon the scene properties, this may or may not be an advantage. The flat data structure we use seems more efficient for a hardware implementation, since we do not need to stop the pipeline as often to determine the visibility of objects. In [13], Greene introduces an optimized variation of the HZB technique, including a non-conservative mode.

A closely related technique is the Hierarchical Occlusion Maps of Zhang et al. [33]. For each frame, objects from a precomputed database are chosen to be occluders, and are rendered (possibly) in lower resolution to get a coverage footprint of the potential occluders. Using this image, OpenGL's texture mapping functionality generates a hierarchy of image-space occlusion maps, which are then used to determine the possible occlusion of objects in the scene. Note that in this technique, the depth component is considered after it is determined that an object can potentially be occluded. One of the main differences between HOM and cPLP is that HOM relies on preprocessing the input to find its occluders, while cPLP uses PLP for that purpose. HOM also utilizes a strict front-to-back traversal of the object-space hierarchy.

The work by Bartz et al. [3, 4] addresses several of the same questions we do in this paper. They provide an efficient technique for implementing occlusion culling using core OpenGL functionality, and then propose a hardware extension which has the potential to improve performance. Similar to the previous methods, Bartz et al. use a hierarchy for the 3D scene. In order to determine the visible nodes, they first perform view-frustum culling, which is optimized by using the OpenGL selection mode capabilities. For the actual occlusion tests, which are performed top-down in the hierarchy nodes, they propose to use a *virtual occlusion buffer*, which is implemented using the stencil buffer to save the results of when a given fragment has passed the z-test. In their technique, they need to scan the stencil buffer to perform each visibility test. Since this has to be performed several times when determining the visible nodes of a hierarchy, this is the most time consuming part of their technique, and they propose an optimization based on sampling the virtual occlusion buffer (thus making the results only approximate). In their paper, they also propose an extension of the HP occlusion culling test [25] (see [3] for details). At this time, the HP occlusion test simply tells whether a primitive is visible or not. Bartz et al. propose an extension to include more detail, such as number of visible pixels, closest z-value, minimal-screen space bounding box, etc. There are several differences between their work and our own. First and foremost, our techniques are designed to exploit multiple occlusion queries at one time, which tend to generate a smaller number of pipeline stalls in the hardware. Also, our hardware extension is more conservative in its core functionality, but has the extra feature that it would support multiple queries. One additional difference is that, similar to Greene et al. [14], cPLP incorporates an effective technique for filling up the depth buffer so as to minimize the number of queries. We do not believe that it would be difficult to incorporate this feature within the framework of Bartz et al.

The technique by Luebke and Georges [19] describe a screen-based technique for exploiting "visibility portals", that is, regions between cells which can potentially limit visibility from one region of space to another. Their technique can be seen as a dynamic way to compute information similar to that in [28]. One can think of cPLP's obscured front as a single occluder, which has a few holes. If we think of the holes as "portals", this is in certain respects analogous to the work of Luebke and Georges. In the context of their colonoscopy work, Hong et al. [15] propose a technique which merges Luebke and Georges's portals with a depth-buffer based technique similar to ours. However, in their work, they exploit the special properties of the colon being a tube-like structure.

HyperZ [21] is an interesting hardware feature that has been implemented by ATI. HyperZ has three different optimizations that improve the performance of 3D applications. The main thrust of the optimizations is to lower the memory bandwidth required for updating the z-buffer, which they report is the single largest user of bandwidth on their graphics cards. One optimization is a technique for lossless compression of z-values. Another is a fast z-buffer clear, which performs a lazy clear of the depth values. ATI also reports on an implementation of the hierarchical z-buffer in hardware. Details on the actual features are only sketchy and ATI has not yet exposed any of the functionality of their hardware to applications. Consequently, it is not possible at this point to exploit their hardware functionality for occlusion culling.

Another recent technique related to the hierarchical Z-buffer is described by Xie and Shantz [32]. They propose the Adaptive Hierarchical Visibility (AHV) algorithm as a simplification of HZB for tile architectures.

Alonso and Holzschuch [2] propose a technique which exploits the graphics hardware for speeding up visibility queries in the context of global illumination techniques. Their technique is similar to our item-buffer technique. Westermann et al. [29] propose a different technique for using the OpenGL

histogram functionality for occlusion culling. Their work involves histogramming the stencil buffer, instead of the color buffer as done in our work.

6 Conclusions

In this paper we presented a novel conservative visibility algorithm based on the non-conservative PLP algorithm. Our approach exploits several features of PLP to quickly estimate the correct image (and depth buffer) and to determine which portions of this estimation were incorrect. To complete our conservative approach, we required an efficient means of performing visibility queries with respect to the current estimation image. We showed how to implement these visibility queries using either hardware or software. If fragment-counting hardware is available (such as on HP fx, Diamond FireGL, SGI IR3), this is clearly the best choice. Otherwise, the item-buffer technique is the next best option. As graphics hardware continues to improve, and if the OpenGL histogramming features are further optimized, this approach may offer the highest levels of interactive rendering.

Our cPLP approach has several nice features. It provides a much higher level of interactivity than traditional rendering algorithms, such as view frustum culling. As opposed to PLP, cPLP provides a conservative visibility culling algorithm. The preprocessing required by our algorithm is very modest, and we are not required to store significant occlusion information, such as view-dependent occluders or potentially visible sets. We are also able to run our algorithm on all (polygonal) data sets since we do not require any underlying structure or format, such as connectivity information.

Further investigation is necessary to study the feasibility (cost) of adding our hardware extension proposed in Section 3.6 to current architectures. As we show in this paper, it can further improve the performance substantially over techniques that provide a single counter of the fragments that pass the depth-test, such as the HP occlusion-culling extension, since it is able to perform several test in parallel.

Acknowledgements

We thank Craig Wittenbrink for help with the occlusion culling capabilities of the HP fx series accelerators. Craig provided us with much needed material, including documentation and a sample implementation that showed us how to use the HP occlusion test. Many thanks to Prof. Carlo Sequin and his students at the University of California, Berkeley for the SODA Hall model used in our experiments. Thanks to Ned Greene for providing us with the Skyscraper dataset, and for helping in tracking down hard to find references. Thanks to David Kirk of Nvidia for discussions about the complexity of adding the per-fragment histogramming extension. Finally, we thank Dirk Bartz for comments on an earlier version of this paper.

References

[1] OpenGL histogram documentation. http://www.opengl.org/developers/documentation/Version1.2/1.2specs/-histogram.txt.

[2] L. Alonso and N. Holzschuch. Using graphics hardware to speed-up your visibility queries. *Journal of Graphics Tools*, to appear.

[3] D. Bartz, M. Meißner, and T. Hüttner. Extending graphics hardware for occlusion queries in OpenGL. *1998 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 97–104, August, 1998.

[4] D. Bartz, M. Meißner, and T. Hüttner. OpenGL-assisted occlusion culling for large polygonal models. *Computers & Graphics*, 23(5):667–679, October, 1999.

[5] F. Bernardini, J. T. Klosowski, and J. El-Sana. Directional discretized occluders for accelerated occlusion culling. *Computer Graphics Forum*, 19(3):507–516, August, 2000.

[6] Y. Chrysanthou, D. Cohen-Or, and D. Lischinski. Fast approximate quantitative visibility for complex scenes. *Computer Graphics International '98*, pages 220-229, June, 1998.

[7] D. Cohen-Or, Y. Chrysanthou, and C. Silva. A Survey of Visibility for Walkthrough Applications. *Submitted for publication, 2000*. Also in "Visibility, problems, techniques, and applications", ACM SIGGRAPH 2000 Course #4, 2000.

[8] D. Cohen-Or, G. Fibich, D. Halperin, and E. Zadicario. Conservative visibility and strong occlusion for viewspace partitioning of densely occluded scenes. *Computer Graphics Forum*, 17(3):243–254, 1998.

[9] R. Cunniff. Visualize fx graphics scalable architecture. In *Hot 3D Proceedings*, Graphics Hardware Workshop 2000, Interlaken, Switzerland, August, 2000.

[10] F. Durand. *3D Visibility: Analytical study and Applications*. PhD thesis, Universite Joseph Fourier, Grenoble, France, July, 1999.

[11] F. Durand, G. Drettakis, J. Thollot, and C. Puech. Conservative visibility preprocessing using extended projections. *Proceedings of SIGGRAPH 2000*, pages 239–248, July, 2000.

[12] H. Fuchs, Z. M. Kedem, and B. Naylor. On visible surface generation by a priori tree structures. *Proceedings of SIGGRAPH 1980*, pages 124–133, 1980.

[13] N. Greene. Occlusion Culling with Optimized Hierarchical Buffering. In *Proc. ACM SIGGRAPH'99 Sketches and Applications*, page 261, August, 1999.

[14] N. Greene, M. Kass, and G. Miller. Hierarchical Z-buffer visibility. In *Computer Graphics Proceedings, Annual Conference Series, 1993*, pages 231–240, 1993.

[15] L. Hong, S. Muraki, A. Kaufman, D. Bartz, and T. He. Virtual voyage: Interactive navigation in the human colon. *Proceedings of SIGGRAPH 97*, pages 27–34, 1997.

[16] J. T. Klosowski and C. T. Silva. Rendering on a budget: A framework for time-critical rendering. *IEEE Visualization '99*, pages 115–122, October, 1999.

[17] J. T. Klosowski and C. T. Silva. The prioritized-layered projection algorithm for visible set estimation. *IEEE Transactions on Visualization and Computer Graphics*, 6(2):108–123, April - June, 2000.

[18] V. Koltun, Y. Chrysanthou, and D. Cohen-Or. Virtual occluders: An efficient intermediate pvs representation. *Rendering Techniques 2000: 11th Eurographics Workshop on Rendering*, pages 59–70, June, 2000.

[19] D. Luebke and C. Georges. Portals and mirrors: Simple, fast evaluation of potentially visible sets. In *1995 ACM Symposium on Interactive 3D Graphics*, pages 105–106, 1995.

[20] D. Meagher. Efficient synthetic image generation of arbitrary 3-d objects. In *Proceedings of IEEE Conference on Pattern Recognition and Image Processing*, pages 473–478, June, 1982.

[21] S. Morein. ATI Radeon Hyper-Z technology. In *Hot 3D Proceedings*, Graphics Hardware Workshop 2000, Interlaken, Switzerland, August, 2000.

[22] B. F. Naylor. Partitioning tree image representation and generation from 3D geometric models. In *Proceedings of Graphics Interface '92*, pages 201–212, May, 1992.

[23] C. Saona-Vazquez, I. Navazo, and P. Brunet. The visibility octree: A data structure for 3d navigation. *Computers and Graphics*, 23(5):635–643, 1999.

[24] G. Schaufler, J. Dorsey, X. Decoret, and F. X. Sillion. Conservative volumetric visibility with occluder fusion. *Proceedings of SIGGRAPH 2000*, pages 229–238, July, 2000.

[25] N. Scott, D. Olsen, and E. Gannet. An overview of the visualize fx graphics accelerator hardware. *The Hewlett-Packard Journal*, May:28–34, 1998.

[26] K. Severson. VISUALIZE Workstation Graphics for Windows NT. HP product literature.

[27] Silicon Graphics, Inc. SGI Visual Workstation OpenGL Programming Guide for Windows NT. Document Number 007-3876-001. https://www.sgi.com/developers/nt/sdk/files/-OpenGLEXT.pdf

[28] S. J. Teller and C. H. Séquin. Visibility preprocessing for interactive walkthroughs. In *Computer Graphics (SIGGRAPH '91 Proceedings)*, volume 25, pages 61–69, July, 1991.

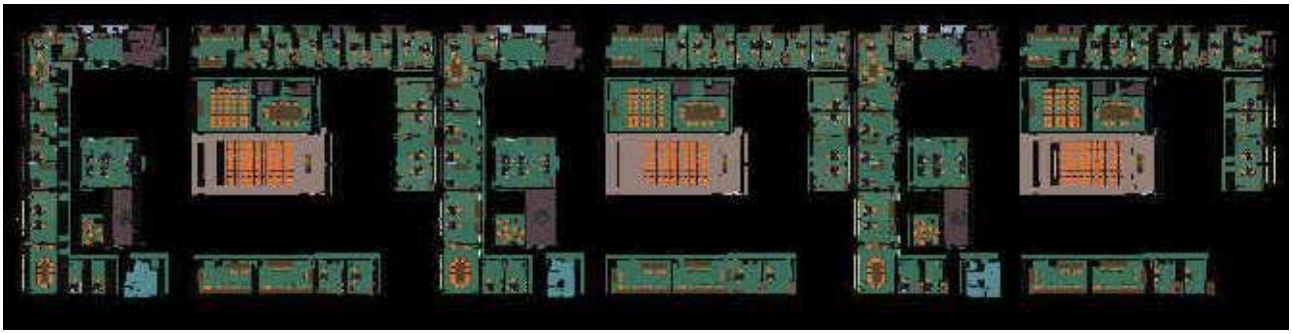
[29] R. Westermann, O. Sommer, and T. Ertl. Decoupling Polygon Rendering from Geometry using Rasterization Hardware. *Unpublished manuscript, 1999.*

[30] P. Wonka and D. Schmalsteig. Occluder shadows for fast walkthroughs of urban environments. *Computer Graphics Forum*, 18(3):51–60, September, 1999.

[31] P. Wonka, M. Wimmer, and D. Schmalstieg. Visibility preprocessing with occluder fusion for urban walkthroughs. *Rendering Techniques 2000: 11th Eurographics Workshop on Rendering*, pages 71–82, June, 2000.

[32] F. Xie and M. Shantz. Adaptive hierarchical visibility in a tiled architecture. *1999 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 75–84, August, 1998.

[33] H. Zhang, D. Manocha, T. Hudson, and K. E. Hoff III. Visibility culling using hierarchical occlusion maps. *Proceedings of SIGGRAPH 97*, pages 77–88, 1997.



(a)

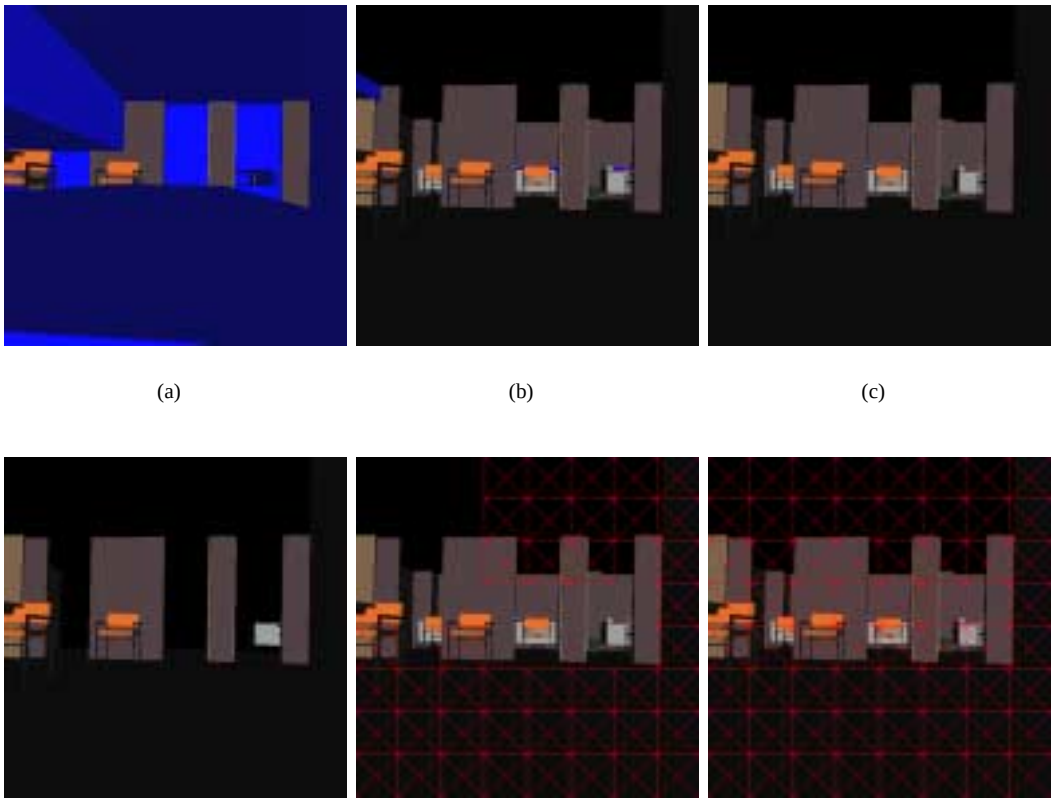
(b)

(c)

(d)

(e)

Figure 9: (a) A top-down view of our dataset. (b)–(e) Sample views of the recorded path.



(a) (b) (c)

(d) (e) (f)

Figure 10: Snapshots during three iterations of our cPLP algorithm. The current front (blue) and completed tiles (red) are highlighted for iteration 1 in (a) and (d), iteration 2 in (b) and (e), and iteration 3 in (c) and (f). The final rendered image is (c).