Fast Event-Driven Refinement of Dynamic Levels of Detail

Christopher Zach*
Konrad Karner†

VRVis Research Center

Abstract

Dynamic levels of detail allow fine grained selection of actually rendered geometry suitable for the current viewing parameters. Therefore, the visual quality of interactive 3D applications is usually higher than the quality achieved by traditional static level of detail approaches. On the other hand, this fine grained representation requires more CPU time for runtime refinement of the displayed mesh. We report on an event-driven refinement method that decreases the selection time significantly. Our approach exploits coherence between successive frames explicitly to reduce the number of tests necessary for node refinements. For large virtual environments we observed, that our method performs best when the actually displayed mesh is very complex. In these cases our approach is several times faster than the primal methods for dynamic level of detail refinement.

Keywords: Dynamic Levels of Detail, View-Dependent Multiresolution Meshes, View-Dependent Rendering, Interactive Visualization

1 Introduction

Dynamic levels of detail (often referred as view-dependent multiresolution meshes) comprise a general tool to allow smooth interactive animation of complex mesh geometry. The displayed mesh is highly adjusted for the current viewing parameters and usually only few changes are applied to the mesh between successive frames. Therefore sudden changes in the visible geometry (often called *popping artefacts*) do not appear and even small changes in geometry can be smoothened using runtime geomorphs.

This highly adaptive mesh visualization approach has some disadvantages: The selection (or refinement) of the mesh to display is rather computationally intensive, since a large number of decisions is required for every rendered frame. Further, dynamic levels of detail do not easily exploit efficient rendering primitives like triangle strips. In this work we address the first shortcoming to reduce the time for mesh selection. This goal is achieved by minimizing the required number of decisions for every frame using an event-driven approach to mesh refinement. Our method exploits coherence between frames directly, since successive frames with close viewing parameters require only few decisions to be performed to adjust the displayed mesh.

Since our approach requires more complex calculations for every decision, in the worst case our method can be slower than traditional refinement schemes. Nevertheless we observed considerable speedup on average, when the user navigates through complex meshes in some reasonable manner.

2 Related Work

The key prerequisite for view–dependent mesh visualization is the generation of a multiresolution data structure containing a set of smooth levels of detail of the original model. The most popular multiresolution concept is the *progressive mesh* [5], which is essentially a sequence of meshes with successively lower geometric accuracy and complexity. Starting with the original mesh the next sequence element is obtained by applying one *edge collapse* operation (Figure 1) to remove one edge and several triangles from the previous mesh. The sequence of edge collapses is chosen such that the overall shape of the model is preserved. Edge collapses are performed until the mesh distortion (according to some quality metric) is larger than a user specified threshold. A very successfull metric to guide the simplification procedure is the quadric error metric proposed by Garland and Heckbert [4].

*zach@vrvis.at
†karner@vrvis.at

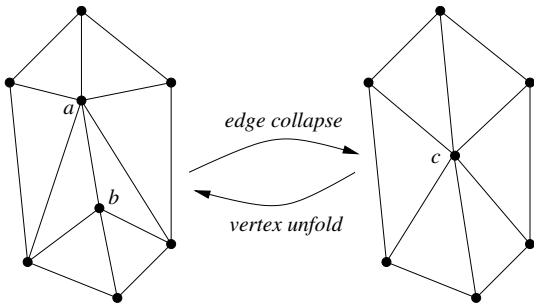


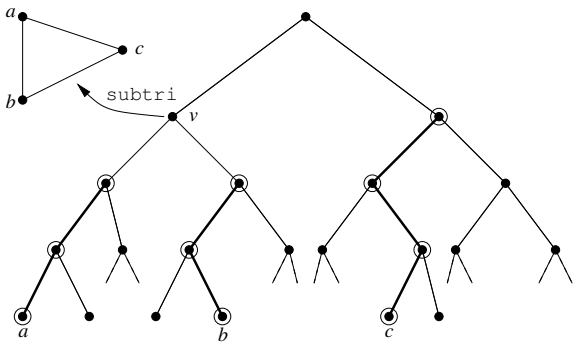
Figure 1: Edge collapse operation. The vertices a and b are collapsed into vertex c, therefore removing one edge and 2 triangles. The inverse operation, *vertex unfolding* (or *vertex split*), is applied at runtime for mesh refinement.

Several authors noticed that the linear sequence of edge collapses can be generalized to a partial ordering represented by the *vertex tree* [13], [6], [3], [12]. Selective refinement at runtime determines the currently displayed mesh. Our framework is based on `VDSlib` [9], which is described by Luebke and Erikson [11]. Figure 2 gives an overview of the most important data structure, the *vertex tree*. At runtime the nodes in the vertex tree fall into three categories: *active nodes*, which contain the currently rendered triangles (called *subtris*), *boundary nodes*, which correspond to vertices in the displayed mesh, and *inactive nodes* (which do not contribute to the rendered mesh). For each frame to render, the vertex tree is traversed in top down order and active nodes are determined according to some screen error metric. Figure 3 illustrates the procedure for the top-down adjustment of the vertex tree.

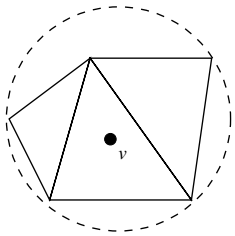
The selection of the displayed mesh is most easily described as a top-down traversal of the vertex tree [11]. Faster mesh selection can be achieved if the refinement starts with the boundary nodes of the previous frame. We observed that the gain of such an approach is rather marginal.

To decrease the time spent for mesh refinement Hoppe [6] proposed a scheme to amortize refinement time over several frames. Before a frame is rendered only a fraction of the boundary nodes are tested for necessary refinement or coarsening. Other authors (e.g. Luebke [11]) proposed separate threads for mesh refinement and rendering to achieve amortized mesh selection. In all amortizing approaches the displayed mesh is not optimally adjusted for the current viewing parameters, which is especially noticeable when the user changes his viewing parameters very quickly.

It is very interesting to recall the ratios of the refinement time to the rendering time given in the early papers on view-dependent multiresolution meshes: Hoppe [6] reports that only 14% of the frame time is spent for mesh refinement. Similar numbers are given by El-Sana et al. [1]. Our own experience with current PC hardware shows, that the performance of graphics hardware has increased faster



(a) The vertex tree



(b) Associated bounding sphere

Figure 2: (a) The vertex tree as found in `VDSlib` [11]. Every node stores geometric data (point coordinate, color, a bounding sphere etc.), its children and a list of associated triangles, the so called *subtris*. The triangle (abc) is a subtri of v, because it exists only if v is unfolded at runtime. The nodes emphasized with a circle are possible corners of (abc) chosen for rendering. (b) The bounding sphere of node v contains all subtris of v and its ancestors.

than the performance of the CPU, mainly because of internal parallelization of vertex and fragment processing. We observed that the original `vdslib` spents about as much time for mesh refinement as in the rendering pass on current hardware configurations.

Recently, Levenberg [8] mentioned a simple, event-driven refinement procedure for a terrain rendering framework, but he considered only potential unfold events to refine the mesh. He did not report on the effectiveness of such a scheme.

Dynamic level of detail approaches do not directly support efficient rendering primitives like triangle strips. Hoppe [6], [7] utilizes a greedy algorithm to generate triangle strips dynamically at runtime. El-Sana et al. [1] proposed a method to maintain triangle strips in the context of view-dependent multiresolution meshes.

El-Sana et al. [2] presented several techniques to accelerate rendering of view-dependent meshes. Their main contribution consists in a modified mesh selection procedure, that is based on a coarse spatial hierarchy and an estimate of the likelihood a set of nodes needs to be re-

```
adjustTreeTopDown(node, unfoldTest)
   if (node->status == ACTIVE)
      // Node was active in the previous frame
      if (!unfoldTest(node))
         // Now it does not meet the screen
         // size criterion.
         // Fold node and all its ancestors
         // (make them inactive)
         node->foldSubtree()
         return
   else
      // Node was inactive in the previous frame
      if (unfoldTest(node))
         // Node meets the screen size criterion:
         // Unfold this node and make it active.
         node->unfold()
      else
         // Node remains inactive (as well as
         // its children)
         return

   // Refine all children
   for (int i = 0; i < node->nChildren; ++i)
      Node * child = node->children[i]
      // Note: we need not to test child nodes
      // that are leaves, since these node cannot
      // become active.
      if (child->nChildren > 0)
         adjustTreeTopDown(child, unfoldTest)
```

Figure 3: Pseudo code for top-down refinement. The function `unfoldTest` returns true if the node should be unfolded according to the current viewing parameters. Our `unfoldTest` criterion is explained in Section 3.

fined. When selecting the mesh suitable for display the coarse hierarchy is traversed at first and nodes not needing a refinement are quickly determined.

3 Our New Approach

We use the following simple observation as the basis of our faster refinement approach: if the user moves slowly through the virtual environment, only few nodes will be refined or coarsened at the beginning of every frame. If some maximum translational and angular velocities are known in advance, we can further estimate the earliest point in time when a node needs to be tested again. We identify points in time with (discrete) frame numbers and use these terms interchangeably.

In the following discussion we restrict to the case of a pure screen space error metric to guide the mesh refinement. This means that the screen space deviation of the vertices of the displayed mesh from the original vertices is less than some user specified threshold. Nevertheless, our method is applicable to more general error metrics e.g. incorporating silhouette information ([11], [6]) as well.

If a node is currently folded, we estimate the earliest time for a change in the node status as the maximum time of two events:

1. A node has to change its status because the user moves directly towards this node.
2. The user changes his viewing direction and consequently this node becomes visible.

If a node is already unfolded, then the earliest time for folding this node is given by the minimum time of the following events:

1. A node requires folding because the user moves directly away from this node and the size of the projected bounding sphere becomes smaller than the user specified threshold.
2. The user rotates his viewing direction, hence this node becomes potentially invisible.

We describe the calculations in more detail for boundary nodes that can be possibly unfolded in the future. Assume that the user is restricted to move at most ρ units and can change his viewing direction by ω degrees per frame at maximum. We denote the current viewing position with $eyePos$ and the current viewing direction with $eyeDir$. For a given boundary node in the vertex tree, let c and r be the center resp. the radius of the associated bounding sphere.

Foldtest Criterion We derive the event times from the basic test whether a node should be folded or unfolded. In our implementation a node is unfolded if and only if the following expression is true:

$$\begin{aligned} unfold \;\; = \;\; & insideTest \vee \\ & (visibilityTest \wedge thresholdTest), \end{aligned}$$

where $insideTest$ is a boolean predicate indicating if the eye point is within the bounding sphere. We added the $insideTest$ predicate to allow simple (and fast) implementations of the $visibilityTest$ and $thresholdTest$ predicates. The predicate $visibilityTest$ checks whether some part of the sphere is inside the viewing frustum and finally $thresholdTest$ is true, if the projected size of the bounding sphere is larger than some user specfied threshold θ.

The procedure shown in Figure 4 is very similar to the code found in the `VDSlib`.

Let t_i be the potential time of entering the bounding sphere of some node, and t_r resp. t_m are the closest frame numbers of entering the viewing frustum and attaining the requested screen size threshold. Therefore, if some node is currently folded (has status boundary or inactive), then the earliest time of becoming active for this node is

$$t = \min(t_i, \max(t_r, t_m)).$$

The time t_i can be easily calculated as

$$\frac{\|eyePos - c\| - r}{\rho}$$

```
unfoldTest(node)
   // D .. distance vector between viewer
   // and node
   Vector D = node->center - eyePos
   float distance = ||D||
   // inside test: is viewer within
   // bounding sphere?
   if (distance < node->radius) return true
   float phi   = asin(node->radius / distance)
   float alpha = acos(<eyeDir, D> / distance)
   // visibility test: is bounding sphere
   // (partially) inside the viewing cone?
   if (alpha - phi > sqrt(2) * fov/2)
      return false
   // threshold test: is estimated projected
   // screen size large enough?
   return (node->radius / tan(fov) / distance)
           >= theta
```

Figure 4: The fold/unfold test criterion.

(recall, that c and r are the center resp. radius of the bounding sphere and ρ denotes the maximum translational movement per frame). Analogeously the time for a potential fold of an active node can be calculated using the dual formulation.

Translational Movement If we assume that the node is within the viewing frustum, but the projected size of the bounding sphere is currently smaller than the threshold θ, then the following holds:

$$\frac{r}{\tan(fov/2)\,\|c - eyePos\|} < \theta.$$

The projected size of the bounding sphere grows fastest if the user moves directly towards the center of the sphere (see Figure 5). The distance d the user has to move, such that the threshold is attained, can be determined from

$$\frac{r}{\tan(fov/2)\,(\|c - eyePos\| - d)} = \theta.$$

Since for every frame the user can move ρ units at most, the earliest time for unfolding this boundary node because of translational movement can be calculated as

$$t_m = \lceil d/\rho \rceil = \left\lceil \frac{\|c - eyePos\| - r/(\tan(fov/2)\,\theta)}{\rho} \right\rceil$$

Change of the Viewing Direction If the currently inspected node is outside the viewing frustum and therefore folded, the earliest frame number the bounding sphere of this node is at least partially visible is

$$t_r = \left\lceil \frac{\alpha + \phi - \sqrt{2} fov/2}{\omega} \right\rceil,$$

where $\alpha = \angle(eyeDir, c - eyePos)$ is the angle between the viewing direction and the relative position of the

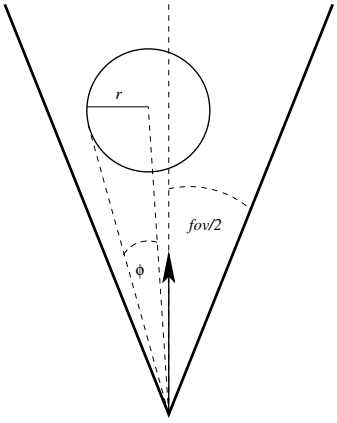


Figure 5: Derivation of potential fold/unfold events due to translational movement.

sphere and $\phi = \arcsin(r/\|c - eyePos\|)$ is the spherical size of the bounding sphere. Remember, that ω denotes the maximum rotation angle of the viewing direction per frame. This situation is illustrated in Figure 6.

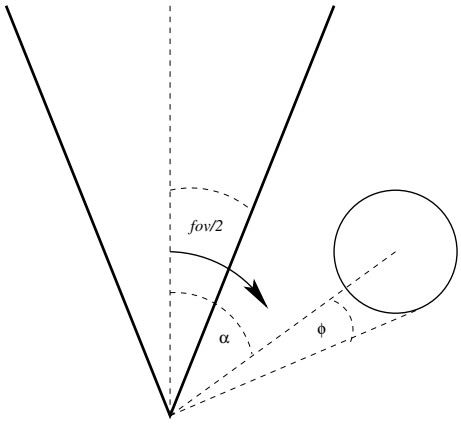


Figure 6: Derivation of potential fold/unfold events due to rotation of the viewing direction.

If a node is already unfolded (active) we can compute the times of potential fold events in a similar manner.

Queue Management The event times determined from the previous paragraphs are relative to the current frame number. A queue containing nodes sorted by absolute event times is maintained during runtime. At the beginning of every frame the events in the queue up to the current frame number are handled. The pseudo code of the main event handling loop is given in Figure 7.

Some parts of the code needs further explanation. We store boundary nodes associated with their potential unfold time and collapsable active nodes in the event queue. An active node can be collapsed if all child nodes are classified as boundary nodes. The functions `potentialFoldTime` and `potentialUnfoldTime` calculate this earliest frame

```
queueMgr.advanceTime() // Start new frame
while (n = queueMgr.popNext())
  if (n->status == ACTIVE)
    if (!unfoldTest(n))
      // Make a previously active node inactive
      n->fold()
      long t = potentialUnfoldTime(n)
      n->queuePos = queueMgr.insertEntry(t, n)
      for (int i = 0; i < n->nChildren; ++i)
        RuntimeNode * child = ne->children[i]
        // Skip leaves.
        if (child->nChildren > 0)
          queueMgr.eraseEntry(child->queuePos)
      // Register parent node if all its
      // children are inactive.
      parent = n->parent
      if (parent != 0 &&
        allChildrenAreBoundary(parent))
        long t = potentialFoldTime(parent)
        parent->queuePos =
            queueMgr.insertEntry(t, parent)
    else
      // No change, reinsert this node for
      // future test
      long t = max(1, potentialFoldTime(n))
      n->queuePos = queueMgr.insertEntry(t, n)
  else
    if (unfoldTest(n))
      // Make a previously inactive node active
      n->unfold()
      long t = potentialFoldTime(n)
      n->queuePos = queueMgr.insertEntry(t, n)
      // Erase the parent of n from the event
      // queue, since one if its children became
      // active
      parent = n->parent
      if (parent != 0 && parent->queuPos != 0)
        queueMgr.eraseEntry(parent->queuePos)
      // Insert all non-trivial children of n
      // into the queue for potential unfolding.
      for (int i = 0; i < n->nChildren; ++i)
        child = n->children[i]
        // Skip leaves.
        if (child->nChildren == 0) continue
        long t = potentialUnfoldTime(child)
        child->queuePos =
           queueMgr.insertEntry(t, child)
   else
     // No change, reinsert this node for
     // future test
     long t = max(1, potentialUnfoldTime(n))
     n->queuePos = queueMgr.insertEntry(t, n)
```

Figure 7: The main loop to handle potential fold and unfold events for the next frame.

number of possible collapse and unfold events as described in the previous paragraphs.

Essentially the code tests initially for every node, which needs to be handled, whether its old status has to change according to the current viewing parameters. If this is not the case, the node is inserted again into the event queue at some appropriate future frame number. Otherwise the node needs to be folded resp. unfolded. If a previously active node is folded, all children are removed from the queue and its parent is registered for a potential fold event, if all siblings of the current node are already folded. Additionally the node itself is registered for an potentially upcoming unfold event.

If a former boundary node is unfolded, its parent is eventually erased from the queue, the node itself is inserted and all non-trivial children are registered. In order to remove invalidated nodes efficiently from the queue, every node stores a pointer (iterator) to its current position in the queue (`queuePos`). This iterator allows fast removal of the node from the event queue.

The structure `QueueMgr` has the following interface:

```
struct QueueMgr
   type iterator

   void       advanceTime();
   Node     * popNext() // returns 0 if no more events
   iterator   insertEntry(int time, Node * node)
   void       eraseEntry(iterator pos)
```

The semantics of these functions are straightforward. The subroutine `eraseEntry` removes an event and invalidates the queue position stored in the deleted node.

The simplest implementation of `queueMgr` is based on multimaps[1]. A faster method uses arrays of lists, which comprise a circular buffer. For better performance we cache some values calculated inside `unfoldTest` (like the distance of the bounding sphere to the eye position) and use them in special implementations of `potentialFoldTime` and `potentialUnfoldTime`. Since folding or unfolding of nodes may cause subsequent collapsing of parents resp. unfolding of subnodes, the runtime queue must be able to handle insertions with `t` equal to zero, which means that a node needs a test within the current frame.

Rapid Movements Even if the user moves faster than the specified velocities ρ resp. ω, our incremental method can still be applied. In case the user navigates faster the current frame number is incremented by the appropriate amount. Therefore more outstanding events are handled and the overall selection time increases. If the user changes his viewing parameters very fast, due to more expensive computations our incremental refinement scheme can be slower than e.g. the direct top-down refinement approach. We observed that this is the case only for very abrupt changes, which cause problems for other amortized schemes as well.

4 Results

We compared our mesh refinement procedure with the top-down selection method found in the `VDSlib` [9]. Since our project is aimed on interactive visualization of urban datasets, we selected three complex scenes related to our project. The *Graz* dataset (Figure 8(a)) consists of a coarse block model of the historic center of the town of Graz.

[1] A multimap is an associative container, in which there may be more than one element with the same key.

The second dataset comprise a digital elevation model of the terrain in north-west Styria (Austria, Figure 8(b)). The third dataset is a digital elevation model of an urban area obtained by an airborne laser scanner (Figure 8(c) and (d)). Some important numbers of these datasets are summarized in Table 1. We set ω to 0.02rad for all datasets.

Dataset name	triangle count	extent	ρ
Graz	162146	2970	1
NW Styria	524288	73704	50
DEM	720000	1032	1

Table 1: Characteristic numbers for the tested datasets. The given extent corresponds with the diameter of the dataset. Since the North-West Styria dataset has a much larger extent than the other scenes, ρ is set accordingly.

For each dataset we recorded a path, which resembles a typical movement through the virtual scene, and we compared our mesh selection method with the top-down mesh adjustment procedure. Graphs showing the obtained selection times are given in Figure 9–11. These figures demonstrate that our approach can effectively reduce the time spent for mesh selection in those cases where the top-down adjustment requires many refinement tests. For those viewing parameters, where only few tests are required, our procedure is comparably fast as the top-down refinement. In these cases the absolute time spent for mesh selection is small, anyway.

5 Conclusion and Future Work

We presented a fast dynamic level of detail refinement method based on screen space error metrics. Our approach can be extended to more complicated refinement criteria incorporating silhouette or lighting information. In these cases the foldtest criterion is more complicated and therefore the closest time of a potential node change is the appropriate combination of several estimated event times. E.g. if silhouette information is incorporated as described in [11], the time of a change in the node's silhouette state has to be estimated additionally.

The speedup of our method is significant; we observe that only about one third of the time of the original selection time is required if the displayed mesh is very complex. Therefore our method usually reduces effectively the overall rendering time, whenever a lot of time is spent for pure rendering. By optimizing math expressions the mesh selection procedure can be accelerated significantly [10] without changing the basic approach. This speed-up applies to traditional refinement methods and to our event-based approach as well.

Since our approach is based on local events, it is of limited use in a time-critical setting, where e.g. a fixed number of rendered triangles must not be exceeded. Nevertheless our method could be beneficial, if the allowed screen-space error is adjusted between frames, such that it is lowered if the budget is exceed and raised if the budget is not exploited completely.

Future work will address the acceleration of refinement strategies in the time-critical setting. Another important topic is the utilization of accelerated rendering primitives like vertex array ranges and triangle strips for view-dependent rendering.

References

[1] J. El-Sana, E. Azanli, and A. Varshney. Skip strips: Maintaining triangle strips for view-dependent rendering. In *Proceedings IEEE Visualization '99*, pages 131–138, 1999.

[2] J. El-Sana and E. Bachmat. Optimized view-dependent rendering for large polygonal datasets. In *Proceedings IEEE Visualization 2002*, pages 77–84, 2002.

[3] L. De Floriani, P. Magillo, and E. Puppo. Building and traversing a surface at variable resolution. In *IEEE Visualization'97*, pages 103–110, 1997.

[4] M. Garland and P. S. Heckbert. Surface simplification using quadric error metrics. In *Proceedings of SIGGRAPH '97*, pages 209–216, 1997.

[5] H. Hoppe. Progressive meshes. In *Proceedings of SIGGRAPH '96*, pages 99–108, 1996.

[6] H. Hoppe. View-dependent refinement of progressive meshes. In *Proceedings of SIGGRAPH '97*, pages 189–198, 1997.

[7] H. Hoppe. Smooth view-dependent level-of-detail control and its application to terrain rendering. In *IEEE Visualization '98*, pages 35–42, 1998.

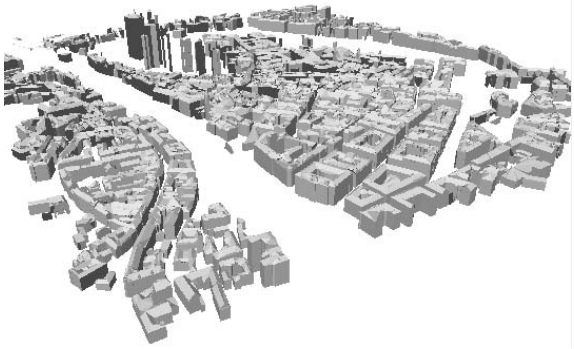
[8] J. Levenberg. Fast view-dependent level-of-detail rendering using cached geometry. In *Proceedings IEEE Visualization 2002*, pages 259–266, 2002.

[9] D. Luebke. View-dependent simplification library. http://vdslib.virginia.edu.

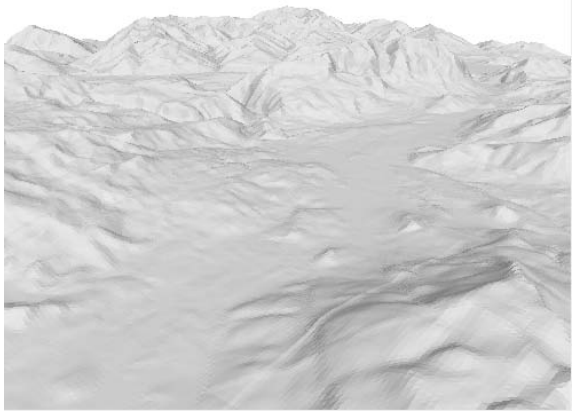
[10] D. Luebke. *View-Dependent Simplification of Arbitrary Polygonal Environments*. PhD thesis, University of North Carolina, 1998.

[11] D. Luebke and C. Erikson. View–dependent simplification of arbitrary polygonal environments. In *Proceedings of SIGGRAPH '97*, pages 199–208, 1997.

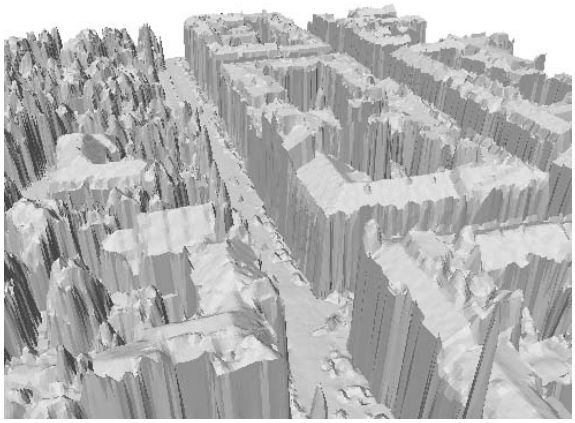
[12] R. Pajarola. Fastmesh: Efficient view-dependent meshing. In *Proceedings Pacific Graphics 2001*, pages 22–30, 2001.



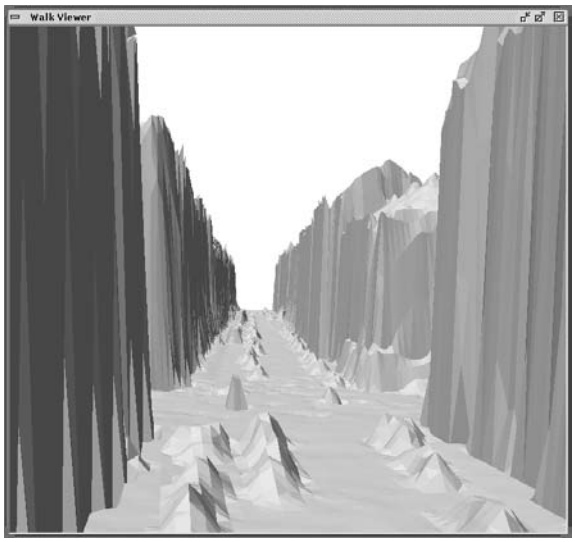
(a) The Graz dataset



(b) A view on the North-West Styria dataset



(c) The DEM dataset



(d) A snapshot from the evaluated path through the DEM dataset

Figure 8: The tested datasets.

[13] J. C. Xia and A. Varshney. Dynamic view-dependent simplification for polygonal models. In *IEEE Visualization '96*, pages 335–344, 1996.

This work has been done in the VRVis research center, Graz and Vienna/Austria (http://www.vrvis.at), which is partly funded by the Austrian government research program Kplus.

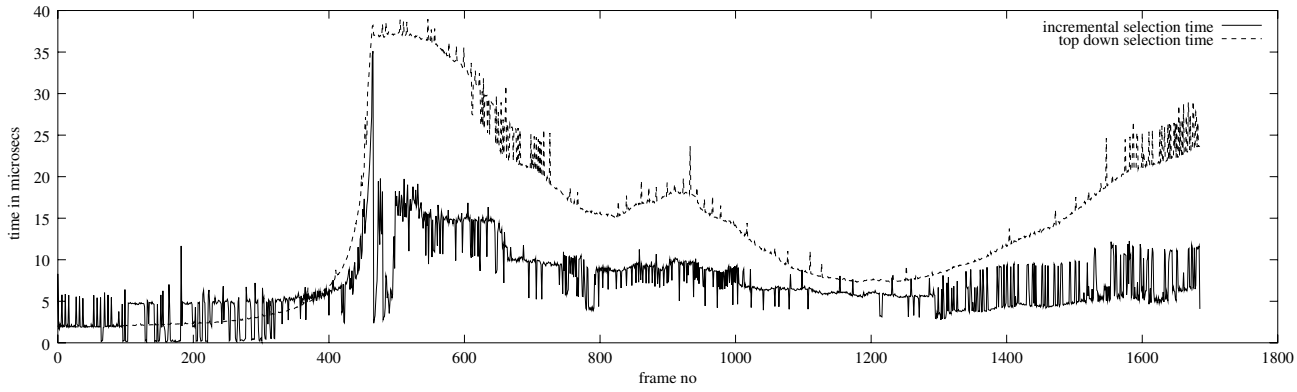


Figure 9: Selection times for the Graz dataset.

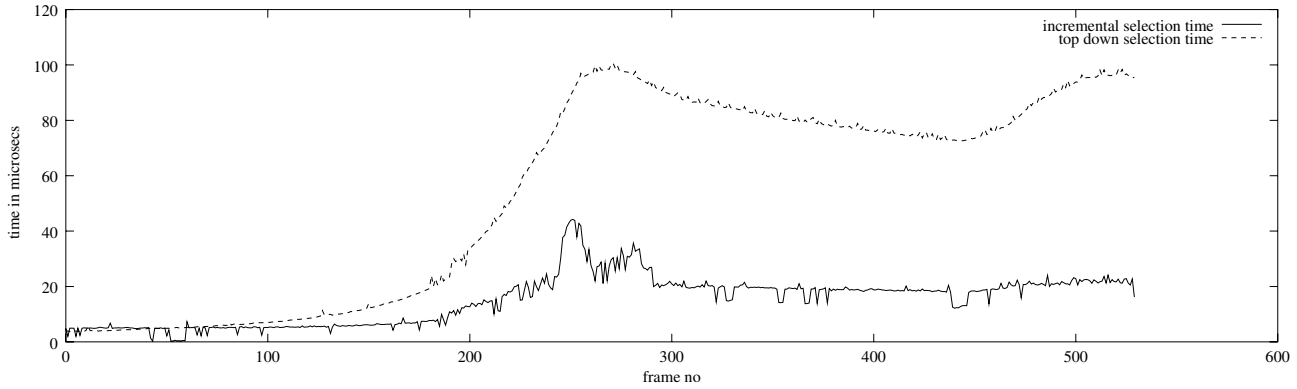


Figure 10: Selection times for the DEM dataset.

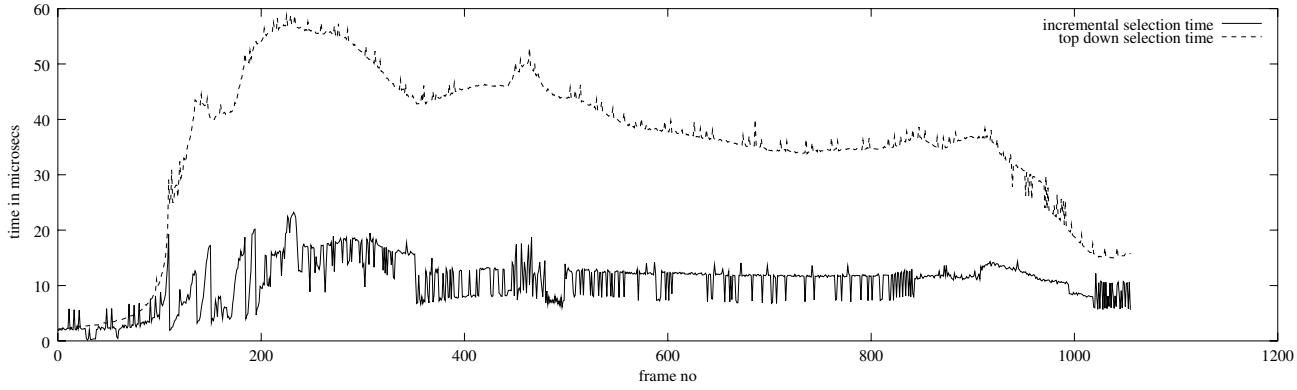


Figure 11: Selection times for the North-West Styria dataset.