Rendering of Large and Complex Urban Environments for Real time Heritage Reconstructions

J.Willmott (willmott@sys.uea.ac.uk) L.I.Wright (liw@sys.uea.ac.uk)
D.B.Arnold (dba@sys.uea.ac.uk) A.M.Day (amd@sys.uea.ac.uk)

University of East Anglia, School of Information Systems, Norwich, NR4 7TJ

Abstract

In this paper we describe a rendering package, which brings together a number of rendering techniques and optimisations to render large and complex urban environments at interactive frame rates. The package has been built on top of a proprietary Scene Graph structure developed for the CHARISMATIC project. The paper presents an integrated approach combining Real-time Optionally Adapting Meshes, View Frustum Culling, Occluder Shadows, Level of Detail for Houses and Motion Captured Avatars in a single application.

We discuss the performance issues associated with our approaches and give comprehensive implementation details. We also consider implications of the trends in high performance graphics sub-systems for PCs in the context of data types and animations required for populated interactive virtual heritage experiences. Finally we present empirical results to demonstrate the importance of employing such techniques and the implications for practical virtual heritage applications.

Keywords: Urban Environments, View Frustum Culling, ROAM, Occluder Shadows, OpenGL, Avatars, Level of Detail, Culling.

1 Introduction

Real-time rendering of large and complex urban environments, populated by virtual humans, is fast becoming an important application in computer graphics. Rendering such large environments at interactive frame rates throws up many specialised problems [4]. For example, a pilot in a flight simulator views large parts of a virtual world at low detail levels, whereas a pedestrian in an urban environment, who only views small parts of the virtual world, expects to see those parts at a considerably higher detail.

The main problem inherent in all urban simulations is the shear complexity of the environments. We believe by combining culling (see section 3) and level of detail (see section 4) algorithms with rendering optimisations and techniques (see section 5), these high fidelity urban environments can now be rendered in real-time using modem day consumer hardware.

In this paper we present a set of both new and existing techniques for rendering large and complex urban environments, innovatively combined together in a single application with of objective of achieving a real-time walkthrough of Medieval Norwich.

2 Scenegraph

At its most basic level, a scenegraph [6, 12] is a data structure that holds all objects in the scene. Most scenegraph structures are tree shaped, with an all-encompassing root node through which all elements of the scene can be accessed.

This is a useful concept in the context of large scenes because it lends itself to object oriented programming. It is intuitive to create a class for each element type that stores all element data and any processing functions associate with it. For visible objects classes, such as Walls or Windows, there will be 'draw' functions that render the object to screen. The root node of the scenegraph has a 'draw' function, which calls the 'draw' functions of its the first-level objects. Each object's draw function, in turn, calls 'draw' on its child objects.

Our research shows that the cities we seek to model consist of approximately 90% vernacular buildings, so many buildings are reused in the scene. Our scenegraph design stores only one copy of each type of building, an instance object is then defined for each copy of the object.

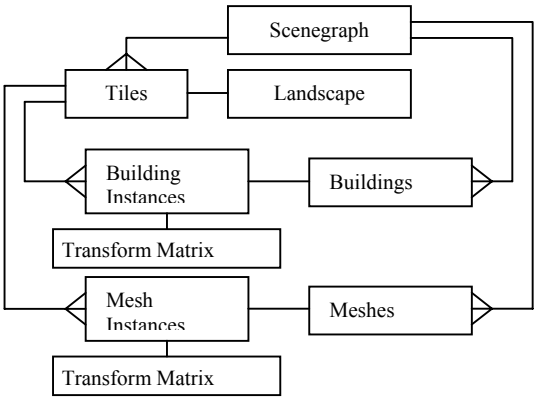


Figure 1: Skeleton diagram of our scenegraph.

Our scenegraph has been especially targeted at urban environments and the methods of culling and of level of detail reduction we intend to use. Our code is written in C++ and uses OpenGL [htt00h] for all its graphics.

Figure 1 shows the root scenegraph node with attached lists of BuildingGeometry and MeshGeometry reference objects. The scene is spatially subdivided into 32x32 or 16x16 Tiles for culling (section 3) or landscape LOD (see ROAM, section 4.1). Each tile

has an attached list of instance objects which are only drawn if the tile is not culled.

Other parts of the scenegraph are also designed for speed, for example the BuildingGeometry structure pictured in Figure 2. The building data are hierarchically structured into Levels, Planes (Walls) and "ExternalGeometry" objects, which are attached to walls. This makes it easy to turn off rendering of detail on the walls of buildings when the viewer is too far away to see any detail (see house LOD, section 4.2).

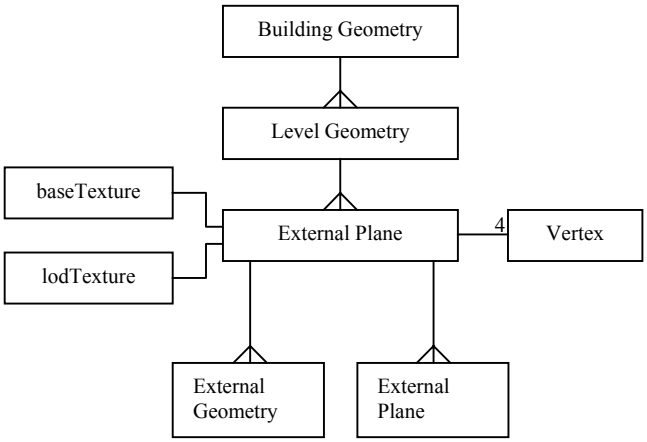


Figure 2: Building geometry structure.

All of the culling and level of detail techniques discussed in this paper are supported by our underlying scenegraph data structure. It is outside the scope of this paper to describe the rest of the structure in full, but suitable reference and explanations of the pertinent parts of the scenegraph will be provided where required.

3 Culling Techniques

3.1 View Frustum Culling

3.1.1 Introduction

View frustum culling [htt00a, htt00b, htt00c] is an object based culling algorithm that only displays objects within the current viewing volume. This viewing volume is defined by six clipping planes: front, back, left, right, top, and bottom, forming a cut pyramid. If any part of the object is within the frustum, it is deemed visible and displayed, however if all parts are entirely outside the frustum, the object is deemed invisible and culled (see figure 3).

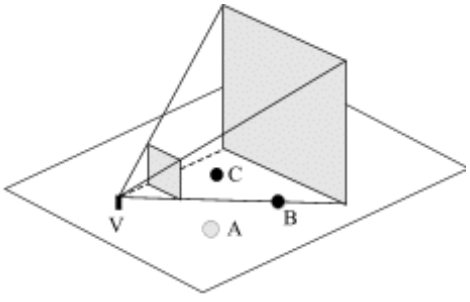


Figure 3: Object A, which is entirely outside the view frustum, is culled. Object B (partially inside) and object C (entirely inside) are both displayed.

3.1.2 Constructing the frustum

The view frustum is constructed once per frame from the OpenGL viewing matrix [htt00d]. Below is the pseudo code to construct the frustum from OpenGL's PROJECTION and MODELVIEW matrices.

```
constructViewFrustum()
{
  ~ get the current PROJECTION matrix from
  OpenGL
  ~ get the current MODELVIEW matrix from OpenGL
  ~ combine the PROJECT and MODELVIEW matrices
  ~ extract the RIGHT, LEFT, BOTTOM, TOP, NEAR
  and FAR  planes and normalise
}
```

3.1.3 Culling the objects

Any object in the scene, such as tiles of the terrain, avatars, buildings, trees and meshes, can potentially be culled by the view frustum. The culling is achieved by testing the visibility of the bounding box entirely encapsulating the object against the frustum. If the bounding box is deemed entirely outside, the object is culled.

3.1.4 Results

The computer used to collate the results presented in this paper was a Pentium III 733mhz processor with 512MB memory and a GeForce3 graphics card.

A model based on Medieval Norwich comprising 800 randomly placed buildings producing 222,912 polygons, created by our Modelling [2] and Scene Assembler [9] packages, was used to test the performance of the View Frustum culling algorithm. We recorded a walkthrough passing through both sparse and densely populated areas of the model. The average frame rate for the walkthrough is shown in figure 4.

Model Complexity	Without Culling	With Culling
222,912 polygons	1.83	30.45

Figure 4: Average number of frames per second for the walkthrough of Medieval Norwich using View Frustum culling.

These results show that significant performance gains can be made using View Frustum culling as large sections of the model can be culled high up the display pipeline.

3.2 Occluder Shadows

3.2.1 Introduction

Occluder Shadows [1, 5, 7, 10, 11, 13, 14] is an image based culling algorithm that exploits specialised graphical hardware to rapidly cull the parts of the scene obscured by large objects, such as buildings, allowing only those parts visible to the viewer to be displayed.

The concept of occluder shadows is based on the following observation: a large building in front of the viewer may totally obscure any number of buildings directly behind the building in question. These buildings are totally invisible (see figure 5) and

in a naïve rendering system would still be displayed. The occluder shadows algorithm allows relatively large sets of occluders to be rendered and automatically combined using graphics hardware, enabling real-time occlusion culling within large and complex urban environments.

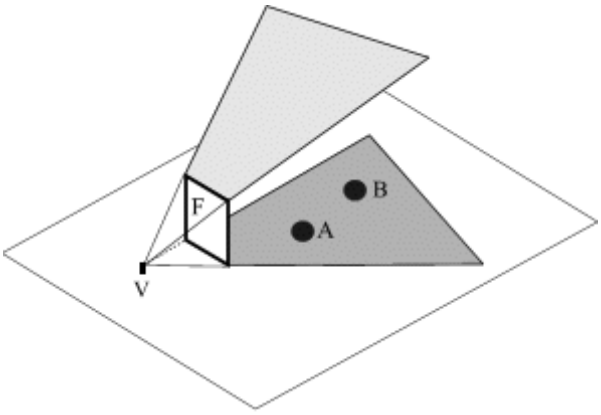


Figure 5: Buildings A and B are totally obscured by building front (F), as seen from the viewpoint V.

Our implementation of the Occluder Shadows algorithm is based on the paper by Wonka and Schmalstieg [14].

3.2.2 An Occluder Shadow

Given a viewpoint *V*, an occluder polygon *P*, defined by the viewpoint and two end points of an occluder line, casts an occluder shadow into the scene, occluding an area lying directly behind the occluder (see figure 6). This area, determined by the occluder and the viewpoint, forms a shadow frustum, and any object entirely in this shadow frustum is deemed invisible and can be culled.

Due to the potentially high occlusion rate of buildings in urban environments, the Occluder Shadows algorithm is ideally suited to this type of occlusion problem as hundreds, or even thousands, of buildings can be culled by just a relatively small set of these occluder shadows.

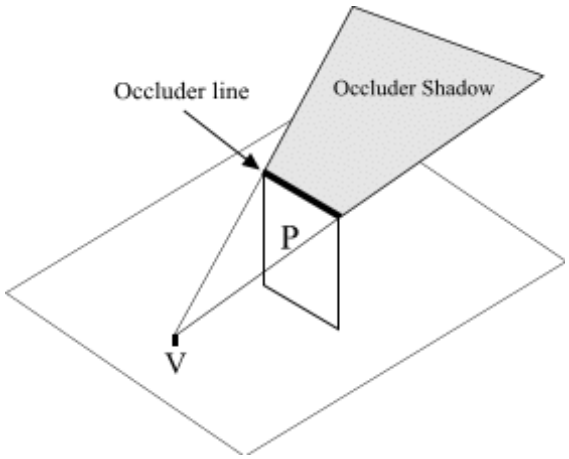


Figure 6: An occluder shadow defined by a viewpoint V and occluder line.

3.2.3 Algorithm outline

A 2D regular grid is created that maps directly onto the terrain, this grid is known as the cull map. Each cell of the cull map represents a physical area on the terrain. Per frame, a set of occluders are selected and rendered into an auxiliary buffer on the graphics card. Each pixel of this buffer (image space) corresponds to a cell in the cull map (object space). Once all the occluders have been rendered to the frame buffer, the depth component is read back and stored in the cull map. Visibility for each building in the scene is determined by comparing the maximum height of the building with the z value from the corresponding cell in the cull map. If the height of the building is less than the corresponding value in the cull map, the building lies beneath the occluder and can be culled.

3.2.4 Setting up the cull map

The resolution of the cull map determines how fast and accurate the algorithm performs occlusion culling. A higher resolution cull map produces more accurate occlusion, however it takes longer to read back the pixels from the frame buffer. The complexity of the scene, and size of the terrain, also influences the resolution of cull map. If the same accuracy of building occlusion is required, the resolution of the cull map scales proportionally with the size of the terrain.

The data type of the cull map also affects the performance and accuracy of the algorithm. We used unsigned shorts, as this proved to be an acceptable compromise between the speed at which the pixels are read back from the frame buffer and accuracy of occlusion (see section 3.2.7).

We used a 256 by 256 cull map and a terrain size of 2048m by 2048m in our implementation. Each cell of the cull map represents an 8m by 8m area of terrain.

A 2D orthogonal OpenGL projection, coincident with the ground plane and matching the resolution of the cull map, was used to represent the cull map.

3.2.5 Selecting the occluder set

Ideally only the set of occluders producing the best possible occlusion would be used to create the cull map, however finding such a set can be a very computationally expensive process.

We favoured a more simplistic, brute force approach to selecting the occluder set. Instead of using valuable computation time to identify a good set of occluders, we use the set of occluders belonging to each building within each tile of the local model (see figure 7).

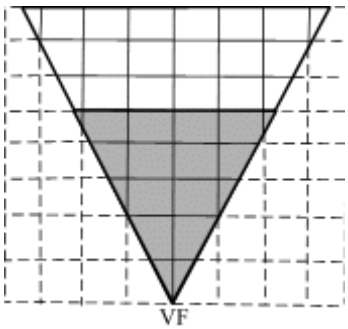


Figure 7: Only occluders belonging to tiles within the local model (grey area) and the view frustum (VF) are used to build the occluder set.

A set of occluder lines are identified for each building at the modelling stage [2]. These occluder lines are usually the facades, or tops, of the buildings, and are defined by two end points, *Va* and *Vb* (see figure 8).

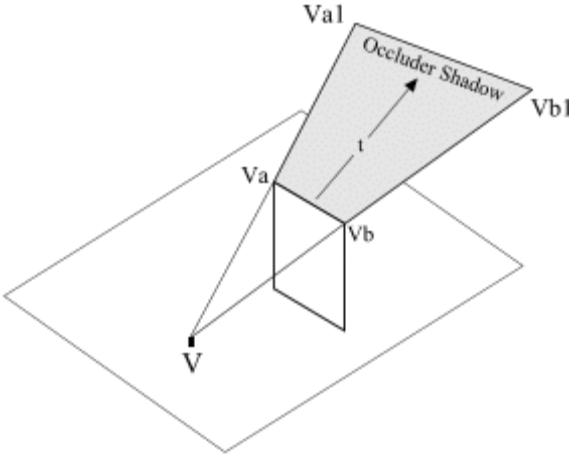


Figure 8: An occluder shadow formed by the viewpoint V and occluder line Va to Vb.

When we render the set occluders to the frame buffer, we disable back-face culling. Whether an occluder is front or back facing is irrelevant in our implementation, as all we are interested in is the depth component of the frame buffer. Regardless of its orientation, an occluder polygon only gets rendered once per frame.

3.2.6 Drawing the occluders

The occluders, selected in section 3.2.5, are now rendered to the cull map using the graphics hardware.

We initially clear the frame buffer with all zeros, and set the depth function to GL_GREATER. Then for every tile within the local model, if it has not been view-frustum culled, construct the occluders between the viewpoint *V* and the occluder lines for every building also within the view frustum. Since we cannot project the occluder shadow to infinity, we simply assume a large value for *t* (ranging from $0 < t < \infty$) so that the edge of the shadow `Va1` to `Vb1` lies fully outside the cull map (see figure 8).

The calculation used to create the occluder shadow is shown below.

```
Va1 = Va + (t * (Va - V))
Vb1 = Vb + (t * (Vb - V))
```

Because OpenGL produces a non-conservative estimation of occlusion in the cull map, due to under sampling, we have to shrink each occluder by an amount *s,* half the physical width of each cell in the cull map (see figure 9). This shrinking of the occluder ensures (1) the z values in the cull map are correct and (2) we avoid rendering occluder shadows over pixels that are only partially occluded.

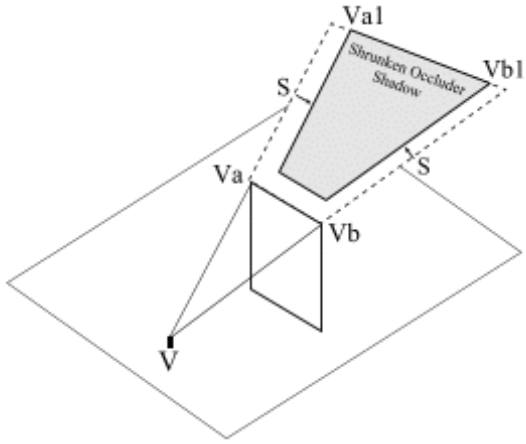


Figure 9: Shrinking an occluder shadow by amount s.

Once the occluder has been shrunk, we need to find if the two edges `Va` to `Va1` and `Vb` to `Vb1` intersect. If there is an intersection point `Vd`, an occluder shadow (a triangle between `Vd`, `Va1` and `Vb1`) is formed and rendered to the cull map.

If there is no intersection point, an occluder shadow (a quadrilateral between `Va`, `Va1`, `Vb` and `Vb1`) is formed and rendered to the cull map. Below is the pseudo code to construct the cull map.

```
drawCullMap()
{
  T = tile
  VF = view frustum
  B = building
  OL = occluder line
  OP = occluder point
  OS = occluder shadow
  VP = viewpoint

  For each T in local model
    If T is within VF
      For each B in T
        If B is within VF
          For each OL in B
            ~ Build OS from VP and OL
            ~ Shrink OS by amount s
            ~ Draw OS
}
```

3.2.7 Reading back from the frame buffer

Once all the occluders have been rendered, the depth component of the frame buffer is read back and stored in the cull map as unsigned shorts. Reading back floating point numbers produces the greatest occlusion culling accuracy, but it is significantly slower than reading back unsigned shorts.

Once the pixels have been read back and stored in the cull map, the values within the cull map range from 0 to 65535. These values represent a scale between the near and far clipping planes of the orthogonal projection. By dividing the height information in the cull map by the difference between the far and near clipping planes, the height of the occluder in real world co-ordinates is found.

3.2.8 Occlusion culling the Buildings

Occlusion culling of the buildings is achieved by testing the buildings occluder point (the x and z component being the centre, and the y component being the highest point, of the building) against its corresponding height in the cull map. The pseudo code for occlusion culling a buildings occluder point is shown below.

```
pointOccluded(p)
{
  If height of p >= height in associated cull
  map cell
    Return true
  Else
    Return false
}
```

For every tile within the view frustum, each building also within the frustum is checked to see if it is within the local model. If it is, the building is assumed visible and not checked for occlusion, as an occluder shadow can sometimes cull out the building it was cast from. If the building is not within the local model, the buildings occluder point is checked against the cull map too see if the building can be culled. Below is the pseudo code for occlusion culling the buildings.

```
drawBuildings()
{
  T = tile
  VF = view frustum
  B = building
  OP = occluder point

  For each T
    If T is within VF
      For each B in T
        If B is within VF
          If B is within local model
            Draw building
          Else
            If pointOccluded(OP)
              Draw building
}
```

3.2.9 Results

The same model (see section 3.1.4) used to test the performance of the View Frustum culling algorithm was used to test the Occluder Shadows algorithm. Figure 10 shows the average frame rate for the walkthrough of the model.

Model Complexity	Without Culling	With Culling
222,912 polygons	1.83	7.54

Figure 10: Average number of frames per second for the walkthrough of Medieval Norwich using Occlusion culling.

These results show that, in the case of our walkthrough of Medieval Norwich, the average frame rate was quadrupled using the Occlusion culling algorithm.

4 Level of Detail Techniques

4.1 Real-time Optionally Adapting Meshes

4.1.1 Introduction

One of the biggest problems in terrain visualisation is how to store the features inherent in landscape. Height fields (height maps) are the most popular solution – a two-dimensional array holding the height of the terrain at that point [3, htt00e]. In this paper, we use a height map generated by our Terrain Generator [9].

In our final test model based on Medieval Norwich (see section 6) we used a height map of dimensions 401 points east by 401 points north, which represented an area of terrain 2048m by 2048m. The spacing between points is 5.12m. Rendering the terrain at maximum detail would require 321,602 polygons, which is far in excess of the average home computer's capability. Therefore some kind of level of detail (LOD) algorithm is required.

4.1.2 Real-time Optionally Adapting Meshes

Real-time Optionally Adapting Meshes (ROAM) [8, htt00e, htt00f] is a view-dependant level of detail algorithm suitable for large terrain data sets. Our implementation of the ROAM algorithm is based on the paper by Duchaineau et al [8] and the article by Turner [htt00e].

We will not discuss the implementation of the ROAM algorithm in this paper, however we do quote some results for the performance increases (see section 4.1.3) in using ROAM.

4.1.3 Results

Our implementation of the ROAM algorithm was tested with a height map resolution of 401 by 401. Figure 11 shows the benefits of using ROAM for the terrain.

Resolution of Height map	Without ROAM	With ROAM
401 by 401	9.26	43.52

Figure 11: Average frames per second for a walkthrough of the 401 by 401 height map.

These results show using ROAM allows large terrain data sets to be rendered in real-time, whilst still preserving a high level of detail around the viewer.

4.2 House LOD

The data structure for buildings pictured in figure 2 is especially designed for a LOD technique developed especially for houses. A performance advantage is gained by turning off the rendering of and geometry attached to building's walls. This includes both 'External Geometry' and 'External Opening' objects. Figure 12 shows the difference between these two entities.

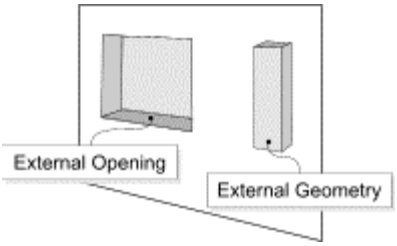


Figure 12: External Opening and External Geometry attached to a wall.

4.2.1 Reducing popping

Simply switching rendering of external detail will have a noticeable visual impact. The occurrence of 'popping' in an interactive scene is unwelcome. Our technique reduces this in two ways.

1) The geometry 'pops in' behind the plane of the wall and is translated forward as the view approaches. External openings start off flush and become deeper.
2) The wall has a special texture map applied that represents any missing external geometry.

We will discuss these two methods first in isolation and then how they can work together.

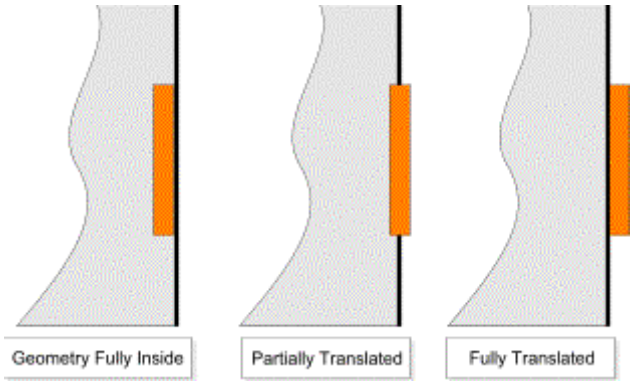


Figure 13: External Geometry pushing through a wall.

Method (1) provides an improvement over simple switching because the eye is not so apt to see a gradual change in the scene. Also, since the amount of translation is governed by the user's distance from the building, the geometry only moves if the user is moving. Our experiments show that the eye is even less capable of noticing sliding geometry when the viewpoint is changing.

However, method (1) on its own is not effective in the case of geometry with a leading plane that is parallel to the wall it projects from. In cases like these, popping starts to occur because a large area of geometry becomes visible all at once. Unfortunately, the majority of desired external geometry items, such as window frames and doors, have this parallel-plane property.

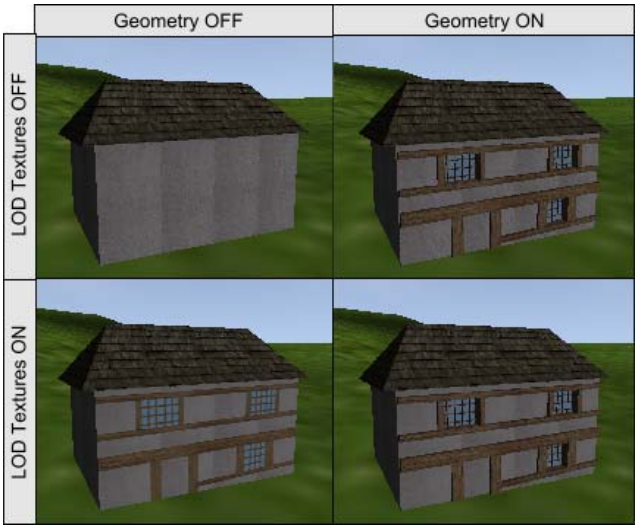


Figure 14: Comparison of house with and without a lodTexture.

Method (2) provides an improvement over simple switching because the texture map that remains still represents the missing geometry in some way. Since it is difficult to notice parallax at a distance, the fact that all the geometry has suddenly become flat is not too noticeable when the switch is made at a great distance. However, as the distance of the switch is increased, so does the complexity of the scene, so it is desirable to choose a switching distance as close as possible to the building.

Bringing both of these methods together overcomes the parallel plane problem of (1) and the distance-switching problem of (2). The sudden appearance of the leading edge of a wooden windowsill in a plaster wall is not noticeable if the wall's texture already includes the windowsill.

Similarly, the problem with (2) is the sudden change in depth of external objects. Since method (1) adjusts the amount of projection of the external geometry smoothly, the switch distance can be brought much closer to the building.

4.2.3 Translating External Geometry

To make building modelling more intuitive, the external geometry components are modelled at their full level of detail position. This means that the render applies full translation when the geometry first switches in. This translation is then relaxed as the viewer approaches.

The amount of translation applied to the external geometry of an object is dependant on three independent factors:

1) The distance of the viewer, *viewDistance*.
2) The depth of the geometry item, *geometryDepth*.
3) A user controlled global scalar held in the scenegraph, *houseLODScalar*.

Two distances, *nearLOD* and *farLOD* are determined as follows:

```
NearLOD = houseLODScalar*geometryDepth;
farLOD = 4*(houseLODScalar*geometryDepth);
```

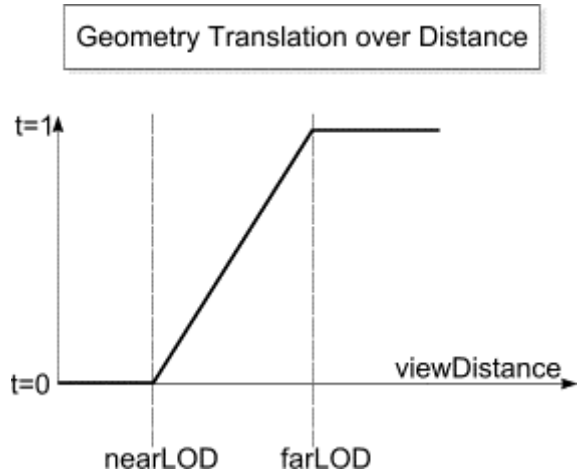


Figure 15: The function used to calculate translation amount.

These two distances are then used to find *t*, the percentage of translation. The graph in figure 15 shows the linear function encoded below.

```
t=1-((farLOD/(farLOD-nearLOD))-
(viewDistance/(farLOD-nearLOD)));
if (t>1) t=1; //Clamp T
if (t<0) t=0;
```

4.2.4 Creating and Grabbing LOD Textures

Figure 2 shows each ExternalPlane object with two textures attached – *baseTexture* and *lodTexture*. The *baseTexture* object holds the background texture data that would be rendered without House LOD turned on. The *lodTexture* object must be pre-generated when the renderer starts. This is done by going round each wall of every house and constructing an orthogonal projection matrix that exactly captures the plane (rendered with its base texture) and any external geometry in front of it.

It is possible to derive the projection matrix from the wall vertices. This is done by finding the corner vertices of the bounding volume shown below.

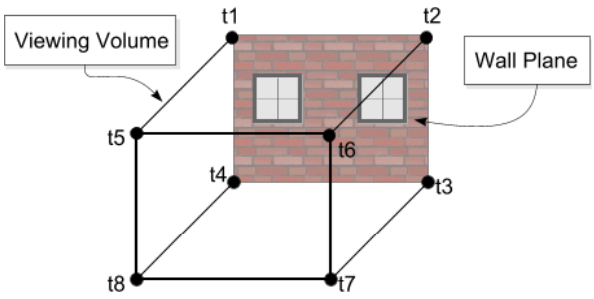


Figure 16: The target view volume t1..t8.

OpenGL requires vertices to be transformed into the cube of space between (-1,-1,-1) and (1,1,1), so the correct projection matrix is the one which transforms vertices t1..t4 to s1..s4 and t4..t8 to s4..s8 where:

```
s1(-1,-1, 1) s5(-1,-1,-1)
s2( 1,-1, 1) s6( 1,-1,-1)
s3( 1, 1, 1) s7( 1, 1,-1)
s4(-1, 1, 1) s8(-1, 1,-1)
```

Now s1..s8 and t1..t8 are known, the projection matrix can be derived from a system of linear equations.
Once the projection matrix has been set, the wall can be rendered at its full detail level. A *glReadPixels()* operation is then used to grab the new texture to system memory where it is placed in the scenegraph. Finally, the *lodTexture* object is associated with this new texture and is used when rendering the wall from that point on.

4.2.5 Results

The same model (see section 3.1.4) used to test the performance of the View Frustum and Occlusion culling algorithms was used to test our house LOD algorithm. Figure 17 shows the average frame rate for the walkthrough of the model.

Model Complexity	Without LOD	With LOD
222,912 polygons	1.83	12.15

Figure 17: Average number of frames per second for the walkthrough of Medieval Norwich using our specialised house LOD.

5 Rendering Pipeline Techniques

5.1 Static objects

A major disadvantage to object oriented scenegraph approaches like ours is that the object encapsulation leads to many redundant state changes. This is caused because each object is a separate entity which must ensure that the GL state is set correctly before rendering. There are two common approaches to this:

1) Ensure that the state is correct by setting all important objects before glBegin().
2) Read the current state and only switch state if it is not correct.

Approach (1) is the most robust method but can cause many hundreds of redundant state changes. Approach (2) can be more efficient, but making unnecessary state queries is also inefficient.

A preferred approach is to pre-order the rendering of objects so that state changes are kept to a minimum. Our scenegraph orders the rendering of the External Planes of buildings by its texture number. The texture state change was chosen because it is the most expensive type of switch.

When state ordering is enabled, the *draw()* function in the *ExternalPlane* class does not immediately render the wall quad. Instead, it adds it to a list of quads to be rendered in the *BuildingRenderer* object. The *BuildingRender* object's data structure is pictured in figure 18, below.

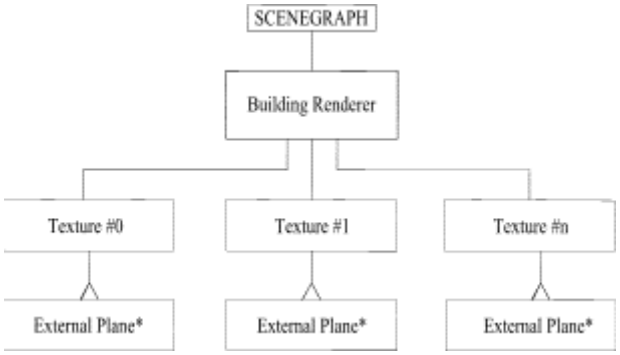


Figure 18: Building Renderer structure

When all the *draw()* functions on *Building* objects have all been called, the *draw()* function is called on *BuildingRenderer* object. The Building Renderer can now step through its list of quads and render them with the minimum number of texture switches. Finally, the *BuildingRenderer* object has its lists cleared for the next frame.

We are currently working on more advanced state ordering method which handles all types of static geometry and uses vertex arrays for extra speed. We are also researching methods which do not require the Building Renderer's lists to be cleared every time.

5.2 Dynamic objects

The model of Medieval Norwich was populated with a number of non-intelligent virtual humans (avatars). These avatars are simple key-frame interpolated motion captured avatars that plays single motions from a pre-built library. As these mesh-deforming avatars are by definition dynamic objects, performing the key-frame interpolation and rendering is a time consuming problem.

Very little pre-processing, such as using call lists, can be used to speed up the rendering of an avatar's mesh as it is constantly being deformed. We instead used the OpenGL extension vertex arrays [htt00g] to speed-up the rendering and set a variable number of animations per second for each avatar.

We also are looking at other ways to speed up the rendering of the avatars. Performing the key-frame interpolation on the Graphics Processing Unit (GPU) of the GeForce3 [htt00g] by using vertex programs and using a level of detail algorithm to simplify the avatars depending on their distances to the viewpoint are just a two examples.

6 Results

The model based on the Medieval Norwich (see section 3.1.4) was combined with the 401 by 401 height map (see section 4.1) to produce a new model comprising 544,002 polygons. We now present some preliminary results (see figure 19) of combining the four approaches, View Frustum culling (see section 3.1), Occlusion culling (see section 3.2), ROAM (see section 4.1) and our own specialised house level of detail (see section 4.2), in a single walkthrough of the new model based on Medieval Norwich comprising both buildings and terrain.

Model Complexity	Without Speed-ups	With Speed-ups
544,002 polygons	1.60	35.54

Figure 19: Average number of frames per second for the walkthrough of Medieval Norwich using all four rendering speed-ups.

These results show that environments previously too complex to render at interactive frame rates on high-end home users PCs can now be rendered in real-time by combining the four rendering speed-ups presented in this paper in a single application.

7 Conclusion

Rendering large and complex urban environments is a rapidly evolving area of computer graphics. Only recently, with the advent of faster and cheaper computers, has it has become more attainable to other sectors, such as education and commerce.

The Charismatic project aims to produce very large urban models, as shown in our test scene of Medieval Norwich, at very high levels of fidelity. We believe such a high fidelity model can only be achieved by combining, in a single application, level of detail and culling algorithms with rendering optimisations and the very latest in graphics hardware

8 Future Work

The techniques presented in this paper are just an example of the work we are doing here at UEA. We intend not only to improve on our View Frustum and Occlusion culling, ROAM and specialised house LOD implementations, but include many more rendering speed-ups such as OpenGL extensions, Cells and Portals and Impostors in our CHARISMATIC rendering package.

Acknowledgements:

This work has been conducted as part of the CHARISMATIC project, which is a fifth framework project (IST-99-11090), and hence part-supported by the European Union. Our thanks go to other partners in the project and other members of the UEA team (Peter Birch, Shaun Browne, Philip Flack, and Vince Jennings) for discussion and comments on the approaches taken.

References:

Papers:

[1] P.Agarwal, S.Har-Peled and Y.Wang. Occlusion Culling for Fast Walkthrough in Urban Areas. URL: http://valis.cs.uiuc.edu/~sariel/papers/00/occlusion.html

[2] S.P.Browne, P.J.Birch, V.J.Jennings, A.M.Day and D.B.Arnold. Rapid Procedural Modeling of Architectural Structures. In proceedings of VAST, Athens, November 2001.

[3] S.P.Browne. Internal CHARISMATIC working paper "*Av evaluation of existing terrain data formats with regard to inclusion in the Charismatic terrain editor*". University of East Anglia.

[4] S.P.Browne, J.Willmott, L.I.Wright, A.M.Day and D.B.Arnold. Modelling & Rendering Large Urban Environments. In proceedings of EGUK 2001, UCL, London.

[5]] S.Coorg and S.Teller. Real-Time Occlusion Culling for Models with Large Occluders. In proceedings of 2001 Symposium on Interactive 3D graphics, North Carolina, USA.

[6] S.Cunningham and M.J.Bailey. Lessons from scene graphs: using scene graphs to teach hierarchical modelling. ISSN 0097-8493.

[7] L.Downs, T.Moller and Sequin. Occlusion Horizons for Driving through Urban Scenery. In proceedings on 2001 Symposium on Interactive 3D graphics.

[8] M.Duchaineu, M.Wolinsky, D.Sigeti, M.Miller, C.Aldrich and M.Mineev-Weinstein. ROAMing Terrain: Real-Time Optionally Adapting Meshes. URL: http://www.llnl.gov/graphics/ROAM/

[9] P.A.Flack, J.Willmott, S.Brown, A.M.Day and D.B.Arnold. Scene Assembly for large scale urban reconstructions. In proceedings of VAST, Athens, November 2001.

[10] R.Germs and F.Jansen. Geometric Simplification For Efficient Occlusion Culling In Urban Scenes. In proceedings of WSCG, Pilse, February 2001. URL: http://wscg.zcu.cz/

[11] V.Koltun and D.Cohen-Or. Selecting Effective Occluders for Visibility Culling. In proceedings of Eurographics, Interlaken, Switzerland 2000.

[12] J.Rohlf and J.Helman. IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics. In proceedings of SIGGRAPH 94, Orlando, Florida.

[13] G. Schaufler, J.Dorsey, X.Decorwt and F.X.Sillion . Conservative Volumetric Visibility with Occluder Fusion. In proceedings SIGGRAPH 2000, New Orleans, Louisiana

[14] P.Wonka and D.Schmalstieg. Occluder Shadows for Fast Walkthroughs of Urban Environments. In proceedings of Eurographics, Milan, Italy 1999.

URLs:

[htt00a] "*Culling your scene to the View Frustum*" by S.Baker / http://web2.iadfw.net/sjbaker1/frustcull.html

[htt00b] "*Geometry Culling in 3D Engines*" by P.Laurila / http://www.gamedev.net/reference/programming/features/culling/

[htt00c] "*Faster 3D Game Graphics by Not Drawing What Is Not Seen*" by K.Hoff III / http://www.cs.unc.edu/~hoff/papers/vfc/vfc.html

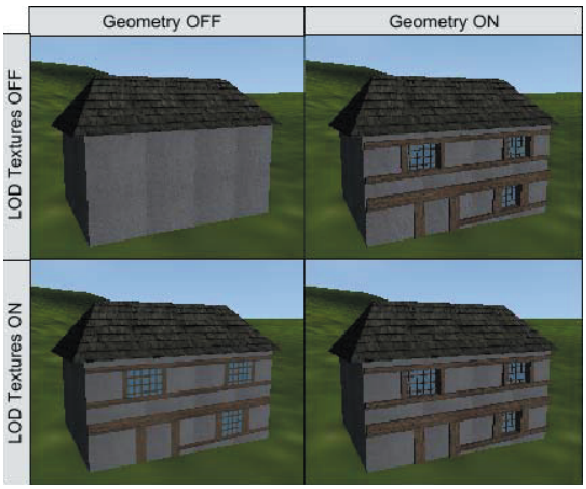
[htt00d] "Frustum Culling in OpenGL" by M.Morley / http://www.markmorley.com/opengl/frustumculling.html

[htt00e] "*Real-Time Dynamic Level of Detail Terrain Rendering with ROAM*" by B.Turner / http://www.gamasutra.com/features/20000403/turner_01.htm

[htt00f] "*ROAM Implementation Optimizations*" by Y.Gyurchev / http://www.flipcode.com/tutorials/tut_roamopt.shtml

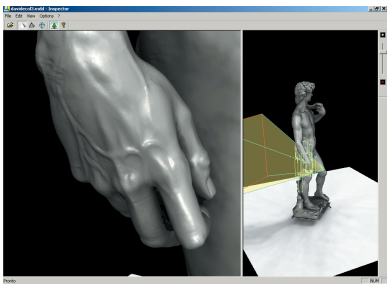
[htt00g] http://www.nvidia.com

[htt00h] http://www.opengl.org

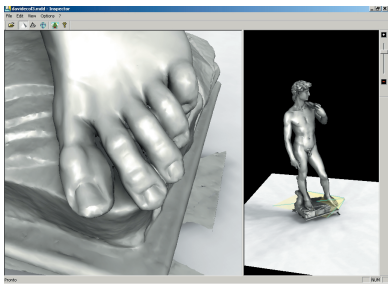


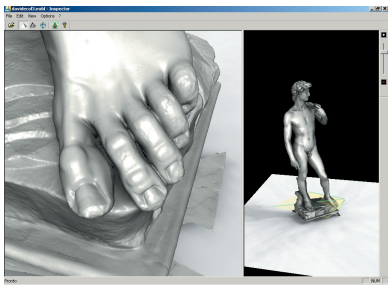
Comparison of house with and without a lodTexture.

Willmott, Wright, Arnold, Day: **Rendering of Large and Complex Urban Environments for Real time Heritage Reconstructions,** pp. 111-120.

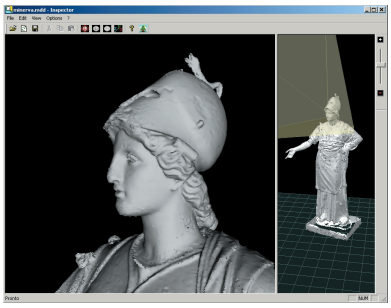


A tighter field of view is chosen, by using the vertical slider on the right.





A tight look of left David's foot. In the left image we show the lower resolution model choose for interactive phases while in the right image we show the high resolution model rendered in background when the user stop interaction.



A snapshot of the Minerva model is shown.

Rendering cast shadows greatly affects the resulting expression of the David's face. On the left a standard OpenGL rendering with Phong lighting, on the right a OpenGL rendering with a more correct precomputed lighting.

Borgo, Cignoni, Scopigno: **An Easy-to-use Visualization System for Huge Cultural Heritage Meshes,** pp. 121-130.