Navigation Rendering Techniques

Whitepaper

Filename	:	Navigation Rendering Techniques.Whitepaper
Version	:	PowerVR SDK REL_3.4@3164023a External Issue
Issue Date	:	30 Sep 2014
Author	:	Imagination Technologies Limited

Contents

List of Figures

1. Introduction

Visualization of navigation data is a complex task: the graphics should be appealing, informative, running at a high frame rate and utilizing 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 was given to the various restrictions found in navigation systems in respect to memory constraints, paging, etc.

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

For further reading and a more comprehensive overview please have a look at the documents which can be found in the PowerVR SDK (see Section 0). It is highly recommended to read the application development recommendation whitepapers to gain 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. 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 point-of-views covered in this document are:

- **2D top-down**: the standard perspective found in a lot of navigation devices from a bird's eye view. 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, it is possible to integrate panoramas to augment the scenery with images and efficiently achieve a higher level of realism.

This document covers 2D and 2.5D. The following sections explain how to visualize the most common cues like streets, road signs, landmarks, etc. in an efficient manner.

1.2. Sample Data

The sample data used throughout the navigation demo has been kindly provided by NAVTEQ. In particular the Chicago sample set in ArcView™ format was compiled into a binary format, which was then used for a navigation demo. The following sections are describing the optimisation techniques used for the demo. Implementations of the algorithms and techniques can be found in the PowerVR SDK (see Section 4) and an explanation of the various navigation data tools used to convert the sample data can be found in Section 0.

2. Data Organisation

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, e.g., considering types of mass storage, display resolution, orientation, etc. Therefore, easily adaptable algorithms are presented instead of specially tailored versions.

The following sections explain approaches to optimize visibility queries through spatial hierarchies, on how to improve performance by batching geometry and how to triangulate mapping primitives in a hardware suitable representation.

2.1. Spatial Hierarchies for Efficient Culling

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 (visible volume enclosed by the screen) or covered by other objects. Simply checking each object against the view frustum is not a good approach though, as the computational complexity increases linearly with the number of objects. Instead, a 2D version of a binary tree search can be applied, commonly known as quadtree.

A quadtree builds a spatial hierarchy on top of a dataset which speeds up certain operations like spatial queries. Each node in the quadtree 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 then looks like that:

- Beginning at the root of the tree each child is consecutively checked if its bounding rectangle intersects the view frustum.
- If a child is not contained all its children are culled at once, vastly pruning the search domain.
- If a child is not a leaf node, repeat the intersection test for each child.
- All intersected leaf nodes can be considered visible and the contained objects rendered.

Basically, a quadtree is built recursively, starting with the whole dataset and a bounding box enclosing it. The bounding box is then subdivided into four bounding boxes, 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, like 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, but it should be considered to generate them for the current working set. For example, if paging regions of the map during runtime, it is possible to dynamically generate them in the background.

Figure 1 illustrates a simple quadtree containing a set of points. 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 and if observed closely it will show that there is a dot outside of the culling rectangle which is coloured green. This is a false positive, but rendering it may cost less than doing a more fine-grained culling. Transferring this scheme to a map full of streets, landmarks, etc., can save you a lot of CPU and graphics core time, which can be used for other tasks.

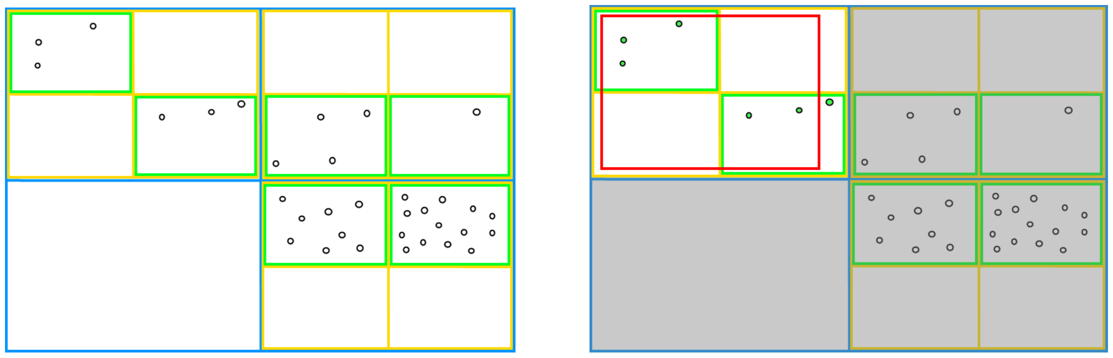


Figure 1. Quadtree used for hierarchical culling

Summarizing, a total of nine intersection tests where 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 thirty-three tests required for the individual objects, if no spatial hierarchy is available, this is almost a 75% saving. In most cases the tree is even more densely populated, further increasing the savings. 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 and further subdivisions of the quadtree could even give far better results/savings.

Also note that the depicted quadtree is a very simplified abstraction; indeed there are different variants of spatial partitioning schemes available and the one that best suits the user's needs should be picked (see Section 0).

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.

Figure 2 illustrates the result of a quadtree. All nodes intersecting the view frustum are drawn with a green border and the culled ones with a red border. It is apparent that there is a lot of geometry being drawn which is actually outside of the view frustum, but the rendering cost of this is negligible.

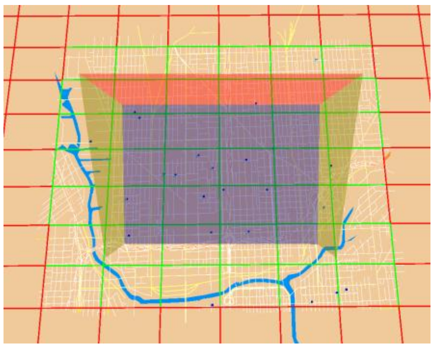


Figure 2. Culling scheme applied to streets layer

The navigation demo itself uses a slightly altered approach for spatial hierarchy. The navigation map is split into big geometry quadrants which contain smaller index quadrants. At first the viewing frustum is tested against the geometry quadrants and the index quadrants are only inspected if their geometry quadrant did not fail the test. As the number of quadrants in the navigation demo is quite low, performance-wise, this approach proved sufficient but does not scale well with the number of quadrants.

2.2. Batching Indexed Geometry

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

For example, a recommended solution is to triangulate multiple roads as a triangle list and submit the whole list of triangle indices with a single draw call rather than submitting each individual road by itself. In order to make a qualified decision for a certain method, the user will have to benchmark the various methods on the target platform.

The Navigation demo uses a mixed approach to optimize the number of draw calls, by producing larger batches and utilizing index buffers. The navigation map is subdivided into large blocks, each containing a big chunk of independent geometric data. The geometric data itself is made up of vertices which are converted into a vertex buffer object on demand during runtime. This scheme allows a block-based swapping of map parts during runtime and can easily be adapted to the underlying system architecture (available memory, data source access latency, etc.).

The blocks in turn contain index buffers referencing the vertex data. These are split according to a quadtree subdivision scheme to allow a fine grained culling of geometry within the block boundaries. During runtime a list of visible index blocks is computed and maintained based on the viewing frustum. These index blocks are then submitted for drawing, reducing the total number of draw calls to a bare minimum.

Figure 3 depicts the partitioning scheme: the sketch view to the left illustrates the map block identified by its blue border and the index buffer bins drawn in yellow. The image to the right shows a screenshot of a debug view of the application: each small rectangle in the big blue rectangle contains an index buffer and shares the same set of vertices with the other rectangles in the same blue rectangle. The blue rectangle contains the vertex data and holds the reference to the vertex buffer object (VBO). The number of draw calls required in this case to submit the entire set of index bins was twelve, which is very good considering the amount of streets actually drawn. Besides that, very little geometry outside of the view frustum is being submitted as can be seen in the empty rectangles in the screenshot.

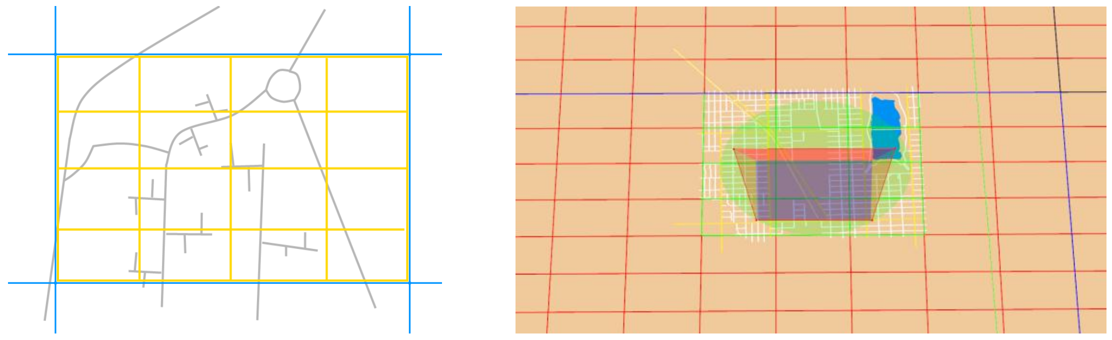


Figure 3. Map blocks containing index data (left: sketch, right: debug view)

2.3. Geometry Triangulation

Rendering roads, signs and landscape are the most important visual parts of a navigation system. Using the primitive data straight away as it is given by the map provider is not practical in most cases. The following subsections introduce techniques to prepare the data for the following rendering steps.

2.3.1. Road Triangulation

Using the street coordinates and simply drawing them as line primitives does have several drawbacks. The supported line width is hardware dependent and line anti-aliasing might not even be supported at all. Furthermore, it is inadvisable to make extensive use of line primitives, instead it is recommended to use triangles to draw roads which are more adequate for the targeted hardware.

In general, roads given by the GIS database are made up of 2D line strings, in some cases augmented by an elevation index describing the relative height of the road. These line strings have to be triangulated in order to be rendered efficiently. The process of road triangulation consists of two steps:

- Allocate storage space for vertex and index data. Preferably, this is done for a whole batch of streets to prevent memory fragmentation due to repeated reallocation.
- Calculate vertex positions and generate index data.

The allocation step is straightforward and the amount of memory to be allocated can be calculated by simply looping over the data and checking the amount of line segments.
In the following example (Figure 4) the line strings are represented by red lines, the generated geometry is illustrated by translucent grey lines and the letters represent the original line string's vertices. The green line illustrates a directional vector, which is used to calculate the blue lines. These 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.

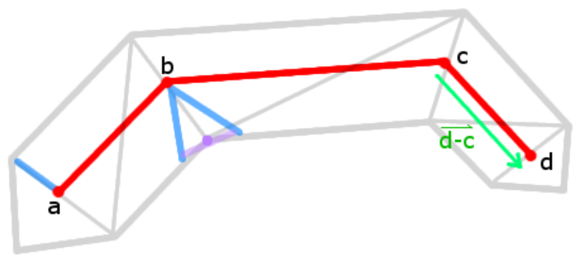


Figure 4. Vector calculus used to triangulate line string

The formula used is:

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

Where `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 normalized 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.

Special attention has to be given to the bends in the line string as there will be gaps depending on the degree of the bend. As illustrated in Figure 5 this can be fixed by calculating the intersection of the blue vectors spanning the sides of the road and using the new vertex (indicated by a red dot) for both line segments.

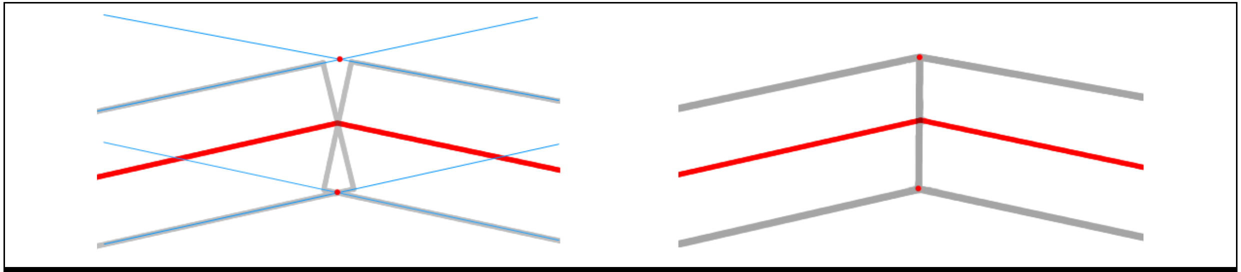


Figure 5. Gap between two triangulated line segments (left) and the stitched version (right)

Another consideration is to insert additional triangles depending on the angle of the bend. If the angle between the line segments is very small (see Figure 6) unnatural sharp bends are generated by the method described above. Subdividing these based on the angle produces a visual more appealing look.

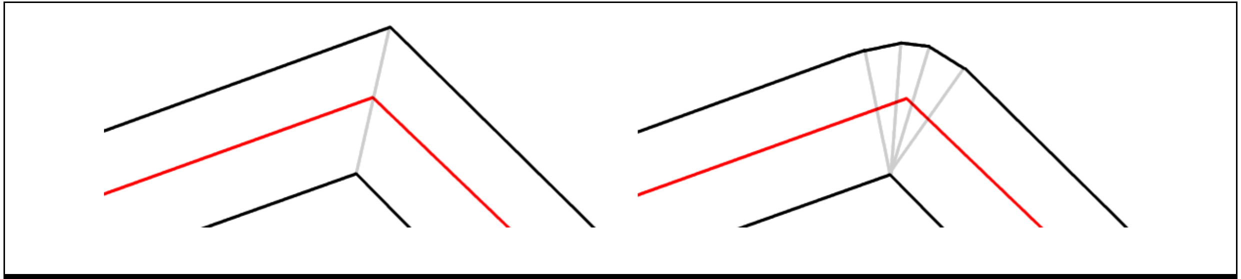


Figure 6. Tessellated road segments to prevent sharp bends

2.3.2. Intersection Triangulation

The previous section described a method to triangulate roads from line segments. Unfortunately, if special texturing techniques like drawing outlines or road markings are to be applied, simply letting road segments overlap will not give pleasing results. Overlapping segments at road intersections will overdraw each other in an arbitrary order and the result will be more or less random.

In order to allow for advanced techniques based on the triangulated geometry, it is mandatory to implement proper intersection triangulation handling. Figure 7 illustrates the idea of overlapping road segments and triangulated intersection. There are several ways to achieve proper intersection triangulation handling: one being a constructive geometric approach, which calculates all the intersections of the triangles and applies Boolean geometric operations to achieve proper geometry. The other one is an analytic approach, which tries to triangulate intersections based on their layout and order of streets.

The following will describe the latter approach as it has been implemented in the Navigation demo and can handle a lot of the intersections in navigation map data. There are intersections which produce artefacts when triangulated with this technique, especially when they meet at a very steep angle. Note that an intersection consisting of just two roads can be handled like in the previous section.

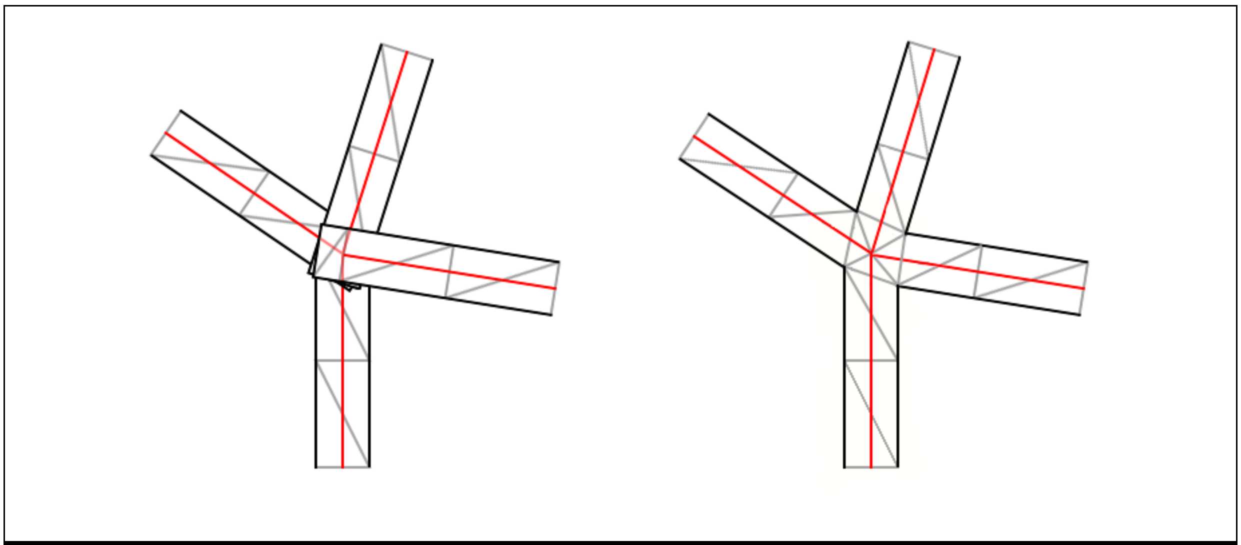


Figure 7. Overlapping road segments (left) and triangulated intersection (right)

The algorithm requires the set of intersecting roads as input and consists of the following steps:

1. Pick an arbitrary road as pivot element.
2. Calculate the angle between the pivot element and all other roads.
3. Sort the roads in ascending order according to the angle.
4. Calculate intersections between neighbouring roads and adjust vertices.
5. Patch all resulting holes with additional triangles.

Picking the pivot element is the first step in the procedure (see sketch (a) in Figure 8). In the navigation demo this involves simply picking the first road which is stored in the intersection vector. Based on this, the angles between this and all the other roads in the intersection are calculated: the direction vector of the original un-triangulated line segment is used and the dot-product calculated. The dot-product will provide the cosine of the angle between one vector and another, but it is necessary to make sure that the direction vectors are of unit length.

In the next steps all road segments are adjusted pair-wise in a counter-clockwise order (see example pair coloured yellow and green in sketch (b) in Figure 8). The intersection of the inner geometric lines (the blue lines and the red intersection point in sketch (c)) is determined and the vertices of the individual road segments adjusted (see sketch (d)), resulting in a proper corner. The whole procedure has to be repeated for each neighbouring pair and should result in a proper triangulated intersection.

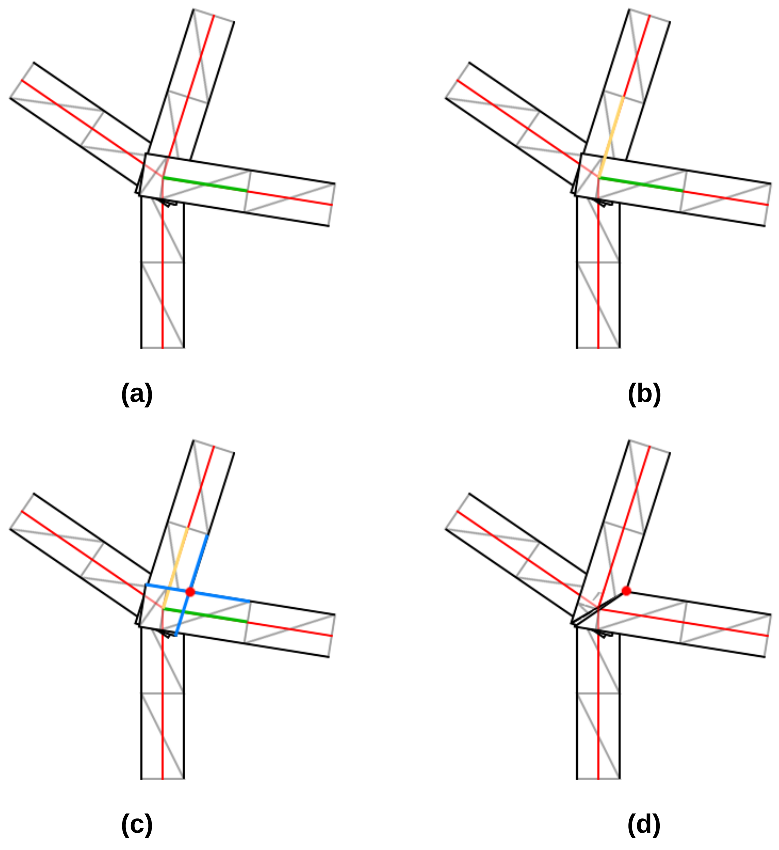


Figure 8. Intersection geometry adjustment steps

Due to the outward movement of the vertices there will be a gap in the middle of the intersection. This can be closed by adding an additional centre vertex (green dot in Figure 9) and generating triangles from the adjusted vertices and the new centre vertex.

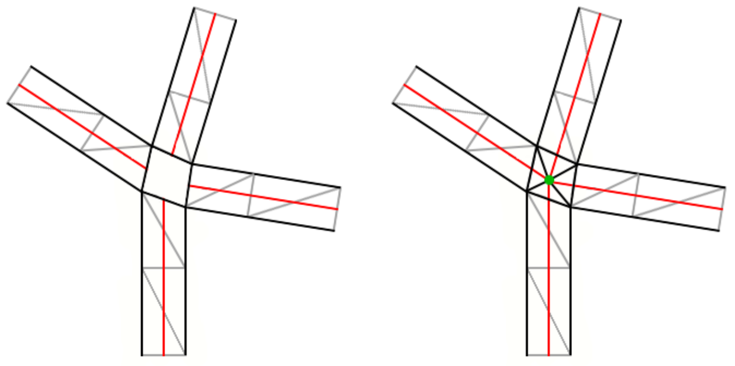


Figure 9. Adjusted intersection with hole (left) and stitched (right)

Please note that this algorithm is capable of handling a wide array of intersections, but there are cases in which it generates artefacts. Fortunately, these can be detected and treated separately. One of the most frequently occurring of these cases is when two adjacent intersections are closer to each other than the road width.

As illustrated in Figure 10 the two adjacent intersections, which are too close to each other, result in overlapping road segments. This could have been detected by looking at the length of the linking road segment. The intersection triangulation code handles these cases by virtually merging the intersections for the intersection triangulation step which reduces the artefacts in some cases.

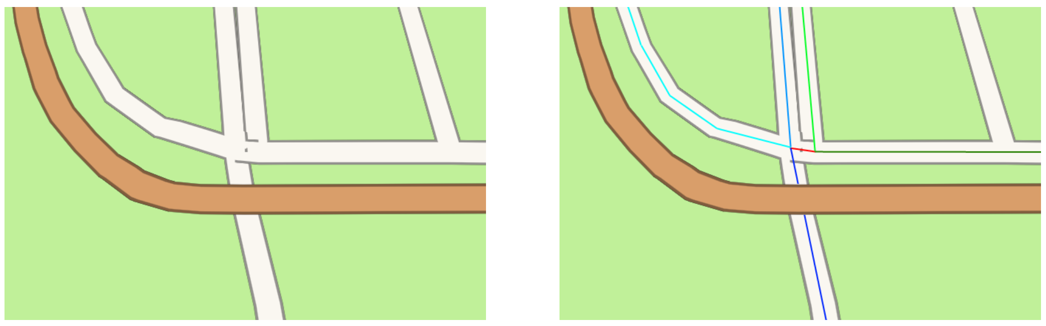


Figure 10. Artefact example: too near adjacent intersections

2.3.3. Polygon Triangulation

One of the primitive types employed in navigation maps other than points and lines are polygons. A polygon is described by a set of points given in a certain order, which define the shape of the polygon (see Figure 11).

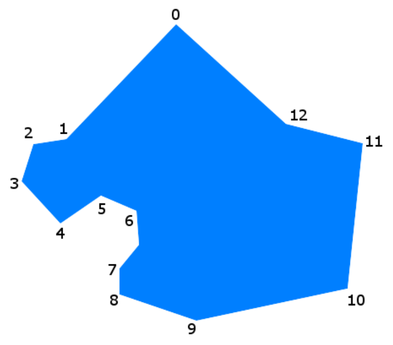


Figure 11. Polygon with numbered vertices

Furthermore polygons can contain holes and self-intersections (see Figure 12). Holes are described by a sub-polygon which literally 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, for example, is useful to describe islands in water areas, which are described as polygons. Self-intersections occur when one line strip of the polygon crosses another line strip in the polygon.

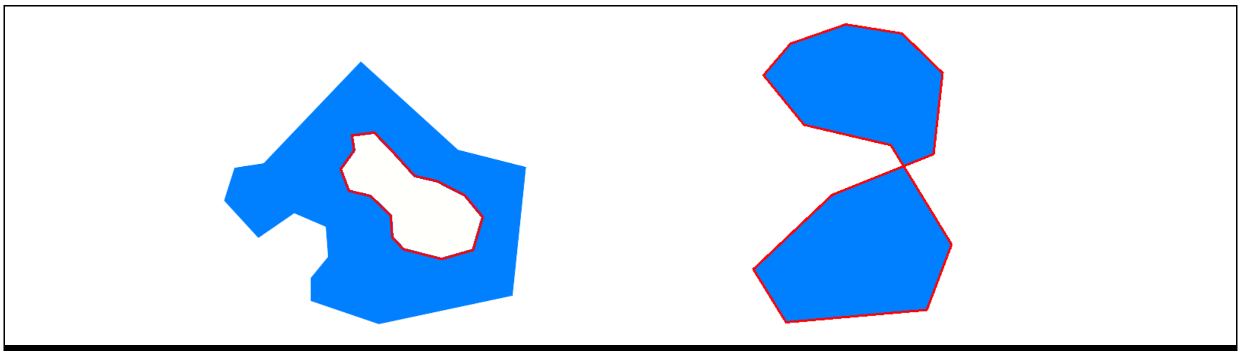


Figure 12. Polygon with a hole (left) and self-intersection (right)

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

1. Direct rendering by use of the stencil buffer, which is a buffer in addition to the colour and depth buffers which can control the rendering on a per pixel basis.
2. Triangulation into triangle primitives and rendering of those.

Method 1 is quite easy to implement but can perform very poorly at runtime. It uses the stencil buffer to generate the final shape of the polygon and stencil test to render it. Depending on the shape and size of the polygon this can cause a severe overdraw ratio for the individual triangles and is not recommended from a performance point of view.

Method 2 might prove difficult to implement, but performs a lot better than the previous method at runtime, as the polygon will be triangulated before use and the triangle representation cached for all further operations. However, the general difficulty is based on the complexity of the polygon, that basically means it is non-self-intersecting and does not contain any holes. In that case a simple triangulation technique called ear-clipping can be used.

This iterates over the polygon edges, successively building triangles out of two consecutive edges (vertices *0*, *1* and *2* in the middle example in Figure 13) and testing the shared vertex of both edges (vertex number one in Figure 13) against all other triangles in the polygon. If its area is positive and no other vertex of the polygon is contained within the triangle it can be safely clipped away. 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.

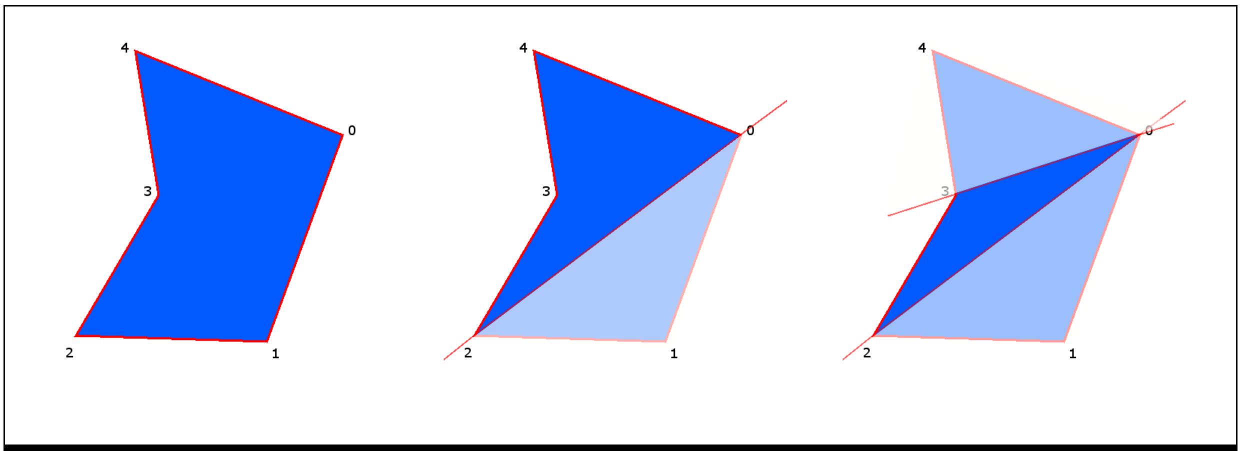


Figure 13. Polygon triangulation with the ear-clipping technique

The algorithm can be expressed as follows:

1. If the polygon exactly contains three vertices, add them to the triangle set and terminate.
2. Take two consecutive edges from the polygon.
3. Test whether the area is positive. If not go to step 2 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 2 and repeat with next pair of edges.
5. Remove shared vertex from polygon and add triangle to set of triangles. Go to step 1.

A working implementation of this algorithm can be found in the Navigation demo utilities. Please note that this algorithm cannot handle self-intersecting polygons and polygons with complex structures like holes.

2.3.4. Floating Point Precision

One of the more unexpected problems when dealing with navigation data is floating point precision errors. Navigation data is usually stored in a reference coordinate system and represents positional information defined within a scale from a few meters to several kilometres, stretching over thousands of kilometres. Thus very small fractions of floating point numbers are used to describe vast distances. From a pure representational point of view this accuracy proves to be enough.

However, employing vector calculus on this data and using arithmetic operations like the ones which are used in the previous sections may cause floating point errors. This is due to the internal storage representation of floating point numbers and round-off errors during calculation. Symptoms of these errors are gaps between streets which are supposed to meet or even visible jitter of the camera during runtime.

One possible solution to this problem is to store the coordinates relative to a local reference point and to perform the vector calculus on those local coordinates. This is done in the navigation demo, where after the construction of the spatial hierarchy, all coordinates are transformed relative to the minimum coordinates of the bounding box of their respective tree node.

$$p_{new} = p_{old} - p_{ref}$$

The only thing to keep in mind is that during runtime the camera matrix has to be manipulated so that the camera is offset by the same amount as the current coordinate set.

$$p'_{cam} = p_{cam} - p_{ref}$$

In the case of the navigation demo this helped to remove jitter which occurred occasionally due to vector maths that was used for path interpolation.

3. Rendering Techniques

The previous section dealt with the handling of geometric primitives and how to convert them into a format suitable for rendering. The following sections give attention to the various techniques used in the Navigation demo.

3.1. Adjusting Near and Far Planes

Setting optimal near and far plane values has two important motivations: the first one is that we base our culling on the frustum of the camera. A tight view frustum means that we are able to cull more geometry batches which are not sent through the rendering pipeline, saving CPU and graphics core time.

The second reason is the precision of the Z buffer. The Z buffer determines which objects are hidden behind other objects. It is limited in its precision and if the precision is not sufficient artefacts like objects showing through other objects, called “flimmering” or “Z-fighting”, might occur.

The relationship between clipping plane values and Z buffer precision can be understood when looking at the Z buffer value distribution. Z values are non-linearly stored in the range [0, 1]; more precisely this means that the resolution of Z values near the eye (the near clipping plane) is much higher than further away. One of the most common mistakes is to place the near clipping plane at a very short distance.

In order to calculate the view frustum intersection with the ground plane, we analyse the intersection of four view frustum viewing direction vectors (highlighted with yellow lines in Figure 14) with the ground plane. Projecting the shortest distance vector onto the viewing direction vector and calculating the length gives the near clip plane distance, repeating the same procedure for the longest distance vector gives the far clip plane distance.

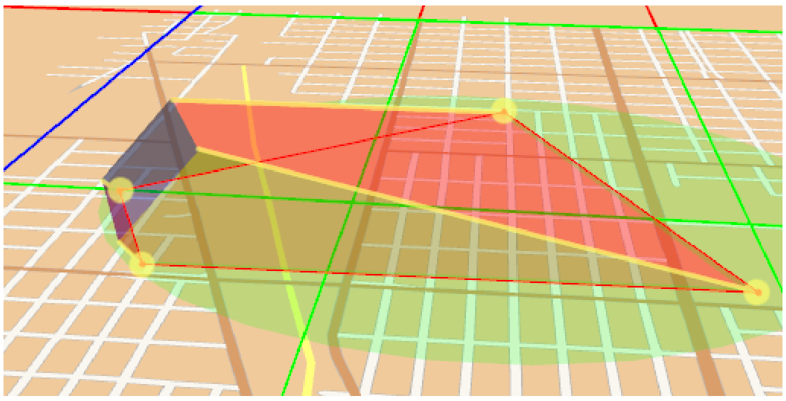


Figure 14. View frustum intersections with ground plane

Furthermore, the calculated intersection reference points can be reused for the culling step. Based on these points a bounding circle can be calculated which will serve as the intersection primitive against the quadtree. This is quite useful as the test whether a circle intersects a rectangle is computationally inexpensive in comparison to an arbitrary intersection test. Not only does this simplify the bounding primitive against quadtree traversal, but if the user enlarges the bounding circle slightly and introduces a gentle shift along the viewing direction (see the grey arrow in Figure 15) he/she will add nodes to the active index list which are not yet, but soon will be, visible leaf nodes (yellow highlighted area).

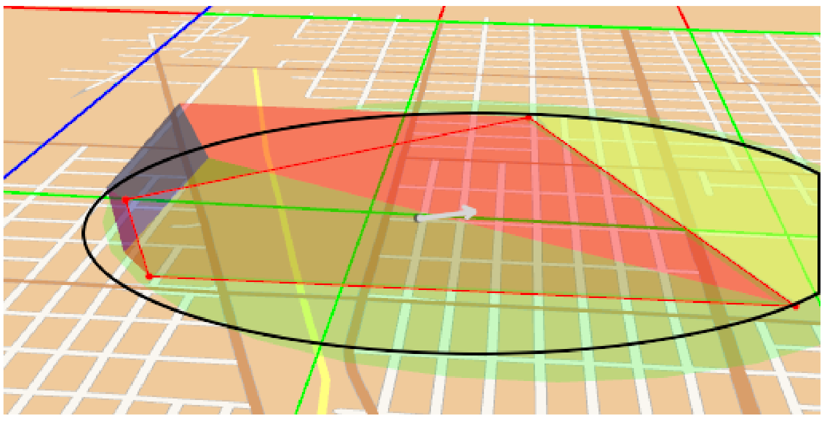


Figure 15. Bounding circle with slight shift

This can prevent the sudden appearance of geometry which is loaded during runtime, called popping. Considering that the view frustum against geometry culling and caching step is only done a few times per second in the demo no visible popping is noticeable during runtime due to this measure.

3.2. Anti-Aliased Lines

One of the most recurrent issues in computer graphics is aliasing. Scientifically, 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 (see Figure 16) or visual artefacts when texture mapping (see Figure 17).

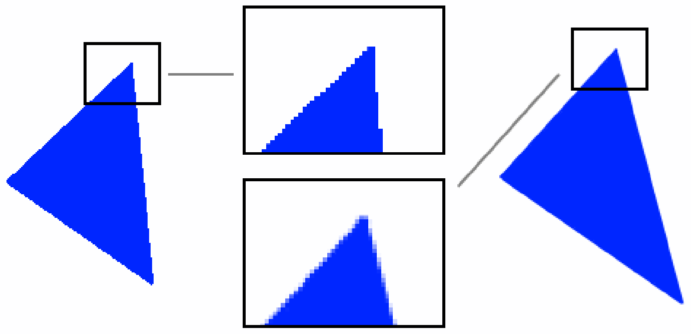


Figure 16. Comparison of rendering with anti-aliasing disabled (left) and enabled (right)

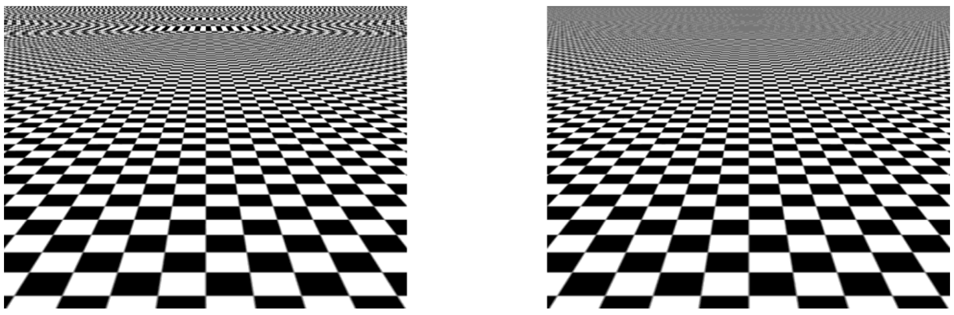


Figure 17. Texture aliasing: linear filtering (left) and linear filtering with MIP mapping (right)

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 (which are commonly called “jaggies”). These require special hardware support that is present in all modern 3D hardware, but might incur in some performance cost.
In the case of rendering simple road geometry, however, we are able to help ourselves without the need for special hardware support. All that is required is the capability to texture map triangles and blending which is supported on all OpenGL ES capable platforms. The technique which is applied in this section is introduced in detail in the Anti-Aliased Lines training course in the PowerVR SDK (see Section 4).

As can be seen in Figure 16 the most visible artefact is the abrupt change between the triangle edge and the background. When looking closer at the anti-aliased example the triangle edges seem to blend with the background at continuous varying levels of transparency.

In order to achieve a similar high quality anti-aliased result, we misuse the blending hardware to emulate this effect and gradually blend the edges of the geometry with the contents of the framebuffer. Therefore, we draw the road triangles with a special texture map applied and enable blending.

The texture map in Figure 18 shows a specially constructed translucency map which represents the alpha values. It is encoded as a single channel 8-bit grey value map which only contains two values: 0 and 255. These values represent the levels of opacity and the fractional values of opacity in between are generated during runtime by the linear texture filtering. Those fractional levels and the resulting blending in and blending out effect produces the anti-aliased look.

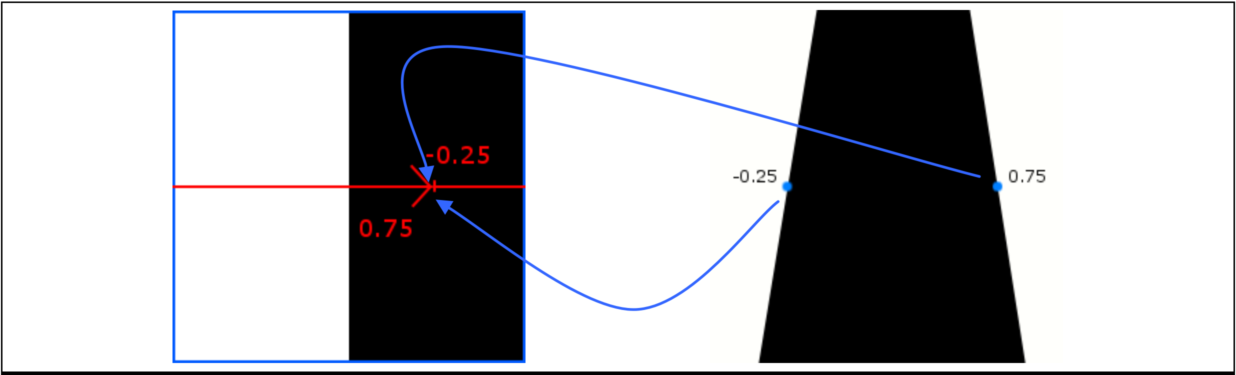


Figure 18. Single channel alpha map (left, added blue border for visibility) and applied onto a road texture (right).

In order to make use of these values we need to augment the vertices which have been generated in the triangulation step in Section 2.3.1 with specially adjusted texture coordinates (see Figure 18). Considering a top down view on a road, it works by assigning texture coordinates to the left side of the road which point into the middle of the right half of the translucency map. Instead of just using 0.75 as horizontal texture coordinate, -0.25 is used (this will be explained later on). The right side of the right will point to the exact same location, this time using 0.75 as texture coordinate.

The OpenGL texturing mechanism will interpolate linearly between both texture coordinates and cause it to cross the right texture map border, halfway in between both coordinates. Thus the appropriate border handling has to be assured in the OpenGL state machine: the wrap handling mode of the texture (`glTexParameter()`) for the `GL_TEXTURE_WRAP_S` parameter has to be set to `GL_REPEAT`. This trick not only saves storing one third of the required texture space, but actually is useful when mip-mapping is employed.

As it is visible in the visualized MIP levels (see Figure 19) the texture is able to preserve its original appearance, despite a large shrinking factor. This makes it possible that even roads of two pixels width can be properly rendered, as the second-lowest MIP level (the 2x2 pixel block in Figure 19) still contains the original values at the texture coordinate sampling positions. At the lowest MIP level the texture is of 1x1 width and height and can only store one discrete value, which is not enough for the employed texture interpolation scheme. Thus the developer has either to make sure that lines are at least wider than one pixel or the resulting artefacts must be considered acceptable.

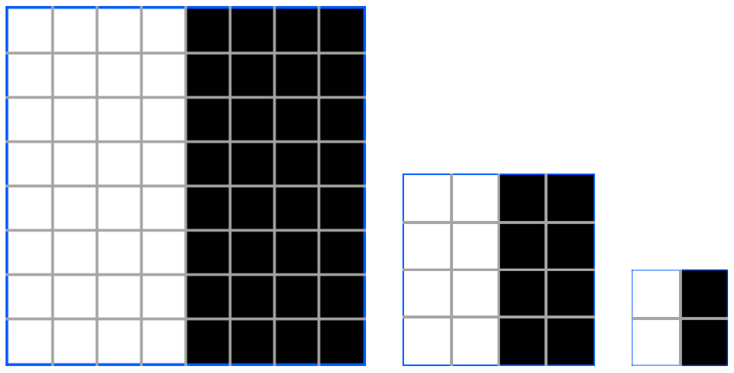


Figure 19. Several MIP-map levels of the translucency texture

One important thing to consider is that the rendered road will be half as wide as the actual geometry due to the alpha blended parts. There are two possible workarounds: widening the triangulated geometry or offsetting the texture coordinates. The first option is straightforward and should only require the adjustment of a single parameter. The second option shifts the texture coordinates by a fractional amount towards each other so that less of the translucent part of the texture is rendered. This is employed as an example in the Navigation demo.

3.3. Billboards

Efficiently drawing text onto the screen is a vital aspect of navigation software. The same goes for other navigation elements like road signs, speed limits or general signs like public park indicators. In this context it is strongly advised for optimal performance to use OpenGL ES for rendering text or drawing bitmaps, as using operating/windowing system primitives may compromise performance.

For the purpose of rendering any element mentioned above the user would submit a textured rectangle (please see Section 3.4 for texturing optimisations). Unfortunately, depending on the orientation of this rectangle there are only a few viewer positions where the rectangle lines up with the viewer and is actually readable (see Figure 20).

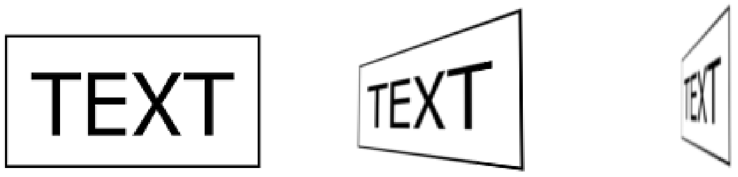


Figure 20. Screen space aligned and perspective text

In order to be able to easily read the text we have to face it in the direction of the viewer somehow. Furthermore, it is desirable that nearby text is larger than text farther away. The first aspect is more difficult, as the reorientation of signs has to be done every frame as the viewer moves through the scenery. The second aspect can be achieved by simply applying a perspective projection.

The general algorithm behind billboarding is derived from the camera's intrinsic parameters. As illustrated in Figure 21 the camera orientation is defined by three orthogonal vectors which span the camera coordinate system: the x-axis (red), the y-axis (green) and the z-axis (blue).

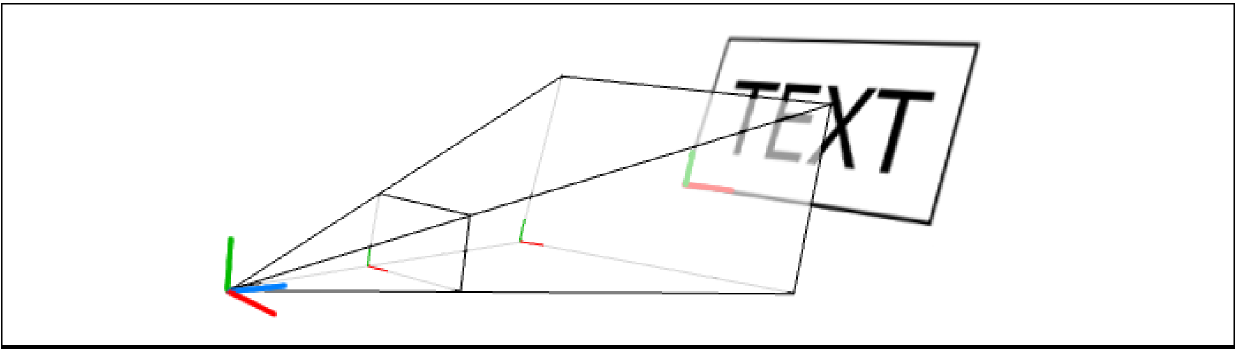


Figure 21. Camera coordinate system and spanned rectangle

Utilizing the x-axis and y-axis vectors to span the textured rectangles allows us to have them oriented towards the viewer. The only input required is the origin and the width and length of the rectangle. One method to retrieve the camera vectors is by extracting them from the model-view-projection-matrix (see Figure 22).

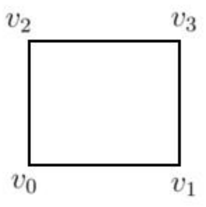
$$\begin{pmatrix} m_{01} & m_{02} & m_{03} & m_{04} \\ m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \end{pmatrix}$$

Figure 22. Highlighted columns corresponding to camera vectors

The following two subsections give examples on how to achieve the previously described effect on the CPU and the graphics core. Both methods have their advantages and disadvantages and the most suitable one for a certain use-case can be picked and adapted. Subsection 3.3.3 presents a method to extend the described algorithm to render multiple billboards aligned along a certain axis. This is especially useful if every letter in a string is represented by a single billboard and it is desired to align in screen space in order to construct a word.

3.3.1. CPU Technique

Implementing billboarding with the use of the CPU is straightforward. First of all the user has to define the origin and the texture coordinates for the four vertices and the width and height of the spanned rectangle. In order to screen-space align the rectangles it is necessary to extract the camera coordinate system vectors as illustrated in Figure 22 and do some vector calculus to calculate the final position of the individual vertices. Assuming the origin defines the lower left corner, the formulae for the other vertex positions are:



$$\begin{aligned} v_0 &= v_{origin} \\ v_1 &= v_{origin} + v_{cam_x} * width \\ v_2 &= v_{origin} + v_{cam_y} * height \\ v_3 &= v_{origin} + v_{cam_x} * width + v_{cam_y} * height \end{aligned}$$

The vertex positions have to be recalculated if the relative position between the viewer and the billboard changes. This might become impractical if a huge number of billboards are visible, e.g., a lot of text is being rendered onto the screen. Additionally the vertex data has to be altered frequently which might have a severe performance impact when using vertex buffer objects on some platforms. For these cases a technique entirely running on the graphics core is presented in the next section.

3.3.2. Graphics Core Technique

The billboarding technique presented in the last section can be implemented on the graphics core when applying minor modifications. The biggest advantage in comparison to the CPU method is that once the initial data is created, the user does not have to maintain it during runtime when the relative viewpoint changes.

Modern 3D hardware has dedicated programmable shader capabilities that allow a very high degree of flexibility when processing vertices and pixels. The first step of this technique is submitting two triangles for each rectangle to be drawn as usual, but the vertex positions of the triangles now correspond to the origin of the billboard. The spanning of the rectangle and thus the calculation of the final position of each billboard vertex is then done in the vertex shader. The most complicated part of this is that the relative vertex location (lower left corner, lower right corner, etc.) is not accessible in the vertex shader, so it does not have to be derived somehow. There are two basic identification methods which differentiate between storage, bandwidth and computational cost:

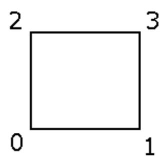
- Storing spanning coefficients as vertex attributes, using them as a spanning vector mask.
- Storing a relative triangle index as vertex attribute, doing case-based vector calculus.

The first method expects spanning coefficients as vertex attributes in addition to the usual vertex attributes like position, uv-coordinates, etc. The formula stated previously can then be expressed in a very compact way:

$$v_{world} = v_{origin} + v_{cam_x} * s_{width} * \mathbf{s_{mask_x}} + v_{cam_y} * s_{height} * \mathbf{s_{mask_y}}$$

Each vertex is translated among the different camera coordinate vectors based on the value of masking scalar. The user could make use of this scalar for more than just simple masking, e.g., define a per vertex width/height modifier which enables the creation of more shapes than rectangles.

The second method requires only one additional vertex attribute, but conditional branching has to be used in the shader. The per vertex translation vector is then calculated based on the triangle corner index parameter which is specified as the additional vertex attribute:



$$\begin{array}{ll} if(i == 0) & v_{world} = v_{origin} \\ else\ if(i == 1) & v_{world} = v_{origin} + v_{cam_x} * s_{width} \\ else\ if(i == 2) & v_{world} = v_{origin} + v_{cam_y} * s_{height} \\ else\ if(i == 3) & v_{world} = v_{origin} + v_{cam_x} * s_{width} + v_{cam_y} * s_{height} \end{array}$$

Based on the encoding type of the parameter this method can allow saving to four bytes per vertex.

3.3.3. Billboard Arrays

Both techniques presented in the previous section supports the ability to efficiently render a huge number of screen space aligned rectangles. The biggest drawback though is that it is not possible to draw several individual billboards so that they are always aligned relative to each other.

However, such a property would be desirable when rendering text, as it is not feasible to represent each string with an individual texture which can be mapped to a single billboard. Instead, each character of a string is represented by a single textured billboard, which has to line up with the other characters in order to build the original string (see Figure 23).

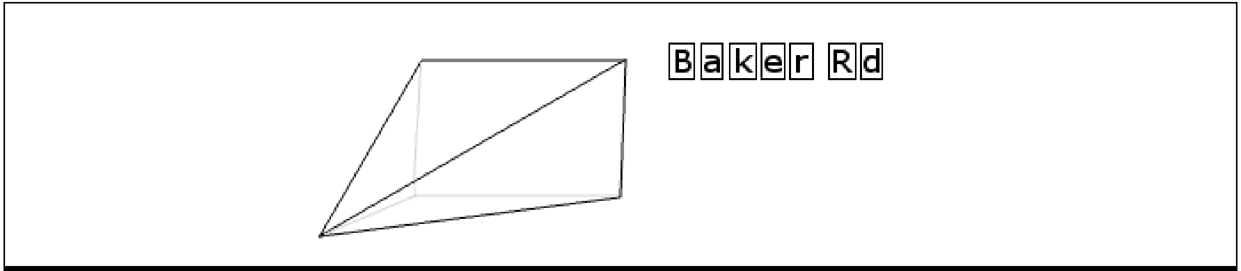


Figure 23. Multiple billboards aligned to build a string

Figure 24 highlights two prevalent issues when rendering text with single billboards: the first misalignment illustrated in the left sketch shows a simple rotation around the viewing direction, which causes the rotation of the individual letters. The road name is still readable in that case, but the letters do not line up anymore and additional rotation would make it much harder to read up to the point where the letters are facing top down.

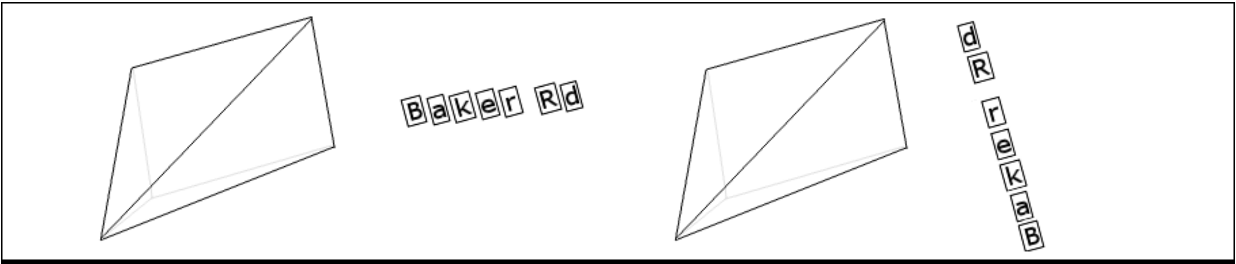


Figure 24. Illustration of occasional problems when rendering text as billboards

The right sketch in Figure 24 illustrates the problems arising from a different viewpoint: depending on the relative position of the camera, the text could be upside down and the readability would suffer severely.

In order to solve these issues we are going to introduce a technique to draw arrays of billboards which are aligned along a certain axis. The central idea behind this approach is that multiple billboards share a single origin which is called “pivot element” from now on. This shared pivot element is used as reference point for all succeeding offset calculations and thus automatically takes care of alignment issues (see Figure 25).

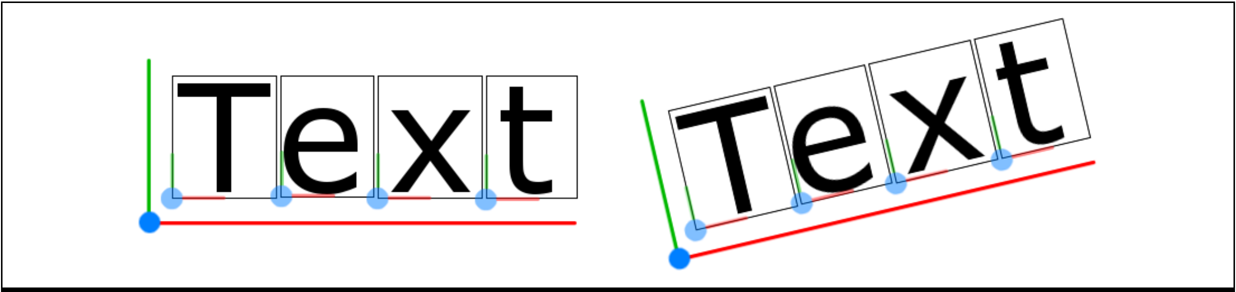


Figure 25. Billboard array before and after rotation

Starting from the shared pivot element (dark blue dot) it is possible to calculate a character-local pivot element (light blue dots) which is indeed the individual origin of the previous billboard techniques. Figure 26 illustrates the individual pivot elements in detail. Based on that local pivot element the previously described graphics core based billboarding technique can be applied.

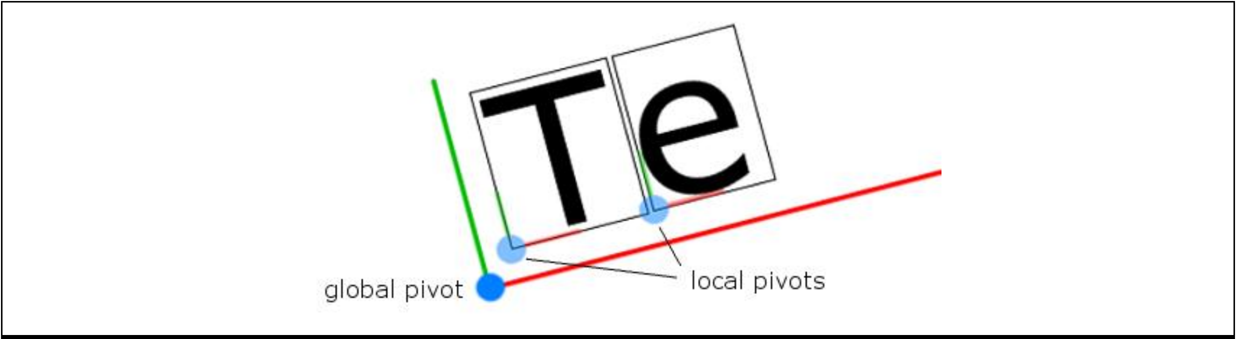


Figure 26. Global and local pivots illustrated

The vertex attributes have to be augmented by a few additional parameters in order to make this technique work:

- A shared pivot element: each letter in a string will have the same shared pivot element.
- An offset scalar: this attribute serves as an offset calculator within the string.

Using the global pivot as an anchor the local pivot element can then be calculated with the following formula:

$$v_{local\ pivot} = v_{global\ pivot} + v_{cam_x} * s_{offset}$$

3.4. Texture Atlases

As previously mentioned, one of the primary goals of optimisation is to reduce the number of submitted draw calls and state changes during rendering as far as possible. The techniques introduced in the previous sections have shown that this is possible by grouping similar primitives, but there is still the issue that several primitives in a batch might apply different textures. Unfortunately, textures are set per batch, so we would have to split those batches again according to texture usage, diminishing the positive effects of the illustrated batching techniques. Additionally, altering texture map bindings is one of the most costly state changes in most rendering APIs, thus an ideal candidate for optimisation.

Figure 27 illustrates the issues associated with texture mapping and geometry batching: although we manage to draw a screen-space aligned row of quads with a single draw call, we would have to split these batches according to the respective textures and change texture bindings for each letter drawn.

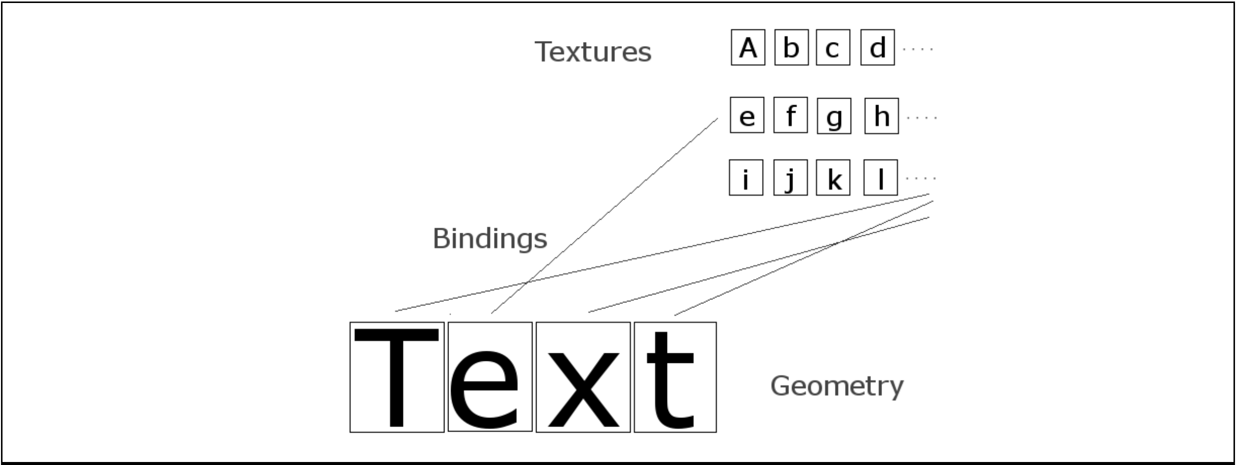


Figure 27. Rendering a batch of glyphs without texture atlas

In order to solve these issues we introduce a technique called texture atlases. A texture atlas assembles multiple textures in a single texture, allowing one to draw multiple batches of primitives without having to rebind different textures (see Figure 28).

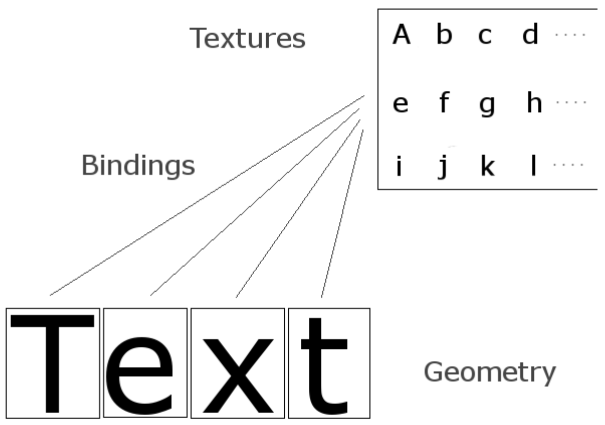
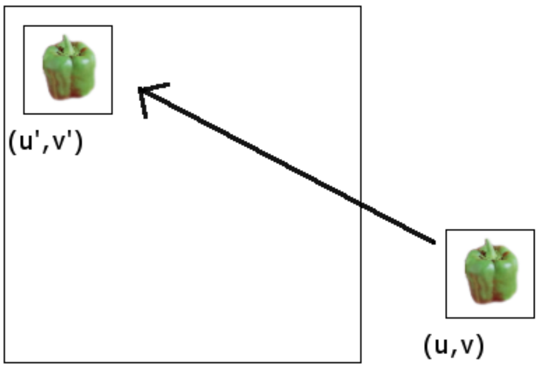


Figure 28. Rendering a batch of glyphs with a texture atlas

The first step when creating a texture atlas is to determine the layout of the various textures in the atlas. There are tools that can automatically generate texture atlases out of a set of textures. After laying out the textures in the atlas the original texture coordinates have to be mapped to match those of the new location. In those cases where textures are being mapped onto quads or triangles it is easy and sufficient to simply replace the old texture coordinates per vertex with the new ones. In more complex cases, e.g., where parts of a texture are spanned over several triangles and vertices do not coincide with texture map corners, this simple readjustment will not work. In these cases the user has to calculate the transformation from the source texture with its texture coordinates into the new location. This can be written down as a 3x3 matrix which is able to describe all translation, scaling and rotation transformations. Each texture coordinate is then simply augmented by a third component which is set to 1.0, called the homogenous component and the new texture coordinate is calculated by multiplying the texture coordinate vector by the transformation matrix (Figure 29).



$$\begin{pmatrix} 1 & 0 & dx \\ 0 & 1 & dy \\ 0 & 0 & 0 \end{pmatrix} * \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \begin{pmatrix} u' \\ v' \\ 0 \end{pmatrix}$$

Figure 29. Simple texture atlas layout example with texture coordinate mapping calculation

When distributing the textures in the atlas there are various things to consider:

- **MIP mapping:** due to the logarithmic decrease in texture size, the textures within an atlas will collide at a certain MIP-map level. This means that they will start to smear into each other and special consideration has to be taken in order to shift these artefacts back into the MIP-map chain as far as possible. Generally textures within an atlas should be aligned to power-of-two coordinates and they should not cross power-of-two lines, e.g., given a 512x512 texture atlas and a 64x64 texture one should try to place the texture in one of the corners instead of the dead centre. The texture atlases in Figure 30 illustrate a perfect layout in the left image, where the first collision turns up in the very last MIP-map level. The right image illustrates a non-power-of-two aligned case, where the first texture collision appears in the fifth MIP-map level.

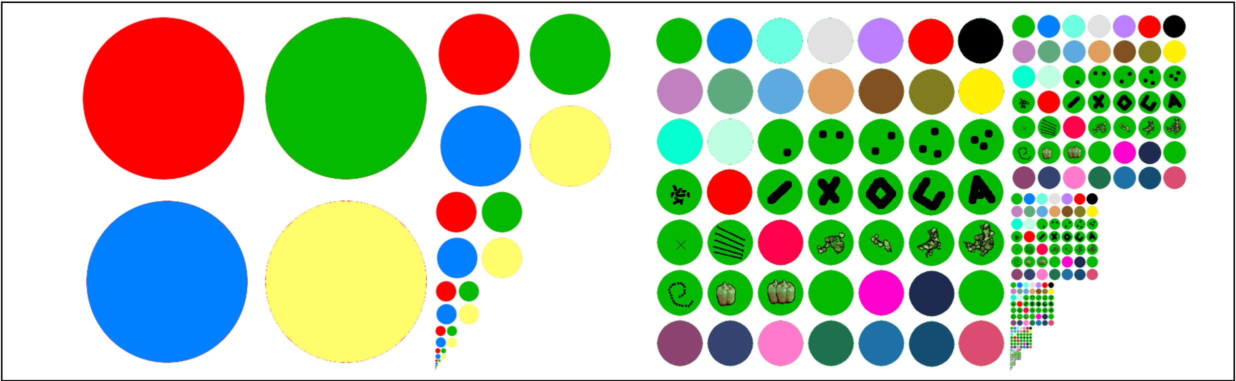


Figure 30. MIP-map chain of power-of-two aligned textures (left) and non-power-of-two aligned textures (right)

- **Texture address modes:** when using texture atlases the user has to take care of the texture address mode being used for the various textures. Usually, if your texture coordinates are not within the range [0, 1] (e.g., when using a grass texture which should be repeated forever), the addressing mode which can be clamp, wrap or mirror has to be specified. However, as soon as a texture is contained within a texture atlas these modes are not applicable any longer. Coordinates not contained within the mapped texture coordinates will point into other textures contained within the same texture atlas. There are workarounds, e.g., replicating textures in an atlas or handling the special cases in the pixel shader. But in most cases it is recommended to leave textures that require special address modes in their original texture.

3.5. Outlines

Rendering apparently simple outlines adds a lot to overall visual quality. A quick solution would be to just add additional geometry that resembles the shape of the outline, but this is not recommended due to the additional processing cost. Unfortunately, it is very difficult to add outlines without properly triangulated geometry. If road intersections are not properly triangulated, the outlines make the overlapping of the various road segments visible. Thus it is important to have a proper intersection handling as described in Section 2.3.2. Figure 31 illustrates map examples with disabled and enabled outlines.

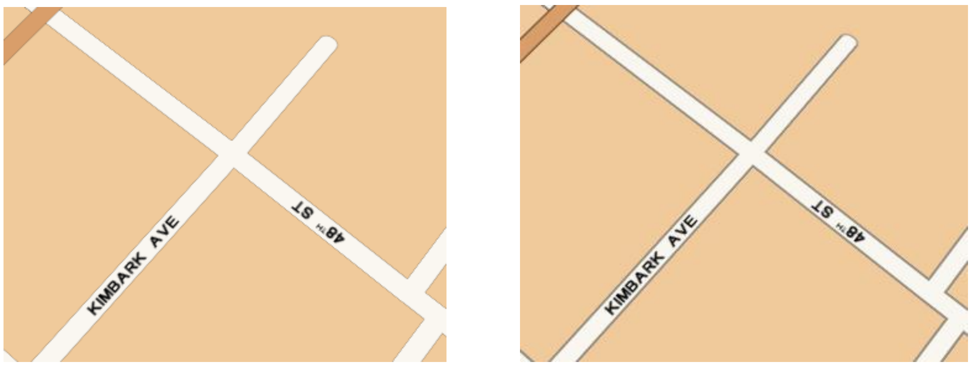


Figure 31. Map examples with disabled (left) and enabled (right) outlines

With properly triangulated intersections it is possible to employ a texture based solution: as we are already using a texture to render anti-aliased lines, it is possible to augment it with an additional channel that contains the outline information (see Figure 32). The resulting texture uses the OpenGL luminance-alpha storage format to store the outline as the luminance information. It is important to make sure that the outline width in the texture is sufficiently thick, as it will be propagated through the MIP-map chain and become thinner at each MIP-map level.

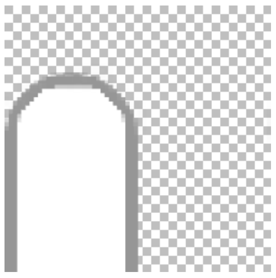


Figure 32. Texture containing outlines for anti-aliased lines technique

During runtime this texture is sampled as usual and, in addition to the alpha information, the luminance part is used to generate the outline. In case of the Navigation demo the luminance value is used as a scalar which is multiplied with the road colour, darkening the colour in the outline regions. In addition to the outline at each side of the road it is possible to generate a texture-based rounded cap at the end of dead-end streets. This can be achieved by adding small caps to the end of these roads and setting the texture coordinates accordingly.

As illustrated in Figure 33 the `v` texture coordinates map to the middle and top of the texture, whereas the horizontal `u` coordinates resemble the previous ones.

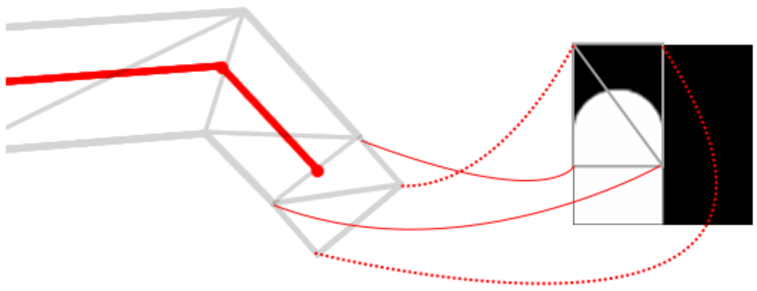


Figure 33. Adding a rounded cap to the end of a road

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