



3D Navigation Rendering Techniques

Whitepaper

Copyright © Imagination Technologies Limited. All Rights Reserved.

This publication contains proprietary information which is subject to change without notice and is supplied 'as is' without warranty of any kind. Imagination Technologies and the Imagination Technologies logo are trademarks or registered trademarks of Imagination Technologies Limited. All other logos, products, trademarks and registered trademarks are the property of their respective owners.

Filename : 3D Navigation Rendering Techniques.Whitepaper
Version : PowerVR SDK REL_17.1@4658063a External Issue
Issue Date : 07 Apr 2017
Author : Imagination Technologies Limited

Contents

1. Introduction	3
1.1. Point-of-View Types	3
1.2. Sample Data	3
2. Data Organisation	4
2.1. Data Conversion	4
2.2. Spatial Index	4
2.3. Occlusion Culling	6
3. Rendering Techniques	9
3.1. Skybox	9
3.2. Shadows	9
3.3. Batching	11
4. References	12
5. Contact Details	13

List of Figures

Figure 1. City tiles intersecting the view frustum.....	5
Figure 2. Working set and per-object culling.....	6
Figure 3. View of the Chicago city model as rendered from the demo application.....	6
Figure 4. Combine wireframe and shaded views of the geometry.....	7
Figure 5. Visibility set determined by camera position.....	8
Figure 6. Visibility set and per-object frustum culling.....	8
Figure 7. Comparison of a scene with a black background (left) and a skybox (right).....	9
Figure 8. Comparison of a scene without shadows (left) and with shadows (right).....	10
Figure 9. Visualization of the extruded shadow volumes.....	10
Figure 10. Hierarchical overview of vertex (VBO) and index (IBO) buffer objects for a city block	11

1. Introduction

Visualization of large datasets is a complicated and demanding task. Efficient techniques have to be employed in order to achieve interactive frame rates when rendering huge datasets. The first document on this topic (please see the “PowerVR™ Navigation Rendering Techniques” document) deals with the aspects of 2D and 2.5D map rendering.

This document introduces techniques related to the visualization of 3D datasets. It provides additional information on the topics of optimization, extending those found in the “PowerVR 3D Application Development Recommendations” document.

1.1. Point-of-View Types

There are several high-level approaches to rendering a navigation system. They mainly differ in the point of view and the amount of detail being rendered. From a different perspective this means they differ in the minimum hardware specs they require from the targeted device to be able to run at an appealing frame rate. The following gives an overview of the most common types:

- **2D top-down:** the standard bird’s-eye perspective found in a lot of navigation devices. It features a very limited field of view, concentrating on basic features like streets, signs and landmarks and can be rendered using an orthographic projection scheme. The terrain and all the landmarks are specified in a single plane.
- **2.5D:** this perspective shares the same set of features with the plain 2D one, but the camera is slightly tilted to offer a wider field of view. Due to the viewing angle and the perspective projection, artefacts like line-aliasing have to be considered. Furthermore, it is desirable to add 3D models of important buildings to provide reference points for the user. As with the previous view all the landmarks are specified in a single plane.
- **3D:** this view is similar to the 2.5D view, but now all coordinates have an additional z-coordinate which makes it possible to illustrate additional landscape features like elevation. In addition to the 3D coordinates, the map data can be augmented by enhanced features like 3D models of the actual city buildings.

This document covers the last entry in the list, on how to efficiently render 3D city models.

1.2. Sample Data

The sample data used throughout the 3D navigation demo has been kindly provided by NAVTEQ.

In particular, sample data from NAVTEQ Enhanced 3D City Models for Chicago, provided in the Collada™ interchange format, was used and transformed into Imagination Technology’s binary POD format, which in turn was used for the 3D navigation demo. The city models are split into tiles of approximately 400m by 400m side length, where each tile is stored as a separate POD file. Each tile contains textured buildings, roads and bridges, which resemble the original city.

The following sections describe the optimization techniques used for the demo and the sample data. Implementations of the algorithms and techniques can be found in the PowerVR SDK and an explanation of the various navigation data tools used to convert the sample data can be found in the “Navigational Data Tools Reference Guide”.

2. Data Organisation

Rendering the 3D models out-of-the-box as they are delivered is not possible. The models themselves are stored in the Collada interchange format, which is primarily used as a standardized storage format for data exchange between authoring software packages.

The format itself is a human-readable XML format which is not meant for deployment. It includes metadata which is not required for the rendering process and the file size itself is a multiple of comparable binary deployment formats.

As the models have to be deployed to mobile platforms, the data set has to be reduced to the bare minimum. Deployment size matters due to limited storage and slower transmission speeds so the data is better in an efficient binary format which is fast to load. The PowerVR SDK provides a set of tools that accomplish these tasks.

The next section will introduce a few of these tools and how they can be used to compile and generate the required deployment content. The later sections deal with the process of generating spatial index data to accelerate the rendering process.

2.1. Data Conversion

As already mentioned in Section 1, the Collada file format is not suitable for rendering the data out-of-the-box. The data has to be converted into the PowerVR POD file format for deployment. The PowerVR SDK includes a set of tools to accomplish this task:

- **PVRGeoPOD**: a plug-in for Autodesk 3DS Max 6, 7, 8 & 9 and Autodesk Maya 7 & 8. It exports model data into the optimized PowerVR POD file format. It also supports special features, such as tangent space generation and bone batching. This tool can be used if you have to modify models in a 3D authoring package instead of directly using the source data.
- **PVRGeoPOD Standalone**: a standalone tool that converts Khronos COLLADA Digital Asset geometry and animations into the optimized PowerVR POD file format. This tool comes in two flavours, a GUI version and a command line version, which can be easily integrated into content pipelines.

In the 3D navigation demo the data has been converted¹, using the vertex, normal and texture coordinate information stored within the Collada file, almost halving the file size in most cases. The transformation process is depicted in Figure 1. Once the data has been transformed into the POD file format, the PowerVR SDK can be used to read and process those POD files. For more information, see the PowerVR SDK documentation.

2.2. Spatial Index

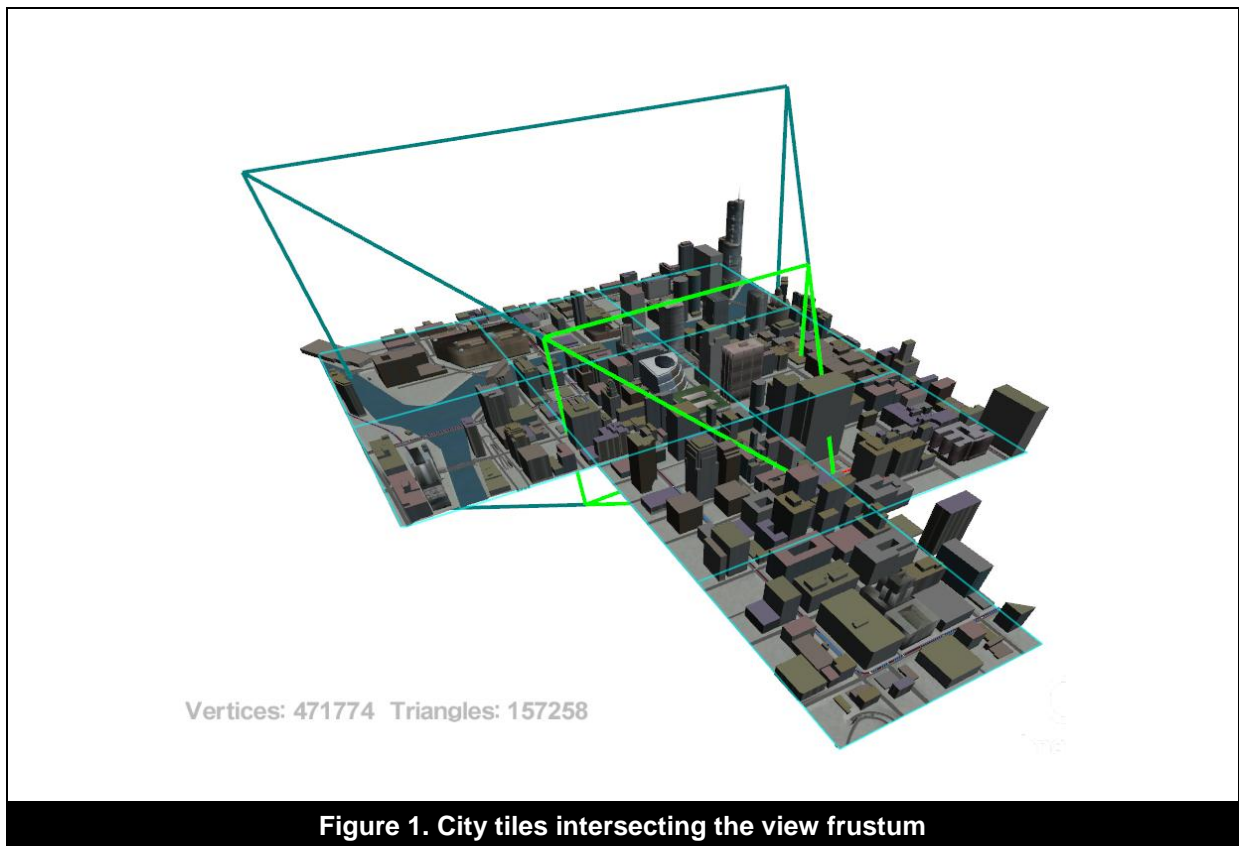
After converting the data into the PowerVR POD deployment format, it can be easily loaded and rendered in any 3D application. Unfortunately, the city model data set itself proves to be very complex and rich in detail so that further steps have to be taken in order to reduce the amount of rendered geometry.

Although the PowerVR Tile Based Deferred Rendering (TBDR) architecture applies various techniques to optimize rendering, such as reducing the amount of overdraw per pixel to a bare minimum, geometry bandwidth will be the limiting factor as all of the geometry has to be sent to the graphics core. It is, therefore, crucial to only actually submit geometry which is visible from the camera's point-of-view.

The easiest optimization is to generate an additional set of data that logically augments the 3D model set. For every city model tile the 2D bounding box coordinates are calculated and stored in a separate file, the spatial index, which then can be used to efficiently cull the tile's bounding box against the camera view frustum (see Figure 1, where the light blue lines represent the tiles' bounding boxes).

¹ Using Collada2Pod with the following options:

-Indexed=1 -Interleaved=1 -ExportGeometry=1 -ExportMaterials=1 -ExportNormals=1 -PrimitiveType=TriList -PosType=float -NorType=float -UVW0Type=float



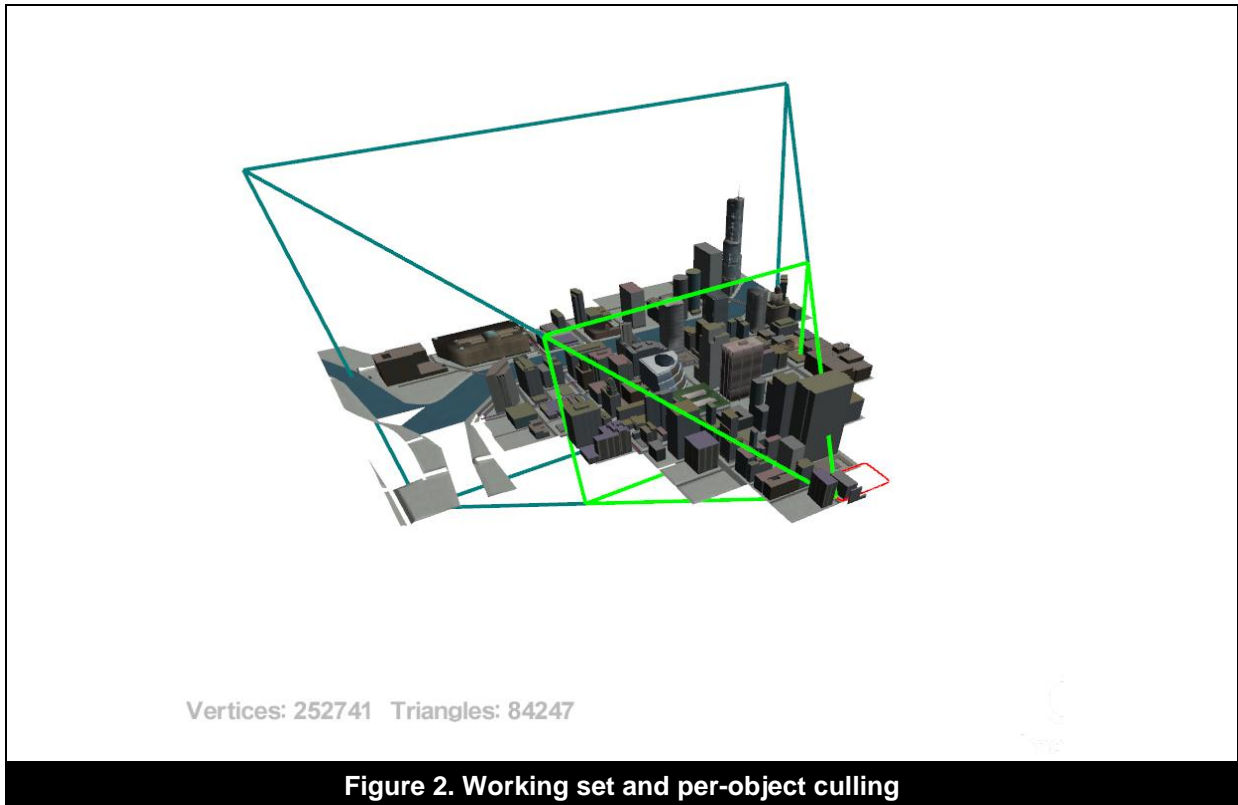
As can be seen in the example scene in Figure 1, the amount of geometry that has to be rendered seems to be manageable due to the view frustum culling (the view frustum is depicted with light and dark green lines in the figure), but still almost contains 500 000 vertices which would have to be sent each and every frame to the graphics core.

Another optimization is to extend the culling from a tile basis to a per-object basis. Instead of rendering each building within each intersected tile, test each building against the view frustum. The spatial index is, therefore, augmented with each object's bounding box which can be used for the enhanced detailed view frustum culling.

The whole rendering procedure then can be rephrased to: if a tile is intersecting the view frustum, test each object within that tile against the view frustum and only render an object if it is itself intersecting the view frustum.

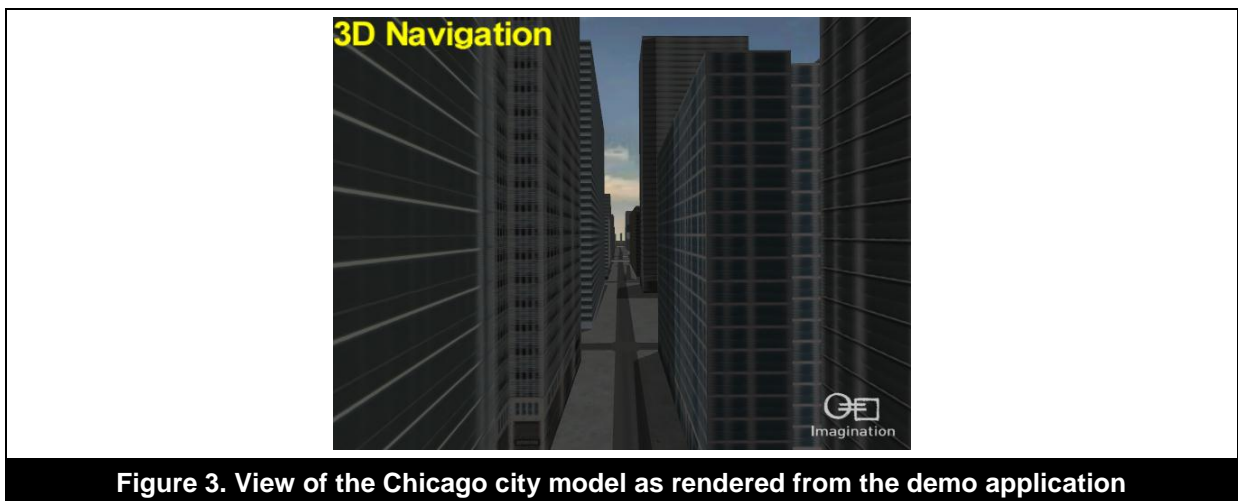
As Figure 2 illustrates, the amount of geometry that has to be rendered has been vastly reduced. Now only objects actually lying within the view frustum have to be rendered. The amount of vertices in this example has been reduced to approximately 250 000 vertices from 500 000 vertices without any visual difference.

It has to be noted though that these optimizations are not completely free, as we have to calculate on the CPU whether a bounding box intersects the view frustum or not. It is advised that a hierarchical data structure shall be used for the bounding boxes (see the section about quad-trees in the first navigation whitepaper) to gain a more efficient culling performance.



2.3. Occlusion Culling

Applying simple optimization techniques vastly reduced the amount of geometry that has to be processed by the graphics core, as can be seen in Figure 1 and Figure 2. Nonetheless, having a closer look at a screenshot from the 3D Navigation demo (see Figure 3) and comparing it against a screenshot from a bird's eye view (see Figure 2) it can be noticed that some of the geometry is not visible from the camera's point of view.



Some of the buildings are occluded by buildings near to the camera. Having a closer look at a wireframe rendering of the same viewpoint reveals that a lot of geometry has been submitted to the graphics core that does not actually contribute to the final image.

Figure 4 illustrates the geometry which has been sent to the graphics core as red overlaid lines over the actual image. As can be seen on the upper image this amounts to a huge amount of redundant

geometry, whereas applying an occlusion culling technique reduces the amount of geometry even more.

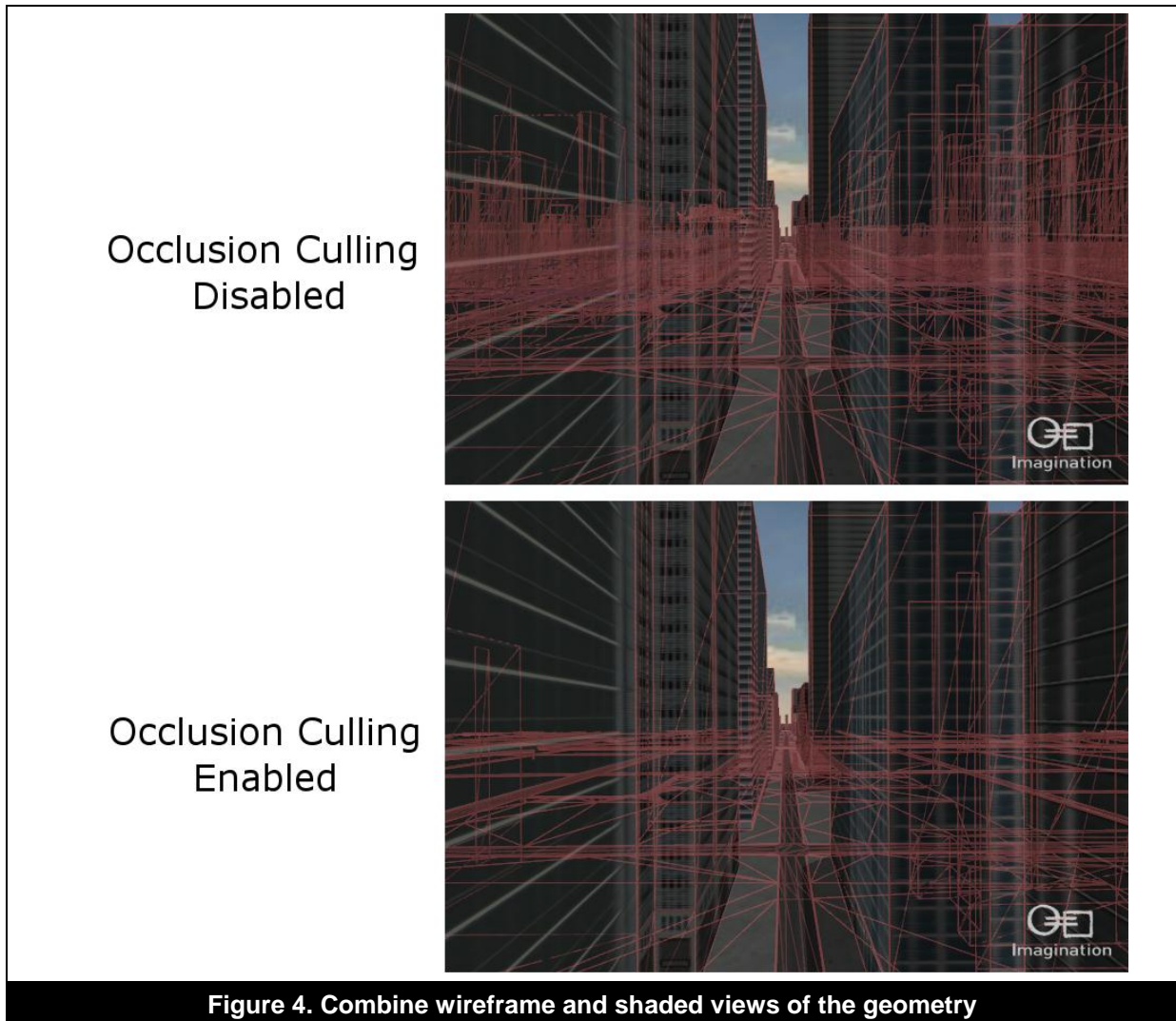
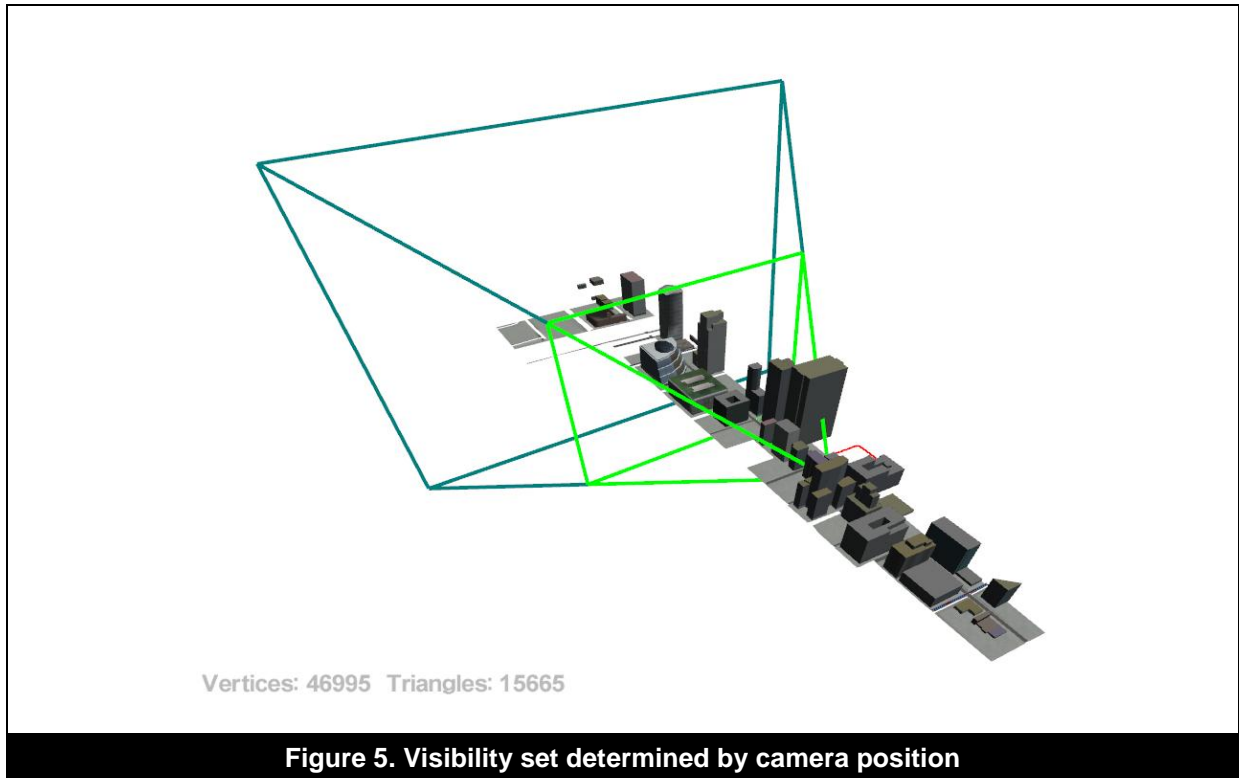


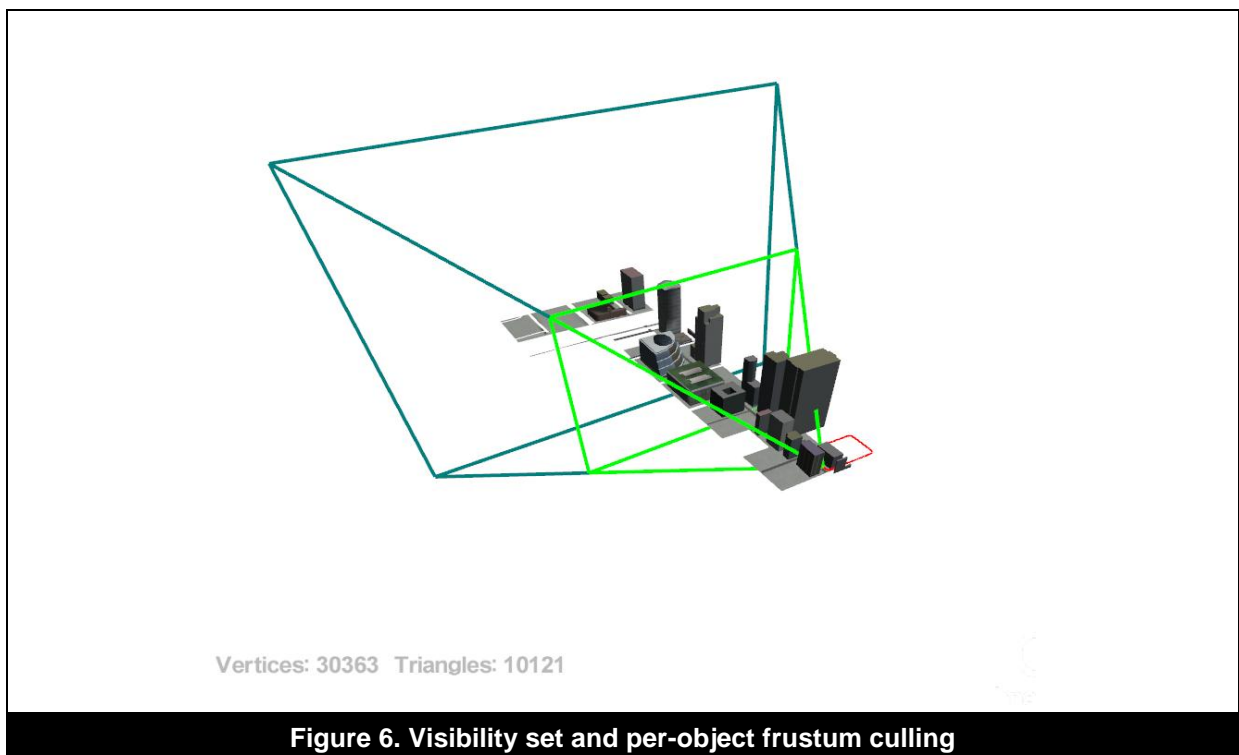
Figure 4. Combine wireframe and shaded views of the geometry

Occlusion culling is a very popular technique to speed up the rendering of large scenes that have a high depth complexity. The particular technique employed in the 3D Navigation demo is generally known as Portal Visibility Sets (PVS), in which the scene is divided into regions that contain an index list of visible objects from that particular region. At runtime this index is then sampled and each visible object successively rendered.

As the bird's eye view in Figure 5 illustrates the amount of geometry has been dramatically reduced: the number of triangles in this particular viewing position went from approximately 90 000 to 16 000. Note that as the index list contains all viewing directions from a particular point of view, objects behind the camera are included in the rendering process.



To further optimize the rendering procedure, view frustum culling can be employed on a per object basis which almost halves the amount of geometry in the particular example (see Figure 6). After determining the set of visible objects from the index, which includes all objects from an omnidirectional point of view, each object's bounding box is tested against the camera view frustum and only rendered if it falls within or intersects the frustum.



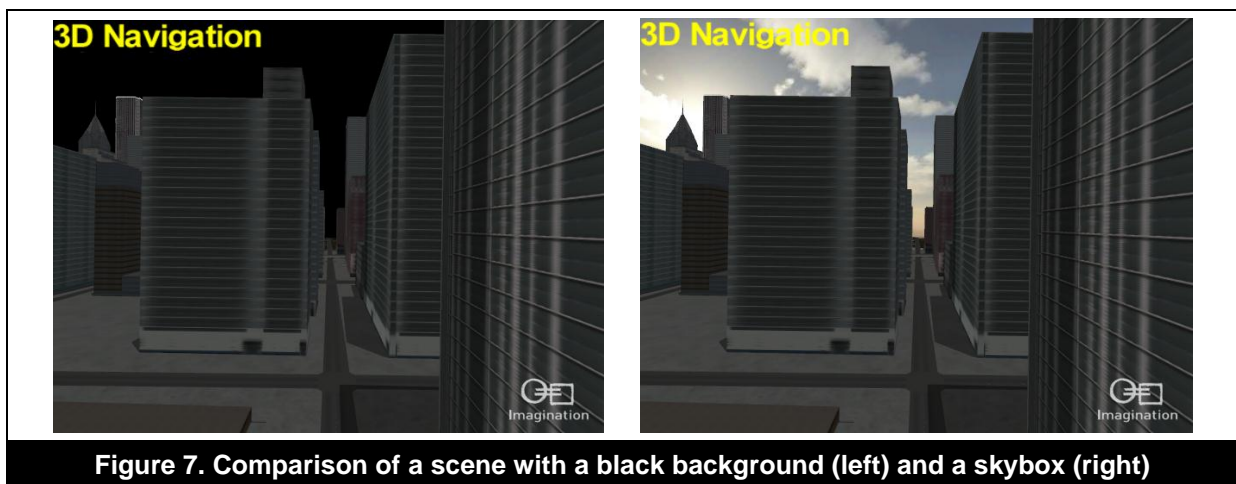
3. Rendering Techniques

The previous section dealt with preliminary steps on how to reduce the actual data that needs to be sent to the graphics core for efficient rendering. This section is about techniques on how to add visual fidelity and further improve performance from a technical point of view.

3.1. Skybox

A very simple technique to add a lot of realism to outdoor scenes is to include a skybox in the rendering process. The *Skybox2* demo and the code in the 3D Navigation demo in the PowerVR SDK demonstrate on how use the PowerVR tools to implement a simple but convincing skybox.

As Figure 7 illustrates, the skybox changes the whole mood of the resulting image. Skyboxes can be changed depending on the time of day, the weather, the geographic location of the user and many more conditions by simply exchanging the skybox texture.



From a performance point of view, due to the Tile Based Deferred Rendering architecture in PowerVR graphics cores and the implicit hidden surface removal, no special measures have to be taken in the rendering process to efficiently render the skybox.

3.2. Shadows

One of the most important visual cues for the human perception system is shadows. Without shadows virtual objects are difficult to locate in a three dimensional space and seem to hover. It is even more difficult to establish spatial relationships between objects. As Figure 8 illustrates the addition of shadows add a lot of depth to the final image.

The particular shadow technique employed in the 3D Navigation demo is known as “stencil shadow volumes”. A reference implementation can be found in the PowerVR SDK training course called *ShadowVolume* as well as in the 3D Navigation demo.

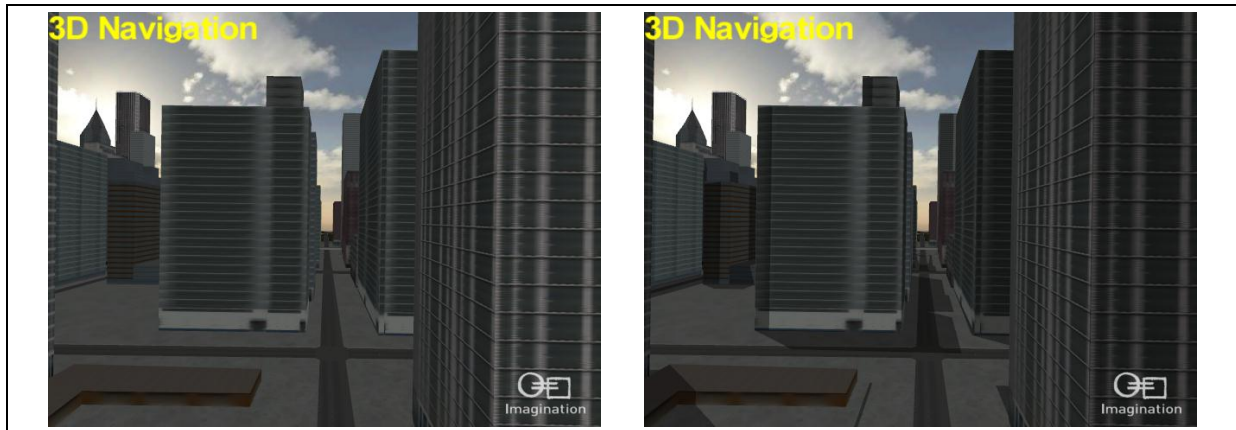


Figure 8. Comparison of a scene without shadows (left) and with shadows (right)

The abstract concept behind the stencil shadow volume algorithm is to determine shadowed and lit surfaces by counting the number of volumes (see Figure 9) that are visible between the camera's point of view and the geometric surfaces that are visible. The volumes are generated by extruding geometry along the light direction. For a more in-depth description of the stencil shadow volume algorithm and recommended reading see Section 4 of this document.

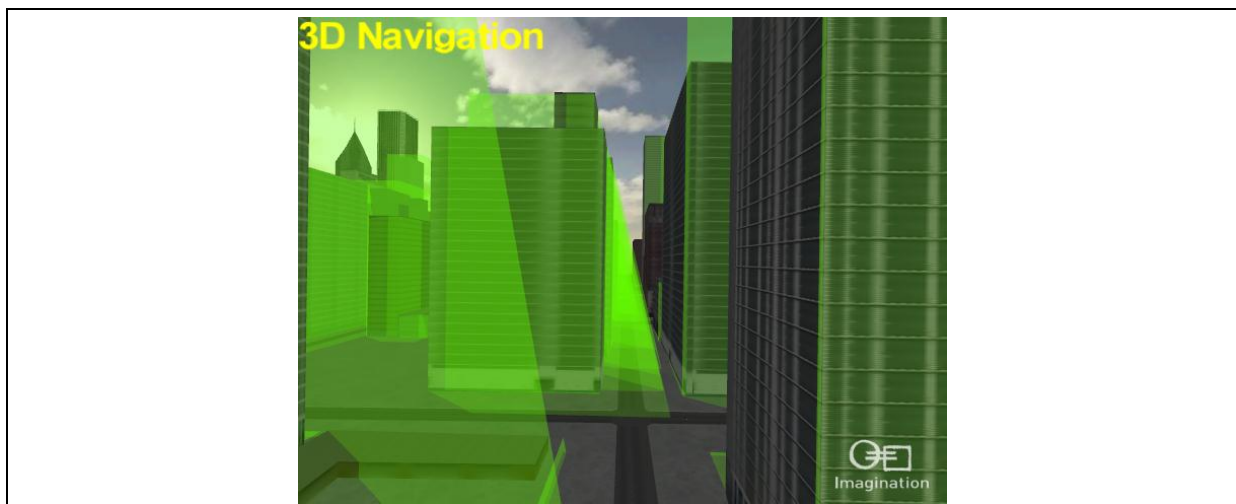


Figure 9. Visualization of the extruded shadow volumes

After determining the shadowed areas in image space they are multiplied by a scalar in the [0-1] range, effectively darkening them. This is not an accurate physical representation of real light interaction but provides a quite convincing effect with very good performance.

The shadow volume performance is highly dependent on the geometric detail of the shadow casting model, but another optimization which can be found in the 3D Navigation demo is to simply take a low polygonal representation of the shadow casting object. In case of the demo it is just the bounding box of each individual object, which can cause artefacts concerning the shape of the shadow, but which are barely visible during runtime. The shadow volume extrusion itself is then handled by an easy to use set of tools provided by the PowerVR SDK.

3.3. Batching

Another very important technical aspect besides the geometric management of the scene is the internal representation for rendering. For example it is recommended to draw as many primitives as possible with as few API calls as necessary. For a comprehensive list and explanation of recommendations please see the development recommendations whitepapers in the PowerVR SDK.

In order to achieve good runtime performance it is recommended to make use of vertex (VBO) and index (IBO) buffer objects. In the demo, during the initialization phase, every model in a city block is merged into a single index and vertex buffer object, which in turn only needs to be bound once frame.

The individual indices are offset to match the new vertex positions in the unified vertex buffer.

Each individual object can then be rendered by simply using the bound vertex and index buffer objects and specifying an offset into the index buffer (see Figure 10) when drawing with `glDrawElements`. This minimizes the amount of necessary API calls for binding vertex and index buffer objects but still keeps the flexibility of dynamically loading and unloading city blocks during runtime.

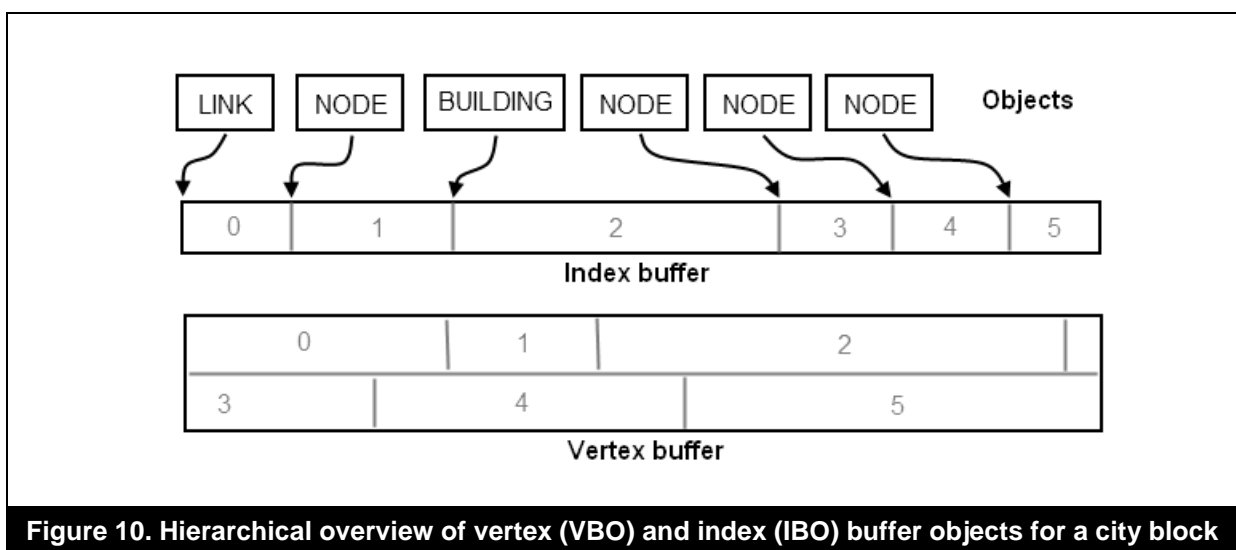


Figure 10. Hierarchical overview of vertex (VBO) and index (IBO) buffer objects for a city block

4. References

Overview about spatial partitioning schemes on Wikipedia:

http://en.wikipedia.org/wiki/Space_partitioning

Navigation data kindly provided by NAVTEQ:

<http://www.navteq.com/>

Stencil Shadow Volumes (Wikipedia):

http://en.wikipedia.org/wiki/Shadow_volume

Further Performance Recommendations can be found in the Khronos Developer University Library:

<http://www.khronos.org/devu/library/>

Developer Community Forums are available at:

http://www.khronos.org/message_boards/

5. Contact Details

For further support, visit our forum:

<http://forum.imgtec.com>

Or file a ticket in our support system:

<https://pvrsupport.imgtec.com>

To learn more about our PowerVR Graphics SDK and Insider programme, please visit:

<http://www.powervrinsider.com>

For general enquiries, please visit our website:

<http://imgtec.com/corporate/contactus.asp>

Imagination Technologies, the Imagination Technologies logo, AMA, Codescape, Enigma, IMGworks, I2P, PowerVR, PURE, PURE Digital, MeOS, Meta, MBX, MTX, PDP, SGX, UCC, USSE, VXD and VXE are trademarks or registered trademarks of Imagination Technologies Limited. All other logos, products, trademarks and registered trademarks are the property of their respective owners.