Occlusion Horizons for Driving through Urban Scenery

Laura Downs
laura@cs.berkeley.edu

Tomas Möller *
tompa@acm.org

Carlo H. Séquin
sequin@cs.berkeley.edu

Computer Science Division
University of California at Berkeley

ABSTRACT

We present a rapid occlusion culling algorithm specifically designed for urban environments. For each frame, an occlusion horizon is being built on-the-fly during a hierarchical front-to-back traversal of the scene. All conservatively hidden objects are culled, while all the occluding impostors of all conservatively visible objects are added to the $2\frac{1}{2}$D occlusion horizon. Our framework also supports levels-of-detail (LOD) rendering by estimating the visible area of the projection of an object in order to select the appropriate LOD for each object. This algorithm requires no substantial preprocessing and no excessive storage.

In a test scene of 10,000 buildings, the cull phase took 11 ms on a PentiumII 333 MHz and 45 ms on an SGI Octane per frame on average. In typical views, the occlusion horizon culled away 80-90% of the objects that were within the view frustum, giving a 10 times speedup over view frustum culling alone. Combining the occlusion horizon with LOD rendering gave a 17 times speedup on an SGI Octane, and 23 times on a PII.

CR Categories and Subject Descriptors: I.3.7 [Computer Graphics]: I.3.7 Three-Dimensional Graphics and Realism.

1. INTRODUCTION

Culling away objects that are definitely not visible and avoiding the expense of sending them through the rendering pipeline is arguably the most effective contribution to speeding up rendering time. In interior scenes, effective visibility culling may be obtained from the many opaque walls [7]. In exterior scenes, one can try to cull away parts of the scene that are totally occluded by other objects [7, 5, 9]. However, this approach is in general much more difficult than managing visibility based on cells and portals in interior scenes. Each occluder derived from an individual object may, by itself, be a rather ineffective occluder. Only when several such objects are fused into a larger occluder, can we expect effective culling. Determining when this can be done, in the general case, is difficult.

We present an algorithm that fuses occluders, needs little preprocessing, requires little extra information, and is simple to implement. It is most useful for urban scenes and performs best when the viewer is located where many humans usually are, less than 2 meters above street level. If the viewer is located above the buildings in the scene, then no occlusion culling algorithm will work well, because most buildings will be visible. We also combined our occlusion scheme with a simple LOD technique, where the LOD is selected based on an estimate of the area of the visible part of an object. Color plate 1 shows the algorithm's effectiveness.

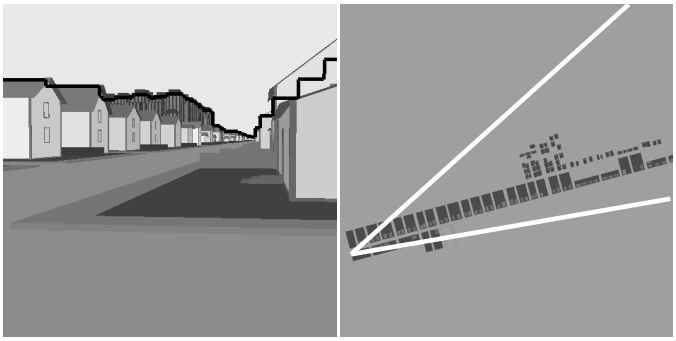

Figure 1: Left: city with occlusion horizon. Right: top down view of occlusion effectiveness.

The cull algorithm is based on a visible horizon in a projected horizontal view. The projection fails if the view direction includes the vertical axis (up). This limitation could be avoided by splitting the view into multiple parts, using only view frustum culling in the vertical direction and occlusion culling in more level directions. Despite this limitation, we find that this is a powerful and efficient algorithm that yields good results with small processing time and no onerous preprocessing or storage requirements.

2. PREVIOUS WORK

Out of the vast quantity of literature on occlusion culling, we review only a small subset of the material that is relevant to our context.

Horizons have been used by several researchers in different ways. Floating horizons can be used to solve the hidden surface problem for plotting functions of the form $z{=}f(x, y)$. The function is processed from front to back, maintaining an upper and a lower horizon for the function graph. See, for example, Roger's book [4] for a discussion and references.

Wimmer et al. [8] render all geometry within a certain radius without occlusion culling. Then, image caching and ray casting is used for the distant geometry, resulting in complexity proportional to the number of pixels of the rendering window. Stewart [6] computes accurate horizons at all vertices in a terrain in a preprocessing step.

* On leave from Chalmers University of Technology, Sweden

Coorg and Teller [2] present a cell-based technique that selects a small set of large occluders, and determines via supporting planes whether a bounding box of an object is hidden behind, i.e., in the shadow of, a large occluder. Occluders are fused only when the resulting occluder can also be represented as a convex polygon.

Schaufler et al. [5] present an algorithm that voxelizes space, identifies interior voxels and fuses them to represent larger occluders. The projection of these voxels forms a shadow shaft which is used to cull away objects.

Wonka and Schmalstieg [9] have developed an algorithm for visibility culling in urban environments. They use graphics hardware to describe shadow volumes in an orthographic top-down view. This requires the frame buffer to be read for each view point, which is slow on some architectures and limits the accuracy of the algorithm to the resolution of the frame buffer. Later, this technique was extended to a cell-based method and used as a preprocessing step [10]. The upgraded algorithm organizes the information hierarchically, but the preprocessing still takes about 9 hours for 8 million polygons, where more than 50% of the time is spent reading the frame buffer and the extracted visibility information requires a good deal of storage space.

Funkhouser and Séquin [3] present a level-of-detail (LOD) rendering algorithm for constant frame rate. Andújar et al. [1] describe a framework for combining LOD rendering with occlusion culling. They use hardly visible sets (HVS) and a visibility octree to bound the image error, and report a speed-up of 2.3.

3. OCCLUSION CULLING WITH CONSERVATIVE HORIZONS

Our algorithm assumes that most of the occluding geometry in the model can be described by a height field. We use a piecewise constant approximation to this field in the form of vertical prisms. A plane is swept in the viewing direction, away from the eyepoint across the model, accumulating a horizon of occluding objects. Any object that is not obscured by the horizon at the point that it is first encountered is added to the list of visible objects for display, and its occlusion information is added to the horizon. This simple representation covers the dominant sources of occlusion in urban or suburban environments and allows for very fast merging of occluders. Overhanging structures and bridges will not contribute to occlusion in the scene, but they will be correctly culled or deemed potentially visible. Terrain does contribute to the occlusion horizon, since it is well represented as a height field, however the model we created for testing uses a flat ground plane.

3.1 Occluder Volumes

Our algorithm requires outer and inner approximating volumes for each object. As an outer volume, we use a simple bounding box, aligned with the object's frame of reference. This box is used to test whether an object is visible. The inner volume of an object is described by a collection of convex vertical prisms (CVPs), each contained entirely within the object's geometry.

A CVP is defined by a convex polygonal 2D cross-section in the xy-plane and a z height. The occlusion extent of a single object may be approximated by the union of several CVPs, as shown in Figure 2, in order to better approximate the occlusion silhouette of the object. The union of the ground plane with the CVPs for all objects in the scene forms a height field which is a conservative estimate of the occluding geometry in the scene. The occlusion properties of most buildings, even those with sloped roofs, can be represented quite well with a small set of CVPs (Fig. 2).

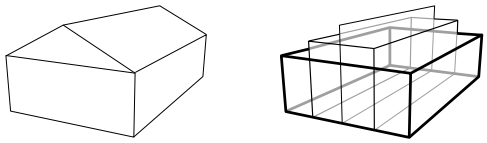

Figure 2: A simple house, and a set of three CVPs.

3.2 CVP Generation

Although we used a model with pregenerated occluder volumes for each building, the CVPs for a building could be generated from the building geometry if they were not provided explicitly. The goal is to find a small set of convex prisms that are wholly contained within the geometry of the building and which provide as large a silhouette as possible.

To generate the CVPs for a building, a sweep plane is passed from the ground to the top of the building. At each z slice, the outer contour of the building is computed. The intersection of all such contours so far is an outer bound on the possible prisms that could be placed within the building. At any point where this intersection is about to grow smaller, a CVP or multiple CVPs may be created. The CVP cross-sections may be generated by a convex decomposition of the contour or by computing the maximum rectangle that can be inscribed within the polygon. The former is favored for simple models and the latter for very detailed models. The decision of whether to create a new CVP at any particular z value depends on the occlusion area added by the new CVP over all CVPs already created and on the total number of CVPs one is willing to create for that building.

3.3 Scene Representation

The input model is a 3D scene graph in which some objects have predefined occlusion volumes (CVPs), as described above. Every object is assumed to be defined in a frame of reference where xy is the ground plane and z is the vertical axis. Objects with CVPs may be rotated about the z-axis and translated by any amount; we call such transformations *prismatic*. These transforms can be updated from frame to frame, and if they remain prismatic, this allows for movable occluders and occludees. If an object is transformed non-prismatically (e.g., rotated about the x-axis), then its CVPs may no longer form a height field, and they will not be considered for occlusion.

The scene hierarchy is flattened down to the level of objects that have occluder volumes or transforms that are not prismatic. This creates a flat list of the buildings and other objects in the scene. Next, the model is organized into a 2D quadtree in the xy-plane. Objects are inserted into the quadtree at the lowest level cell whose minimum dimension is larger than the object's maximum dimension. This allows small objects that straddle high-level quadtree cell boundaries to be culled away more reliably than if they were included in the high-level cells in the tree. Any object can be included in as many as four cells by this method. In order to avoid duplicating objects, an object is stamped with the current frame number when it is first encountered during the plane sweep. Each quadtree cell records the maximum height of any object within it or any of its descendants, so that the entire cell can be culled away, if it lies below the horizon.

3.4 Occlusion Horizon

A person standing at street level in an urban environment will mostly see the nearby buildings. Due to the effect of perspective, distant buildings will appear smaller and are frequently hidden behind the

closer buildings. Based upon this observation and the fact that horizons have been used to solve the hidden surface problems [4], we have been inspired to use an occlusion horizon to cull away hidden geometry in urban environments.

The occlusion horizon is simply a description of the highest projected points in the height field that are visible from a given vantage point within a certain distance. More distant objects that lie below this horizon may be safely discarded as invisible. We use a piecewise constant approximation to this horizon (Plate 2).

The horizon is a conservative mask of the space occluded by all buildings encountered so far in the sweep. It is described in the reference frame of the observer as a projection onto the $y = 1$ plane for an observer at the origin looking down the y-axis. In the view plane, the z-axis points up and the x-axis points to the right. The horizon is approximated as a piecewise constant function in x and represented in a binary tree (Fig. 3). Each node in the tree has an x range. Every leaf node has a z value for the mask height. The minimum and maximum mask height of all children are stored at every internal node.

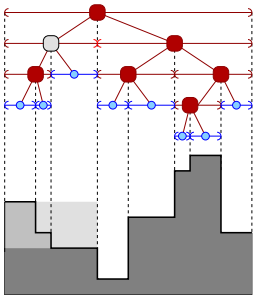

Figure 3: Binary tree representation of an occlusion horizon. The minimum and maximum mask height is shown in light grey for the the second internal node from the left.

Each occluding volume adds a rectangular mask to the current horizon. Because the prisms are vertical and the projection plane is parallel to the vertical axis, the vertical edges of the prisms project to vertical lines in the projection plane. To compute the occluding mask of a CVP, we first compute the perspective projection of the prism top (Fig. 4a). The minimum and maximum x values correspond to the left and right silhouette edges of the prism. The occlusion properties of the prism will now be represented by the vertical slice of the prism through those two points (Fig. 4b). A rectangular mask for the prism in the projection plane will have the left and right x values of the projected points, a height equal to the minimum z value of the projected points (Fig. 4c), and an event distance equal to the larger y value of the actual points (ev). This rectangle will be added to the occlusion horizon, when the sweep plane reaches the event distance for the rectangle mask.

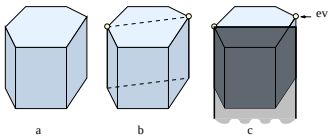

Figure 4: Underestimating the occlusion extent of a CVP with a rectangular mask.

3.5 Occlusion Culling Algorithm

For the sake of simplicity, assume the viewer is looking in a direction parallel to the ground plane. As we will see later, the solution remains correct even if the view is pitched up or down.

The core of the occlusion culling algorithm is an event queue that represents the position of the sweep plane as it moves through the scene. This sweep plane is parallel to the view plane and runs from behind the viewer, forward in the viewing direction, out to the far edge of the scene geometry. Every potentially visible object and every potentially visible node in the quadtree will have an event on the queue at the first time that the sweep plane touches that object or node. Objects that lie entirely behind the viewer are culled away without contributing to visibility or geometry, but objects that intersect the viewer's plane may be visible and may cause occlusion.

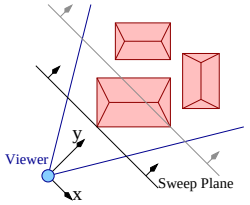

Figure 5: Progress of the sweep plane through a scene.

Potentially visible objects are compared with the occlusion horizon at their minimum y value. CVPs are added to the mask at their maximum y value. This ensures that no object can accidentally occlude an object that passes in front of it at some point. The minimum z value for the occluder object is used for the height of the rectangular mask which will be added to the horizon.

Each potentially visible object is assigned a rectangular bounding frame in the view plane that contains the projection of its bounding box. This mask is a conservative overestimation of the screen area that the object can occupy; thus if the mask is hidden by the horizon, the object will be as well.

The test of whether a given bounding frame is visible above the occlusion horizon is performed recursively on the binary tree representation of the horizon. The test can terminate without recursing to the leaves of the tree if the mask is below the minimum value or above the maximum value of a portion of the horizon. Because the tree representation is kept balanced, the recursion is never very deep, and thus the visibility test is very fast.

When an object or an area of terrain is determined to be visible, all CVPs associated with that object are added to the event queue at the time when the sweep plane touches the farthest point in each CVP's effective occluding geometry. Also, any geometry is added to the display list and any child elements are added to the event queue. The event queue will be empty when the sweep plane has passed through the entire scene. After the sweep, all objects that have been placed in the display list are drawn.

A rectangular mask is added to the horizon through a recursive tree traversal. At each node of the tree, the mask is compared to the minimum and maximum node heights. If the mask completely hides the node, then the subtree below that node is removed and the node becomes a child node with the mask's height. If the mask is completely hidden by the node, then recursion stops with no effect. If the node is a child node, it is split. Otherwise, the mask is passed down to the children that it overlaps. The tree is kept balanced by recording subtree depth at each node and rebalancing as necessary. A small effort is made to merge adjacent children with identical heights.

3.6 Levels of Detail

The same type of recursion that is used to determine whether a mask is visible above the horizon can also be used to calculate an upper bound on the visible area of an object. Instead of a simple boolean result of "visible" or "not visible", the area of the mask above the horizon is computed. This area can be used to determine at what Level of Detail (LOD) the object should be rendered. Objects which potentially occupy no more than a few pixels in the scene can be omitted entirely.

3.7 Tilted Views

The occlusion is described for a viewer looking in a direction parallel to the ground plane. However, if the viewer pitches up or down, the intersection of the view frustum with the y=1 plane can be used to find a view window in the horizontal direction that encompasses the new view. As long as the view does not include the z-axis, such a projection will exist. Since the observer has only rotated between the two views, occlusion is unchanged. The algorithm as described is used to compute visibility for the horizontal view but the display list is drawn with the actual viewer transform. If the new view does contain the vertical axis, then the view area must be partitioned into a region where occlusion culling is computed and a near-vertical region where only view frustum culling is used.

4. TEST RESULTS

4.1 Urbania Generator

In order to test our program with urban scenes of different sizes, we wrote a program that automatically creates such scenes. All generated cities have the following features:

The model contains many long straight streets which afford views containing a lot of geometry and a complex horizon. Each building model is kept simple (120–180 polygons), which requires the cull process to be very efficient in order to show good speedups over unculled scenes. The buildings are kept small with a good amount of space between them to make occluder fusion more difficult. The city is rotated 15° to prevent unnatural alignment of the model with the quadtree cells. The city contains some tall buildings that interrupt the horizon causing a fair amount of recomputation when they are encountered.

4.2 Measurements

Using our procedural urbania generator, we created two urban models, called *urban1k*, which has 1,000 buildings and 178,577 polygons, and *urban10k*, which has 10,396 buildings and 1,724,749 polygons, which we used for performance measurements.

In Plate 3, we report the statistics obtained on an SGI Octane R10k and on a PC PII 333 Mhz with an nVidia GeForce2 GTS graphics card. The statistics were gathered during predetermined walkthroughs in both *urban1k* and *urban10k*. The following culling algorithms are reported: hierarchical view frustum culling (VF), VF plus level-of-detail (LOD) rendering, occlusion horizons (OH), and OH+LOD. In the table below, we also report average frame rate (AFR), minimum frame rate (MFR), and average speed-up (ASU) compared to hierarchical view frustum culling (VF).

For both models, the OH+LOD algorithm generated the best average speed-up as expected – ranging from 3.6–17 on the Octane, and from 5.2-23 on the PC. We also noticed that the occlusion horizon on average culled away 80-90% of the objects within the view frustum. The main algorithm consists of about 3,000 lines of C++ code and took two weeks to develop.

	VF			**OH**			**OH+LOD**		
	AFR	MFR	ASU	AFR	MFR	ASU	AFR	MFR	ASU
	SGI Octane								
u1k	4.4	1.8	1.0	13.8	6.2	3.1	16.0	6.4	**3.6**
u10k	0.5	0.2	1.0	5.0	1.7	10	8.7	2.9	**17**
	PC 333 MHz GeForce2 GTS								
u1k	11.1	3.8	1.0	38.3	18.5	3.4	57.6	32.2	**5.2**
u10k	1.1	0.2	1.0	12.0	2.3	10.9	25.2	4.5	**23**

5. CONCLUSION AND FUTURE WORK

We have combined level-of-detail rendering with a new occlusion culling algorithm. The resulting algorithm is well suited for rendering of urban environments, and we present speed-ups from 3.6 to 17.4 over frustum culling, depending on the model. The algorithm builds a $2\frac{1}{2}$D occlusion horizon during a front-to-back traversal of a 2D quad tree of the scene. Every object is tested against the occlusion horizon and culled if it lies below the horizon. Otherwise, its occlusion extent is added to the horizon. The area of a (partly) visible object is estimated and used as a measure to select the appropriate LOD for the object during this process.

Future work includes extending the algorithm to deal with cell-based visibility. The occlusion mask describes a hidden volume from a particular view point. The set of objects that are completely hidden from all points in a cell can be computed by tracing parallel rays from all of the vertices of the cell for each ray between the eye and an occluder and choosing the most conservative estimate for occlusion.

6. REFERENCES

[1] Carlos Andújar, Carlos Saona-Vásquez, Isabel Navazo, and Pere Brunet. Integrating occlusion culling and levels of detail through hardly-visible sets. *to appear in Computer Graphics Forum*, 2000.

[2] Satyan Coorg and Seth Teller. Real-time occlusion culling for models with large occluders. *1997 Symposium on Interactive 3D Graphics*, pages 83–90, April 1997.

[3] Thomas A. Funkhouser and Carlo H. Séquin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. *Computer Graphics (Proceedings of SIGGRAPH 93)*, pages 247–254, August 1993.

[4] D. F. Rogers. *Procedural Elements for Computer Graphics*. McGraw-Hill, 1985.

[5] Gernot Schaufler, Julie Dorsey, Xavier Decoret, and Francois X. Sillion. Conservative volumetric visibility with occluder fusion. *Computer Graphics (Proceedings of SIGGRAPH 2000)*, pages 229–238, July 1000.

[6] A. James Stewart. Hierarchical visibility in terrains. *Eurographics Rendering Workshop 1997*, pages 217–228, June 1997.

[7] Seth J. Teller and Carlo H. Séquin. Visibility preprocessing for interactive walkthroughs. *Computer Graphics (Proceedings of SIGGRAPH 91)*, 25(4):61–69, July 1991.

[8] Michael Wimmer, Markus Giegl, and Dieter Schmalstieg. Fast walkthroughs with image caches and ray casting. *Computers & Graphics*, 23(6):831–838, December 1999.

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

[10] Peter Wonka, Michael Wimmer, and Dieter Schmalstieg. Visibility preprocessing with occluder fusion for urban walkthroughs. *Rendering Techniques 2000: 11th Eurographics Workshop on Rendering*, pages 71–82, June 2000.

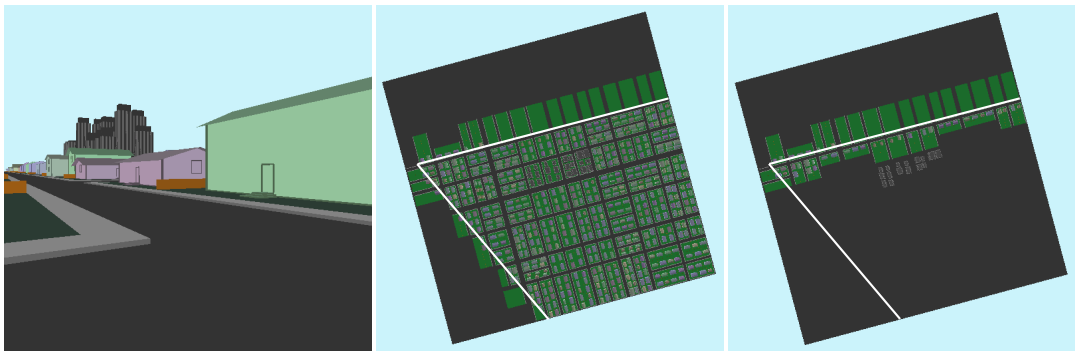

Plate 1: To the left, an image is shown from the observer's view; the middle image shows a bird's eye view of the city with view frustum culling; and to the right we show the same view with occlusion horizon culling.

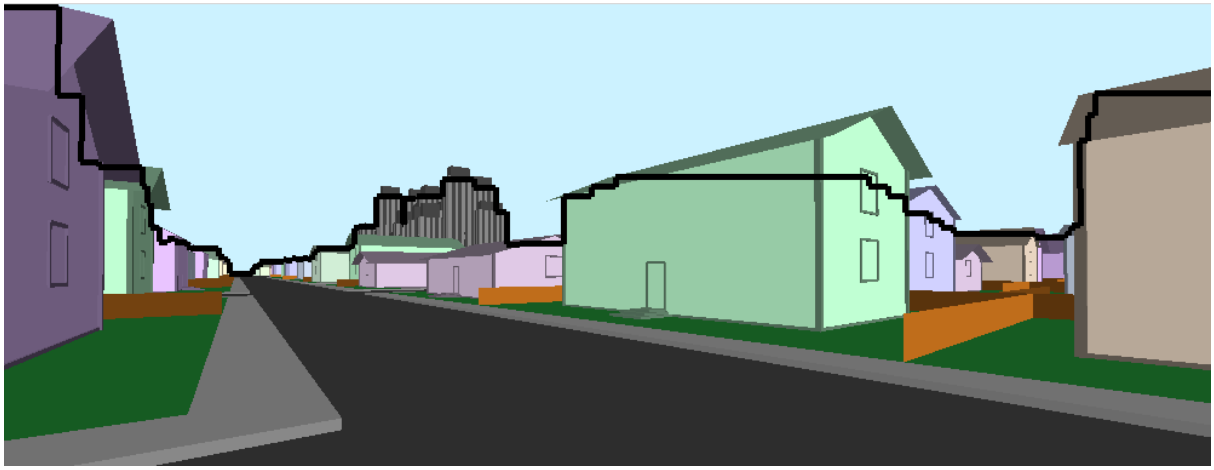

Plate 2: Occlusion horizon overlaid on city view.

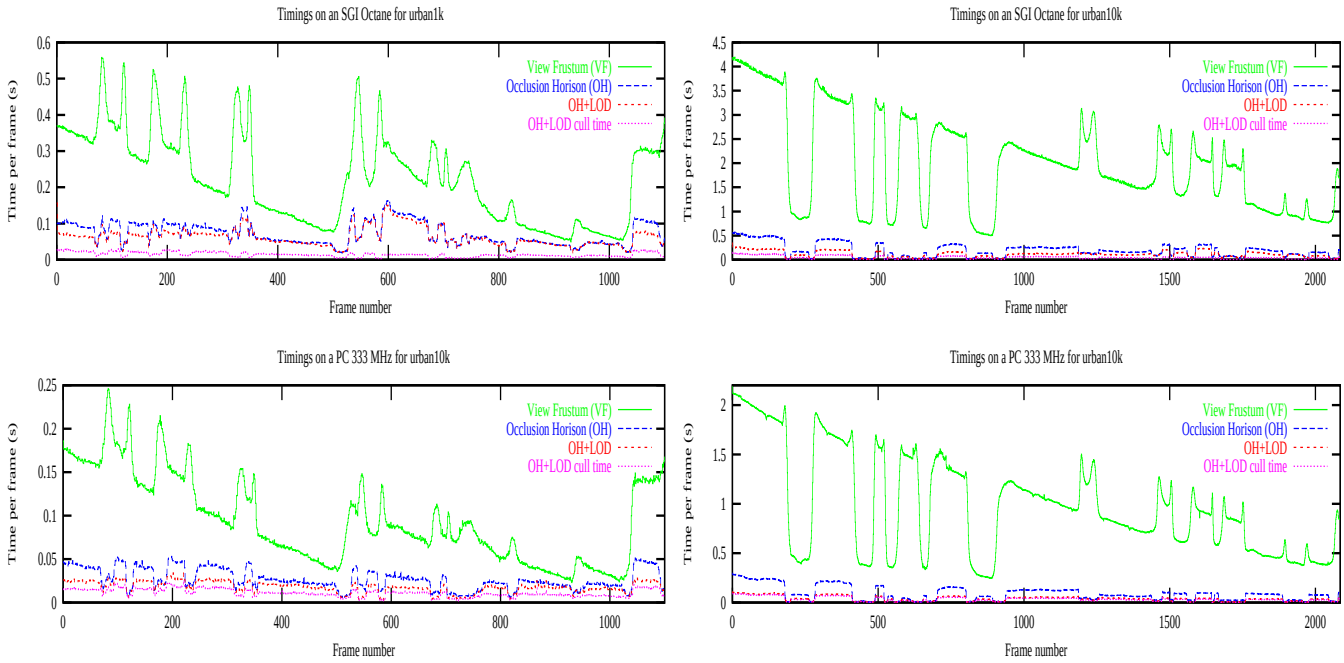

Plate 3: Timing graphs.