



## Navigation Rendering Techniques Whitepaper

Public. This publication contains proprietary information which is subject to change without notice and is supplied 'as is' without warranty of any kind. Redistribution of this document is permitted with acknowledgement of the source.

Filename : Navigation Rendering Techniques.Whitepaper  
Version : PowerVR SDK REL\_18.2@5224491 External Issue  
Issue Date : 23 Nov 2018  
Author : Imagination Technologies Limited

## Contents

<b>1. Introduction .....</b>	<b>3</b>
1.1. Point-of-View Types .....	3
<b>2. Sample Data .....</b>	<b>4</b>
2.1. Node .....	4
2.2. Way.....	4
2.3. Data Flow.....	4
<b>3. Tips and Tricks.....</b>	<b>5</b>
3.1. Spatial Subdivision for Efficient Culling .....	5
3.1.1. Initial Tile Setup.....	6
3.1.2. Tile Clipping Algorithm .....	6
3.1.3. Final Setup .....	10
3.2. Optimisation: Quad-tree .....	10
3.3. Batching Indexed Geometry .....	11
3.4. Geometry Triangulation .....	12
3.4.1. Road Triangulation .....	12
3.4.2. Intersection Triangulation .....	13
3.4.3. Polygon Triangulation.....	14
<b>4. Rendering Techniques .....</b>	<b>16</b>
4.1. Anti-Aliased Road Outlines.....	16
4.2. Shadows .....	17
4.3. Lighting .....	18
<b>5. Contact Details .....</b>	<b>20</b>

## List of Figures

Figure 1. Tiled Data Structure .....	5
Figure 2. Tile Clipping .....	6
Figure 3. Intersection Point Outside of Triangle Edge .....	8
Figure 4. Zero Edges Intersecting.....	8
Figure 5. One Edge Intersecting .....	9
Figure 6. Two Edges Intersecting .....	9
Figure 7. Quad-tree for Hierarchal Culling .....	11
Figure 8. Vector calculus used to triangulate line .....	13
Figure 9. Polygon .....	14
Figure 10. Polygon with hole (left) and self-intersecting polygon (right) .....	14
Figure 11. Polygon triangulation with the ear-clipping technique .....	15
Figure 12. Anti-aliasing disabled (left) and enabled (right) .....	16
Figure 13. Texture aliasing: bilinear filtering (left) and trilinear filtering (right) .....	16
Figure 14. Planar shadows (without lighting) .....	18
Figure 15. Gouraud shading with shadows.....	18

# 1. Introduction

Visualisation of navigation data is a complex task. The graphics should be appealing, informative, running at a high frame rate, and utilising low power consumption at the same time. This whitepaper deals with the efficient rendering of navigation maps on the tile-based deferred rendering architecture of the PowerVR chipset families.

The navigation demo, which can be found in the PowerVR SDK, implements the optimisation techniques described in the following sections. Particular attention is given to the various restrictions found in navigation systems for memory constraints, paging, and suchlike.

The geometry generation techniques described in the first few sections illustrate a possible approach. There may be other, more suitable solutions depending on the input data.

For further reading and a more comprehensive overview, it is recommended to look at the documents packaged in the PowerVR SDK, particularly the application development recommendation whitepapers. These offer a good understanding of performance pitfalls and general purpose optimisations when developing for mobile graphics solutions.

## 1.1. Point-of-View Types

There are several high-level approaches to render a navigation system, which mainly differ in the point of view and the amount of detail being rendered. This means that they differ in the minimum hardware specifications they require from the targeted device to be able to run at an appealing frame rate.

The points-of-view covered in this document are:

- **2D top-down:** The standard perspective found in a lot of navigation devices, providing a bird's eye view. It features a very limited field of view, concentrating on basic features like streets, signs, and landmarks. It can be rendered using an orthographic projection scheme, and the terrain is specified in a single plane, along with any landmarks.
- **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. 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, it is possible to integrate panoramas to augment the scenery with images and efficiently achieve a higher level of realism.

This document covers both the 2D and 3D cases, and for the most part the algorithms and techniques discussed apply to both the 2D and 3D navigation application found in the SDK. Any differences in techniques used will be clearly marked.

The following sections explain how to visualise the most common cues like streets, buildings, road signs, and landmarks in an efficient manner.

## 2. Sample Data

The sample data used throughout the 2D and 3D navigation demos is provided by Open Street Map (<https://www.openstreetmap.org>). OSM is an open source platform that creates and distributes free geographical data; furthermore the raw map data is freely available to download in an XML format.

The two most important tags within the OSM XML data from the navigation demo's point of view are the node and way tags. The navigation demo relies on these constructs to build the renderable map. The next section provides an overview of what these terms mean and how they are used within the navigation demo.

### 2.1. Node

OSM defines a node as a single point in space which consists of a unique ID and its coordinates in space, represented by latitude and longitude. A single node on its own is not particularly useful, however several nodes can be used together in a way to build more complex features. The navigation demo uses the nodes read from file to build triangles through a process known as triangulation. This process will create new nodes in order to transform the geometry from line primitives into triangle primitives. A node starts out as a single point in space, after triangulation of the nodes they can be considered vertices on a triangle.

### 2.2. Way

OSM defines a way as an ordered list of nodes. A way is used to build complex objects such as roads or buildings by grouping a series of nodes. Ways also commonly hold a list of tags which provide extra information about the way such as the road name. A way may also be open or closed. A closed way will share the last node with the first node, for example a roundabout. An open way is a linear feature which does not share the first and last nodes, such as a section of road.

Internally, the navigation demo extends the way structure, so that the ways can define a polygon made up of one or several triangles. Nodes held in a way are used to define triangles, which in turn are used to define a complex polygon, such as a road or building.

### 2.3. Data Flow

The following outline is a brief overview of the data flow within the navigation demo:

- OSM data is read from file using a document object model XML parser called pugixml, which is shipped as part of the SDK
- Temporary objects are created to hold the data before being added to the appropriate data structures
- Before a node is added to the internal data structure, its coordinates are converted from latitude and longitude into metres, which is a more efficient metric for the internal algorithms to work with
- Nodes are then processed by an algorithm to convert them from simple line strips, which represent road axes or building outlines, into triangle strips. This effectively provides surface to the objects, such as turning the one-dimensional line representing the road axis into a two dimensional area representing the road's surface. This newly triangulated data is then split up into tiles, which are used for efficient culling. At this point the triangles must be clipped against the borders of the tile
- Finally vertex and index buffers are created and filled, and then command buffers are pre-recorded, before the appropriate data is sent to the GPU for rendering

## 3. Tips and Tricks

One of the most important aspects of the whole optimisation process is data organisation. It is not possible to deal with every different hardware configuration in this document. Therefore easily adaptable algorithms are presented instead of specially tailored versions.

The following sections explain approaches to optimise visibility queries through spatial hierarchies. They also cover how to improve performance by batching geometry, and how to triangulate raw map primitives into a representation suitable for the underlying graphics hardware.

### 3.1. Spatial Subdivision for Efficient Culling

The navigation demo uses a tiled data structure, illustrated below, where each tile in the structure represents a small subsection of the entire map. Each tile contains all of the vertices which are contained within that area, and also holds a vertex buffer, an index buffer and a command buffer. The command buffer is used to store the consumable data and draw the tile.

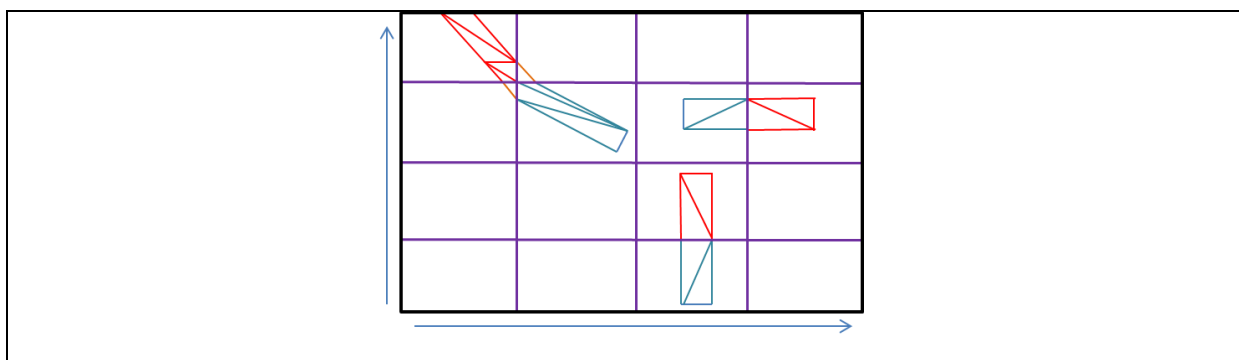


Figure 1. Tiled Data Structure

This approach has two main advantages:

- It is fairly easy to implement
- Visibility checks are straightforward and lightweight to perform, which means large amounts of geometry can be quickly and efficiently discarded before ever reaching the GPU

In most cases the best rendering optimisation is to not render something at all, at least if it is not visible. This can be achieved by culling objects which are outside of the view frustum, which is the visible volume enclosed by the screen.

Culling benefits the GPU in two ways:

- It improves memory transfer efficiency by discarding redundant data
- It reduces the amount of wasted GPU clock cycles on processing vertex data, which will inevitably be culled by the GPU later in the pipeline

The navigation demo performs culling by performing a visibility test against the view frustum. Each tile is tested against four planes. For the 2D demo these are the top, bottom, left and right. For the 3D demo these are the left, right, far and near - these planes are derived from the view projection matrix. The visibility test consists of determining whether the tiles bounding box is inside, partly inside or completely outside the frustum. A tile fails if its bounding box is completely outside all test planes. If the tile fails, all the geometry held in the tile is culled. In other words, it does not get drawn.

As the amount of geometry generated by the navigation demo is quite low, performance-wise, this approach has proven to be sufficient. However, it may not scale well with more complex geometry, such as very dense tiles. This approach could be extended further to allow for culling of geometry, such as individual objects, enclosed by individual tiles. This could be achieved by further subdividing

each tile into a spatial hierarchy such as a quad tree. This finer grained spatial hierarchy could then be used to provide visibility information for individual objects within the tile.

### 3.1.1. Initial Tile Setup

The initial setup of the data structure involves calculating the number of tiles that are required. This can be broken down into calculating the number of rows and columns required. This is calculated based on the minimum and maximum extents of the map loaded into the system. Next the individual dimensions (minimum and maximum coordinates) for each tile are calculated, and subsequently the data structure is populated with several tiles. It should be noted that the tiles are evenly distributed to fill any map no matter the dimensions meaning that each tile is of equal size.

After the tiles have been initialised the next step is to fill each tile with actual map data. This is done once the data has been processed from raw OSM data into renderable triangulated node lists – which should now be considered as lists of vertices that define triangles. The clipping algorithm which fills the tiles with vertex data is described in the next section.

### 3.1.2. Tile Clipping Algorithm

The tile clipping algorithm used in the 2D and 3D navigation demo is recursive in nature.

This means that it:

1. consumes a triangle (three vertices)
2. recursively breaks the primitive into progressively smaller triangles until the generated triangle/triangles are completely enclosed by a single tile

At this point the triangle is added to the appropriate tile and the algorithm exits. This algorithm was chosen for its robustness and simplicity. The only test is a single triangle against a single plane, as many times as required.

The output is:

- One, if the triangle does not intersect the clipping plane
- Two, if the clipping plane passes through a vertex of the triangle
- Three, if the clipping plane typically passes through a triangle, see below

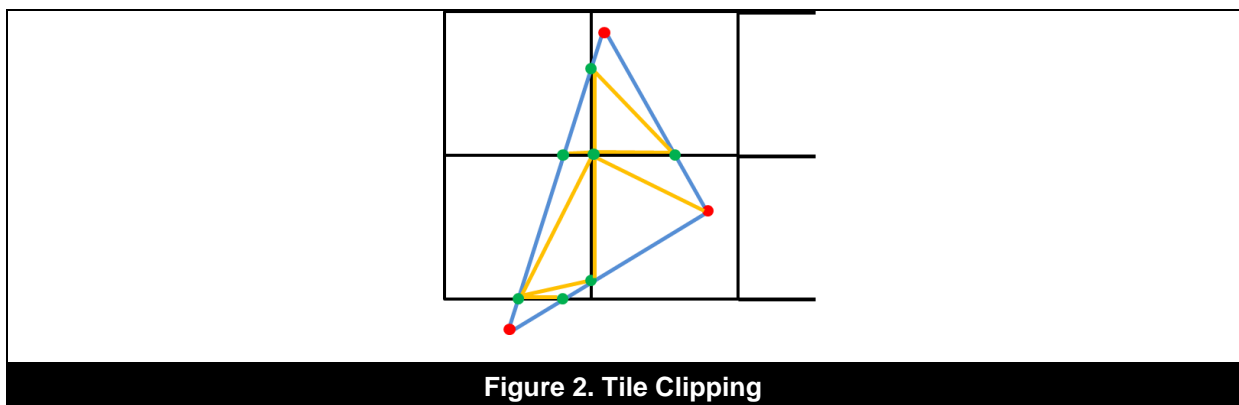


Figure 2. Tile Clipping

The figure above shows a visual representation of how a triangle, which crosses multiple tile borders and extends beyond the map bounds, might be clipped against the tile boundaries.

- Red dots represent original vertices
- Green dots represent the newly generated vertices
- Blue lines represent the triangles original edges
- Yellow lines represent new edges created from new and existing vertices

Below is a full detailed description of the algorithm:

1. The index of the tile each vertex occupies must be found, such as 0, 1. Note that this may be outside of the map area entirely.
  - a. If any of the vertices lie outside of the map extents (they do not occupy a tile), then the algorithm must clip against the map boundaries first before clipping against internal tile edges. Any triangles produced by the clipping that lie outside tile bounds are discarded, and the rest will feed the next iteration, recursively breaking the triangle into more triangles until all of them are within the map bounds. This step uses the algorithm defined further down.
2. The three indices of tiles occupied by the vertices of the triangles are now obtained. The next step requires finding their minimum and maximum extents which is the bounding box that encloses the triangle. This ensures that all tiles which the triangle occupies are considered when clipping.
3. The minimum and maximum tile indices are now found, and all vertices lie within the map bounds. The algorithm can now begin to clip the triangle against all of the internal tile edges that the triangle covers.

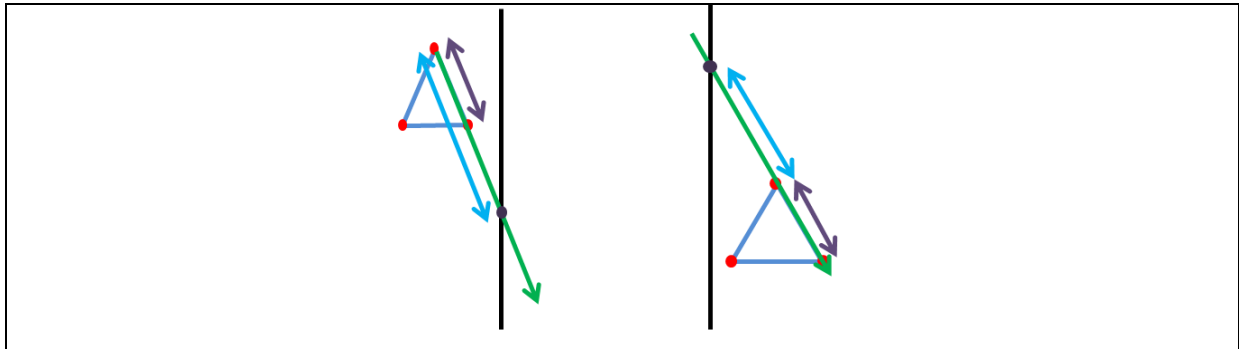
This is a recursive function that splits a single triangle up into multiple triangles if necessary. The output of the previous iteration feeds the next iteration of the algorithm until the triangle under consideration occupies only a single tile.

*Note: It is important to remember that the algorithm may produce only a single triangle if the initial input triangle is fully enclosed by a single tile. Alternatively, it may produce many triangles if the initial input triangle crosses multiple tile boundaries, so each step may produce one, two or three triangles.*

The algorithm is as follows:

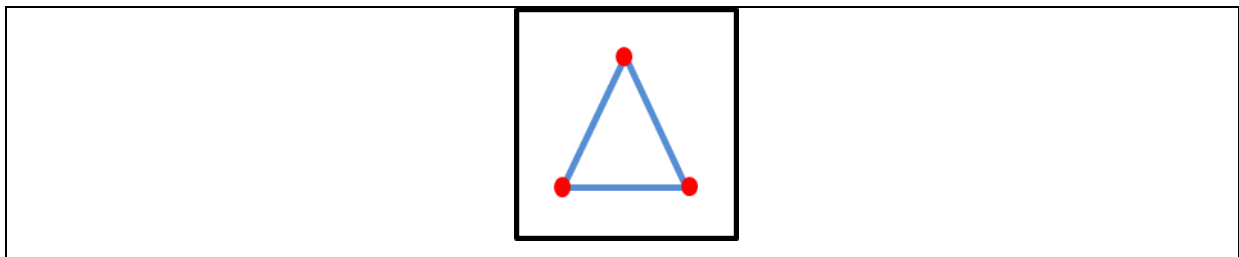
- a. It recursively clips the triangles against a single plane. To simplify this task, the algorithm recursively clips against the X and Y axis (the top and bottom or right and left of a tile) sequentially. The clipping plane is defined by an origin point and a normal.
  - i. The origin used in this case is the top right point, the tile's maximum point, of the "middle" tile. This is the tile that falls closest to the centre of the minimum and maximum tile indices. By taking the middle tile the search space is cut in half each time, which helps to reduce the recursion depth.
  - ii. The plane normal, a normalised vector perpendicular to the plane, is set to either:
    - 1.0, 0.0 if clipping against X (the left and right of the tile)
    - 0.0, 1.0 if clipping against Y (the top and bottom of the tile)
- b. Once the clipping plane is defined, the triangle can be clipped against it.
  - i. A direction vector between each pair of vertices in the triangle,  $p_1 - p_2$ , has to be calculated. These vectors are used as an input to the intersection function, and to calculate the new vertices used to define new triangles if an intersection with the plane is found.

- ii. Calculate the Euclidean distance between each pair of vertices in the triangle. These values help in checking that a valid intersection has taken place.
- iii. Now the initial state is setup, test each side of the triangle for intersections:
  - First, use the Framework function “`intersectLinePlane`” to determine if an edge is intersecting the plane. This function will return the signed distance from a vertex along the edge of a triangle. If the function returns false, the edge is parallel, and therefore does not intersect.
  - If the distance is negative, as shown on the right of the diagram below, or greater than the length of the side, as shown on the left, the edge does not intersect the plane.
  - If the distance is between zero and length, then the edge intersects.



**Figure 3. Intersection Point Outside of Triangle Edge**

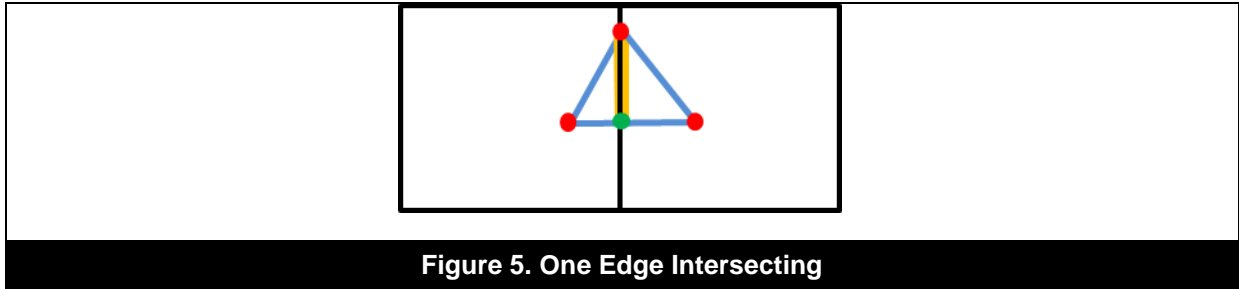
- iv. Now it has been determined which pairs of vertices, or edges, are intersecting the plane, there are three potential scenarios to consider:
  - There are zero edges of the triangle intersecting the plane, in which case the triangle does not need to be clipped.



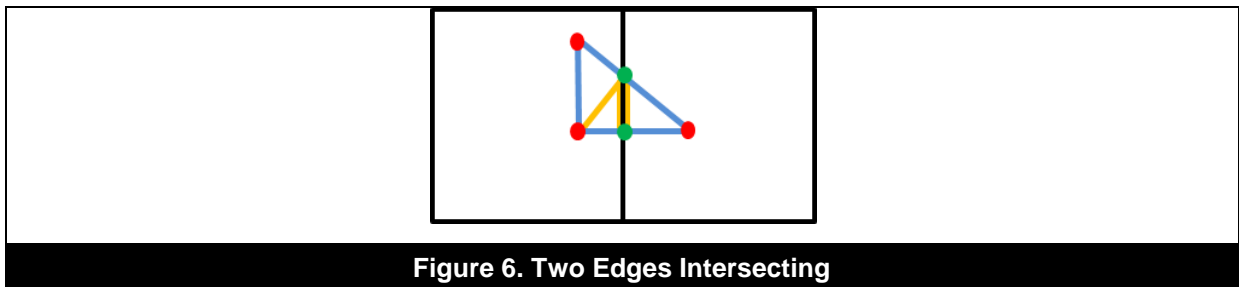
**Figure 4. Zero Edges Intersecting**

- There is one edge intersecting the plane. This scenario means that one of the vertices is exactly on the plane. The original triangle must be split into two sub-triangles.





- There are two edges intersecting the plane, in which case the original triangle must be split into three sub-triangles.



- v. With the intersections found, now the triangle has to be split into other triangles - ones that completely lie on one or the other sides of the triangle. This is shown in the above images by yellow lines. Using the three scenarios described above:
- Zero edges intersecting. There is no need to split the triangle, so all that has to be done is to check which side of the plane the triangle resides - left or right. This can be achieved by taking the dot product of the vector ( $p_0$  - plane origin, where  $p_0$  is any vertex of the triangle) and the plane normal, and checking whether the value returned is negative or positive.
  - One edge intersecting. Two new triangles must be generated. First, the new vertex which lies on the plane has to be calculated. As  $p_0 + \text{clip distance} * \text{edge0}$ ,  $p_0$  is a vertex on the triangle,  $\text{clip distance}$  is the distance from the vertex to the plane, and  $\text{edge0}$  is the directional vector that defines an edge on the triangle. Note that the selected variables for this operation will depend on which edge is actually intersecting the plane. Once the new vertex has been generated, two new triangles can be created from it and the three existing vertices. Finally, determine on which side each triangle resides, which can be found by performing the same operation as in scenario one.
  - Two edges intersecting. Three new triangles have to be generated in this example. The steps are similar to case two. The differences are that two new points of intersection have to be calculated instead of one, and three new triangles have to be created instead of two. The variables used to calculate the new vertices will depend on which two

edges are intersecting the plane. Also, note that two of the new triangles will reside on the same side of the plane.

4. A single tile is reached when the minimum bounds equal the maximum bounds. It is certain that the triangle is now fully enclosed within a single tile and can be safely added to the current tile. This terminates the recursive loop for this particular triangle.

*Note: This algorithm will check for degenerate triangles at various stages during execution. A degenerate triangle is when two vertices of the triangle lie at exactly the same co-ordinates, or when all three vertices lie on the same line – within epsilon, a small error margin for values. This is where values whose difference is smaller than epsilon are considered equal. This prevents the algorithm becoming stuck in an endless recursive loop due to floating point imprecisions.*

### 3.1.3. Final Setup

Once the tiles have been filled with the now clipped vertex data, the various buffers can be filled with data that can be consumed directly by the GPU. Each tile holds a secondary command buffer, as well as a vertex and index buffer. The final step to setting up the tiles for rendering involves filling these buffers and recording the command buffer.

Note: A tile holds only one Vertex Buffer Object (VBO) and Index Buffer Object (IBO) in order to avoid rebinding many buffer objects when rendering. This means that data for different map elements such as road types, buildings, parks, and so on, are all held together.

Firstly, the vertex buffer is filled. Because each node holds its own position, the vertex data is simply created by copying each node's position (a `vec2` in this case) into the VBO.

Next, the index buffer is filled by copying the index (unsigned integer) of the node's position in the vertex buffer into the IBO. As all map elements are held together in a single buffer, an extra step is taken while filling the IBO, which involves storing offsets. These offsets are based on how many roads, buildings, and parks nodes are present in the tile. They are then used to index into the IBO when recording the draw commands. This is the element offset into the IBO and count.

Finally the secondary command buffer is filled, which involves recording all commands such as:

- binding the VBO and IBO
- binding pipelines
- setting up uniforms
- issuing the draw commands, which is where the offsets become useful

The offsets allow the use of different pipelines with different settings applied (fixed function state) and different uniforms for any given set of vertices in the map such as roads or buildings.

## 3.2. Optimisation: Quad-tree

A quad-tree builds a spatial hierarchy on top of a dataset, which speeds up operations like spatial queries. Each node in the quad-tree contains either references to geometry, or references to child nodes. If it does not reference any child nodes, it is considered to be a leaf node.

A typical spatial search sequence occurs in the following steps:

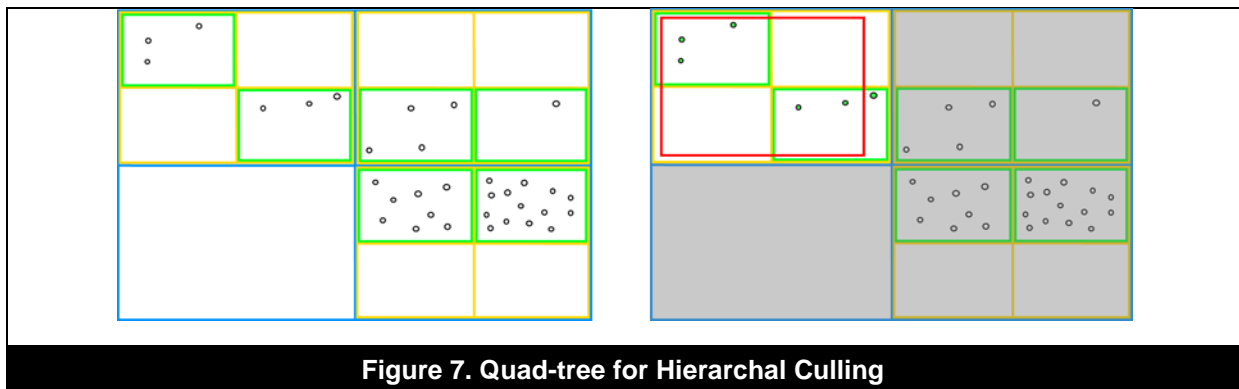
1. Each child's bounding rectangle is tested for intersection against the view frustum
2. If a child is not contained, all its children are culled at once, vastly pruning the search domain
3. If a child is not a leaf node, the intersection test is repeated for each child
4. All intersected leaf nodes can be considered visible and the contained objects rendered

A quad-tree is built recursively, starting with the whole dataset and a bounding box enclosing it. The bounding box is then subdivided into four bounding boxes, known as the children, and each object is

assigned to the bounding box it is contained in. This process is repeated until a certain criterion is met, such as a maximum number of recursions, or a minimum number of objects left per bounding box.

In some cases it might not be possible to build a spatial hierarchy containing the whole dataset beforehand. However, it is possible to generate nodes for the current working set. For example, if paging regions of the map during runtime, it is possible to dynamically generate nodes in the background.

The image below illustrates a simple quad-tree containing a set of vertices. The root node is indicated with blue lines and all child node levels are coloured differently. The green rectangles are the bounding boxes for the leaf nodes and contain references to the contained geometry. The image to the right shows the result of a search, where the red search rectangle is consecutively tested against each quadrant of the root node. If a sub-quadrant is not intersected then all of its children are culled immediately.



The set of intersected objects is coloured green. Upon inspection, there is a green dot outside of the culling rectangle. This is a false positive, but rendering it may cost less than doing a more fine-grained culling. Transferring this scheme to a densely packed map can save a lot of CPU and graphics core time, which can be used for other tasks.

In summary, a total of nine intersection tests were applied in this example:

- One intersection test against the root node
- Four against the first level of children, immediately culling three children
- Four intersection tests against the remaining children at the second level of the tree

Compared to the 33 tests that would be required to test each individual object against the frustum, if no spatial hierarchy was available, this equates to almost a 75% saving. In cases where the tree is even more densely populated, savings could be increased further. As this is a very theoretical illustration, it should be noted that in actual situations the culling primitive, in this case the red rectangle, is represented by the camera frustum. Further subdivisions of the quad-tree could give far better results/savings.

In order to determine the number of subdivisions, it is always a good idea to use benchmarks to estimate the cost of rendering the data. Heuristics should be applied based on the maximum/minimum numbers of primitives per leaf node or specify a globally targeted tree depth.

Note: The depicted quad-tree is a very simplified abstraction. There are several different variants of spatial partitioning schemes available and the one that best suits the developer's needs should be picked.

### 3.3. Batching Indexed Geometry

A very important aspect of optimisation is to batch geometry for draw calls. The larger each batch is, the more efficiently the hardware is able to deal with the workload. This is because larger batches mean fewer graphics API calls, which in turn means less work for the driver. In turn, this results in reduced workload for the CPU, and a more consistent utilisation of the graphics core.

For example, a recommended solution is to:

1. Triangulate multiple roads as a triangle list
2. Submit the whole list of triangle indices with a single draw call rather than submitting each individual road by itself

The precise size that gives the best results can be tweaked based on target data or platform.

The Navigation demo uses a mixed approach, optimising the number of draw calls by producing larger batches and using index buffers. The navigation map is subdivided into tiles, which are sufficiently large enough to contain a significant amount of independent geometric data. This data can be dispatched to the GPU with only a few draw calls for each tile – rather than each element of the map being drawn individually.

*Note: The tile size is fully configurable, and it can be changed based on data density or expected zoom levels to optimise for different scenarios.*

Each tile contains an index buffer which references the geometry, and could be split up further into a spatial hierarchy, such as a quad-tree. This allows for fine grained culling of geometry within the tile boundaries. This would require maintaining a list of visible sub-blocks during runtime, which would be calculated from the view frustum.

As described earlier, the partitioning scheme is fairly coarse grained in its approach but does a good job at keeping the number of a draw calls to a minimum. It does this by batching up geometry, while being flexible enough to perform visibility checks to cull large amounts of geometry which is off-screen.

## 3.4. Geometry Triangulation

Rendering roads, buildings, signs, and the landscape is the most important visual part of a navigation system. Using the primitive data straight away as it is given by the map provider is not possible in most cases. The following subsections introduce techniques to prepare the data for the GPU to render. These algorithms are used for both the 2D and 3D demos.

### 3.4.1. Road Triangulation

Using the street coordinates and simply drawing them as line primitives has several drawbacks. The supported line width is hardware dependent. This means it could possibly be very thin, and line antialiasing might not even be supported at all, making the finished visual too simple to be acceptable. It is generally inadvisable to make extensive use of line primitives. Instead, it is recommended to use triangles to draw the map elements which are more appropriate for the graphics hardware.

Roads given by OSM are defined as ways which themselves are made up of nodes, known as line lists. These line lists have to be triangulated in order to be rendered efficiently. The process of road triangulation consists of three main steps:

- Generate triangles from line lists, contained in the ways from the XML file
- Perform tile binning and clipping
- Allocate and fill vertex and index buffers used for drawing

In the following example:

- The red line is the line list
- The grey translucent lines are the generated geometry
- The green line illustrates a directional vector, which is used to calculate the blue lines
- The blue lines are the spanning vectors used for calculating the triangle vertices. The spanning vectors can be easily derived from the directional vectors, as there is only one unique perpendicular vector to another in a two-dimensional plane
- Letters represent the original line list vertices

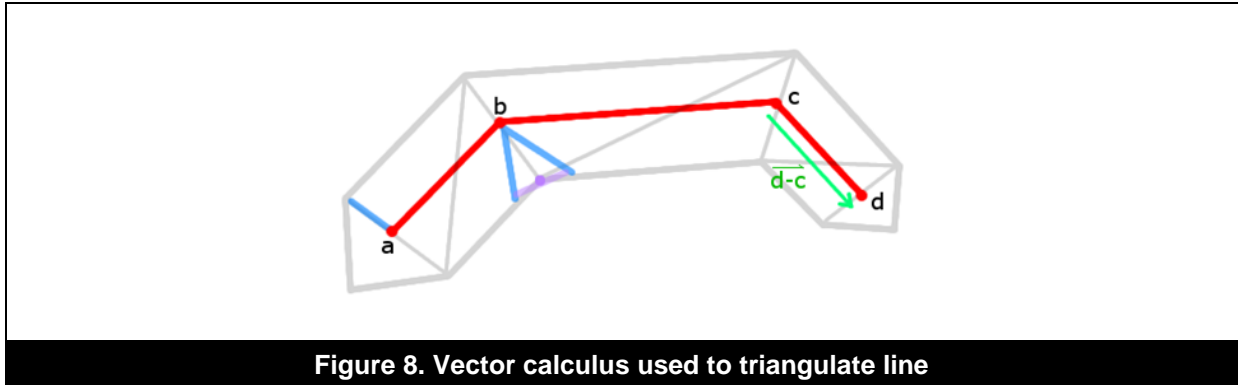


Figure 8. Vector calculus used to triangulate line

The formula used is:

$$v_{perp} = \begin{pmatrix} -v_{dir_y} \\ v_{dir_x} \end{pmatrix}$$

$perp$  denotes the perpendicular vector and  $dir$  denotes the directional vector. As the different line segments can be of different length, the perpendicular vector has to be normalised first and then scaled by the desired road width. Applying simple vector calculus, it is possible to calculate the individual triangle vertices belonging to the various line segments that define the road.

### 3.4.2. Intersection Triangulation

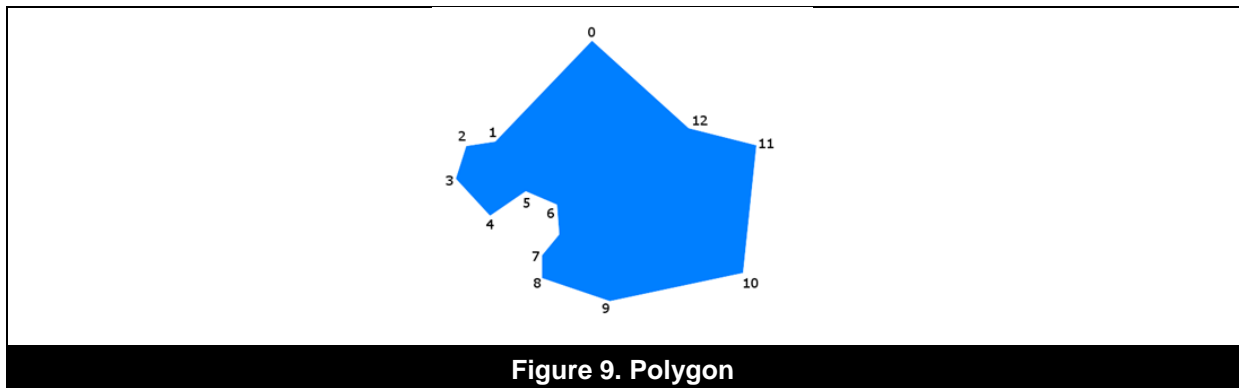
Intersections are a very broad subject with several different techniques and possibilities, where code complexity, robustness, and possible rendering quality all need to be considered. The approach used in the demos is as generic as possible, with as few special cases as possible. Different widths, very acute, or very oblique angles with different widths, and other special cases, tend to create corner cases that need to be handled.

The technique opted for is as follows:

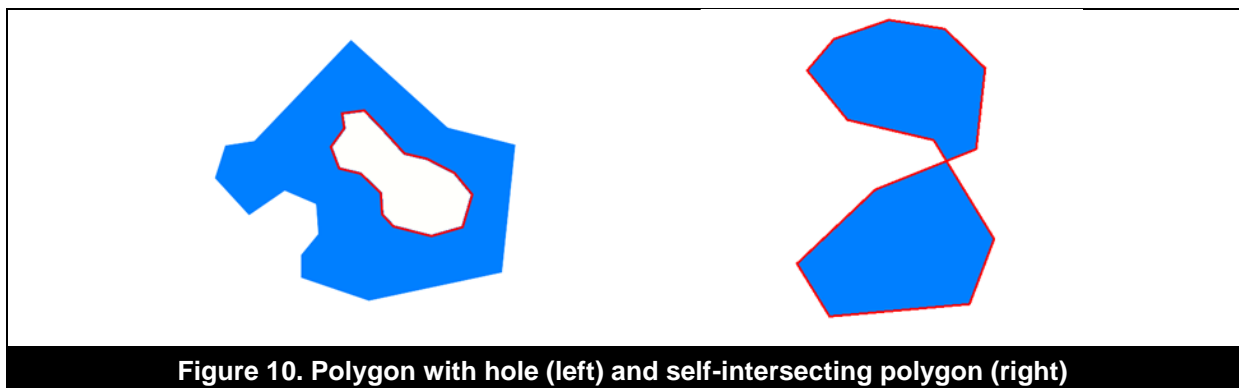
- Pre-process the data in such a way that intersections will always be at the endpoints of roads, and there are no loops. All roads are considered, and whenever a road is found to contain an intersection node in a middle point, it is split up on the intersection into two roads. Afterwards, if a loop is found, the road is broken up into two roads at an arbitrary point. This will generate several two way intersections
- For each intersection:
  - If it has only two incoming roads, connect the corresponding sides of the road with each other so that there is continuity
- For three or more incoming roads:
  - Create a point in the centre of the intersection
  - For each road:
    - Intersect the “left” side of this road with the “right” side of the next road, and move the corresponding endpoints to this intersection
    - If this move puts the points further back from their previous points (as in, the previous points now lie inside the intersection), perform the same procedure for the previous segments of the road. So instead of moving the last point on the last road segment, move the next-to-last point on the next-to-last segment, and connect those together instead. This procedure may create some degenerate triangles
    - Add a triangle connecting the two end vertices of the road to the centre of the intersection
  - This procedure is deceptively simple, but has a few corner cases that have to be handled, mostly to do with parallel lines or with intersection centres that fall outside the outline of the intersection itself

### 3.4.3. Polygon Triangulation

One of the primitive types employed in navigation maps other than vertices and lines are polygons. A polygon is described by a set of vertices given in a certain order which enclose an area and define the shape of the polygon, as illustrated below.



Polygons can contain holes and self-intersections. Holes are described by a sub-polygon which cuts a hole into the parent polygon. It is possible to cut holes in holes by defining polygons within the sub-polygons and the whole procedure can be repeated. This is useful to describe elements which have inner areas that are cut-out of the main parent polygon, which are described as polygons. Self-intersections occur when one line strip of the polygon crosses another line strip in the polygon. The image below shows both examples, with a polygon with a hole on the left and a self-intersecting polygon on the right.



Polygons are commonly used to describe items in a map which have area, like buildings, recreational parks and special zones. There are two ways of rendering a polygon:

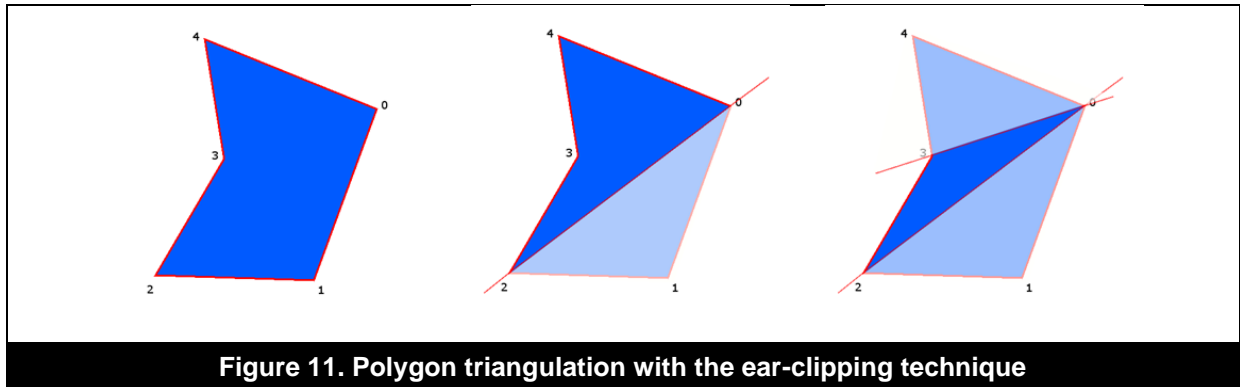
1. Triangulating the convex outline of the polygon and using the stencil buffer to cut out holes and concave parts
2. Using more complex triangulation techniques to separate the whole polygon into triangles

The first method is quite easy to implement but can perform poorly at runtime. It uses the stencil buffer to generate the final shape of the polygon, and takes advantage of the stencil test to render it. Depending on the shape and size of the polygon, this can cause overdraw for the individual triangles and is not recommended from a performance point of view.

The second method might prove more difficult to implement due to the difficulties of handling holes and concave parts. However, it has a much better expected runtime, as the polygon will be triangulated before use and the triangle representation cached for all further operations. The general difficulty is based on the complexity of the polygon, which means it is non-self-intersecting and does not contain any holes.

The simplest triangulation technique is known as ear-clipping:

1. This technique iterates over the polygon edges, successively building triangles out of two consecutive edges, such as vertices zero, one, and two in the middle example below.
2. It also tests the shared vertex of both edges, such as vertex one in the below example, against all other triangles in the polygon.
3. If its area is positive and no other vertex of the polygon is contained within the triangle, it can be safely clipped away.
4. The resulting triangle is added to the triangulated set, the shared vertex removed from the polygon and the whole procedure restarted with the next pair of edges.



**Figure 11. Polygon triangulation with the ear-clipping technique**

*Note: While polygons with holes (inner areas) cannot be handled directly by this algorithm, they can be handled indirectly. This is done by drawing the inner triangulated polygon over the top of the main parent polygon. This essentially cuts a hole in the parent polygon.*

The algorithm can be expressed as follows:

1. If the polygon exactly contains three vertices, add them to the triangle set and terminate
2. Otherwise take two consecutive edges from the polygon
3. Test whether the area is positive. If not, go to step two and repeat with next pair of edges
4. Test whether there is a polygon vertex which lies within the triangle. If yes, go to step two and repeat with next pair of edges
5. Remove the shared vertex from the polygon and add a triangle to the set of triangles. Go to step one and repeat until the polygon is sub-divided into triangles

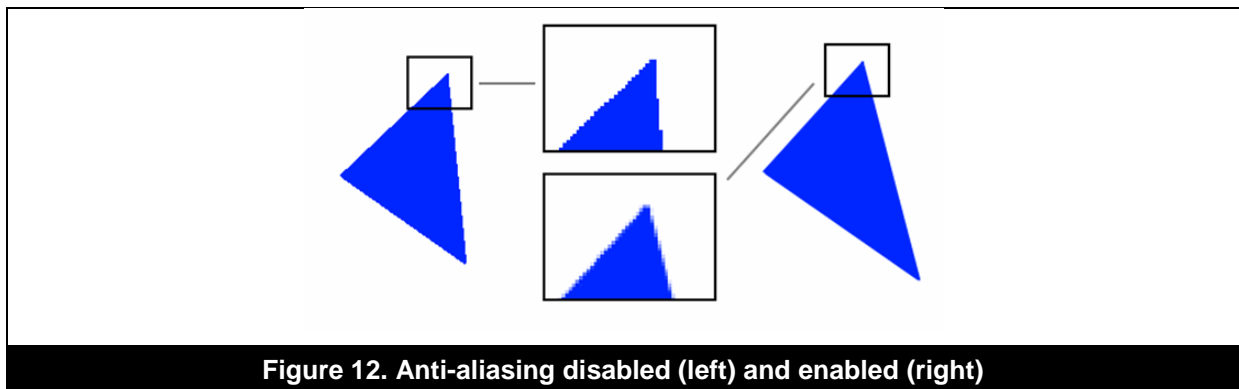


## 4. Rendering Techniques

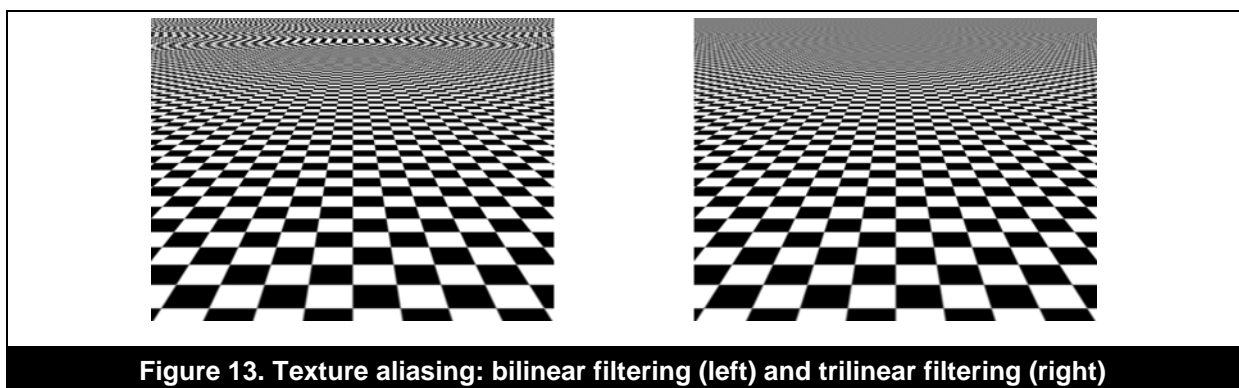
The previous section dealt with the algorithms to handle the pre-processing of geometric data, and how to convert them into a format suitable for rendering. The following sections focus on the various rendering techniques which were employed to enhance the visual quality of the demo. Note that the sections entitled “Shadows” and “Lighting” describe techniques which only apply to the 3D navigation demo.

### 4.1. Anti-Aliased Road Outlines

One of the most recurrent issues in computer graphics is aliasing. Aliasing occurs when the sampling theorem is not fulfilled, and a signal is sampled at too low a frequency.



In computer graphics, it is noticeable as staircases at the edges of geometry, illustrated above, or visual artefacts when texture mapping, as shown below.



There are several techniques, like multi-sampling or super-sampling, used to perform anti-aliasing in order to get rid of the stair-case artefacts, commonly called jaggies. These require special hardware support that is present in all modern 3D hardware, but might incur a very high performance cost.

In the case of rendering simple road geometry, it is possible to achieve the desired results without the need for special hardware support. In this section, a relatively simple and efficient method to cope with aliased lines is introduced. This method is used to add the road outlines and greatly improve the quality of the edges of the road geometry.

Select two constant values which will be assigned to each vertex in the data set. The assigned value alternates between odd and even vertices. This means that one side of the road will receive value X, and the other will receive value Y. This data will be uploaded to the graphics hardware as vertex data, for use in the fragment shader to calculate the final alpha value.

The values assigned to the vertices allow distinguishing between each side of the road, as in, the left and right hand-sides. The values will be linearly interpolated by the hardware, because the values are



being used in the fragment shader. The newly interpolated values can then be used to calculate the distance that the current fragment has from the edge of the road.

The target is to render the centre road opaque, and then gradually introduce transparency a few pixels near the edge of the road polygon, by decreasing alpha, finally reaching zero at the edge of the road polygon.

Since the base “distance from axis” values are interpolated from the vertex data by the hardware, and that has already been calculated, the final alpha can be worked out.

The base interpolated value is used as an argument to the partial derivative functions which GLSL provides. These functions are `dFdx` and `dFdy` which calculate the rate of change of a value in screen space along the X and Y axis respectively, typically over a 2x2 grid of fragments. This indirectly provides:

- The distance from the edge, relative to road width
- The “rate of change” of this distance

Based on this rate of change, the appropriate alpha value can be calculated for any given fragment. This calculation is the key to the whole algorithm, as the resulting value determines the percentage of fragments that define the edges of the road. As a consequence, the calculation determines how many fragments must be blended in order to give a smooth outline.

Using the derivative functions makes it possible to determine whether the object is taking up a large percentage of the screen or not:

- If the object is small on the screen due to being zoomed out, the pixels needed to blend to achieve smoothness are a larger percentage of the total pixels
- Conversely, if the object is large on the screen, the pixels needed to blend are a smaller percentage of the total distance from the centre of the road

When zoomed in, then the rate of change will be low, as the base values will be interpolated over many fragments. This means that in order to blur the same number of fragments, it is necessary to start blurring on larger values, percentage-wise, closer to the edge of the road.

In order to allow several optimisations, in the first pass only the road outlines are drawn, which means drawing all of the road geometry but with the anti-aliasing shader bound. Then, the road geometry is drawn again over the top of the outlines but with a simple flat colour to fill in the road. After both passes, the result is a road which has a flat fill colour accompanied by sharp, crisp outlines on either side of it.

The advantages of this method compared to previous, texture-based anti-aliased methods are:

- There are no texture fetches, which may help reduce the memory bandwidth for the entire application
- This approach is relatively cheap in terms of cycle count
- Unlike texture-based methods, it does not rely on mipmaps or maximum texture size, so it has no limitation in either zoom-in or zoom-out
- Therefore, it produces exceptionally crisp, high quality outlines independent of the object’s scale and orientation

## 4.2. Shadows

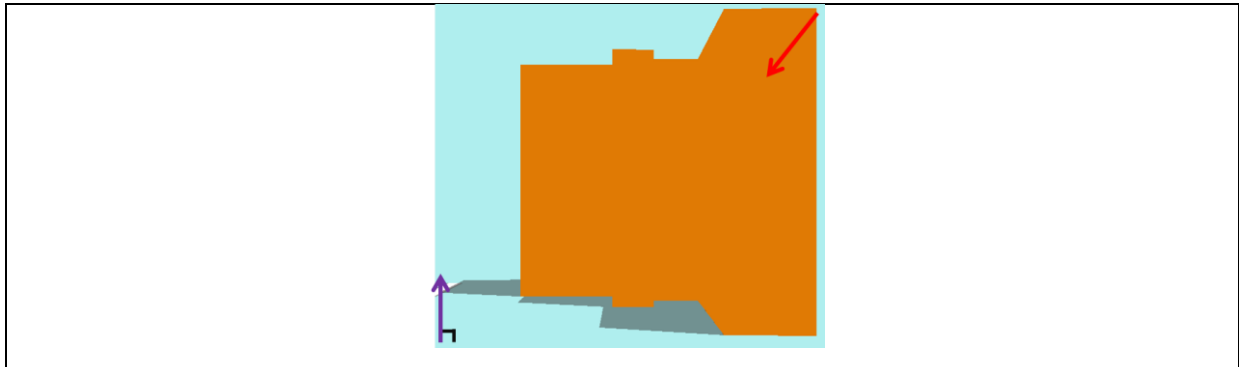
This section covers the technique used to add shadows to the navigation demo.

*Note: This section applies to the 3D demo only.*

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.

The shadows in the 3D navigation demo are generated by:

1. Projecting all of the 3D geometry (the buildings) onto a 2D plane with respect to the ground which is represented by a single normal, and the light direction. This can be achieved by constructing a 4x4 matrix from the light direction, ground normal, and the dot product of the light direction and ground normal.
2. Multiply the vertices which define the 3D object by this matrix in order to calculate the shadow area which is cast by the object. Note that the light is static so this matrix is calculated once and reused. In the diagram below the red arrow represents the incoming ray of light from the imaginary light source and the purple arrow represents the normal to the ground plane.



**Figure 14. Planar shadows (without lighting)**

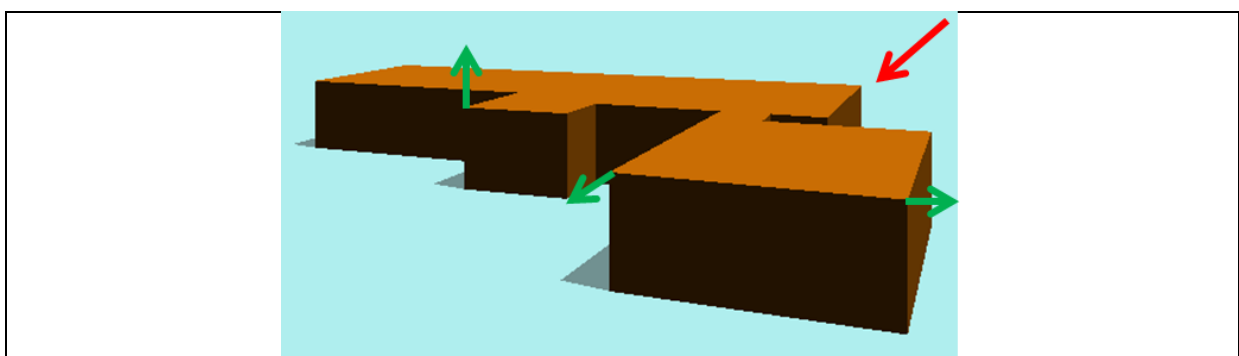
This technique on its own can produce some artefacts where multiple shadow areas are overlapping and therefore end up being drawn on top of each other. The artefacts are a result of two factors:

- Blending of multiple shadow areas
- Z fighting

To prevent this, employ a stencil test to ensure that only a single shadow area is drawn at any one pixel location. This test checks whether a shadow has already been drawn at a particular pixel location. If so, then the stencil test will fail and the hardware will discard the fragment.

### 4.3. Lighting

This section deals with the simple lighting technique that is used in the 3D demo, as even simple lighting can enhance the visual fidelity. It makes the scene much easier to interpret from a developer's point of view by helping to add some perception of depth to the scene. Lighting goes hand in hand with shadows. A scene with shadows but no lighting can look strange as there are no visual cues between faces that are in shadow and those that are not.



**Figure 15. Gouraud shading with shadows**

The lighting model employed in the 3D demo is Gouraud shading, also known as per vertex shading. This works by calculating the lighting contribution per vertex, which is the result of the dot product between the vertex normal and the light direction (light position -  $\mathbf{v}$ ). It then uses the calculated contribution to modify the object's colour. The advantage of employing this method is that it is a very cheap lighting technique that produces "good enough" results for the navigation demo. More sophisticated lighting models could be employed at the expense of computation time.

## 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 Tools and SDK and Insider programme, please visit:

<http://www.powervrinsider.com>

For general enquiries, please visit our website:

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