Exploring Portals in Consumer Class Applications

Master's Thesis

by

Thomas M. B. F. Gjørup

Abstract

This thesis explores certain acceleration techniques in the area of 3D computer graphics. It deals with the subject of visualising large world models in real time.

The thesis describes visualisation techniques for creating ‘interactive walkthroughs’ of world models, as used in e.g. Virtual Reality, simulation and computer games. For such applications, the demands of high update-rates and short response times are very high.

The described techniques are based on a complete subdivision of the environment into cells, where inter-cell visibility is only allowed through explicit ‘portals’. Portals are comparable to *windows* placed on the shared boundary between two cells, and they can be viewed as edges in a graph describing the connectivity and general structure of the world model.

The techniques described are of general use, but their usability is also reviewed in the perspective of a certain application area, namely the broad industry of software for entertainment, simulation and other real-time visualisation aimed at consumer level hardware. Apart from posing demands of interactive frame rates on a common hardware base, the applications in this area often require a high level of real-time world modification. This type of dynamics is not asserted in most of the previous work on virtual walk-troughs; in fact the work often exploit the presumed static nature of the input world to create pre-computed information for speeding up the rendering process.

The thesis describes a specific case in which portal techniques were successfully used for accelerating a - by common standards - large world in a computer game. Apart from using the portals for accelerating the rendering process, the implementation did also benefit from the underlying data structures for other purposes.

Finally, the thesis suggests a new approach to make automated portal and cell classification techniques achieve more ‘natural’ subdivisions of certain environments, namely environments with irregular shapes and a lack of ‘obvious’ portal locations. When using current techniques, such environments are often subdivided beyond sense, and they create an unreasonable time- and space overhead from the acceleration itself.

Contents

1 Preface

This thesis was written in the first half of 1999 by Thomas Gjørup at the Computer Science Department of the University of Aarhus, Denmark, under the supervision of Jørgen Lindskov Knudsen. Part of it builds on the author's experience from several years of professional work within the computer game industry, and part of it describes specific experiences from the author's involvement in the development of a commercial game project (chapter 3). The new methods described in chapter 4 were developed specifically for, and as a part of this thesis.

1.1 Introduction

This thesis explores certain acceleration techniques in the area of 3D computer graphics. It deals with the subject of visualising large world models in real time.

More specifically, it deals with visualisation techniques for creating 'interactive walkthroughs' of world models, as used in e.g. Virtual Reality, simulation and computer games. For such applications, the demands of high update-rates and short response times are very high. The application area is typically very large virtual worlds inside which an observer is allowed to move around and observe the world structures from a highly mobile viewpoint, literally straining every part of the graphics pipeline.

The general growth in computer performance and in graphics hardware in particular is very high, so a relevant question is whether such acceleration is necessary at all. Will the general performance rate be able to catch up with the highest demands in this area, rendering such cumbersome acceleration techniques superfluous?

We shall suggest that the demands are growing likewise, and yet far from being achievable using a brute-force approach. In the light of this answer, we further define the goals we wish to achieve, and finally we identify the target areas in which these techniques seem feasible and the target areas we wish to address in this thesis.

1.2 Motivation

It is common in computer science to evaluate algorithms under their asymptotic behaviour, i.e. for infinitely large sets of input data. Indeed, there seems to be a persistent demand of working on increasingly bigger data sets as hardware restrictions in storage and performance is lowered.

Graphic models of today still seem to be expandable several orders of magnitude both inwards and outwards. In CAD/CAM systems, models are usually described to the finest level of detail, but for other systems the models seem to stop at a level above the finest achievable detail. For systems visualising a virtual world, the overall size of the world is usually restricted by storage and/or performance level.

In general, the storage and performance restrictions imposed by a given system will make large worlds lack detail or detailed worlds lack size. In this context, large worlds could be e.g. a large city model where building details like windows etc. are merely represented by textures. Detailed worlds could be highly detailed architectural models, where building complexes are split into separate buildings or buildings are split into separate floors. The demand of higher visual quality also requires textures to be of increasing detail and materials to be more complex, including textures for bump mapping and light maps etc.

Under these assumptions, we do not consider it a valid claim that world models have reached anything close to their highest desirable size or detail complexity. Therefore, we assert that techniques for accelerating the visualisation of complex worlds are highly relevant. We assert that the computational powers used in a brute-force attempt to render a world without acceleration is better spend increasing the detail in the *visible* part of the rendered world.

Although a roughly detailed model of limited size will suffice for certain application areas, the ultimate goal of Virtual Reality is to create a world that appears real to the observer. This implies that the ‘horizons’ of world size and detail complexity should be no more apparent to the observer than in the real world.

1.3 Goals

In the context of worlds of ever increasing size, we seek solutions that are scaleable, i.e. solutions with computational and storage requirements that do not explode as world sizes increase. We also seek a solution that shows a certain degree of ‘local’ behaviour. Updates or queries inside a local area of the world should not receive any computational penalty from the total size of the world. Nor should it require any knowledge of - or access to - distant parts of the world.

The first requirement is necessary to make the acceleration technique compete with the brute-force approach under asymptotic conditions. Accepting the numerical inaccuracies introduced by a depth buffer, the brute-force technique is at most linear in the size of the input world. If distance-sorting of graphic entities is required, it is at most $O(n \log n)$ in world size. Working with fixed visibility horizons or if the data set comes with some hierarchical ordering, the cost can be even lower.

We can to some extend justify a more costly solution, if the increase in computation time is spend in pre-processing the world, ultimately yielding better interactive performance. This is due to the fact that virtual worlds are often created once and viewed many times. However, pre-processing can also be a strait-jacket for dynamically changing worlds, if the data structures created by the pre-processing are not allowing for efficient dynamic updates to the world.

The second requirement concerning the local behaviour of the solution is induced by the quest for an ‘optimal’ rendering solution. We consider a visualisation to be optimal, if it only requires work proportional to the size of the *visible* part of the world for any given observer. If deploying a level-of-detail scheme that omits detail features of the actually visible part of the world, the work should be proportional to the chosen amount of visualised detail.

A benefit of such a (close to) optimal visualisation is that it has good caching behaviour. The algorithm need only access parts of the world database that are ‘local’ to the observer, i.e. with a high possibility of being visible to the observer.

More specifically, we seek methods for efficiently creating some ordering or subdivision of the world allowing us to cull away parts of the world that are distant or invisible to the observer. Note that this is not achieved with acceleration schemes like simple Hidden Surface Removal as performed by painter’s algorithm, view frustum culling or detail-omission by level-of-detail. These methods still consider invisible parts of the world occluded by e.g. a large polygon close to the observer.

1.4 Target Applications

‘World models’ is a broad term covering numerous classes of data that could be subject to visualisation. First, we limit ourselves to such models that have a polygonal representation. These could either be a polygon mesh (typically a data structure consisting of shared vertices and faces) or plainly a so-called ‘polygon soup’ (a collection of polygons with no presumed ordering or connection).

Furthermore, we recognise that certain world classes show features that can be exploited by highly specialised algorithms due to data being e.g. significantly repetitive or bound under certain constraints. One such example would be terrains generated from height fields, which are often represented by one single surface elevated over a uniform grid. Terrains are usually - on a big scale - relatively flat and show a low degree of self-occludence from most viewpoints. Therefore, terrains may be more efficiently rendered by specialised algorithms that exploit their uniform representation (e.g. in a quad tree) for efficient world-to-frustum clipping and level-of-detail selection [Lin96]. Other input data that comes with some kind of clustering or hierarchy (e.g. a hierarchical ordering kept from the creation or assembly process) may also be subject to specialised algorithms.

Finally, we shall make a distinction between worlds that show a *low* or a *high* degree of self-occludence. Since our goal is to use occludence to also cull invisible parts of the world that are *inside* the view frustum, we choose to regard only those classes of worlds for which this approach

seems optimistic. Work has been done on occludence culling in environments without particular self-occlusion such as [CoTe97] and [ZMHH97]. These techniques show relatively small and extremely view-dependent improvement (typically in the order 2-5 times) compared to work on densely occluded environments such as [Tel92], where up to 99% of the world is culled away.

A typical example of a 'hard case' to accelerate is a model of a town like New York with a multitude of high rises. At ground level it may be possible to pick out a few large occluders near to the observer, culling away most of the city (except when looking directly down a street or an avenue). However, from a bird's eye there *is* an incredibly amount of visible data, and any acceleration seem rather pessimistic, probably using most of its time in computational overhead with little gain. In this case, level-of-detail may very well be the most fruitful approach.

Therefore, our target worlds shall be only worlds with high self-occludence. Typical examples of such worlds would be building interiors, sewer systems or caverns. As argued by [Tel92] there seems to be a relatively low upper bound on the distance you can see inside any particular building (e.g. measured in the number of visible rooms). This seems intuitively correct. A person inside a building would not expect the general visual *complexity* to increase, because a new floor was added on top of the building, or if a new building was attached on the side of the first one. A 'soft' argument is that hallways connected to a lot of rooms present the hardest case inside a building, but even hallways come to an end, no matter the overall size of the total building complex.

Ultimately, we shall only briefly discuss the integration with acceleration techniques based on level-of-detail. This is not because such techniques can not be extremely efficient or very applicable to our target worlds, but because they are mainly orthogonal and complementary to our approach. Level-of-detail may well be implemented on top of the visibility culling [Fun93].

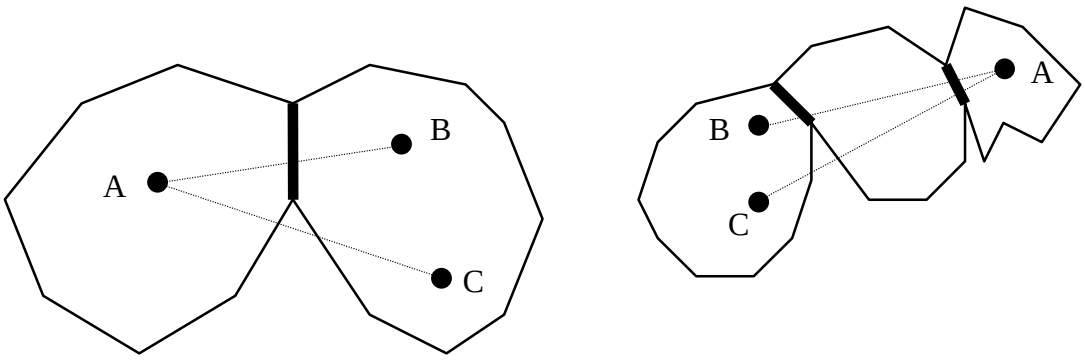
2 Portals

2.1 Basic Definitions

‘Portals’ is a relatively new technique in computer graphics, originally proposed by Jones [Jon71] who suggested constructing a world subdivision based on ‘portals’ and ‘cells’ for hidden line / surface removal. The technique was extended by Airey [Air90] to involve two steps. A pre-processing step where the technique - using a portal / cell subdivision - would construct a ‘Potentially Visible Set’ (PVS) assigned to each cell, and a rendering step where the technique would use the pre-processed information to accelerate rendering. The technique was further refined by Teller [Tel92] by taking an analytic approach and achieving more accurate Potentially Visible Sets.

2.1.1 Definition of Cells and Portals

In the most general form, we define a ‘cell’ to be a connected region of space and a ‘portal’ to be a connection between such two cells through which lines of visibility can ‘travel’. For two points in two distinct cells to be mutually visible, the (otherwise unobstructed) line between them must intersect (or ‘stab’) a portal sequence leading from the one cell to the other.



Points A-B being mutually visible, points A-C mutually invisible
(bold lines indicate portals).

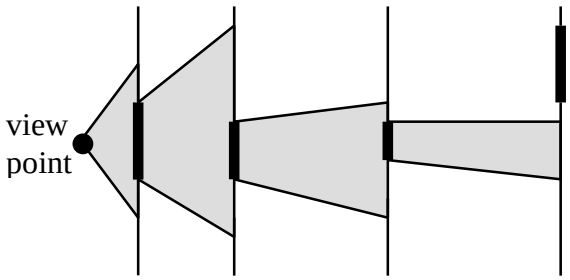
Figure 2.1

Teller uses the notion of a ‘conforming spatial subdivision’ to describe a cell complex of *convex* cells and explicit portals. He further restricts a portal to be a planar, convex region on a shared boundary between two cells and regards any part of a cell boundary not covered by a portal to be an *occluder*. We shall later discuss the use of concave cells and overlapping cells.

An intuitive way of thinking about cells and portals is to regard cells to be ‘rooms’, portals to be ‘doors’ or ‘windows’ and occluders to be ‘walls’, ‘floors’ or ‘ceiling’. In this way the definition is close to how one conceptually perceives a large number of real life structures, e.g. building interiors.

Furthermore, it seems like a logical approach towards visibility determination. An observer placed in a room can only see things outside the room through the ‘openings’ in the rooms boundary (doors and windows). The openings represent an important ‘discontinuity’ in the visibility, because a window will (typically) cover only a small portion of the observers field-of-view and thus restrict the view frustum extending on the far side of the window.

The approach also seems feasible, at least in ‘densely occluded’ environments like e.g. building interiors where occluders are common and portals are sparse. This is because the projected area of a portal falls off with the square of the distance, thus quickly reducing the field of view when traversing a portal sequence.



Narrowing and termination of the view frustum by a sequence of portals (bold lines) and occluders.

Figure 2.2

2.1.2 The Portal Graph

Another interesting aspect of the cell and portal subdivision is that the structure of the subdivision corresponds to a graph, where cells constitute vertices and portals constitute edges. Whereas other hierarchical space subdivision techniques such as Binary Space Partitioning may impose a ‘global’ view on the world (since the top levels of the hierarchy embed the entire world), the graph structure allows a more ‘local’ view. Rendering can start in the observer’s cell and proceed along graph edges (i.e. through portals) only as long as a line of visibility exists, effectively preventing the algorithm to consider distant parts of the world not visible to the observer.

The rendering process (or visibility determination process in general) can be described as a recursive, depth-first traversal of the graph, terminating when visibility is no longer possible through any

portal sequence in the search tree. Given an initial view frustum and a cell containing the observer position, the algorithm renders everything inside the cell and identifies the portals of the cell incident on the view frustum. For each visible portal, the view frustum is restricted using the viewpoint and the (clipped) portal boundary, and the rendering is called recursively for the cell beyond the portal with the new view frustum.

An important notion is that the search tree may visit a vertex more than once (e.g. if two doors lead into the same room or if concave rooms are allowed). However, for planar portals, the search tree cannot recurse infinitely, because a visibility line can not intersect the same portal twice. Furthermore, if a vertex is visited twice (i.e. through different portals), the algorithm will render different parts of the cell on each visit (assuming that the two portals are not overlapping). In this case, the view frustums through two distinct portals originating in the same viewpoint can not be overlapping.

Finally, it is important to notice that the graph traversal provides front-to-back (or back-to-front) rendering of the world by rendering as the graph is recursed (or on return from the recursed path). This makes the technique applicable to Painter's algorithm (if the world is not fully subdivided, the contents of each cell must be sorted individually, but at least the cells are visited front-to-back or back-to-front).

2.1.3 Pre-processing and Dynamic Visibility

The work by Airey and Teller proposes the construction of a PVS for each cell in the subdivision. The idea is to calculate cell-to-cell visibility for a *generalised* observer, i.e. from any position inside a cell. That is, if *any* part of cell B is visible from *some* position inside cell A, cell B must be included in cell A's PVS. At rendering time, given the *actual* observer's cell, the algorithm can quickly pick out the parts of the world that may be visible to the observer and disregard the (ideally much larger) part of the world invisible to the observer.

The PVS is a *superset* of the cells visible to the actual observer (since it includes cells visible from *any* point inside the observer's cell). In the *dynamic step* of the visibility computation, the set is further restricted to reflect only the exact visible set (or a very tight superset, depending on implementation). The dynamic visibility calculation can be compared to the naive restrict-frustum-on-graph-traversal as described in the previous subsection. Teller uses a more analytical approach by using 5-dimensional line space to determine the *existence* of a visibility line stabbing a given portal sequence.

Teller also incorporates the notion of 'detail objects' (in the building analogy, these are typically furniture and minor wall details), for which he includes a special visibility calculation. Detail objects typically have no valuable occlusion properties due to their small size or no feasible occlusion properties due to high complexity.

2.1.4 Subdivision and Graph Construction

[Jon71] focused on the visibility calculations and did not propose any method for cell, occluder and portal identification other than manually subdividing the world into convex cells and portals. Teller proposes automated world subdivision using a hierarchical subdivision technique known to produce convex cells such as k-d trees or BSP trees, combined with a heuristic based on common architectural structures.

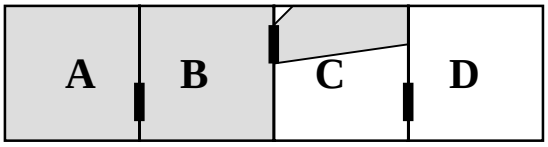
The task of selecting an ideal subdivision, whether manually or automatically, is not trivial. The subdivision can be too coarse or 'unnatural', not providing the optimal hidden surface removal, or it can be too fine, imposing a large computation overhead from basically redundant portal 'clipping'. In addition, the subdivision itself may increase the world size by introducing new polygons through subdivision (e.g. BSP trees). Finally, system performance may dictate *what* the optimal subdivision is, depending on bottlenecks in the renderer's architecture, such as vertex count or polygon size.

This thesis will deal with different kinds of subdivision in different environment types. Portal identification in a conventional architectural model seems relatively straightforward to humans and applicable to simple heuristics. However, in other environment types like e.g. caverns may involve hard-to-identify occluders and portals, as the transitions from e.g. caves to tunnels may be very subtle and the construction of strictly convex rooms become practically impossible.

2.2 Visibility

Given a world divided into a set of cells connected by explicit portals, different types of visibility queries may be posed. Cell-to-cell visibility, point-to-cell visibility and point-to-point visibility are all relevant queries to make in different situations.

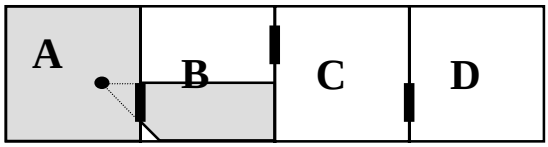
Teller uses an approach where general cell-to-cell visibility is pre-calculated for each pair of cells. This augments a list to each cell, containing the *Potentially Visible Set* (PVS) of that cell, i.e. a list of those cells that can be seen from *at least one* point inside the 'source' cell, denoted a 'general observer' (figure 2.3). In the rendering process, this is used for the first quick world culling. The cell containing the actual observer is located and the part of the world to be considered is limited to the cells in the PVS.



Static visibility information:
Grey area is visible to a *general* observer in A.
Subsequently, PVS(A) is equal to {A,B,C}.

Figure 2.3

Obviously, the PVS for a cell may overshoot the set of actually visible cells from the observer's position. The next step - the *dynamic* step of the visibility computation - is to *refine* the visible set by inferring point-to-cell visibility for the chosen set of cells (figure 2.4). Finally, point-to-point visibility is relevant for a wide range of algorithms relying on a line-of-sight function.



Dynamic visibility determination:
Grey area is visible from the *actual* observer in A, so
C is removed from the set of visible cells.

Figure 2.4

2.2.1 Cell-to-cell Visibility

Under the restriction of cells and portals being convex and portals being incident only with the convex hull of the connecting cells, the cell-to-cell visibility through a given sequence of portals can be greatly simplified [Tel92].

Visibility between two cells is propagated through a sequence of portals, beginning with a portal on the boundary of the one cell and terminating in a portal on the boundary of the other cell. To determine visibility between such two cells it suffices to show the existence of an unobstructed line stabbing all portals in the sequence.

Using the same elegant kind of ‘backwards reckoning’ that is the fundament of ray tracing, the first portal in the sequence can be perceived as an ‘area light’. An area light is a surface that emits light in all directions from its front side. Any surface directly illuminated by this light will be visible to a general observer in the first cell. In this way, the question of cell-to-cell visibility is equivalent to a problem well known to research in e.g. radiosity and computational geometry in general: face-to-face visibility, or the ‘weak visibility’ problem.

2.2.2 The ‘Hourglass’ Data Structure

The set of lines stabbing all portals in a portal sequence has been examined by [Her87] and [Tel92]. The region defined by the complete set of stabbing lines is known as a ‘bowtie’ or ‘hourglass’ due to its characteristic shape (figure 2.5).

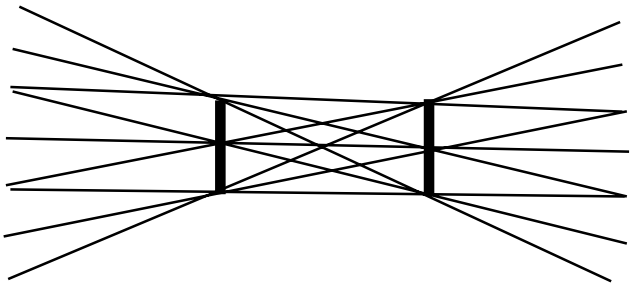
The examination of the hourglass structure is extremely important to this area of visibility determination. As explained previously, the determination of surface-to-surface visibility enables us to predetermine visibility from a general observer in one cell to another cell in the world subdivision. This provides a very efficient tool for choosing a tight subset (the PVS) of the entire world that we need to consider in the rendering step.

Determining portal-to-portal visibility is by no means trivial. At a first glance, it seems sufficient to make e.g. a rough point sampling from points on one portal surface to points on the other portal

surface. However, dividing each portal into a grid of a given resolution, *n* by *n*, yields a total of $O(n^4)$ stabbing line tests. Each test will check the line for incidence with each portal in the portal sequence of length *m*. If the highest number of edges in a single portal is denoted by *e*, the cost of one single surface-to-surface test is $O(n^4me)$.

Furthermore, if one considers a sequence of rooms connected by non-aligned doors, the region where an observer can see a very distant room can be extremely small. This indicates that such point sampling requires a rather high resolution to avoid disturbing rendering artefacts. Potentially, point sampling may always miss important visibility information, creating an incomplete PVS - contradicting the entire idea of the PVS being an *over-shooting* (though ideally tight) estimate of the visible parts of the world.

The exact hourglass region in 3D turns out to be rather complex, bound not only by planes (as one would perhaps intuitively anticipate) but also by quadratic surfaces. However, a tight bound on the exact hourglass can be described by a set of planes, simplifying implementation issues and increasing robustness [Tel92].



A (sub)set of lines stabbing two portals. The ‘middle’ section is enclosed by the convex hull of the two end portals.

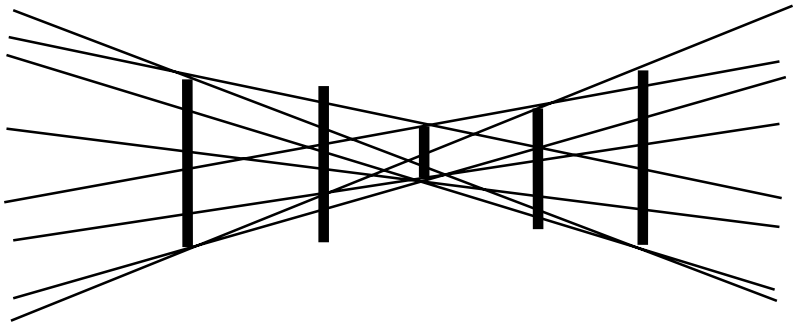
Figure 2.5

In the simplest case with a portal sequence of length 2, the hourglass region *between* the portals A and B is equal to the convex hull of the two portals, CH(A,B).

On the ‘far side’ of each portal, the region has a trumpet-like shape, and it is defined by the extremal stabbing lines or ‘cross-over’ lines. In two dimensions, it will be bound by the upper and lower crossover line. In three dimensions it is bound by the set of extremal planes incident with edges on A and points on B and extremal planes incident with points on B and edges on A.

The hourglass region can also be explained as the *union* of all the view frustums of an observer walking all over the surface of A looking through B, and all the view frustums of an observer walking all over the surface of B looking through A.

When regarding the hourglass structure for portal sequences of lengths greater than two, the structure becomes a bit more complicated.



A (sub)set of lines stabbing a longer portal sequence.
The middle section is now further restricted by mid-portals.

Figure 2.6

The shape of the hourglass at both ends of the portal sequence is still an expanding set of stabbing lines. It can be perceived as the light rays that shine from the surface of A, through the portal sequence and ultimately through B - and vice versa.

In between the end portals, the structure is now smaller than the convex hull of A and B, due to the restrictions imposed by the new mid-portals (figure 2.6). In general there will be a central area where the cross-section (using the general direction A-B) of the hourglass will be minimal, and this cross-section will increase monotonously towards both ends of the portal sequence. In 3D this is of course a rough explanation, since the region is not entirely tubular in shape, making cross-section an ill-defined term, but it serves for an intuitive explanation.

At first, it may seem unclear how to efficiently infer the hourglass structure from a given set of portals. In fact, given a portal sequence $p_1,...,p_n$, the hourglass region for the section between portal p_i and p_{i+1} is bound by the *intersection* of all convex hulls CH (p_a , p_b) where $1 \leq a \leq i$ and $i < b \leq n$, i.e. the convex hulls of any combination of two portals from each side of the section.

However, regarding the way the visibility search trees are traversed, it is natural to view the portal sequences as an incremental data structure. It starts at the source portal (the ‘area light’ polygon) and is increased by adding one portal at a time at the far end of the sequence until visibility through the sequence is no longer possible and the search terminates. Building the sequence step-by-step also makes it easier to maintain certain invariants and rule out data as it loose relevance to the visibility calculation.

Adding the first two portals is straightforward. We assume that the portals are bi-directional (otherwise, we terminate the sequence when a new portal is facing away from the last portal in the existing sequence). Furthermore we assume that neither of the portals are incident with the plane of the other (if this is not the case, each portal should be clipped to the positive half-space of the other). Under these assumptions, the two portals will be fully visible to each other.

The hourglass structure for the first two portals is simply as described in the beginning of this section: In between the portals A and B it is CH(A,B), on each side of the portals it is spanned by the extremal planes through edges on one portal and a point on the other (also known as the *separating* planes). This set of planes can be described as a ‘viewing region’ frustum, i.e. they bound the area

visible from any point within a region, namely the portal on the other end of the sequence. The way to construct this set of planes is for each edge, ***e***, on portal A to find the 'opposite' vertex on portal B. All vertices of B are projected on ***e***×***n***, where ***n*** is A's plane normal, and the extremal vertex in the negative direction is chosen (figure 2.7).

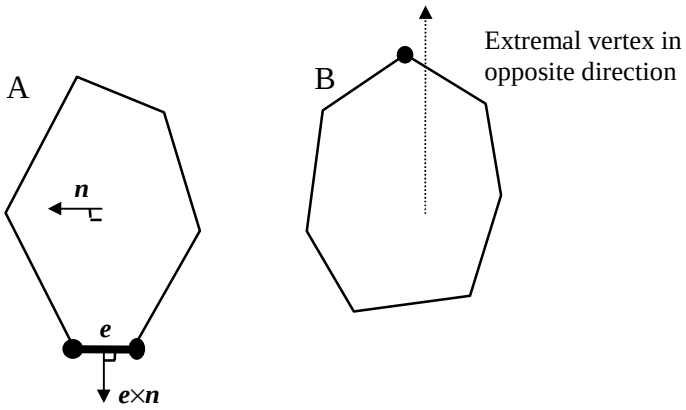


Figure 2.7

Using the two endpoints of ***e*** on A and the opposite vertex on B, a unique separating plane can be constructed. After repeating this process for each edge on A versus opposite vertices on B and likewise for edges on B versus their opposite vertices on A, the full set of separating planes is constructed.

After the initial construction of the hourglass for the first two portals, A and B, adding a new portal B_{new} on the far side of B amounts to a few simple steps, maintaining the hourglass data structure. The first step is to clip B_{new} to the current frustum 'beaming' out of B, since the parts of B_{new} that are not visible from A given the current portal sequence can ever contribute to the visibility calculation. Only the edges of B_{new} that are inside the frustum will be active edges, and the inactive edges can never again go active, since adding portals will monotonously restrict the hourglass shape, 'carving' away parts of it. Consequently, these edges are deleted from the data structure.

The next step is to clip away the parts of A that are not visible from the new portal B_{new}. This is done using the separating planes of B_{new} and the currently active edges of the mid-portals.

Finally the mid-portals are clipped to $CH(A,B_{new})$. No stabbing lines can ever (between the end portals) exceed the convex hull of the end portals, since they have to stab both portals, so any mid-portal edge outside $CH(A,B_{new})$ can be deemed inactive and removed from the structure.

Whenever A or B_{new} is fully clipped away in either of the first two steps, visibility is no longer possible through the sequence, and the sequence is terminated.

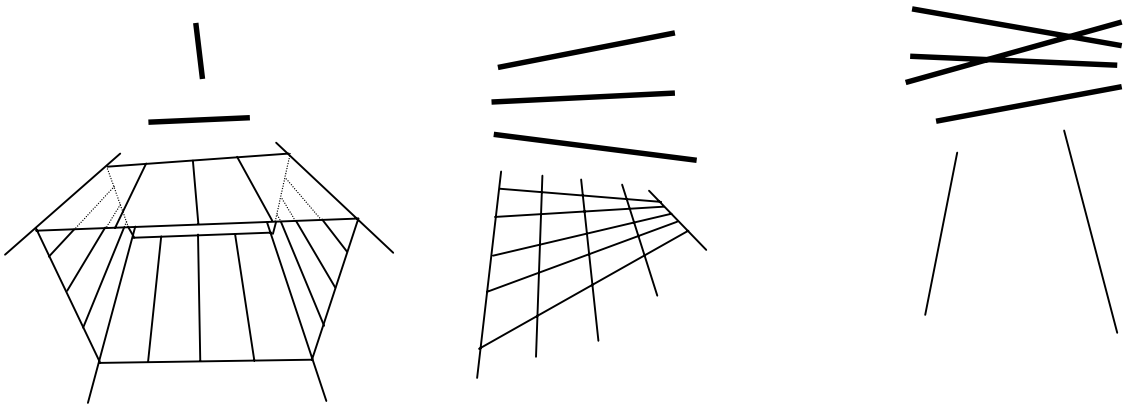
Using that we consider only convex portals, the implementation issues of the incremental hourglass structure can be somewhat simplified by the following observation. Adding an *n*-sided convex portal is equivalent to adding *n* infinitesimally close half-planes one at a time, each one defined by an edge of the portal.

The time complexity of the incremental hourglass algorithm has a high theoretical bound, calculating separating planes against all mid-portals for every new portal added. However, as pointed out by Teller, the number of portal edges is typically bound by a small constant (real-life portals such as doors and windows are typically axial aligned rectangles). The maximal length of a portal sequence is also bound by a rather small constant depending on the nature of the model and the subdivision - in Teller's case around 10. Furthermore, a model-dependent amount of the mid-portal edges will go inactive and not contribute to the calculations. Teller uses a similar approach for his general 3D model and reports of performance levels making the method practically applicable.

2.2.3 Portal stabbing

The previous section explicitly constructs a bound for the set of lines that stabs a sequence of portals, using separating planes. As mentioned, the *exact* bound can not be fully described by planes but it may also include quadratic surfaces.

The bounds of the hourglass volume are formed by lines that, at the same time, are pivoting over two or more edges from different portals in the portal sequence. Therefore, the examination of the hourglass surface amounts to examine the set of stabbing lines incident with two ore more lines in 3D space. For generally oriented portal edges, the lines incident on *two* portal edges span a volume, lines incident with *three* portal edges span a surface, and for *four* generally oriented portal edges exactly two stabbing lines exist (figure 2.8).



The volume spanned by lines incident with two 3D portal edges (bold lines).

The (curved!) surface spanned by lines incident with three portal edges.

The only two lines incident with all four portal edges.

Figure 2.8

Consider three portals in a sequence, each one slightly skewed in respect to its neighbours. The portals are arranged so that a part of the hourglass surface is described by a line 'pivoting' over edges of all three portals at the same time.

In this case, the surface swept out by the set of lines incident on all three portal edges (the *line swath*) will *not* be a plane, but a slightly curved, quadratic surface. In essence, the surface will be slightly 'twisted' around an axis incident with all three portal edges. The bound calculated by the previous method will approximate the curved surface with a set of separating planes, effectively 'over-shooting' the volume containing the set of stabbing lines. However, the curved parts of the hourglass basically amounts to a slight rounding of the sharp edges, and the plane approximation provides a sufficiently tight superset to be used at the application level [Tel92].

Teller takes an analytic approach. Instead of constructing the explicit hourglass structure for a portal sequence to determine if visibility is possible through the sequence, he reduces the problem to *prove the existence* of a line stabbing all the portals.

To treat the portal stabbing in three dimensions analytically, it is convenient to work in a higher-dimensional space and introduce a new co-ordinate system. For this purpose, we use Plücker coordinates and a 5-dimensional geometric object called the Plücker surface, as thoroughly explained by Teller.

A *directed* line can be represented by an *ordered* pair of points $p=\{p_x,p_y,p_z\}$ and $q=\{q_x,q_y,q_z\}$, indicating a direction from p through q. Such a line can be expressed in a 6-tuple, Π, using the Plücker coordinatisation defined by

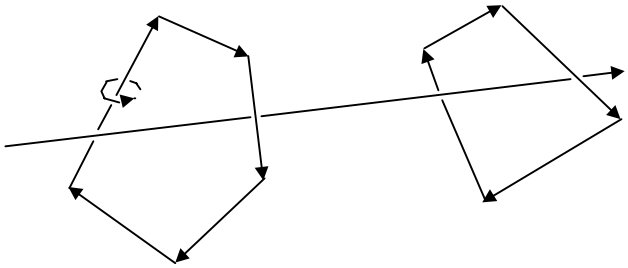
$$\Pi = \{\pi_0,\pi_1,\pi_2,\pi_3,\pi_4,\pi_5\} = \{\ p_xq_y - q_xp_y\ ,\ p_xq_z - q_xp_z\ ,\ p_x - q_x\ ,\ p_yq_z - q_yp_z\ ,\ p_z - q_z\ ,\ q_y - p_y\ \}$$

We now define the permuted inner product of two Plücker coordinates by

$$\Pi_a \bullet \Pi_b = \pi_{a0}\pi_{b4} + \pi_{a1}\pi_{b5} + \pi_{a2}\pi_{b3} + \pi_{a4}\pi_{b0} + \pi_{a5}\pi_{b1} + \pi_{a3}\pi_{b2}$$

The interpretation of this inner product resembles the way the cross product between two vectors is used in 3-space. Considering two *directed* lines a and b, and their Plücker duals Π_a and Π_b, we define the positive circular direction around a using the right-hand rule. Now, the *sign* of the inner product $\Pi_a \bullet \Pi_b$ determines if b 'passes' a in the *positive* or in the *negative* circular direction (if b intersects a, then $\Pi_a \bullet \Pi_b$ is zero).

The practical use of the Plücker representation becomes clear when we regard each convex portal as an ordered set of directed lines, namely the portal edges defined in a clockwise order. For a stabbing line to intersect a portal, we can now require that the inner product of the stabbing line versus each of the portal edges is positive. For a stabbing line to intersect *every* portal in a portal sequence, we simply require that the inner product of the stabbing line versus each portal edge from the entire sequence is positive (figure 2.9).



Portal edges as directed lines. For a line stabbing all portals, the Plücker inner product against each edge must be positive, indicating that the line ‘passes’ each edge on the right side.

Figure 2.9

So in order to determine visibility through a sequence of portals with *n* directed edges, we most show existence of a line, *s*, such that

$$\Pi_s \bullet \Pi_e > 0 \text{ for all edges } e \text{ in the sequence.}$$

Proving existence of such a line is a linear programming problem with a clause for each edge and can be solved in time $O(n^2)$ in the number of portal edges. Furthermore, the experiments conducted by Teller shows that *n* is practically bound by a small constant, in his case typically ‘a few tens’. This is because the number of edges in each portal is typically bound by a small number, and that visibility through portal sequences of length greater than 10 hardly occur. Solving the linear program only involves calculation of inner products and it can be performed robustly.

Teller also shows how to construct the exact hourglass bound, using Plücker coordinates and 5D line space. For this purpose he uses the Plücker surface, a geometric object defined by exactly those Plücker coordinates for which $\Pi \bullet \Pi = 0$. These points corresponds to real lines in 3-space (the points *not* incident with this surface, i.e. for which the inner product with themselves is not 0, have imaginary composants in 3-space).

So to explicitly construct the exact hourglass, Teller constructs a 5D polytope from the hyperplanes corresponding to the Plücker coordinates of the portal edges. This polytope is intersected with the Plücker surface, and the point set of this intersection represents the 3D planes and quadratic surfaces that form the hourglass structure.

Not surprisingly, this calculation of the exact hourglass structure is difficult to implement robustly, so for calculating the hourglass volume Teller also uses the plane approximation. Though the *visibility question* can be determined quickly and robustly using Plücker coordinates and linear programming, the explicit construction of the extended frustum of the hourglass is still desirable. It is convenient to have a well-defined volume to check intersection with, e.g. when classifying visibility of detail objects.

2.2.4 Point-to-cell and Point-to-point Visibility

The hourglass structure and the portal stabbing described in the previous sections are powerful tools for determining surface-to-surface and ultimately cell-to-cell visibility in a portal graph. Cell-to-cell visibility allows us to pre-calculate the set of cells that *may* be visible to an observer inside a given cell.

At run-time, we know the exact position of the observer for a given time frame, so we wish to further restrict the set of cells to those actually visible to the observer. In this way, we ensure that only relevant cells are submitted to the last stage of the rendering pipeline, where the final removal of hidden surfaces will occur, typically by means of a depth-buffer.

A first, naive approach to point-to-cell visibility could be to clip the PVS to the current view frustum. Though removing cells outside the visible region, this approach does not ensure that we remove cells *inside* the view frustum that are occluded by other polygons inside the view frustum. Instead, we need to ensure that visibility from the observer to a cell is possible *through* a portal sequence. Like in the cell-to-cell visibility determination we can take two approaches, the analytic approach that Teller uses or a more geometrical, step-by-step solution.

The analytic approach follows the same framework established for cell-to-cell visibility.
Consider the portal sequence leading from the observer cell to the cell for which to determine visibility. In order to prove visibility from a point to a cell, we must determine *existence* of a line through the observer point and stabbing each portal in this sequence. Again, this means that the line must pass on the ‘inside’ of each portal edge. This can be determined by the permuted inner product on Plücker coordinates - and we get an additional restriction on the solution, namely that the line passes through the observer point. Teller solves this problem efficiently using linear programming in quadratic time in the number of portal edges.

Alternatively, the point-to-cell visibility can be determined by traversing the portal sequence while maintaining the set of planes enclosing the region visible from the observer position. Initially the set of planes is simply the planes of the observers view frustum. For each new portal in the sequence, the view frustum is further restricted by the edges of that portal. Indeed, the border of the screen or the viewport can be regarded as a dynamically moving portal that initially restricts the view of the observer.

First the portal is clipped to the view frustum; if the portal is completely outside the view frustum, visibility is not possible through the portal sequence and the traversal is terminated. Otherwise, the portal edges inside the view frustum, together with the observer position, define a new set of planes restricting the view frustum.

This visibility determination can be determined for each cell, one at a time, or it can more efficiently be embedded in a recursive graph traversal determining visibility for all the considered cells. In this way, the view frustums calculated for the shared part of two portal sequences leading to two different cells can be re-used. In fact, one can choose to do the rendering in the same step, so the point-to-cell visibility is implicitly determined in the actual rendering.

Finally, point-to-point visibility can also be determined analytically by using Plücker coordinates or explicitly by graph traversal. In the first case, we need not prove the *existence* of the line (since the line is now given) but rather test if the line passes on the 'inside' of each portal edge. This is done by examining the permuted inner product of the line and the portal edge.

The graph traversal approach amounts to intersect the line connecting the points with the boundary of the first cell in the sequence (i.e. the cell containing one of the points). If the intersection happens at a portal, we proceed recursively for the cell attached to the other side of that portal, otherwise the search is terminated. When we reach the cell of the other endpoint, we have successfully determined visibility between the two points. The advantage of the latter approach is that we can determine exactly *where* the line-of-visibility intersects the world model. This can be useful for a number of applications, for example determining 3D intersection of a 2D mouse over the viewport or for calculating trajectories inside the world model.

2.3 Acceleration Techniques

Considering the portal framework, a number of more or less model-specific acceleration techniques can be exploited.

2.3.1 Detail Objects

In [Tel92] and [Fun93], the primary test world is (extracts of) an architectural model of a planned building at the Berkeley University dubbed the 'Soda Hall'. The number of polygons describing the building itself amounts to around 30.000. The model is further populated with a set of highly detailed objects such as furniture, plants, lamps, pencils etc. Each of these objects has polygon counts of several hundreds up to several thousands, totalling a polygon count for the entire populated building of around 1,5 million.

In this case, it is natural to make a distinction between polygons describing the building itself and these detail objects. First of all the objects are easily identified as they are *explicitly* described in the model, and furthermore their nature differ from that of the building components (walls etc.) both in the amount of detail and in the obscuration properties. Typically, a plant composed of thousands of polygons would add an incredible amount of computational overhead to the visibility computation without providing any useful visibility restrictions at all. Therefore, these objects are treated separately in the frameworks of Teller and Funkhouser. They are embedded in bounding volumes and their visibility is pre-calculated and stored similar to the visibility of cells, but they do not contribute to the visibility calculation themselves.

Of course, this kind of acceleration is highly model-dependent. It may be very useful for most architectural models, but not all models may come with an explicit distinction between building components and detail objects. Furthermore, not all models may provide such a clear difference between what should be considered obscuring elements and what should be detail objects. The example with the plant is quite obvious, but large, bulky detail objects like e.g. a desk or a refrigerator may actually provide valuable visibility information that could accelerate the rendering process.

2.3.2 Level of Detail

Another important acceleration technique for rendering complex scenes is using ‘level-of-detail’ (LOD). This technique simplifies the representation of parts of the world based on e.g. the distance from the observer. The simplification is typically done by reducing the polygon count for an object when it is too distant (and thus small in screen space projection) to justify a high complexity representation.

We shall only deal superficially with LOD techniques (as mentioned in section 1.3). It is important to notice that the graph traversal (and the rendering process itself) is handled front-to-back. This ensures that anything rendered in the recursion beyond a given portal is at least that far from the observer as the distance of the portal. Therefore, portals are good ‘decision points’ for selecting an appropriate level-of-detail for a sub-tree of the graph traversal. Furthermore, one could introduce a ‘hard floor’ for allowed rendering detail by choosing to *terminate* graph traversal beyond portals smaller than a certain projected area in screen pixels.

[Fun93] deals extensively with LOD techniques placed in the same portal framework and test base as that of [Tel92]. In that world model, the detail objects are the main contributors of visual complexity, and consequently they are chosen to be the subject of the LOD simplification. The framework creates instances of each detail model at several detail levels. Appropriate detail models are chosen per-object based on criteria such as distance from observer, used rendering time and available rendering time. This method runs fairly independent of the actual cell culling as performed by the portal framework.

Generally speaking, the class of target worlds where portal rendering - or indeed *any* visibility deciding algorithm - falls short, are worlds with low degrees of self-occludence and very high and irregular visual complexity overall. In a world where the number of visual elements present to the observer are far exceeding the potential of the graphics pipeline, the only feasible approach is to *reduce* the visual complexity, i.e. to deploy LOD. Consequently, any truly general rendering acceleration framework should include both occlusion culling (such as portals) as well as LOD simplification. This could be implemented either as complementary methods (such as [Fun93]) or as integrated methods, where the structure of the portal graph itself can be dynamically simplified based on e.g. distance from the observer.

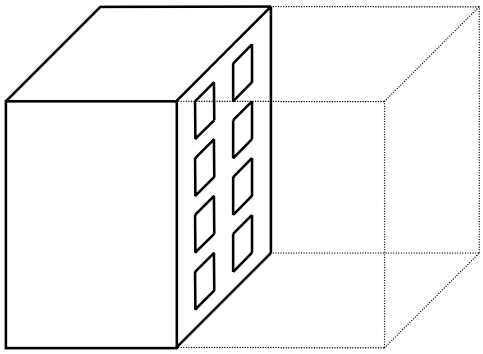
2.3.3 Portal Hierarchies

Teller addresses the problem of cell boundaries with concave portals split into many convex portals or generally complicated portal sets on a single boundary face. Clearly, at some point it will be more feasible to ‘collapse’ a number of portals into one large portal, using their convex hull, when their computational overhead can not be justified by time saved in the rendering process. Criteria for aggregating coplanar portals could be to compare their summed area to their aggregated area or to consider the complexity of the world beyond the set of portals. The portal splitting may not even be worthwhile, if e.g. the entire set of portals only leads into a single room of little complexity.

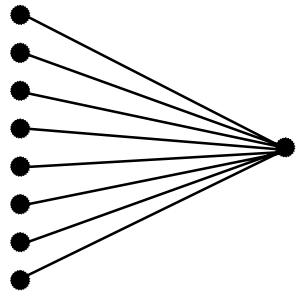
Such aggregation could remove ‘redundant’ portal complexity, but it does not address the problem of very complex sets of coplanar portals leading to *separate* parts of the world. In this section, we suggest a method to handle this problem without complicating the framework of the portal graph.

Though portal rendering sounds like an ‘inside rooms’ rendering technique only, the concept beautifully handles several layers of ‘inside/outside’ conventions in a single world. For example, one could at the same time be *inside* a large dome (e.g. representing the sky), looking at the *outside* of a building (e.g. a high rise), seeing the *inside* of the building through the building windows, possibly looking further beyond doors into other rooms inside the building. In this set-up, the dome will be one graph node and each window in the building a graph edge (a portal) pointing to room nodes inside the building. The dome walls and the *outside* walls of the building will belong to the dome node, whereas the *inside* walls of the building will belong to the node of the room they enclose.

Even if fitting nicely into the portal concept, the world described above calls for one obvious optimisation. If we consider the high rise to be a building with a roof and 4 planar walls, each wall could contain potentially hundreds of windows leading to separate rooms. In the computational framework, this would introduce a large amount of portal handling (figure 2.10). Considering that only two of the building walls can be visible at any time, all portal handling performed for the ‘backsides’ of the building is redundant. Therefore, it would be convenient to introduce a quick test for each wall to test if the wall itself is visible - if not, none of the portals embedded in the wall can be visible. Such a test could be expanded to a hierarchy over the wall by first checking if the entire wall may be visible, then proceeding to sub-sections of the wall, arranged in e.g. a 2D BSP tree.



A building with 8 separate rooms, connected through 8 windows (portals) to the region outside the building.

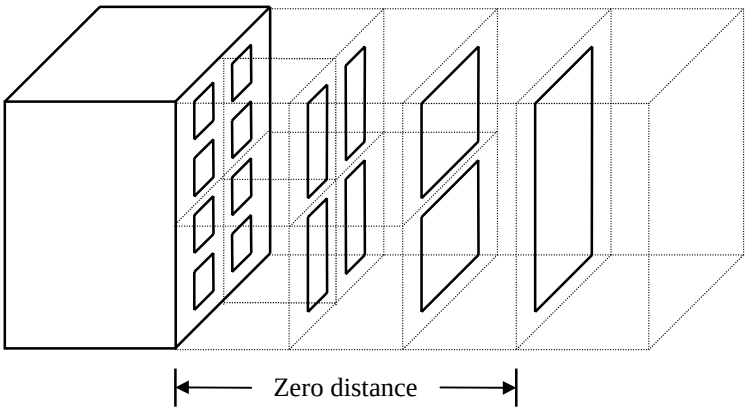


The corresponding portal graph. An observer outside the building must deal with all 8 portals, even if he can see none or only part of the building.

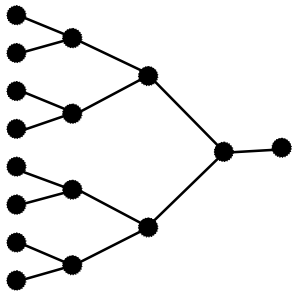
Figure 2.10

Rather than adding new complexity to the data structure of the portal graph, we suggest introducing new ‘zero-volume’ nodes into the portal graph. In this case, such a node would represent an infinitesimally thin ‘film’ over the wall. On the side of this new ‘volume’ facing the building there will be a portal for each window. On the side facing away from the building there will be one single portal, aggregating the area of all the windows on that side of the building. By adding *multiple* layers of

'film' over the wall, a full hierarchy can be constructed, subdividing the wall into appropriate regions (figure 2.11).



Adding a zero-volume hierarchy of portals for efficient early-rejection of invisible sections of the wall.

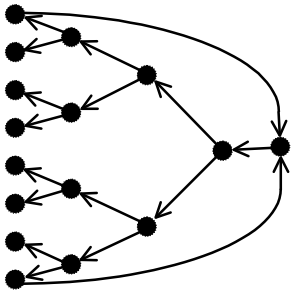


The corresponding portal graph with the newly introduced nodes.

Figure 2.11

Now, when the graph traversal reaches the building, it will first meet the big portals of each of the newly introduced nodes. Only for those two portals visible to the observer, the graph traversal will proceed to the next nodes and ultimately to the individual windows of the building. Of course adding a hierarchy introduces a computational overhead, so it is important to apply such added complexity only where an overall rendering acceleration can be expected. This is similar to the balancing needed when introducing e.g. bounding volume hierarchies in ray tracing [ArKi89].

One should notice, that if the observer is placed *inside* the building, this newly added portal(s) will introduce completely redundant checks, slowing the rendering process. However, if the portal graph is constructed with *directed* edges, this redundancy can be eliminated. The edges representing the windows when seen from the inside can simply *bypass* the newly introduced node(s) and 'look' directly into the node of the region outside the building (figure 2.12).



Bypassing acceleration hierarchy with directed graph edges.
(Note: Shows only bypassing edges from 2 of the 8 rooms.)

Figure 2.12

2.3.4 Dynamic Portal Simplification

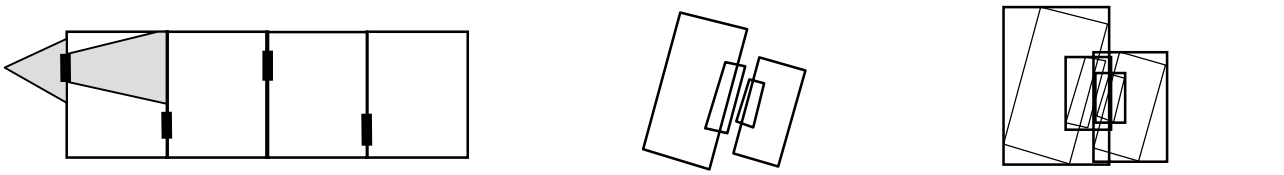
As mentioned in section 2.2.4 the edges of a convex portal together with the observer position can be used to construct and modify a set of planes defining the observers view frustum.

Depending on the chosen or available implementation of the rendering pipeline it may be desirable to represent the view frustum by clipping planes that are axially aligned in camera space. The camera space can be normalised so that these planes are defined as $x=z$, $x=-z$, $y=z$ and $y=-z$, a *canonical view volume*, allowing for efficient clipping and culling [Fol90]. Furthermore, it may be desirable to bound the number of planes defining the view frustum to a constant number in order to achieve simpler and more efficient implementations.

[Lue95] suggests for each portal to use its bounding rectangle in screen space for defining the view frustum. In this way, the clipping of subsequent portals can be done very efficiently, since the current view frustum is simply represented by a viewport with x_{min}, x_{max}, y_{min} and y_{max} values. The simplicity comes at a price, since the visibility calculations are done in screen space, i.e. *after* projection. Using generally oriented frustum planes; we may cull an object (using a slightly more expensive dot product) *before* projection is performed.

This approach seems quite feasible for models with rectangular portals (typically doors in an architectural model) and *especially* for walk-through situations where the camera is *not* tilted. In this case, the bounding rectangles in screen space will provide a sufficiently tight bound on the actual portal perimeter.

However, in scenes where camera tilting is allowed and common (as should be assumed in any, true 6-degree-of-freedom framework) and in scenes where portal rectangularity is not guaranteed, this approach can produce a substantial rendering overhead. An example of this could be a sequence of rooms with non-aligned doors. In a situation where visibility is only possible through the first door, tilting of the camera could result in the entire sequence of rooms being rendered. This would happen because the screen aligned bounding rectangles of the portals would now be overlapping, although the actual portals are not (figure 2.13).



Room sequence with portals (doors) and *actual* view frustum

Doors projected to screen space with tilted camera angle.

Bounding rectangles for doors, all overlapping.

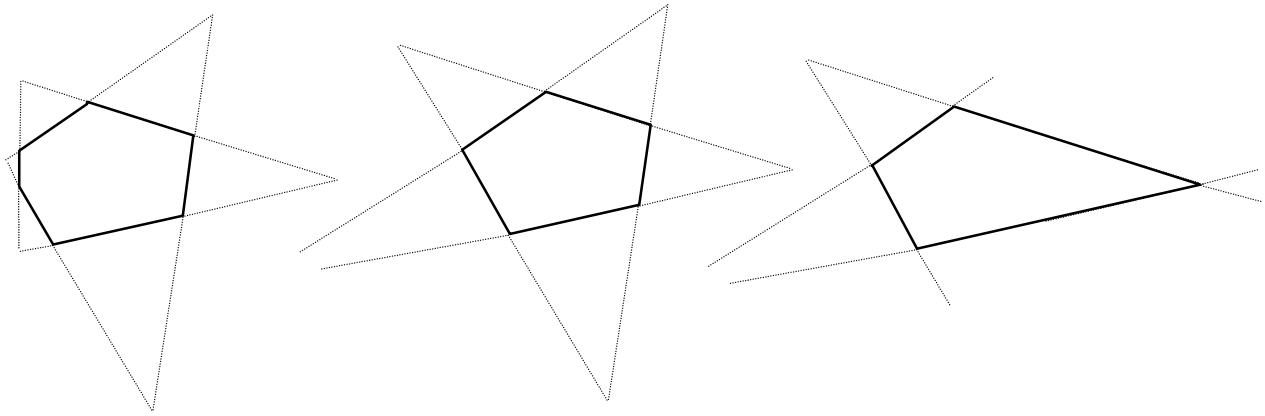
Figure 2.13

To avoid this problem while retaining the simplicity and efficiency, we propose the following method. To achieve as early termination of a portal sequence as possible, the actual projection of the portal in screen space (or equivalently the view frustum defined by its edges) should be maintained.

If desired, one could still choose to construct a screen aligned view frustum for clipping the world model's occluder polygons (and possible detail objects), using the bounding rectangle of each portal.

However, clipping the actual portal perimeters to each other ensures that the view frustum is not needlessly expanded. The worst case is represented by a very narrow portal rotated 45 degrees relative to the camera, where the area of the bounding rectangle can be arbitrarily larger than that of the portal itself. If the perimeter of the last traversed portal is stored as a convex polygon in screen space, the clipping of the next (convex) portal in the sequence simply amounts to finding the intersection of two convex polygons (a convex polygon itself).

Addressing the issue of a low, constant number of clipping planes / viewport edges, we suggest finding a tighter bound on the portal perimeter than that of the screen aligned bounding rectangle. In this way, we combine the best of both worlds - clipping is performed with a low number of clipping planes, and a more accurate bound on visibility through portal sequences is maintained. In order to achieve this, we introduce a method to reduce the number of edges in a convex polygon by edge removal.



Reducing a 6-gon to a 4-gon by successively removing edges resulting in the smallest area expansion.

Figure 2.14

For each edge in a polygon, we calculate the area that the polygon will be expanded with, if that edge is removed and the adjacent edges are prolonged (figure 2.14). Note that this area may be infinite, if the adjacent edges are pointing away from each other. We then remove the edge causing the smallest expansion and update the area calculations for the adjacent edges. We proceed until we reach the desired number of edges (or the smallest expansion is unacceptable). Note that the process can not proceed when we reach either a parallelogram or a triangle, because all subsequent expansions will be infinite. The cost of each edge removal is *O(1)*, but since we need a sorting of edges based on the size of the expanded area, each step adds a logarithmic factor, yielding a *O(n log n)* total cost of reducing an n-gon.

In general for polygons reduced to 4-gons, this method will achieve tighter bounds than the axis aligned bounding box, since each edge of the resulting polygon will be incident with an edge of the original polygon. For the axis aligned bounding box, this may not be the case. Furthermore, this

method has no preference in directions, so camera tilting will no longer be an issue. Finally, it seems that picking a slightly higher bound than 4-gons, e.g. octagons, should be able to approximate *any* convex polygon without considerable error. The ‘ideal’ convex polygon has the shape of a permuted circle and it can be closely fitted by an octagon. Here, ‘closely’ is placed in the context of visibility determination, where we merely need conservative estimates of the visibility in a scene and early termination of portal traversal is more important than accurate view frustums for object clipping.

A final acceleration step that deserves to be mentioned here could be for each portal in the world model to store the centre and radius of the smallest enclosing sphere. When working with generally oriented clipping planes defining our view frustum this provides an efficient way of culling portals outside the frustum. We simply take the dot product of each plane normal and the direction vector to the portal centre and compare it to the radius of the bounding sphere. This provides for a first, quick rejection of invisible portals, and it could even be expanded to bounding sphere hierarchies for clusters of portals.

2.4 World Construction

The previous sections have focused on visibility operations over a graph describing the cell and portal structure of a world model. We have yet to address the construction of this graph. We categorise methods for subdividing an input model and constructing the portal graph in three categories:

- Constructive Models,
- De-constructive Models and
- Re-constructive Models

Constructive models are models that are constructed with the purpose of fitting into a portal framework. Alternatively, it can be a model for which data about the construction process is available, making it easy to infer a cell subdivision and a portal graph for the model. An example of such a model could be an architectural model, where the modeller is already working in concepts of rooms and doors, making the cell and portal information simply an augmentation to the primitives that the modeller uses. Another example of a constructive model will be discussed in chapter 3, where an interactive world is constructed from a set of ‘brick’-like cells with explicit portal information.

De-constructive models describe models for which no portal information - or structural information in general - exist, and for which cell subdivisions must first be created before building a portal graph connecting the cells. Examples of such models could be unstructured CAD models (a so-called ‘polygon soup’) of e.g. buildings or caverns or polygonised scientific or medical data sets (remembering that we restrict ourselves to models with a high degree of self-occludence. Such models require good heuristics for identifying good cell subdivisions and portal connections, and this process can be carried out manually or it can be automated.

Finally, by re-constructive models we describe models *based* on a highly detailed cell subdivision (obtained using either constructive or de-constructive methods). In this approach, neighbouring cells may be collapsed into larger cells in areas where the subdivision seems too fine-grained or too ‘un-

natural' to achieve an optimal rendering acceleration. Also, cell boundaries and portal positions may be re-adjusted (or 'fitted') to obtain a theoretically better subdivision of the model. In this way, reconstructive models can be seen as a structural optimisation technique, that require extremely well-designed heuristics in order to obtain generally better results. In chapter 4 we shall further discuss such techniques.

2.4.1 World Representation

Without going into too much detail about the data structures used to represent a world model, we define a set of desirable properties for an 'ideal' world model. These properties are presumed in most of the algorithms we describe and develop in this thesis. For practical application the input meshes should either be modified to fulfil these properties, or the algorithms should be modified (augmented with special cases) in order to be more tolerant to 'flaws' in the input data.

- First, we assume that polygons are planar and convex.
- A polygon should contain a reference to a sequence of connected edges, arranged in a clockwise order when seen from the polygons 'front' face.
- The 'back' face of polygons are assumed to be invisible, so two-sided polygons can only be represented by double representation of polygons.
- Duplicate vertices and duplicate edges should not exist, because this makes it difficult to deduct connectivity information about the polygon mesh.
- Surface 'cracks' (usually introduced by numeric errors) should be eliminated. Such cracks often appear at so-called 'T-junctions' and they can be eliminated by inserting an extra vertex.

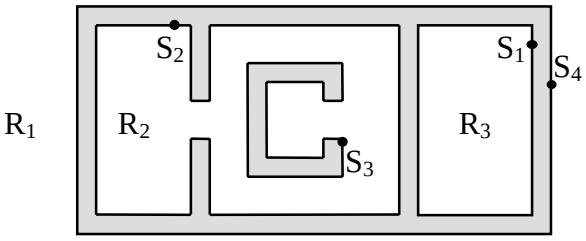
This set of requirements is fairly common in graphics applications and can easily be verified or enforced. Non-planar polygons can be split into planar ones, concave polygons can be split into convex ones, duplicate or very close vertices can be collapsed etc. Such 'mesh cleaning' can typically be obtained using standard software packages or 'filters'.

We introduce an additional set of requirements that is more designed towards our specific application. The basis for these requirements is that we wish to work with *solid* models, defined as models in which no zero-volume structures exist.

More specifically, such models can be described using the analogy of e.g. a house model. Every wall, floor and ceiling has a thickness, and the material *inside* the walls is indifferent to our visualisation and considered *solid*. Volumes that are not solid, we regard as *rooms*, i.e. spaces where an observer can be placed. No polygon may exist inside a solid structure, or in other words, every polygon in the model must be visible from a room. If these conditions are met, all solid structures will be enclosed by a closed, connected polygonal surface with all polygons facing outwards (figure 2.15). Formally, the requirements amounts to:

- No 'gaps' are allowed, i.e. each edge must be shared by exactly two polygons facing in the same direction (when traversing each polygon in clockwise direction, the shared edge should be traversed in different directions).

- No duplicate polygons are allowed, i.e. two coplanar polygons may not overlap.
- For each volume delimited by one ore more polygonal surfaces, each polygon must face either towards or away from the volume. If they face towards the volume, the volume is a *room*, otherwise it is *solid*.



A model with rooms R_1, R_2 and R_3 and surfaces S_1, S_2, S_3 and S_4. Grey areas are solid.

Figure 2.15

In the real world, virtually all of these assumptions are likely to be violated in 'raw' input models. Most models will be created using methods that do not ensure these properties (for example, building a house model by assembling e.g. a set of wall primitives will usually leave a lot of polygons inside or between the solid walls). Even in a creation processes that aim to preserve these properties, the model may be prone to numerical round-off errors (for example, exactly *how* close should two polygons be before they are coplanar and overlapping?) or human errors (e.g. gaps caused by the modeller leaving out a polygon).

Teller uses a combination of automated and manual 'model cleaning' to create close-to-ideal models and furthermore he makes the subdivision algorithms tolerant to certain flaws in the model. Ideally, the algorithms should be designed to be very tolerant in order to ensure robustness. However, this is an art of its own and we shall not go into further detail with this, other than mention a few instances where it has particular relevance to the described algorithms.

2.4.2 De-constructive Models

When facing the task of finding structure - more specifically cell subdivision and portal graph - in an unstructured input model, a number of different approaches can be taken.

The most basic approach is to *manually* identify both cells and portals. This approach was originally taken by [Jon71], using the rather strict requirement that the entire model should be subdivided into a set of empty, convex volumes and that every polygon in the scene should be incident with a cell boundary. [Lue95] uses a more relaxed manual subdivision, where cells need not be empty or convex, basically amounting to identifying conventional rooms and doors in a house model.

A fully manual process has the great advantage of being very tolerant to input errors such as surface cracks etc. Furthermore, human intuition should be excellent for identifying good portal positions,

basically because it is a natural process to identify room divisions in a model. The real drawback of manual subdivision and portal construction is the workload imposed by really large models. Since one of the goals of this thesis is to explore scaleable methods, i.e. methods expandable to extremely large models, the manual approach does not seem like a solution to pursue.

The next step towards automation would be to manually identify good portal positions and let the cell subdivision proceed automatically. Assume that the model meets our requirements posed in section 2.4.1. Then, it is a trivial task to trace boundaries along surfaces between the portals (for example by colouring a ‘seed’ polygon and proceeding recursively to its uncoloured neighbours, until the portal boundary is met). This process would automate a large deal of the work, while keeping the advantage of the human intuition. However, the process will be less tolerant to errors (since mesh surfaces must be robust) and the manual portal identification is still too cumbersome for e.g. large, architectural models.

Therefore, in order to handle very large models, the only approach seems to be a fully automated cell subdivision and portal graph construction. Obviously, this will be the least error tolerant approach, and furthermore it must be based solely on a heuristic for ‘good’ cell subdivision rather than human intuition. Teller takes this approach, using a simple heuristic for BSP-tree subdivision based on common structures in architectural models.

2.4.3 Spatial Subdivision

When subdividing the input world, the only restrictions we put on the subdivision method is that the subdivided model should meet our specific definition of cells and portals. In the end this boils down to a number of choices:

- Should the subdivision be complete, i.e. all cells being empty and all polygons placed on cell boundaries, or will we accept larger ‘rooms’ with some polygon contents?
- Should cells be convex or are concave cells acceptable for our solution?
- Should the subdivision comply with specific criteria for a ‘natural’ or in other way ideal subdivision? (E.g. [Tel92] performs additional subdivision to achieve cell aspect ratios close to 1.)

Most hierarchical space partitioning techniques can be used to create a subdivision applicable to the portal framework. In architectural models where a large amount of axially aligned faces can be expected, a technique like k-d trees [Ben75] can be used, deploying splitting planes orthogonal on - in turn - the three major axes of the co-ordinate system.

A more general subdivision technique is Binary Space Partitioning trees (BSP trees), where splitting planes are chosen incident on existing polygons in the model. This yields a subdivision somewhat closer to the original structure of the input mesh, though especially early splitting planes tend to intersect and split existing model polygons, increasing the overall size of the model.

This hierarchical technique is able to fully subdivide the model into empty, convex cells with polygons only on cell boundaries. This makes it easy to classify cells as ‘rooms’ or ‘solid’ according to our world representation, simply by examining the orientation of the polygons on the boundary of

each cell. In fact, such a hierarchical subdivision is probably always worthwhile to include in the world description. It provides an effective (logarithmic) search method for the point-location problem, where the rendering algorithm need to determine in which cell the observer (or any other object) is located.

However, the drawback of the hierarchical solutions is that they violate our goal of 'localness' in the world model. The primary splitting planes will in general intersect the entire world, even when only chosen to fit polygons within a local region, providing an unnatural bisection of world sections distant from this region. Furthermore, it greatly complicates dynamic updating to the world structure, as the updates may affect top-level splitting planes in the tree and basically require a reconstruction of the entire tree.

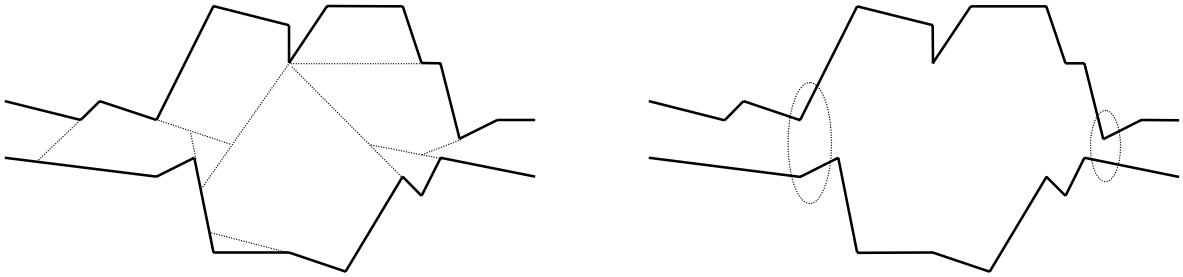
A more 'local' approach could be to use e.g. Delaunay triangulation, splitting the world into convex cells, based on 'local neighbours' as defined by the world model's Voronoi diagram. Intuitively, this seems like a better solution, as the identification of 'rooms' in a model is based on local structure rather than global properties of the model. However, a Delaunay triangulation is not guaranteed to follow the existing 'walls' (polygons) of the input model, so this method may also introduce splitting and thus increase the overall size of the world model.

So, ideally a world subdivision should show the same local behaviour as the Delaunay triangulation but preserve the existing walls as defined by the model for cell boundaries wherever possible.

After subdividing the world into cells, the next question is the placement of the portals. For a pair of adjacent cells, these will be placed on the boundary face shared by the two cells. The first observation is that the boundary can either be fully covered by polygons from the input mesh (fully occluded) or it can be partly or fully 'transparent', i.e. parts of the boundary may *not* be covered by polygons.

This observation can be directly expressed in a graph with cells as vertices, and edges between those adjacent cells for which a shared, transparent cell boundary exist. This graph can be directly adopted to be the portal graph by defining the transparent parts of the cell boundaries to be portals, which is in fact the method used by [Tel92]. In Teller's analytic framework, cells are required to be convex, because cell-to-cell visibility can then be reduced to a portal-to-portal visibility problem. This is because a portal on the boundary of a convex cell will be visible to an observer *anywhere* inside that cell, i.e. a general observer. However, Teller recognises the large overhead introduced especially in 'free space' regions, where numerous (superfluous) cell divisions and portal definitions may emerge in areas where no occludence is present. For this purpose, he introduces *meta-cells*, collections of convex cells, behaving as large, possibly concave cells with only the 'exterior' portals of the cell complex visible to the 'surrounding' world.

This can be seen as an indication of the convex cell requirement being more of a technical requirement than one supporting the cell and portal framework in general. Indeed, more irregular models than Teller's architectural models subdivided with e.g. a BSP tree into convex cells tend to produce a large overhead of subdivision. For example, a cavernous space with ragged walls would produce a multitude of convex cells, even if the 'natural' subdivision would merely be concentrated around a few tunnel entries (figure 2.16).



A cavern-like room completely subdivided into convex cells, creating a large overhead of portals. In 3D, the situation is even worse.

More ‘natural’ regions for inserting portals to achieve a sound and less complicated subdivision.

Figure 2.16

Summarising, the question of an ideal world subdivision may be very model dependant. For architectural models, a BSP tree may seem adequate - though architecture, especially the more exiting kind, does not necessarily conform to axially aligned, right-angled or planar structures. Indeed, it may not be the case that a good portal location is necessarily incident with a boundary of the obtained cell subdivision. In chapter 4 we shall further discuss more generally applicable methods to obtain good subdivisions suited for portal rendering.

3 Applications

Portal rendering is a technique that has quickly gained wide popularity in consumer class applications, more specifically computer games. It has successfully been used in a number of applications and a couple of lessons for the practical implementation of portals can be drawn.

The technique's adoption by the broad entertainment industry can be found in a couple of reasons:

- Much of the workload can be moved to a pre-processing phase, lowering the requirements on the real-time 'performing' equipment, and even without heavy pre-processing the real-time part is very efficient.
- The technique is very scaleable and involves relatively simple data structures and calculations, making it suitable for low-end hardware and not only expensive workstations
- The PC market now embraces areas previously only accessible by workstations, with the late development in pricing and performance of dedicated graphics hardware.

We shall briefly review the background for the application area where portal rendering has been deployed, before moving on to comparing a couple of currently used techniques and discussing more specific experiences and implementation issues from an actual implementation case.

Finally, we discuss a number of possible extensions of the portal framework.

3.1 The Entertainment Industry

At first, this section may seem to drift off into a retrospective and indeed nostalgic review of the evolution of computer games, based on the author's inside experience with the business. Nevertheless, it outlines a series of events, mainly triggered by one single company, that leads directly to the core subject of this thesis.

Hopefully, it also provides an interesting insight into some of the efforts that creative minds had to go through before dedicated graphics hardware appeared on the mass-market scene. It describes a paradigm shift over a few couple of years in which a major industry moved from highly specialised, hardware dictated techniques into using very general, unrestricted techniques, previously only available in work station environments, including dedicated high-performance graphics hardware using OpenGL and similar interfaces to 3D geometry and rasterisation.

3.1.1 Research and the Industry

Most scientific research areas have a ‘real world’ counterpart, an application area where the results of the research is being utilised. Both worlds benefit from this relationship, since experience and demands from the application community goes back into the research community to trigger new paths of research. Medical research e.g. has always been closely linked with hospitals and the medical industry.

Computer graphics is a relatively new area of research by this measure. For this research area, the application areas were initially characterised by a limited number of highly specialised areas, e.g. military flight simulators and medical visualisation, mostly due to the high computational level and thus the demand of resources.

However, the application area for computer graphics has broadened a lot by the introduction of synthesised imagery in the entertainment business, first in the financially strong movie and television industries, but also lately - over the last decade - in computer games. The *increase* in computer performance is popularly estimated to be around 100% every 18 months or so. Over the last couple of years the alarming rate of 300% every 12 months for dedicated graphics hardware has been reported in the average consumer market, namely graphics card for personal computers.

Since computer graphics is no longer limited to highly specialised areas with very strict demands for the application, the research has become more ‘free’ in the sense that the results can be interesting in their own right. If a presentation of - or interaction with - computer graphics is interesting, or even amusing to a substantially large group of people, it becomes per definition entertainment and thus gains the interest of the entertainment industry.

With a total number of released computer game titles exceeding several thousands per year (and a comparable number of development groups producing these games) the struggle to produce the best performing piece of software is immense. Rightful or not, one of the most important measures of quality for a good computer game these days seems to be the quality of the graphics. This is possibly because the number of games is so massive, that games are quickly evaluated on their first appearance - primarily graphics and sound - rather than more subtle elements such as gameplay.

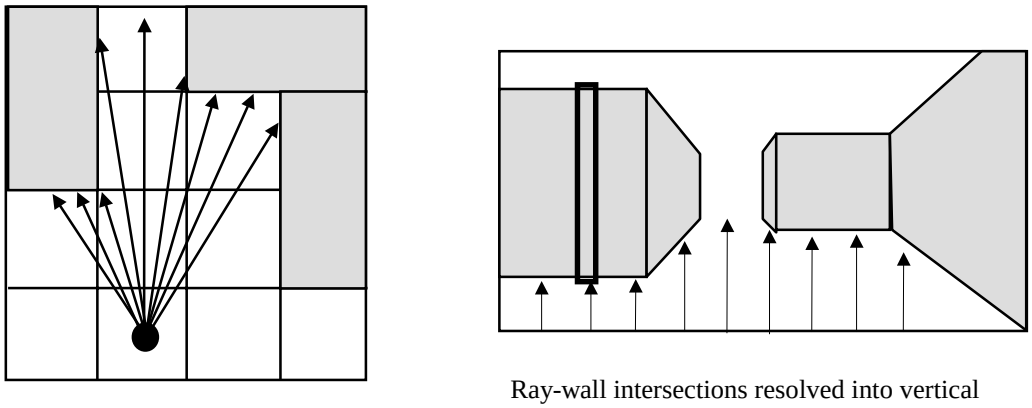
Striving to become King of the Hill, or more humbly to even get noticed, the world-wide community of developer groups literally sucks the results from universities and other research centres, immediately putting them to test in tough competition. Furthermore, such a large group of people constantly exploring the area of computer graphics is bound to come up with innovative ideas once in a while, leading to new areas of research. So like other scientific areas, computer graphics has finally found its own ‘playground’ for the application of research results, hopefully leading to a fruitful cooperation for both communities.

3.1.2 First Generation 2D

One (initially rather small) gaming company has profiled itself in this research-development relationship more than any other, partly due to their declared policy of sharing experience and innova-

tions, and partly due to their ability to pick out research results, that never attracted interest from other parts of the developer community, and apply and combine them in innovative ways to create some of the most important breakthroughs in the history of computer games[1].

'id software' in May 1992 released the computer game "Wolfenstein 3D". The game basically combined two techniques: The first being a relatively simple 'raycasting' over a uniform 2D grid (a floor map consisting of wall-, door- and free-space blocks), very similar to 'voxel walking' acceleration techniques known from raytracing. The second technique was simply a fast vertical bitmap stretcher, implemented by one separate piece of code for each different destination height of bitmaps.



Top view: Casted rays 'walk' over cells

Ray-wall intersections resolved into vertical columns of pixels to build projected walls

Figure 3.1

Utilising the fact that the bitmap stretcher should only render vertical wall strips (as found by the raycaster), the expensive texture perspective correction, that most contemporary games stalled on, needed only be performed per 'wall-strip' (figure 3.1). Furthermore, the texture co-ordinates came at no extra cost from the raycaster. In effect, id software had created the first smoothly running, flawlessly textured, 3D first-person-view 'shooter' - and triggered a whole new genre of games, still being predominant in the business seven years later.

Recall that these were the days of slow, low-resolution graphic cards and PC's generally lacking floating point units. Most 3D games were featuring only wire-frames or flat-shaded polygons, and were still struggling to reach interactive frame rates. Due to the visual outperforming of all competitors, an otherwise well-constructed game and an innovative marketing strategy (involving shareware licenses and internet business), the game became a world-wide hit and set new standards.

Though the algorithms employed were quite clever considering the restrictions imposed by the platform, they were also limited. The efficiency of the raycasting did not scale well over distance, and

[1] Though the term 'computer game' is embracing different areas such as graphic artwork, sound, music, game theory, gameplay etc., this thesis shall only use the term to describe the features of a computer game relevant to computer graphics, primarily the algorithms and data structures on which it relies.

the ‘vertical texturing trick’ (apart from being native to a special VGA graphics mode where screen memory was arranged in columns) inherently prohibited camera rotation other than around the vertical axis. All blocks in the floor map where square and of uniform height, creating worlds that eventually came to look quite blocky and flat.

It is common in a game development cycle that new applicable ideas and research emerge in the middle of the process. Due to the short production cycle it is often not possible to redesign the game to include the improved techniques, so instead they form the basis of the successor of the game. It is very likely that the ideas for id’s next game, Doom, were already being formed as Wolfenstein was being finished.

Meanwhile, Seth Teller published his Ph.D. dissertation at the University of Berkeley. His work focused on techniques for accelerating the rendering of *densely occluded* worlds to interactive frame rates, primarily working on a test case of an architectural model consisting of almost 750.000 polygons including 10.000 detail objects. The interactive frame rates were achieved through reducing the set of potentially visible polygons to an average 1% of that of the input case. Obviously, the scope of this work was focused at workstations with dedicated graphics hardware, and far beyond the limited powers of the contemporary desktop computers. This situation should completely change only a few years down the road.

3.1.3 Second Generation 2D and Restricted 3D

In 1993, the game Doom was released by id software. This game is often described as a ‘quantum leap’ in computer games, and it completely outperformed all contestants who where now just catching up with the raycasting technique. The game was still restricting camera rotation to the vertical axis to provide for efficient vertical and now also horizontal perspective correct texture mapping. The movement included a new (vertical) degree of freedom, and the ‘floor map’ - though two dimensional in nature - had individual altitude settings for floor and ceiling. This allowed for worlds that immediately appeared fully three-dimensional to the player, including stairs, platforms, towers, houses, fences etc. The floor map had been freed of the ‘grid’ restriction that created the very square walls of Wolfenstein. The structure of the worlds were now solely determined by sectors outlined by arbitrary lines, giving the level designers a high degree of freedom and a tool to create much variation.

The underlying techniques of Doom were once again an adoption of computer science research, that had stirred little interest in the gaming community. The data structure chosen to represent the world was a two dimensional Binary Space Partitioning (BSP) tree, first described by Fuchs, Kedem and Naylor some thirteen years earlier [FKN80]. The BSP tree was constructed over the sector boundaries defined in the floor map, dividing it into convex areas for which point-location could be performed in time $O(\log n)$ for well-balanced BSP trees of height $\log n$.

One of the strong limitations of the game’s predecessor was imposed by the raycasting. A travelling ray used time proportional to the distance it travelled; it had to visit each cell on its path, even when crossing large, uninteresting areas. With the BSP tree, cells are allowed to have differing sizes, essentially allowing for the tree only to contain information on the areas of the world, where interesting things happen - i.e. at the cell boundaries. In this way the computations needed for a line cross-

ing a part of the world is merely proportional to the complexity of the traversed land rather than to the travelled distance itself.

Since the BSP tree defines a spatial ordering over the cells it allows for back-to-front or front-to-back traversal of the cells as desired. This essentially allows for the renderer to start at the camera position and work its way out the viewing direction until rendering is terminated by a fully occluded 'visibility horizon'. This can be detected efficiently in Doom because the world structure is 2D in nature. In a 3D world the question of whether full occlusion has been reached is somewhat more computationally complex - equivalent of determining if the view area in screen space is fully occluded by a set of polygons.

Doom immediately set new standards for the realism in interactive worlds in computer games, still using only integer (or at least non-FPU) arithmetic. Again, the game's success must also be dedicated to an intense mood, unique style and not least the introduction of network head-to-head gaming for multiple players. The game now holds a firm position among the top best selling games ever.

However, the game engine still imposed a few restrictions, mainly *(a)* the absence of real 3D structures in the world definition and *(b)* the two lacking degrees of camera rotation - pitch and roll. The missing degrees of camera freedom were still caused by the lack of computational power to render arbitrarily directed polygons with perspective correct texture mapping. Therefore, the game was still restricted to vertical and horizontal strips of pixels orthogonal to the cameras viewing direction (i.e. with constant z-distance), in effect only permitting horizontal faces (floor and ceiling) and vertical faces (walls). A few post-Doom games added slanted surfaces (e.g. Apogee's 'Duke Nukem') and an approximation to limited camera pitching (e.g. Lucas Arts' 'Dark Forces'). The camera was not really rotated, instead the view cone was skewed, keeping the horizontal and vertical pixel strips at uniform projected distance from the camera.

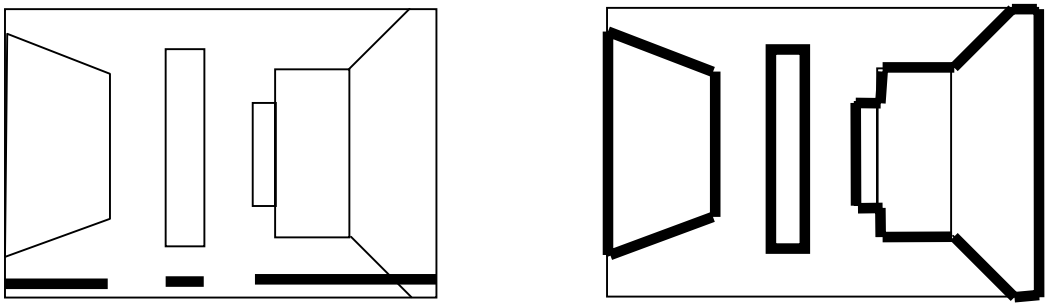
Eventually Doom, its sequel Doom II and a massive number of clones had fully explored the area closed under the restrictions imposed by the engine framework (two-dimensional BSP worlds and 4 (to 4½) degrees of freedom). It became quite clear that the next step would be the transition to full, unrestricted 3D worlds and 6 degrees of freedom. At the same time, floating-point units were becoming standard in desktop computers along with new Pentium processors with parallel pipelines and more muscle. This allowed for perspective correct texture mapping on generally oriented polygons, not necessarily per-pixel, but by subdivision either of polygons or scan lines - and still only with hand-optimised, highly specialised render loops.

3.1.4 Unrestricted 3D

id software early revealed plans of their next 'quantum leap', a full 3D 6-degrees of freedom game dubbed 'Quake'. With id's long experience in utilising BSP trees for real-time games, the extension from 2D to 3D BSP trees was natural. With the demand of general polygons, the era of highly specialised pixel-pushing tricks had ended. The polygon rendering was basically down to implementing well-known techniques that had simply not been available in the PC area before, solely due to the lack of processor power.

Using a BSP tree (or any other hierarchical spatial subdivision), the world can be effectively clipped to the view frustum of the camera. The front clipping plane and the sides of the frustum are initially given, so any part of the world outside the view can efficiently be culled early in the graphics pipeline. The real hard problems are to determine *(a) which* major parts of the world *inside* the view are occluded and should be culled early (e.g. the geometry hidden behind a half-wall), and *(b) when* the visibility horizon has been reached, i.e. when front-to-back traversal of the world can stop, because any more distant part of the world will be fully occluded.

Doom had the advantage of working on visibility progressing over a 2D floor map. An efficient and robust record of occluders (i.e. vertical walls) could be kept in the form of line segments 'masking' out occluded pixel-columns of the view area, i.e. the screen (figure 3.2). Adding an occluder could introduce a new line segment, extend existing line segments or fill the gap between existing line segments, creating one large line segment. When there was only one line segment left, ranging over the entire view area, all pixel-columns of the view area were occluded, and the world traversal could terminate. The same approach in a 3D world would correspond to keeping a record of occluded (or unoccluded) regions of the screen under the operation of adding (or subtracting) general polygons (figure 3.2). In comparison, this is difficult to implement efficiently as well as robustly.



Occluder tracking in a native 2D (floor map) world

Occluder tracking in a 3D world

Figure 3.2

Once again, id software turned to computer graphics research applicable to their particular problem and adopted a strategy that uses methods similar to those proposed by Seth Teller. Instead of dynamically seeking a zero-overdraw[2] solution, the work of Teller uses *pre-processing* to estimate a *potentially visible set* (PVS) from any cell in the spatial subdivision. So, instead of trying to 'cut away' occluded parts of the world at run-time, the engine simply uses a precompiled list (augmented to the camera's cell) to render only those other cells potentially visible from the source cell. The list is valid for a general observer in the source cell (i.e. from *any* position inside that cell), so in most cases it overestimates the visible set of a particular observer (i.e. from an actual camera position). However, accepting a little amount of overdraw in the scene with the bonus of avoiding any ad-

[2] Zero-overdraw indicates the situation, where all hidden geometry is culled before submitted to the polygon renderer, so that no pixel is being rendered and later being overdrawn by a closer surface. Sometimes overdraw is reported as an 'overdraw factor' indicating how many times each pixel on average is being rendered. With that definition, zero-overdraw corresponds to an overdraw factor of 1.0.

vanced real-time evaluation of occluders turned out to be feasible, as Quake achieved interactive frame-rates for even large environments with complex 3D structure.

A problem we have not yet addressed, is a restriction on the dynamic nature of the world that was introduced when id first chose to rely on BSP trees for world subdivision. Basically, this restriction emerges from the fact that the top part of a BSP tree (the early splitting planes) tend to stretch over large parts of the world. This means that locally perturbing the boundary of a cell (a tree leaf) may propagate an effect all they way to the top of the tree. This may induce a full *reconstruction* of the tree rather than a simple reordering or perturbation, since the position of the early splitting planes may affect the subdivision of the world itself and thus the structure of the sub-trees.

In Doom, the restriction was not too imminent, because the BSP tree was in the floor map. Vertical movement was orthogonal to the tree and indeed allowed for a great deal of dynamics in the world - doors sliding up and down, elevators, collapsing stairs and moving floors. In Quake, however, dynamically modifying the structure of the world has immense impact. Since the BSP tree is now 3D, there is no orthogonal direction of movement, so any change will affect the BSP tree, locally or globally. Furthermore, the precompiled PVS information is heavily relying on the structure of the world not changing, so Quake worlds seems to be very static. A few moving objects exist in Quake worlds (doors, platforms and elevators), but in general, they seem more sparsely utilised than the dynamic structures of Doom. Furthermore, they somewhat leave the impression of being inserted as 'special cases' rather than blending smoothly with the environment.

In the end, Quake did not seem to be quite the quantum leap that Doom was. The game was heavily delayed, and some of the competitors had finally caught the drift and moved into 3D 6-degrees of freedom (6 DoF) engines with similar or different approaches, some directly using portal rendering ('Unreal', 'Prey'). Another reason for the 2D-to-3D shift not having the predicted impact could be, that human worlds on a large scale tend to be flat, emerging from the fact that we are surface creatures. Since the 2D worlds of Doom already had some variation into the third dimension, they were not that far from be acceptable as 'real (3D) environments', and Quake added the possibility of *real* 3D worlds. Nevertheless, the average player tends to perceive a very complex 3D world in which he moves, as a multi-storey building, i.e. conceptualise it as a stack of individual floors, based on the moving being primarily in the horizontal plane.

Summarising, the state of the computer gaming world currently seems to be that the gap between work stations and desktop computers is narrowing. This is at least true on the graphics side where aggressive hardware acceleration and unrestricted (6 DoF) 3D languages such as OpenGL is rapidly becoming a standard in the mass market. With the higher growth in graphics hardware performance than in CPU performance, there is evidence that more and more of the trivial tasks of lower level graphics (transforming, projection, scan line conversion) is being moved into silicon. This leaves the more sophisticated world geometry considerations for the CPU.

The high end graphic cards for PC's today are set-up limited rather than fill limited, i.e. the card can scan-convert large polygons faster than we need for most practical applications[3]. However, we can

[3] An example of 1999 performance level being Creative Labs' S3 Savage4 based graphic cards peeking at 125.000.000 pixels per second, equivalent over handling 8.7 times overdraw in a 800x600 resolution at 30 fps. (Expected) price levels around 130 USD.

push set-up information, unneeded textures etc. faster than card can treat them - one of the current bottlenecks here being bus traffic. All together, this suggests that brute-force techniques such as relying on painter's algorithm or z-buffers are not sufficient. Culling major invisible parts of the world early in the graphic pipeline is essential, especially to make the rendering scale well with increasing world size.

3.2 Comparing Current Techniques

As described in the previous section the main acceleration techniques used in real-time applications for densely occluded environments has been BSP trees and, lately, portal rendering. Grid-based or other uniform spatial subdivision algorithms show little promise of scalability and they have been widely abandoned for large worlds.

The comparison between BSP trees and portals is not straightforward. Depending on the context, the techniques can be seen mainly as means of world structuring, or as rendering acceleration techniques. In fact, they may even be regarded as complementary and interleaved techniques.

Though the world model used in portal rendering is explicitly graph based, the method used to obtain the graph subdivision may well be BSP trees. In fact, one could argue that the portal framework only provides a rendering method on top of *any* subdivision method. Likewise, one could argue that BSP trees themselves is hardly an acceleration scheme but rather a world ordering, basically used to present polygons to the renderer in a back-to-front order.

Even if choosing a graph ordering for the world, the rendering algorithm could still use a hierarchical ordering for efficient point-location as described in section 2.4.3. If choosing a strict tree structure for representing the world, the rendering algorithm could still make good use of connectivity information between neighbouring cells, similar to the portal graph. In this perspective, the techniques seem mostly complementary.

Despite these ambiguities we shall outline a series of generally important properties to rendering acceleration and evaluate how these properties are reflected in the two techniques. Formally, a strict BSP rendering framework does not utilise occlusion information and therefore does not require any special occluding properties. However, we shall once again assert that our target worlds are showing a high degree of self-occludence (being a precondition for the portal framework and this thesis).

3.2.1 Data Structure

The portal graph and the BSP tree differ mainly in structure, the graph being a 'flat' data structure with only local dependencies, and the BSP tree being a hierarchical data structure with global dependencies.

The portal graph represents an almost chaotic view of the world without any obvious indication of how to get from node A to node B, if these are not immediate neighbours. Instead, the portal edges represent a natural way to move around in the world, closely reflecting the structure of the input

world. In e.g. a house model, the graph will contain edges only along pathways a (sufficiently small) observer can travel, because the graph subdivides the world using natural ‘room’ and ‘corridor’ concepts. However, this is also the *only* information the graph gives. It does not reveal which rooms are spatially close to each other, if they are not mutually visible or accessible.

The BSP tree, on the other hand, gives a nice, ordered view of the world model, starting from a top level containing everything, and tediously subdividing the world down to the smallest, atomic rooms. It provides an efficient way of locating a point into the model (logarithmic, if the tree is well-balanced), since the BSP tree is a spatial search tree, where you start at the root and recursively ask if the point is in the left or the right sub-tree until you reach the answer to the query - the cell in which the point is located. Neighbour finding is possible too, but it does require a little ‘tree-climbing’, and in the worst case, you have to climb all the way to the top and down the other sub-tree to find an adjacent cell.

Dynamically updating the portal graph is fairly inexpensive. If geometry is changed but the fundamental visibility structure is unaltered, the graph need not be updated at all. I.e. it is possible to ‘push a room around’ and ‘bend a corridor’ as long as visibility paths are not added or broken. Even if the visibility characteristics are changed (a portal removed or two cells suddenly connected by a new portal), the updates to the graph are only local, no matter the size of the total world model. This can be exploited e.g. for doors that can be opened and closed. (Notice that none of this holds true for an accompanying PVS, which is about as static in nature as the BSP tree - or even more, because the construction times are higher.)

BSP trees are of a very static nature because of their global properties and their tight relation to the structure of the input model. Since splitting planes are chosen incidental with model polygons, moving even a single polygon in the model could change the basis for one of the early splitting planes (i.e. a splitting plane close to the top of the tree). This could force a completely new structure for the entire tree with a costly reconstruction as a consequence. If the type and amount of dynamic changes are known beforehand, the BSP tree can be built so the dynamically changing regions are placed in sub-trees as close to the bottom as possible, requiring only local updates in the tree. In practice, a region containing e.g. a moving wall (and its path) is not subdivided until *all* the surrounding volumes have been subdivided. However, this is still a semi-static approach, and too many of such ‘special cases’ could compromise the balancing of the tree.

3.2.2 Visibility and Rendering

The BSP tree can be considered a rendering acceleration technique because of two important properties. First, it provides an ordering of the world polygons with respect to the observer. This gives both *backface removal* (because you always know on which side of a splitting plane the observer is located), as well as *polygon sorting* for painter’s algorithm at no additional cost. You recurse the tree from the top and at each splitting plane, you render the sub-tree *most* distant from the observer before you render the closer one. Second, it provides a spatial subdivision and ordering of the world, making it possible to *efficiently* determine which parts of the world are outside the view frustum. You test the bounding volume of a sub-tree against each clipping plane of the view frustum. If the sub-tree is completely *outside* the view frustum, nothing inside it can be visible and the entire sub-tree is immediately culled. If it is completely *inside* the view frustum, the entire sub-tree is visible

and need not be further tested against the view frustum. In worlds that are very large compared to the volume of the view frustum, this provides culling of the parts outside the view frustum in logarithmic time.

The rendering method used in the portal framework can best be described as ‘progress as you see’. Roughly, it works by constantly reducing the spreading angle of the view frustum every time it passes a portal, until the visibility is blocked by a wall. The rendering process is recursive, so if you look into a room with two doors, the rendering process will first render things visible through one door before rendering things visible through the second door. The rendering process implicitly clips the input world to the observers view frustum, since this is simply the initial frustum with which the recursion is started. Like the BSP method, the portal rendering process (not the data structure itself!) provides back-to-front rendering applicable to painter’s algorithm, due to its recursive nature. Ideally (with perfect frustum calculation and clipping), this method provides an optimal rendering scheme, since it renders everything visible to the observer, and it renders nothing that is invisible to the observer.

Though both methods clip the world model to the view frustum, the portal rendering is more efficient in that it terminates at the ‘occlusion horizon’, i.e. when each path in the search tree is blocked by an occluding wall. The BSP rendering effectively proceeds to the border of the world or the back clipping plane of the view frustum, whichever is encountered first, even if a fully occluding wall is placed right in front of the observer.

There is a good reason why occludence information can not easily be used in the BSP approach. In a 2D-world, the BSP method (as described in section 3.1.3) can easily keep track of occluded parts of the field-of-view (amounts to calculating unions of intervals on a line) and terminate when everything visible to the observer has been rendered. In a 3D-world occluder tracking is much more complicated (amounts to calculating unions of projected polygons) and it can not be performed in the same efficient way. Fundamentally, the BSP approach has no easy way of determining which combinations (or *unions*) of occluders that obscure certain parts of the world. The portal approach reverses the question and determines which sequences (or *intersections*) of non-occluders (i.e. portals) that allows the observer to see certain parts of the world.

Put in another way, the BSP approach starts by assuming that the observer can see *everything* and then has a hard time figuring out which parts cannot be seen. The portal approach starts by assuming that *nothing* is visible and then try to figure out what can actually bee seen. Here lies the reason for the requirement of the target worlds to be highly self-occluding, i.e. only small parts of the world being visible from any particular location. If the set of visible polygons is small, less work is required to build the visible set, than to eliminate invisible polygons from a very large set. In worlds with a *low* degree of self-occludence, the number of visible portals will be high and the search tree for the portal rendering will be very broad. In this case, the portal rendering too is a very pessimistic approach.

Apart from real-time rendering, the portal framework has proven itself applicable to other visibility problems, e.g. for radiosity calculations [TeHa93]. The primary workload in radiosity calculations is the estimation of form factors, describing the visual interaction between all pairs of surfaces in a model. In a self-occluding world with a portal ordering, the number of surfaces mutually visible is low and can be efficiently evaluated by tracing portal sequences in the graph.

3.2.3 Database Management

Point location in the portal framework is a cumbersome linear search (due to the spatially unstructured nature of the graph) and should be accelerated by other techniques (e.g. a BSP tree used as a search tree). However, in practical applications temporal coherence is high and can be exploited. Once the observer's location has been identified in the graph, he is likely to move inside the same cell for several time frames. When the cell boundary is crossed, it will always happen at a portal, i.e. following an edge to a neighbouring node (unless the observer is allowed to walk through walls). Therefore, continuos point location can be performed with expected constant time per time frame. In comparison, a BSP tree without connectivity information must use logarithmic time whenever a cell boundary is crossed (since a neighbouring cell may be very distant in the search tree).

This behaviour is highly relevant when designing databases for representing and retrieving the world model. Since we focus on very large worlds, we can not assume that the entire world description can be present in primary storage, so clearly database access is an issue. Both portal and BSP methods are based on subdivision, so we consider each cell with augmented polygons and possible connectivity information to be the atomic unit for retrieval. The BSP model must further store a representation of the BSP tree, more specifically at least a bounding volume and a splitting plane description for each node in the tree.

Since the portal framework allows direct access to neighbouring cells and it ideally only touches visible cells, it is expected to show good caching behaviour. In fact, the set of cells accessed by the rendering algorithm is very local to the observer and completely indifferent to the overall size of the world model, making promise of a very scaleable method. The PVS as described in chapter 2 has some value to the rendering process (though it requires refinement in the real-time rendering step). However, it may also be used in the data retrieval step for pre-fetching a set of cells that are likely to be visible to the observer when moving around inside the current cell. [Fun93] has extended this approach to pre-fetch a potentially visible set based on predicting the motion of the observer for a couple of time frames.

Even though BSP trees ensure some degree of 'localism' (most neighbour nodes will be relatively close in tree), neighbour nodes *can* be unpredictably distant in tree, requiring access to arbitrary paths in the search tree - even if the tree is decorated with connectivity information. Since the height of the (well-balanced) tree is logarithmic in the number of cells, the cost of data retrieval from the database will receive an impact from the overall world size, making the technique less scaleable.

3.3 Implementing a Real Case

This section describes a commercial project in which the author of this thesis implemented a 3D computer game including a complete 6-degrees-of-freedeom graphics engine with rendering acceleration based on a portal framework.

The creation and programming of a computer game is a manifold exercise involving different areas such as raster graphics, higher level visibility calculations, world database management, sound programming, physics simulation, gameplay logic, input device handling and network communications. The borders between these areas are not always well defined, as for example the sound and physics simulation can also benefit directly from the acceleration techniques used for the graphics rendering. Therefore, this section starts with a short, but broad description of the entire game and the design and construction process behind it, to give an impression of the requirements to the framework.

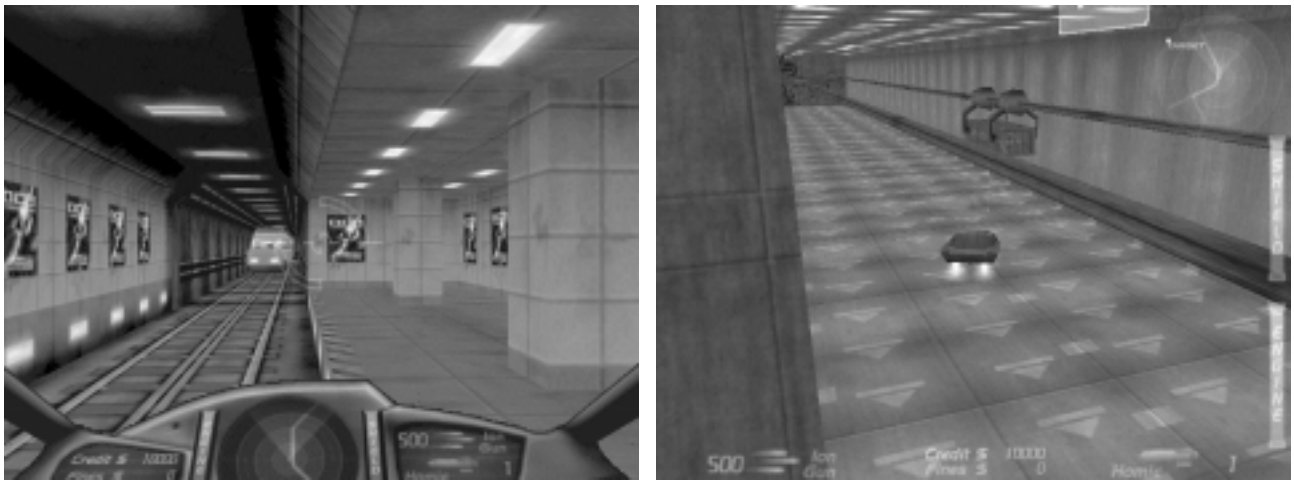
3.3.1 Description

The game takes place in a futuristic science-fiction world where a large city is infected with high crime levels, and renegade characters are terrorising each other as well as innocent people. The player fills the role of a bounty hunter in a gameplay where money and force of arms are the main balancing factors. A balance of power is established between various characters and organisations such as police, gangsters, free-lance bounty hunters and criminal gangs. The objective for the player is to exploit this situation by carefully supporting the right elements at the right time and make money from serving one part and fighting another.

Open city areas, using transparent surfaces for emulating glass and water

Figure 3.3

The city that forms the environment for the game is a large collection of various city parts with individual graphical expressions, such as financial districts, a ‘china town’, industrial complexes, ragged ‘Bronx’ type neighbourhoods, sports stadiums etc. Each of the city parts are relatively open areas with streets, squares and large buildings, and the entire city is connected by a network of main roads, subway tunnels, sewer tunnels and secret pathways.



Closed, tunnel- or cave-like areas

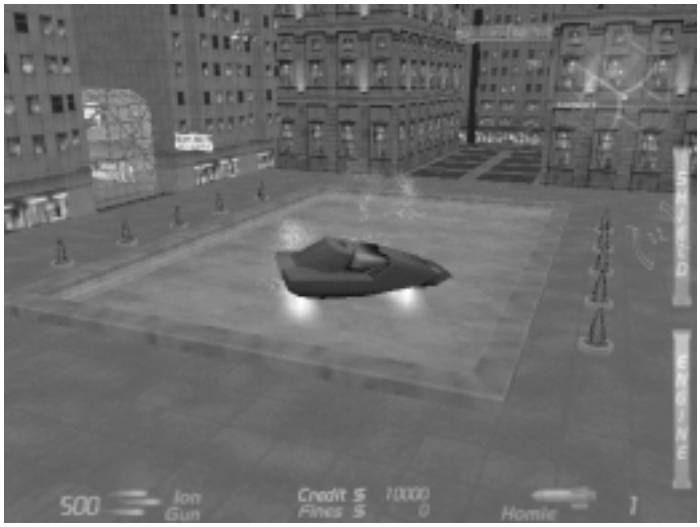
Figure 3.4

All active roles in the game are played by futuristic, flying cars, dubbed *hover jets*. These cars are vehicles that hover on a set of fully turnable jet streams, making them extremely capable, manoeuvrable and, indeed, funny to fly. The game environment manages a large number of cars in real-time, simulating the physics of each car to different extends, based on the particular car's role in the game. Apart from the player's car and possible remote players' cars (in multiple player sessions), the world contains between 20 and 50 computer-controlled enemies and around 500 rather dumb cars. This provides the game with an atmosphere of life and, occasionally, filling the role of unfortunate, innocent bystanders.

A (metro) tunnel connecting to an open area

Figure 3.5

The physics simulation of the game involves a stabiliser system calculating angles and effect of the car's jet streams, based on the car's current movement and the requests from the player. Apart from this, the game performs the notoriously expensive task of collision detection between the car and the environment and a slightly cheaper (bounding sphere based) car-to-car collision. The physical reaction of the car is reflected in the car's movement and in sounds, and it can be physically induced on the player through a force-feedback joystick.



The player's hover jet car, suspended on 4 turnable jet streams

Figure 3.6

For the computer-controlled players, the game includes a simple artificial intelligence, or rather a set of simple heuristics for various decision making, fighting and manoeuvring behaviour, hunting and path finding. The computer-controlled players react both to the (human) player as well as to each other. A computer controlled bounty hunter may, for example, at different times choose to attack another computer player or a human player, based on the reward he may achieve and the dangers involved.

The player directly controls the car through a configurable set of input devices including keyboard, mouse and joysticks. Furthermore, the game has been developed to support a (commercially available) head-tracking device[4] that enables the virtual camera to follow the movement of the player's head, creating a virtually realistic experience.

The realism of the environment is further enhanced by a custom 3-dimensional sound framework that models effects such as stereo, distance attenuation and frequency changing Doppler-effects. Finally, the game is able to connect to other instances of the game on remote machines through any kind of network, including the Internet. The game has been successfully tested with multiple players over a cross-Atlantic connection.

[4] 'UR Gear,' a head set with stereo sound and infrared tracking of two rotational and one translational axis.

3.3.2 Approach and Design

To achieve the properties outlined in the previous section, a number of carefully balanced choices and restrictions had to be made.

The final world contains around 20 different city parts with different appearances. Although textures can be heavily re-used inside each city part, differences between city parts are substantial, so basically each part requires a unique set of textures. Totally, the world is divided into around 1500 segments (corresponding to cells in the portal framework), each with a number of attached objects totalling a polygon count of around 10^5 for the entire world. The 1500 segments are instances of around 500 unique 3D models and the texture base is around 500 textures, each 128×128 pixels with a unique 256-colour palette. Translated to real-life measures, the city covers an area of roughly 10 square kilometres.

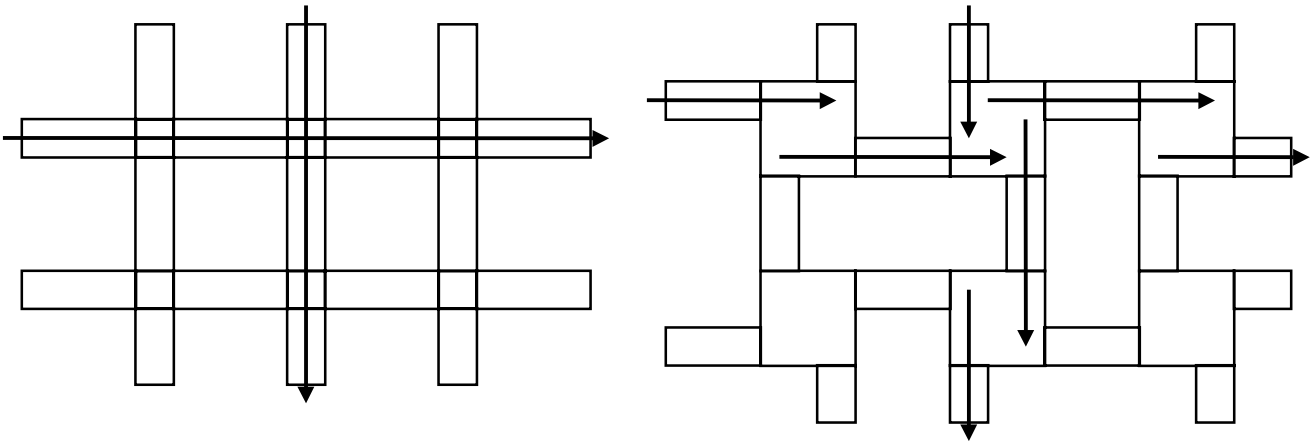
Two main requirements were initially posed for the design process. First, to improve the realistic feel of the world, the entire city should be immediate accessible when flying around, without interruptions for loading new parts. Second, the environment should allow for easy construction and dynamic editing of the city, with as rapid development cycles as possible. In a game design process the final layout and adjustment of the environment can not be done until it can be tested with the final gameplay, because gameplay and environment are highly interdependent.

Based on these requirements and the desired size of the world, it was necessary to impose an efficient structure on the world as well as to accelerate the actual rendering. For this purpose, the portal framework was chosen. Furthermore, it was chosen to build the world as a *constructive model* (see section 2.4). Practically, the world is constructed by combining a set of primitives, for example tunnel or road segments, each with a predefined set of portal positions. This 'brick'-like approach allows for re-use of structural elements and for easy world construction and editing, as segments with predefined portals are easily 'snapped' to each other, using a grid function.

To achieve the dynamic editing required for rapid development cycles, the approach of precalculating a PVS was abandoned in favour of a solely dynamic acceleration technique. For ease of implementation, it was chosen to use only rectangular, axially aligned portals[5] as suggested by [Lue95]. As mentioned in section 2.3.4, this could severely affect the length of traversed portal sequences and hence the performance. This is especially true in our case where the flying vehicle combined with unrestricted camera movement and head tracking allows for arbitrarily tilted camera angles.

In certain areas, this performance degradation became quite evident, so in order to avoid a slowdown the world was explicitly constructed to avoid such situations. When designing a New York-like city, a grid of crossing streets and avenues generates very long portal sequences (and visible complexity in general). Therefore, streets were 'broken' in a zigzag pattern to make the model maintain a high degree of self-occlusion (figure 3.7).

[5] Even though a portal is formally rectangular, it can be partly covered by occluding polygons, allowing for e.g. round tunnel entries.



‘Realistic’ network of street segments permitting very long portal sequences.

Network of street segments designed to avoid long portal sequences.

Figure 3.7

3.3.3 World Editor

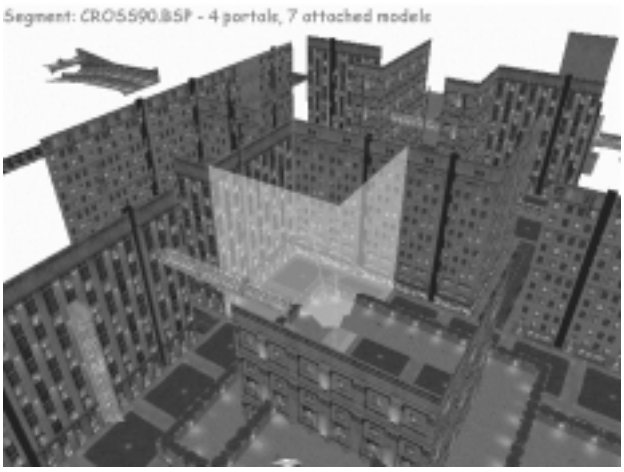
The editor for the world database has been integrated directly into the game and can be accessed on the touch of a single key. The world designer can quickly swap in and out of the editor and immediately put changes to the test inside the game environment. Everything in the editor is displayed using the game’s rendering pipeline, including texture mapping and special effects such as transparency and alpha blending. Since the chosen portal framework requires no heavy pre-computation (such as calculating a PVS), the world editing is fully dynamic and the designer can build the world in a rapid prototyping process.

The atomic unit for the editor is called a *segment* and comes in two flavours, *ordinary* and *attached* segments. A segment is basically a small 3D mesh with texture information, where certain polygons may act as portal definitions (those polygons augmented with a special texture name). If a segment has one or more portals, it is an ordinary segment, typically a road or tunnel segment. If a segment has no portals, it must be *attached* to an ordinary segment. Attached segments are typically separate walls and loose objects such as bridges, elevators or containers (figure 3.8). Practically, each segment has been compiled into a small BSP tree (for efficient backface culling and collision detection). All segments are arranged in a linked list, so that an ordinary segment is immediately succeeded by the segments attached to it.

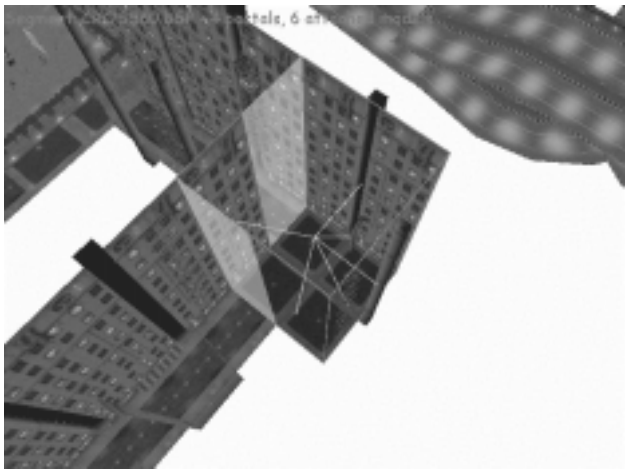
Relatively complex 3D structures created with attached segments

Figure 3.8

Ordinary segments serve the role of *cells* in the portal framework. The non-portal polygons in segments serve the role of *occluders*. Attached segments, that do not have portals, can be considered a kind of ‘additional’ occluders - or in some cases simply detail objects. For a simple tunnel system it may be sufficient with a single segment per cell, defining both the walls of the tunnels and the portals that connects the segments. Attached segments were introduced for more complex structures such as city environments. In this case, a cell typically represents a road segment or a square, incident with the walls of at least two adjacent buildings. In order to create many different buildings in a city, either it was necessary to create a new segment for each combination of adjacent building types, or alternatively to detach the actual walls and make them separate, attachable segments. An additional benefit of such loose walls is that one can choose *not* to use certain portals and rather fill the gap with an attached segment. For example, if turning a road into a dead end or turning a crossroad into a T-junction (figure 3.9).



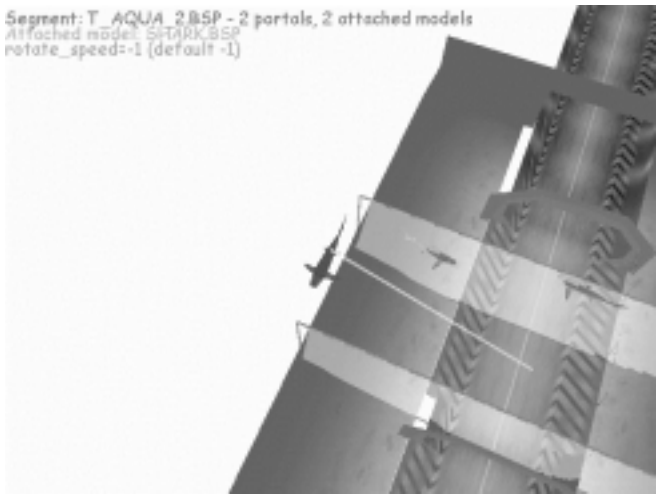
4-portal square where only 2 portals are used; unused portals are filled with attached walls.



Portals only partly used; for unused parts of portals, the gaps are filled with small wall segments.

Figure 3.9

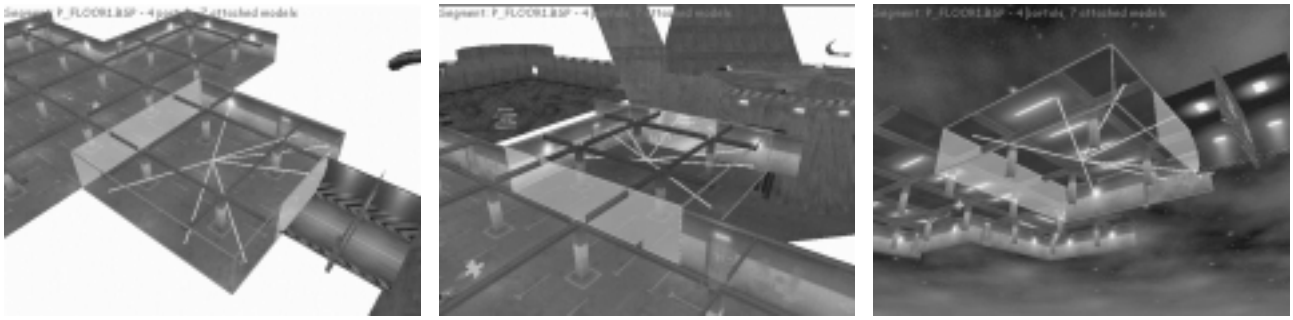
Since the editor directly uses the game's rendering pipeline, it is capable of rendering the world model at highly interactive frame rates (typically above 20 frames per second). It is therefore not limited to conventional display techniques used by many 3D modelling programs, such as wire-frame models in separate XY, ZX and YZ windows. Instead, the model can be rotated and viewed in real-time from any angle with full texture detail, and additional editor information such as outlines and links between segments are visualised with transparent surfaces and lines in different colours (figure 3.10).



The link between ordinary and attached segments (in this case a shark) is visualised with a transparent line to the centre of the embedding segment.

Figure 3.10

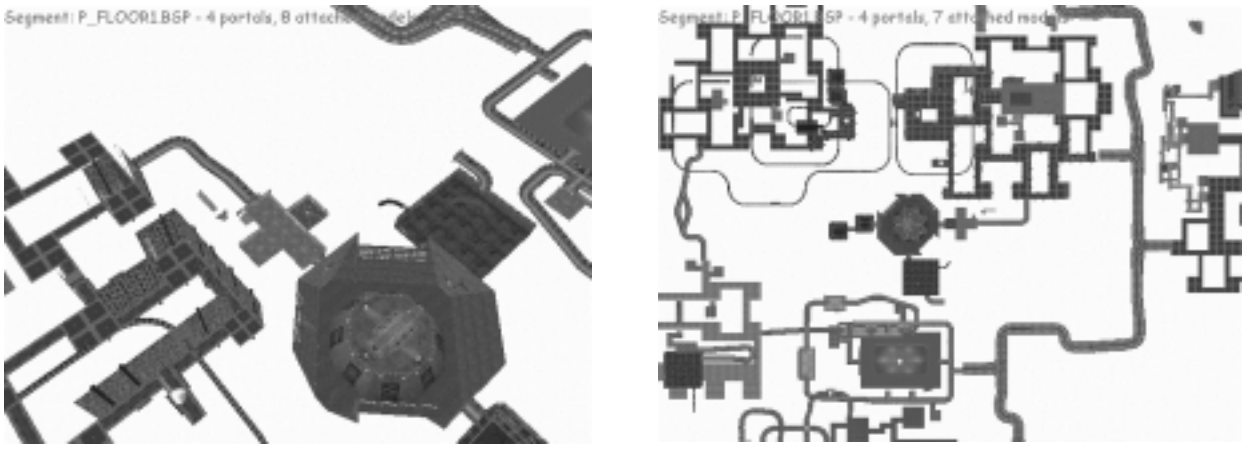
The design of the editor is very simplistic, and all manipulation is done using the mouse together with a small set of modifier keys and action keys. This has turned out to be a great advantage for the project, essentially allowing anyone to use the editor very quickly.



The same room segment viewed from different angles simply by moving the mouse.

Figure 3.11

The default mode for the mouse, if no other keys are pressed, is to view the active segment from different sides by spinning the camera around (figure 3.11). Moving the mouse along the x-axis rotates the camera around the vertical axis while keeping the active segment in focus. Moving the mouse along the y-axis will rotate the camera vertically around the segment, allowing the user to get a side view or a top view of the model. If the user holds down the right mouse button, the mouse y-axis is translated into a zoom factor, allowing the user to quickly zoom in on details or to zoom all the way out to get a full, top-down overview of the world (figure 3.12).



Slanted top-view good for navigating around

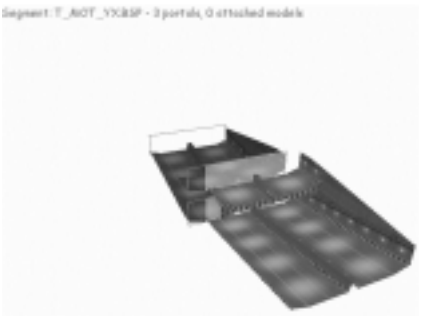
Zooming out to a large top-view of an area

Figure 3.12

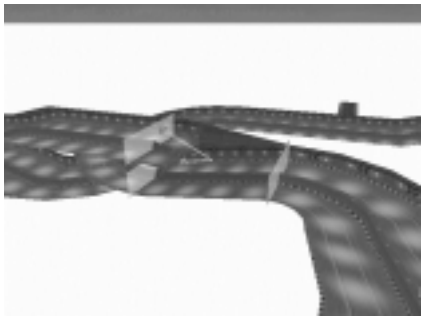
Navigating around the world model is just as simple; by holding down a key for ‘select’ a 3D cursor appears and can be moved around in the horizontal plane using the mouse. As the cursor moves over a new segment, it will be selected as the new active segment, so that when the ‘select’ key is

released, the camera focuses on this new segment instead. Moving segments around happens in a similar way by holding down the left mouse button. The active segment is then directly moved around in the horizontal plane using the mouse and can be rotated in 90 degrees steps using the cursor keys. All segments have a defined grid size, making it easier to align segments and snap portals to each other (figure 3.13).

A single segment with 3 portals, indicated by outlines.



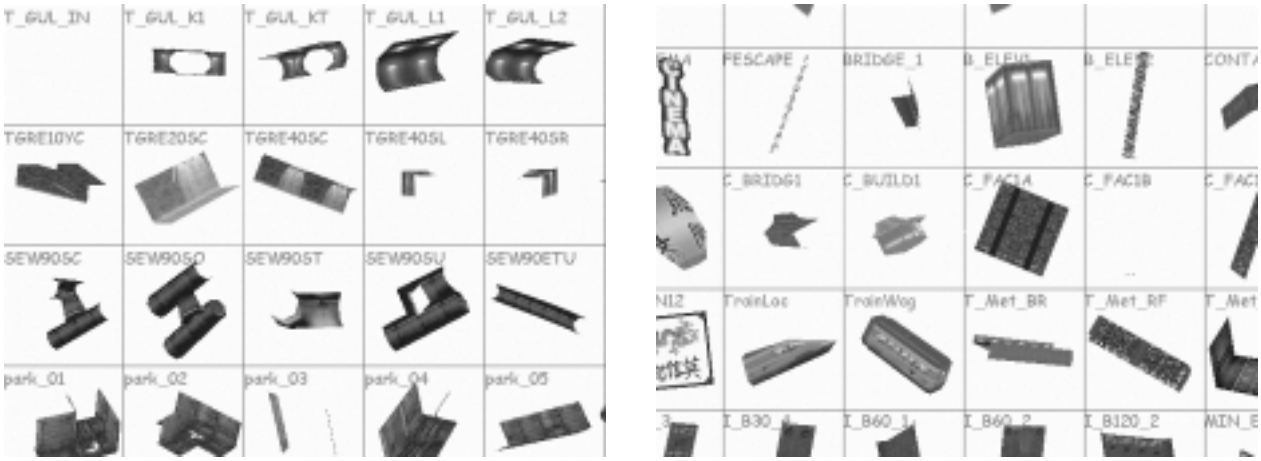
Using grid to snap two segments to each other. Overlapping portion of portals is indicated by a transparent surface.



A more complex set of segments correctly assembled to form a road junction.

Figure 3.13

As previously mentioned, all segments are based on a set of primitives or prototype models. Creating new segments is done by cloning the currently active segment when the user presses the ‘new’ key. Alternatively, the user can select to clone the current segment *including* all attached segments. The 3D model used for the active segment can be replaced at any time by holding down the ‘palette’ key (figure 3.14). In this case, the entire screen is turned into a large palette of all the available 3D models. By moving the mouse over this palette (which will scroll around if larger than the screen area), the user can select a new prototype for the current segment. When the user releases the ‘palette’ key, the prototype pointed out by the mouse will replace the model previously used for the active segment.



Two views of the model palette, using the mouse to pan around over the 500 models.

Figure 3.14

3.3.4 Additional Issues

A number of implementation issues that are directly linked to the portal structure deserve to be mentioned. First, the newly introduced ‘attached’ segments must be fitted into the framework. As the attached segments are considered either additional occluders or detail objects, they are presumed to be located *inside* the segment or cell to which they are attached. Therefore, the portal framework and the portal graph traversal operate only on the *ordinary* segments that store the portal information. Only when an ordinary segment is visible through some portal sequence, the segments attached to this ordinary segment are considered visible and rendered.

This approach is taken even further to also include dynamic objects in the world, such as cars flying around. Each ordinary segment (i.e. each node in the portal graph) stores a linked list to all the dynamic objects currently inside its bounds. Such objects are therefore only rendered whenever the segment in which they are contained is rendered. Each dynamic object must update these references whenever it crosses a portal and moves from one segment to another; this is an *O(1)* operation, where the dynamic object removes itself from the linked list of the exited segment and adds itself to the linked list of the entered segment.

As mentioned in section 3.3.1 a very large number of ‘dumb’ cars are flying around the world to give the impression of life. Given the large overall size of the world, each of these cars is invisible to the player most of the time and should use as little computational power as possible when not visible - except that it shouldn’t turn completely immobile when out of sight, because this would be quite noticeable to the player. Instead, whenever it enters a new segment, it updates its references and quickly calculate (based on the portal graph) an entry point and an exit point on two different portals. Then it estimates the time needed to fly from the entry to the exit, given its individual speed. The only task it needs to perform for the next many frames is to observe whether the travelling time has elapsed and it is time to move into the next segment.

Only when such a car becomes visible (which it does when the containing segment becomes visible) it is necessary to calculate the cars actual position. Using the entry and exit point and the portals' normal vectors, a spline connecting the points is quickly calculated, and the percentage of the elapsed travelling time decides the car's position on the spline. Furthermore, the car's velocity and acceleration vectors are easily calculated from the spline. In practice this works very well, as the player will soon observe that e.g. a number of cars that go into a closed tunnel system eventually comes out again, leaving the impression of a very vivid and consistent world.

A similar approach is taken for limiting the number of active sounds and 'spectacular behaviour' (such as moving elevators and fountains without any gameplay interaction), in order to reduce CPU load. Each object that emits sound or has such behaviour keeps track of when its embedding segment was last visible to the player, and after a while it shuts down its sound or activities until the segment becomes visible again. In practice this works quite well, as the embedding segment typically becomes visible a little bit of time before the object itself, allowing e.g. a fountain object to start up properly before it becomes visible.

The collision detection in the game also relies heavily on the portal framework. Collision is determined in a number of different ways, depending on the involved objects. For example, the 'dumb' cars as described above have no real physical representation, so their collision against the player's car is simply performed using a bounding sphere (which seems sufficient for fast moving cars anyway). This collision check can exploit the fact that dynamic objects are linked to the embedding segment. Therefore, the player's car needs only check collision against other dynamic objects linked to the same segment (and possible neighbour segments, if the car is crossing a portal).

For the car-to-environment collision, a real surface-to-surface collision has to be examined (although it first checks a bounding sphere for early rejection). Again, the collision code exploits the portal graph by using a recursive graph traversal and the bounding sphere radius to select a set of good 'collision candidates'. The search need only proceed beyond a portal if the bounding sphere touches that portal. Again, we benefit from the introduction of attached models, as this provides a good breakdown of the walls of a given cell. Typically, the car will only be close to one of the walls in a road segment with an attached wall on each side, and the other wall is rejected immediately. Finally, when a minimal set of collision candidates is chosen, the BSP tree of the car is efficiently checked against the BSP tree of each candidate by sorting the larger BSP tree down the smaller one, looking for intersecting polygons.

The last type of collision detection is line-to-environment checks performed for e.g. bullet trajectories and camera line-of-sight. This check basically amounts to the portal stabbing and recursive graph traversal described under point-to-point visibility in section 2.2.4. Again, the 'attached segment' breakdown and the BSP structure of the 3D models come in handy for efficiently finding a directed line's first intersection with the environment.

Finally, the connectivity information from the portal graph is explicitly used in two different instances in the game. The computer controlled players use the portal graph directly for path finding, by creating an $n \times n$ matrix describing for any pair of segments A and B, the shortest distance and which one of A's portals that leads towards B. To keep the size of the matrix down, segments with

exactly two portals are disregarded, as they merely constitute pathways between segments that are real decision points. Removal of these segments reduces *n* from around 1500 to around 200.

The other direct use of the portal graph is to present a structural map to the player while navigating around the world. This method directly renders the portal graph onto a radar screen in the car's cockpit (figure 3.15). Again this is done using recursive graph traversal, starting in the player's segment and proceeding until the range of the radar screen (or a vertical limit) is exceeded. This has the great advantage, that only areas that the player can get to within a short distance is visualised, and unnecessary complexity (for example from inaccessible world layers right above or below the player) is omitted.

The round radar screen in the car's cockpit, displaying close parts of the portal graph.

Figure 3.15

3.3.5 Evaluation

Overall, the portal framework and approach appeared to be very advantageous to the design and development process of the game, as well as to the performance of the final game.

The real-time rendering reached the desired performance level, allowing the game to run on today's average consumer PC's with highly interactive frame rates (easily above 30 frames per second on a 300 MHz Pentium with an average hardware accelerated 3D card). In fact, the very large game world could probably not have been visualised in real-time without using occlusion culling as provided by the portal framework (or other occlusion culling methods). The typical number of world

polygons deemed visible to the player in a given time frame is around 1000, or 1% of the total number of world polygons, consistent with the results reported by [Tel92].

In the development process a lot of time and work was saved by directly using the game rendering code for the world editor instead of writing specific tools for assembling segments and maintaining the portal graph. Furthermore, the process of designing the game world was tremendously eased by the integration of game and editor and the short prototyping cycles achieved.

A computer game is an obvious application for constructive models, as the game world can be *designed*, and is typically not required to precisely reflect a real-world counterpart. Even if trying to model a real environment, a game situation usually leaves a little room for deviations and adjustments to accommodate the portal framework. In a computer game, and indeed for some simulation purposes, it may be sufficient to model a particular town using a relatively small set of typical building types rather than model each house individually from its real-life prototype.

Not only does the constructive approach using construction primitives allow for quick and easy world building as well as model and texture reuse. If the ‘rooms’ are created at a reasonable size and with a reasonable amount of detail, it also provides a good breakdown - or subdivision - for the portal framework to work from.

The possible drawback of using constructive models is that the ‘human factor’ from the design process may be quite evident in the final result. It requires great discipline and skill on the designer’s behalf to make the world appear with a large amount of variation and not making the reuse of primitives too conspicuous. When trying to make construction primitives easily fit to each other like bricks, it becomes very difficult to avoid a preference for axially aligned portals and walls. In the end, a designed world easily appears too regular to look fully realistic.

3.4 Other Uses

The portal framework - with its alternative world ordering - invites to a number of interesting uses in a number of application areas that seem very open to research.

3.4.1 Mirrors

With a few restrictions, [Lue95] has implemented static and dynamically moving mirrors in a portal framework. The portal framework lends itself very well to mirrors by handling a mirror as a special kind of portal augmented with a geometric transform. The idea is that the mirror is treated as a ‘window’ into the mirrored world, but instead of mirroring the world, the observers position and co-ordinate system is mirrored in the plane of the mirror portal (figure 3.16).

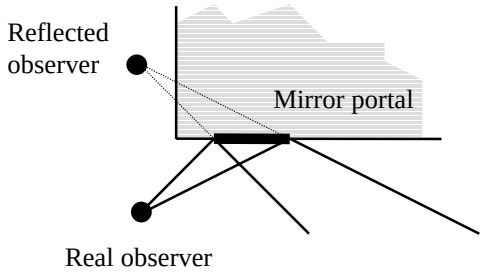


Figure 3.16

Conceptually the mirror behaves like an ordinary portal, so no workload is imposed from a mirror until it becomes visible. When the mirror is rendered, the view frustum is clipped to the boundary of the visible part of the mirror. Apart from the geometric transform on the camera matrix, mirrors are no more expensive to render than windows into ordinary rooms with complexity similar to the reflected part of the world.

An obvious concern for the implementation is that the camera viewpoint is moved to a position behind the mirror when reflected in the plane of the mirror. This means that any geometry between the ‘new’ viewpoint and the mirror should be ignored in the rendering step (figure 3.16). Depending on the chosen portal framework, this may not be a problem. If the world is fully subdivided into convex cells, the mirror will be at the boundary of the cell from which the graph traversal proceeds. Now, the recursive rendering step will only move away from any geometry between the view point and the mirror, so it need never be considered.

However, if cells are allowed to be concave, the mirror portal, the new view point and some obscuring geometry could all be inside the same cell and special measures has to be taken. In this case, it suffices to *alter* the camera’s front clipping plane to be coplanar with the plane of the mirror portal, so no additional computations are required.

3.4.2 Non-Euclidean Space

One of the most exciting properties of the portal framework is the ability to include a geometric transform at every portal (similar to the method described above for mirrors). If this transform is allowed to include arbitrary translations and rotations, the space visualised by the portal rendering will no longer be Euclidean. A simple example is a 10 meter long room with a door at each end. Each door portal links to the door at the opposite side of the room and contains a transformation that translates the camera exactly 10 meters backwards (or forwards). Now, looking out through one of the doors will correspond to looking in through the opposite door and in effect seeing an infinitely long ‘corridor’ of repetitions of the same room. In fact, an observer placed inside this room and looking through one of the doors will look into a room, where he will see himself from the back, looking through a door, etc. (figure 3.17).

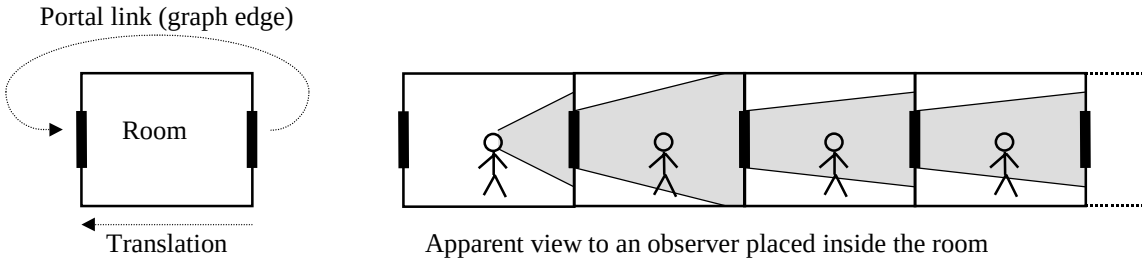


Figure 3.17

In this way, two rooms can physically occupy the same space without being mutually visible, and rooms can be recursively visible to themselves. Apart from being a funny gimmick, this feature can support several important applications. First, in a badly designed or badly defined world, where portals are unaligned or too distant to fit each other, the ‘gaps’ can be repaired by applying geometrical transforms between two portals.

Furthermore, the property that several cells can occupy the same physical space can be very important in unpredictably expanding worlds, where new cells need to be added in an area that is already crowded. An example of such a world could be a 3D visualisation of the World Wide Web, where a web hotel initially would be assigned a certain amount of space (e.g. a model of a hotel building). Now, as more users where joining the web hotel, the hotel ‘owner’ could infinitely expand the virtual space inside the hotel by simply adding new corridors, doors and rooms.

Finally, this property can be exploited for applications where the visualised data is not even defined in terms of Euclidean space in the original form, for example visualisation tools for programming environments or scientific data.

3.4.3 Sound Rendering

As mentioned in section 3.2.2, portal graphs are already successfully being used to speed up calculations involving global visibility questions such as surface-to-surface visibility in radiosity.

The fundamental theory of radiosity is to trace the propagation of energy (especially light) inside a given environment based on the idea that all surfaces either emit, absorbs or reflects energy (or combinations of these). Since the naive approach requires calculation of visible interaction between all pairs of surfaces - and even subdivisions of surfaces - this technique is computationally hard. In environments with a high degree of self-occlusion, generally few pairs of surfaces are mutually visible, and therefore a portal graph can accelerate the radiosity calculations drastically.

Placing realistic sounds into a virtual environment can be done in similar ways, by tracing the propagation of the sound energy inside the environment. These computations are even more complex than those of radiosity [HFGL98] and could be accelerated by placing them into a portal context. The fundamental idea of the portal rendering is merely reformulated to the idea that a *sound wave* can only travel between two rooms if travelling through a portal or a sequence of portals (not necessarily in a straight line, but also reflected from surfaces in the environment). Even with a relatively crude sound model, it should be possible to obtain quite vivid environments with ambience and echo effects.

4 Improvements

[Tel92] implements a method for automated subdivision and portal identification based on a BSP tree subdivision. The method requires the model to be fully subdivided into convex cells and it is biased towards architectural models in the sense that it takes advantage of coplanar structures such as floors and axially aligned walls.

[YaRa96] suggests a method applicable to more irregular models such as cavern-like environments. This method superposes the model on a uniform 2D or 3D grid, and it identifies empty cells and cells containing walls. It groups cells together to identify room-like structures and infers connectivity into a portal graph over the model. However, the method is very sensitive to the scaling of the grid both in relation to the model as a whole but also to the local size of rooms and tunnels. In addition, the memory requirement is quadratic in the scale of the grid in 2D and cubic in 3D.

In search of a less restrictive method for subdividing irregular environments, we shall make some general observations on the nature of certain subdivisions and the qualitative properties of portals. We shall then suggest a new method for automated or semi-automated construction of a portal graph for general 3D models.

4.1 Approach

Initially, we shall distinguish between two different approaches to solving the problem, namely a top-down subdivision process and a bottom-up reassembling process. We classify previous methods in this perspective, and we choose an approach for our implementation.

The choice is based on the assumptions that for irregular environments, we have to abandon the strict requirement of convexity of cells, and we have to introduce some threshold for the detail of the subdivision. Though general convexity is a desirable property for a room, the *conforming* (fully convex) subdivision of a room with ‘rough’ walls will create a multitude of completely redundant portals near the wall surfaces. Therefore, we need our algorithm to ignore details that are small compared to the overall room structure.

4.1.1 Top-down Subdivision

By ‘top-down’ we mean a process that starts by identifying a single (or several) good place(s) to insert a portal, subdividing the model into two parts. It then proceeds recursively for the subdivided parts, and stops when the model is considered appropriately subdivided. The portal graph can either be maintained during the construction process or inferred after the model has been subdivided.

The BSP method used by [Tel92] can be considered a top-down approach. The criteria for finding good splitting planes are based on axial alignment, tree balancing and minimising polygon splitting, rather than explicitly identifying ‘good’ portals. The method proceeds until all occluders are placed on cell boundaries and the model is subdivided into convex cells. For architectural models, it turns out that a sound subdivision is obtained, defining portals incident with conventional doors, windows and room partitionings. However, this method does not guarantee that the ‘good’ portals are identified as the first ones, so the success of this method is depending very much on the fact that the model is fully subdivided.

The top-down approach intuitively seems straightforward to implement, as humans can quickly identify good portal positions for a top level partitioning of a model. However, a good deal of under-the-surface analysis is involved in this ‘pin-pointing’. In essence, it is necessary to have a good perception of the general structure of the portal graph before deciding on good points to divide it.

The problem of identifying the portal graph structure of a model is analogous to finding the *medial axis skeleton*[6] for the model. [TeTe98] creates a Delaunay triangulation of 3D models and removes the cells outside the surface of the model. The Voronoi graph corresponding to this Delaunay triangulation represents the connectivity of the individual cells, and by performing thinning on this graph, a skeleton for the entire model is constructed. Effectively, a bottom-up process is applied in order to obtain the general structural description of the model, which is needed for a top-down approach.

The alternative to deciding good portal positions from a general structure of the model is to make a local analysis on the quality of several portal candidates. Such an analysis involves examining the walls close to the portal candidate. Finding good portals is a process of ‘looking for loopholes’. A loophole can be characterised by the walls on *at least one* side of the portal candidate to be expanding to something larger than the portal itself.

Such an analysis seems far from trivial and can be complicated by factors such as occluders intersecting the middle of the portal, non-convexity of the portal perimeter and non-planarity of the portal. Furthermore, the meaning of ‘local analysis’ is by no means clear, as the adjacent walls can be highly irregular in terms of both size and tessellation.

[6] A medial axis skeleton is defined such that for each point on the skeleton the distances to the *two nearest* model boundary points are equal. In effect, the skeleton is a graph giving a good description of the over-all shape of the model.

4.1.2 Bottom-up Reassembling

By bottom-up we mean a process that starts with a fully subdivided (or generally unstructured) model, and groups smaller elements together in order to obtain an overall structure of the model. The portal graph can be constructed for the fully subdivided model and then maintained during the re-assembly, or it can be created afterwards – though it seems feasible to maintain the connectivity information during the process to easily identify neighbouring cells.

The method used by [YaRa96] can be classified as a bottom-up process, as the model is first divided into 2D tiles or 3D cells, and larger rooms are then identified by clustering close cells without interleaving walls.

Methods that first divide the model far beyond the intended level of subdivision and then works backward to obtain a reasonable subdivision is what we in section 2.4 describe as ‘re-constructive portal models’. It should be noted that the concept of meta-cells as introduced by [Tel92] is also an attempt to reassemble or ‘override’ subdivision in open areas where superfluous portal definitions exist.

A relaxing constraint compared to the top-down approach is that not *all* of the portals found by the subdivision need to be ‘good’ portals. A premise for the reassembling process is that it tries to remove portals that can be classified as ‘bad’ or unfeasible. Of course, it is a requirement for the subdivision that it *also* finds the good portals that should emerge from the process, as no new portals are introduced in the reassembling process.

The reassembling process works by pruning the portal graph, i.e. removing ‘bad’ portals and collapsing cells. Although the quality of a portal can still be a difficult question to answer, the problem is now simpler than in the top-down approach. The algorithm only needs to make a choice between a given number of portal candidates rather than looking for a good position itself, and the available portal graph supplies some useful structural information for the quality measure.

Finally, we shall acknowledge that an automated process can not be expected to always find the ‘optimal’ portal positions – in irregular environments, ‘optimal’ can be a very context-sensitive term. We *do* require that the portals generally are *close* to their optimal position, but the main concern is to get a *sound* subdivision of a model into a relatively small number of rooms. In the end our method should be compared to a conforming subdivision that divides the same model into hundreds or thousands of cells, creating a large overhead of portals. In this context it is not so important if a division between two rooms is made up of one, two or three portal faces, but rather that the general occlusion properties for sequences of rooms is properly reflected.

4.1.3 Subdivision

In the light of the considerations made in the previous sections we choose to take a bottom-up approach to the implementation. Since the reassembling process is depending on the initial subdivision to find good portal positions, we shall first evaluate certain subdivision techniques.

[Tel92] successfully uses BSP subdivision for architectural models. One of the reasons that this works is that such models are very regular, showing a consistent detail level throughout the entire model. Furthermore, the subdivision is most likely to find large sets of coplanar faces, even across rooms that are not directly connected, due to e.g. the outer walls of the building and uniform room sizes. In addition, numerical algorithms are likely to be quite robust in an environment where most polygons are axially aligned and thus either co-planar or orthogonal to each other.

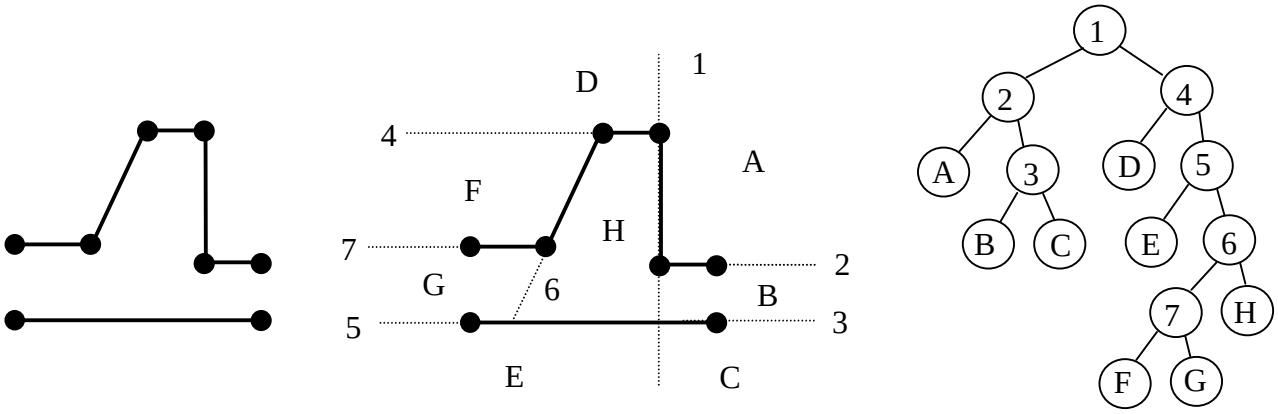


Figure 4.a

Figure 4.a shows an input model, a BSP subdivision of the model, and the corresponding BSP tree. Splitting planes and their equivalent tree nodes are numbered 1-7, and A-H are the convex regions and their tree leaf equivalents. Note that the ‘baseline’ of the model is divided by the top level splitting plane.

In irregular environments such as caverns, large sets of coplanar faces are not likely to be found. In effect, top-level splitting planes in the BSP tree will introduce global artefacts by splitting polygons all over the model. This ‘arbitrary’ splitting is bound to produce a number of numerically difficulties, for example very thin slices of a cell or very small polygons, making portal construction a numerically unstable task.

A more fundamental problem of BSP trees in this context is that they really are not very likely to find good portal positions at all. Splitting planes in the BSP tree are chosen coplanar with existing polygons in the input model, and good portals are mainly orthogonal to the structures in the input model. The reason why it works so well in architectural models is the high expectancy of 90-degree angles. The motivation for inserting a portal is usually some kind of ‘tunnel’ like structure connected to a room, but it is not the walls of the tunnel but rather the adjacent walls in the room that define the portal. The less articulated the tunnel ‘mouth’, the less quality of the portal found by the BSP tree. For a tunnel with a trumpet-like opening, the BSP tree can find no portal nearly orthogonal to the main axis of the tunnel.

[Tel92] introduces ‘free-space’ splitting planes in the subdivision process for further subdividing e.g. long hallways, i.e. finding splitting planes orthogonal to the original model polygons. However, finding good positions for such splitting planes in very irregular environments is equivalent to the analytically hard question of finding good portal positions as described in section 4.1.1.

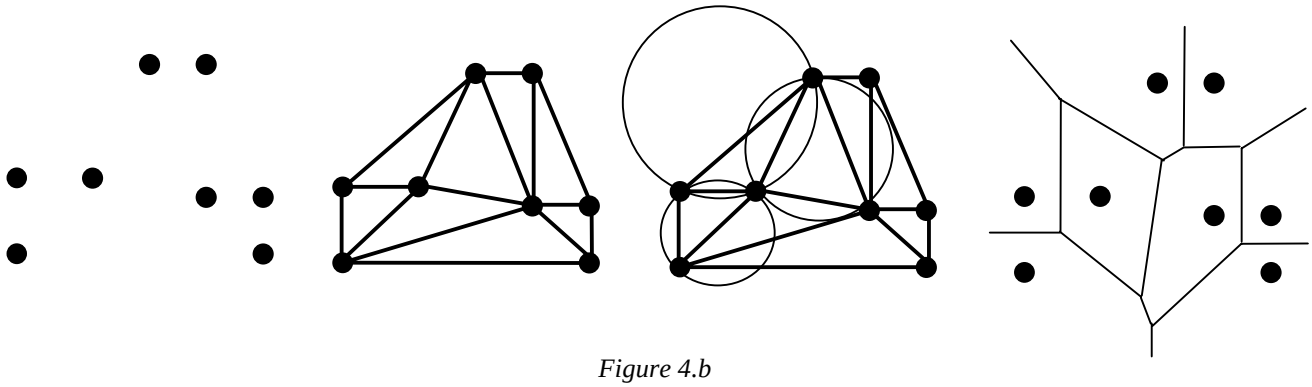


Figure 4.b

As described in section 2.4.3 other methods for subdividing a model exist. The Delaunay triangulation of a 3D model fully subdivides the volume enclosed by the model's convex hull into tetrahedrons. Each tetrahedron is spanned by four original vertices from the input model, and no other vertex exists inside a tetrahedron. Figure 4.b shows the vertices of a (2D) input model, the Delaunay triangulation and the circumcircles for some of the triangles. A property of the Delaunay triangulation is that the circumcircle for each triangle (circumsphere for each tetrahedron in 3D) contains no other vertices.

The last graph of figure 4.b illustrates the Voronoi regions for the input vertices. The Delaunay triangulation is related to the Voronoi graph of the model in such a way that each Voronoi node corresponds to exactly one tetrahedron. Each of the four triangular sides of the tetrahedron is intersected by exactly one Voronoi edge. A property of a Voronoi graph edge is that for each point on the edge, the distance to the two nearest model vertices is equal, similar to the concept of medial axis skeletons as described in section 4.1.1. The construction of the Voronoi diagram is based on the concept of 'natural neighbours', meaning that a Delaunay triangulation is much more 'local' in nature than the global properties shown by a BSP subdivision.

Practically, the Delaunay triangulation reveals few degenerate cases like the very thin, arbitrary 'slicing' seen in BSP trees. Since it is build only from vertex data and has no knowledge of model polygons, it has no problems identifying portals that are orthogonal to e.g. the general axis of a tunnel. Furthermore, it seems that due to the 'natural neighbour' concept it is very good at finding surfaces along existing model boundaries, if the model surface is sufficiently tessellated. A Delaunay tetrahedron may in fact be intersected by a model polygon, but as shall be seen in section 4.3 these cases are few. They can be handled by performing an additional splitting round, where the original model polygons are used to split tetrahedrons where necessary, ensuring that all model polygons are placed along cell boundaries.

4.1.4 Heuristics

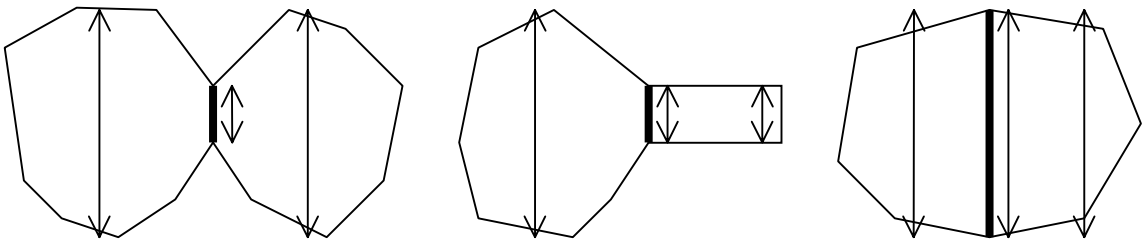
If the chosen subdivision is sufficiently detailed to capture at least all ideal portal positions, the crucial point for the algorithm becomes the heuristics used for cleaning out the 'bad' portals.

Initially, such features as convex cells and convex and planar portals seem desirable properties. In fact, these properties define an ideal goal for the final re-constructed model, but for the 'low-level' heuristics applied in each step of the re-assembling process, they may be directly misleading.

Although you start out with a collection of convex cells and you wish to end up with larger, mainly convex cells, it may not be a good approach to require all the intermediate steps to conform to these constraints. An example of this would be a circle slit into three uniform pieces; though each piece is convex, you cannot reassemble the entire (convex) circle, without temporarily allowing concave pieces to exist.

Another approach could be to make some analysis and measure based on wall polygons adjacent with a given portal. However, one should remember that initially the model is extremely subdivided, and many free-space cells are likely to exist in the middle of rooms, far from the nearest walls. Furthermore, these portals are probably the ones that the heuristic should *first* identify as good candidates for elimination.

Instead we choose a rather simple measure for the quality of a portal, and we shall then further refine this measure in the light of experimental results. We consider a portal to be of high quality if it strongly limits the view between the two cells that it connects. More specifically, we compare the area of the portal to the *projected areas* of the two adjacent cells, i.e. the area of each cell projected onto the plane of the portal. Practically, it turns out that a sufficient requirement is that at least one of the two cells is large compared to the portal (typically tunnel openings). We define our measure as the *ratio* between the area of the portal and the largest projected area of the two adjacent cells.



Good portal: Cell-Portal aspect is 4:1

Good portal: *Best* Cell-Portal aspect is 4:1

Bad portal: Cell-Portal aspect is 1:1

Figure 4.1

An important notice is that in this framework (and especially in the intermediate steps), a multitude of portals between two cells may exist, and the division between the two cells need not even be in a single plane. Practically, we sum the areas of the involved portals to obtain a 'collective' measure for the cost of collapsing the two cells.

A number of additional questions arise, such as a possible user-defined threshold for room sizes and detail levels or an absolute measure for terminating the reassembling. It also becomes an issue to carefully adjust the heuristic to allow region growing inside rooms, but prevent the growing from 'slipping through' doors. These shall be further examined in section 4.3.2.

4.2 Implementation

A test-bed for the proposed method has been implemented in Microsoft Visual C/C++, running on Windows95/98 utilising the 3D package included in the Microsoft DirectX extension.

The implementation includes a basic set of low-level routines for 3D vector operations, vertex and polygon maintenance and polygon and cell clipping.

4.2.1 Subdivision

For the Delaunay triangulation dividing the 3D model into tetrahedrons, we have used a software package dubbed ‘nnsort’ available in the public domain [Wats81].

The package reads vertex coordinates from a text file and outputs another text file with a list of 4-tuples defining the vertex indices spanning each tetrahedron. For *n* vertices, this implementation calculates the Delaunay triangulation in time $O(n^2)$, but implementations that perform 3-dimensional Delaunay triangulation in time $O(n \log n)$ or even better expected times are thoroughly described in the literature [Dwy86].

The package may create a number of collapsed, zero-volume tetrahedrons, typically representing quadrangles near the convex hull of the model. These ‘cells’ are of no value to our algorithm, and they are identified (by determining coplanarity between all four vertices) and discarded at load time.

After reading the triangulated cells, the original polygonal faces of the 3D model are loaded. Each polygon is compared to each cell. If the polygon is incident with the cell, it is clipped to the boundary of the cell. Since each cell is invariably convex in this stage, the clipping amounts to clipping the polygon to the planes of each of the boundary faces.

Now, if the polygon was not entirely clipped away, we examine if it is incident with the cell boundary or if it splits the cell. In the first case, it is attached to the cell if the (visible) front side of the polygon is facing the cell. In the second case, the cell (and any possible attached polygons) is split along the plane of the polygon, and the polygon is attached to the cell incident with the front side of the polygon. Note that splitting a convex cell by a plane ensures that the resulting cells remain convex.

The structure of the algorithm is described below. It reads the tetrahedrons output from the Delaunay triangulation and the polygons from the model. Each polygon is tested and clipped against each cell. If the *front side* of the polygon is only incident with the boundary of the cell, the polygon is attached to the cell. If it divides the cell, the cell is split, and the polygon is assigned to the new cell in front of it.

```
C:=ReadCellsFromDelaunay();
P:=ReadPolygonsFromOriginalModel();
for each polygon p in P do begin
  for each cell c in C do begin
    pclip:=ClipPolyToCell(c);
    if NotEmpty(pclip) then begin
      cfront:=ClipCellAbovePoly(c,pclip);
      cback:=ClipCellBelowPoly(c,pclip);
      RemoveCell(C,c); {since p touches c, either cfront, cback or both exist}
      if NotEmpty(cfront) then begin
        AddCell(C,cfront);
        AddPolyToCell(cfront,pclip);
      end
      if NotEmpty(cback) then begin
        AddCell(C,cback);
      end
    end
  end
end
```

Assuming that the number of edges in model polygons and the number of boundary faces in cells are bound by small constants and that few splits occur (as we shall verify later), this step runs in time *O(nm)*, where *n* is the number of polygons and *m* is the number of cells, since it has to check every polygon against every cell (the two for-loops in the algorithm above).

Using an optimised spatial search algorithm for deciding incidence between polygons and cells, the expected performance should get close to *O(n log m)*, assuming that cell and polygon dimensions are small compared to the overall model size, i.e. that each polygon will be incident with only a small number of cells. In this case, each of the *n* polygons should be able to locate the (expected) small set of incident cells in time *O(log m)*, using a sorting structure like e.g. a balanced BSP tree with *m* nodes and height *log m*.

4.2.2 Graph Construction

The first step of the graph construction is to establish connectivity between cells. At this point, cells have already been augmented with (fragments of) model polygons along their boundaries. These polygons play the role of occluders, so that no connection is established between two cells, if their shared boundary face is fully covered by occluders. Otherwise, the uncovered area(s) are considered portals and a graph edge for each portal is inserted between the two cell nodes.

Practically, the portal identification is done by assuming the entire shared (convex) boundary between two adjacent cells is a portal. For all the (convex) occluders attached to this boundary face, the portal is recursively split by each occluder edge. Each clipped region that was ‘inside’ all edges of an occluder polygon is considered fully occluded and discarded. Any remaining (convex) regions are considered portals, and graph edges (links) are established between the two cells.

When this process is finished, the boundary of each cell will be fully covered by convex polygons that are either occluders or portals, pointing to adjacent cells, except at the convex hull of the entire model. These hull faces are only incident with one cell each, so no portals are established.

The structure of the algorithm is described below. The description is simplified, because the clipping by occluders may generate a *list* of convex portal polygons, rather than a single portal. In our implementation, the bounding boxes of c1 and c2 are first checked for incidence for an early rejection of cell pairs.

```
for each cell c1 in C do begin
  for each cell c2 in C do begin
    b:=SharedBoundaryFace(c1,c2);
    if NotEmpty(b) then begin
      for each occluder o in c1 do b:=DiscardOccludedArea(b,o);
      for each occluder o in c2 do b:=DiscardOccludedArea(b,o);
      if NotEmpty(b) then CreatePortal(b,c1,c2);
    end
  end
end
```

Finally, by simple graph colouring, we identify and discard connected parts of the graph for which no cell has any attached occluders. These regions typically represent the ‘void’ outside the model boundary, or the regions ‘inside walls’ that can never be visible, as described in section 2.4.1.

The algorithm uses the following, recursive graph-colouring function. The function returns **`false`** if no cells in the region contain occluder polygons:

```
function ColourRegion(cell c)->bool begin
  if Coloured(c) then return false; {stop graph cycles}
  ColourCell(c);
  bool result:=CellHasOccluders(c);
  for each connected cell f do result:=LogicalOr(result,ColourRegion(f));
  return result;
end
```

The algorithm itself colours all connected regions of the graph and discard those that cannot ‘see’ any occluders.

```
while un-coloured cells exist in C do
  c:=FirstUnColouredCell(C);
  if not ColourRegion(c) then DeleteRegion(c);
end
```

4.2.3 Decimation Process

The first step of the decimation process is to augment all portal edges with a weight, supplying a measure for the quality of the portal. As mentioned in section 4.1.4 each pair of cells may be connected by several portal edges (see figure 4.2), so practically, each edge is assigned a weight for the *entire* cost of collapsing the two cells. Two cells may be connected by a set of small portals that individually seem good portals, although their aggregated area has an aspect ratio close to one compared to the projected area of the cells.

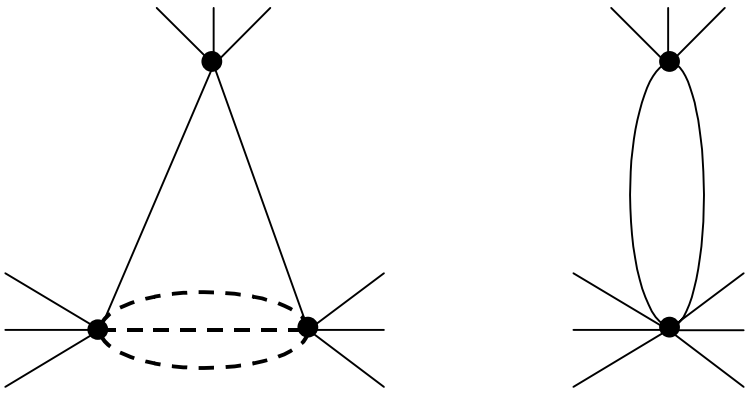
Since all initial cells are tetrahedrons, each adjacent with at most 4 other cells, and assuming that only few cells are split (as experiments will show), the number of portal edges is expected linear in

the number of cells. However, we wish to sort all portal edges according to their weight, so the overall time complexity for the initial weighting of the portals runs in time *O(n log n)* in the number of cells (adding the logarithmic factor for the sorting in each step).

Assume the existence of a function **`PortalWeight(cell c1,cell c2) -> real`** returning the collective quality of all the portals between c1 and c2, i.e. an implementation of the heuristics from section 4.1.4. Then the initial weighting of the graph is performed by the following algorithm. The procedure **`SetPortalWeights(c1,c2,w)`** sets the weight of all portals in c1 pointing to c2. All portals between two cells are assigned the same weight.

```
for each portal edge e do e.weight:=-1;
for each portal edge e do
  if e.weight<0 then begin
    c1:=e.cell1; c2:=e.cell2;
    e.weight:=PortalWeight(c1,c2);
    SetPortalWeights(c1,c2,e.weight);
    SetPortalWeights(c2,c1,e.weight);
  end
```

The actual decimation process now amounts to finding the portal with the lowest weight, and collapsing the two cells that it connects. Practically, this process involves a number of steps. First, the portal polygons (that also represent the graph edges) are removed from the cells. Then, all remaining boundary, occluder and portal polygons are moved to one of the cells and the other cell is discarded. Finally, the references in all portal edges between the new cell and its neighbours are updated, and all the weights for these edges are recalculated using the new (increased) size of the collapsed cell.



Collapsing two nodes of the portal graph

Figure 4.2

We assume that all *portal edges* are sorted in e.g. a traditional heap data structure with the lightest weight on top of the heap. Then structure of the decimation algorithm is as described below. It identifies the best pair of cells to collapse, removes all portals between the two cells, collapses the cells and updates all neighbouring edges.

```
while Heap[0].weight<threshold do
   c1:=Heap[0].cell1; c2:=Heap[0].cell2;
   for each portal edge e between c1 and c2 do HeapRemove(e);
   <<make all portals from c2's neighbour cells point to c1 instead>>;
   <<move all c2's portals and occluders to c1>>;
   Remove(c2);
   for c3:=each neighbour of c1 do begin
     w:=PortalWeight(c1,c3);
     SetPortalWeights(c1,c3,w); {including reordering in heap}
     SetPortalWeights(c3,c1,w); {including reordering in heap}
   end
end
```

The time complexity for each decimation step is determined by the number of edges that need to be updated. In the beginning, the number of edges from each node is rather small, but in some intermediate steps, many edges may lead from a single node. An interior cell of a room that has been grown by collapsing many free-space cells, but has not yet expanded all the way to the walls will be connected to a multitude of small cells along the room boundary. Considering the sorting of the updated portal weights, the time complexity for each decimation step has an upper bound of $O(n \log n)$ where n is the number of updated edges, because each edge has to be updated and the logarithmic factor is added for the sorting of the updated edge. Since the target models for this algorithm are densely occluded environments, i.e. models consisting of many, relatively small rooms, we expect this n to be substantially smaller than the total number of edges in the total model graph. In addition, results from the experiments indicate that the algorithm spends far more time in the subdivision and graph construction process than in the decimation process.

A crucial question for the decimation process is when to stop. In the test environment, we have implemented the possibility to view the portal graph during the decimation process. It is possible to spool through the decimation process or single step through the cell collapsing. Meanwhile, one can observe the portal weight of the next cell pair to be removed. In this context, we have implemented a semi-automated portal construction. It seems reasonable to let the removal of the first large amount of cells be fully automated, by carrying on with the decimation process as long as the portal-cell aspect ratios are close to 1. Only for the last few portals, it seems necessary with an interactive supervision, possibly with the added feature of overruling algorithm choices by backtracking and manually adjusting portal weights.

4.3 Results

The 3D models used for testing are described in a simple, text-based mesh structure, supplying a list of vertex coordinates and lists of vertex indices defining the polygonal surfaces.

Several of the examples are focused on a case with four unaligned rooms connected by small corridors. The motivation for choosing this model is that it represents a 'hard case' for the typical BSP approach. The BSP subdivision at an early stage is forced to make a global splitting along a wall from one room, forcing another (not even neighbouring) room to be unjustifiably split.

4.3.1 Optimised Room Model

This is the model of the four rooms represented by the fewest possible number of triangles. The model has 64 shared vertices (8 for each room and each corridor) and 96 triangles. The Delaunay triangulation produces 160 tetrahedrons, and the splitting of model polygon introduces additional 18 cells, an increase of around 11%. The graph colouring removes 74 ‘outside’ cells.

Figure 4.3 shows the undecorated input model and the model with all the (bold) outlines for the portals found by the graph construction step. Figure 4.4 shows an intermediate step from the decimation process, where some portals have been removed, and the final configuration where the decimation process has been stopped.

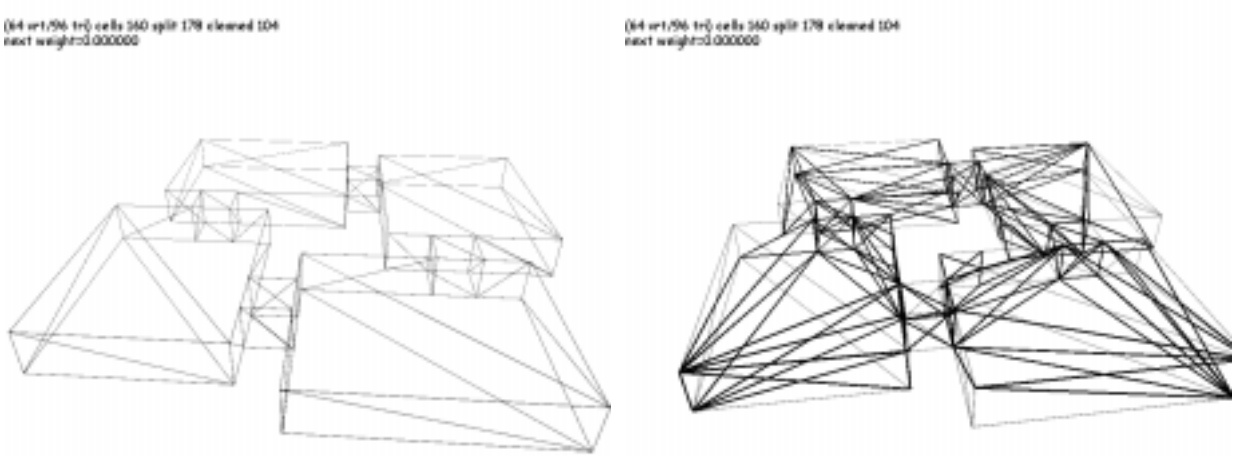


Figure 4.3

The decimation process starts by removing portals with aspect ratios close to one, primarily portals internal to rooms. After removing half of the portals, the portal concentration is very high around the corridors connecting the larger rooms.

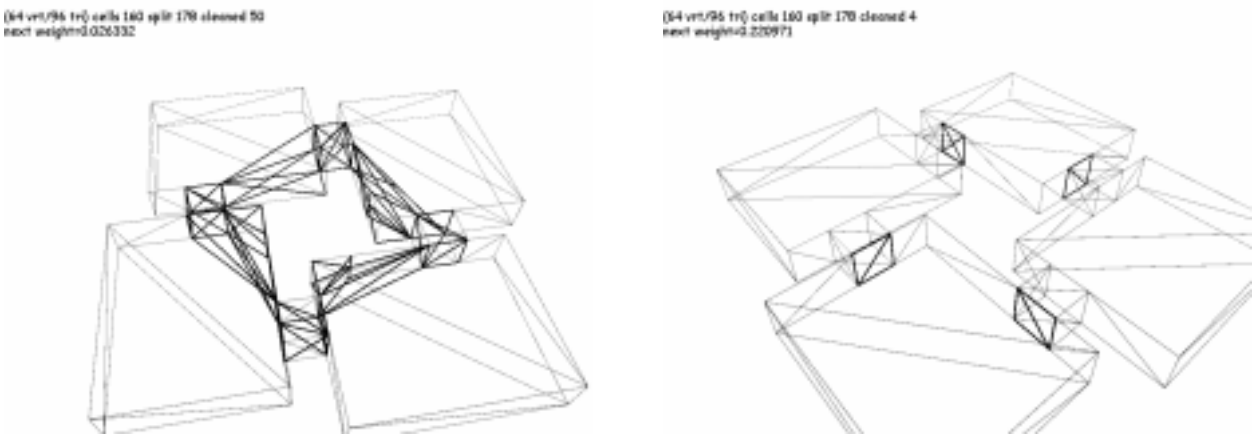


Figure 4.4

The last four portals to be removed by the decimation process are, as desired, portals separating the four major rooms. At this point the point the cell-portal aspect ratio suddenly jumps from around 2 to 5. Since the *ratio*[7] is independent of model scaling, it seems feasible to introduce a fixed termination threshold around 3 or 4 (depending on the desired portal granularity) for fully automated portal classification.

It is interesting to notice that the algorithm only finds portals on one side of the corridors. The reason is that for such short corridors, it is difficult for the heuristics to determine if it is merely growing into a cavity of a wall or if it is entering a tunnel. However, it correctly infers that collapsing two of the large rooms through such a small portal is costly, and therefore identifies the room separations as the last portals to be removed.

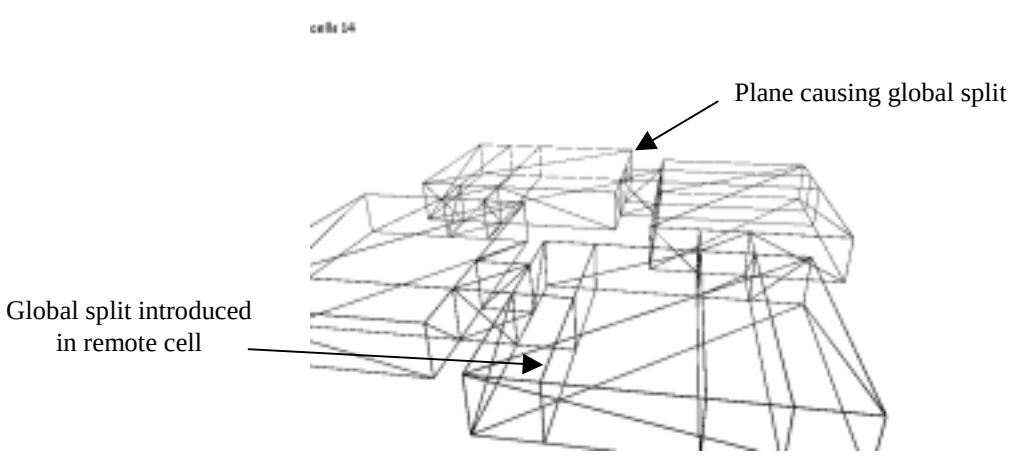


Figure 4.5

Another interesting feature is that the Delaunay triangulation shows the 'local' behaviour we were looking for, by not introducing splitting *across* room boundaries, as the BSP approach does (figure 4.5). For a very regular model like this, the BSP subdivision is fairly well-behaved, introducing only one 'global' split across rooms, but for irregular models these global splitting planes are introducing numerous unnecessary, odd-angled separations throughout the model.

4.3.2 Tessellated Room Model

This model represents the same four rooms, but all the polygons are tessellated to have roughly the same size across the model. The model contains 200 vertices and 400 triangles. The Delaunay triangulation generates 523 cells, of which none (!) are split by model polygons, and the graph colouring removes 125 exterior cells. Again, figure 4.6 shows the original model and subdivision, figure 4.7 shows an intermediate step and the final result.

[7] Note that the weight shown in the screenshots is not the aspect ratio, but rather the weight of the *refined* heuristic as described in the section. 4.3.2

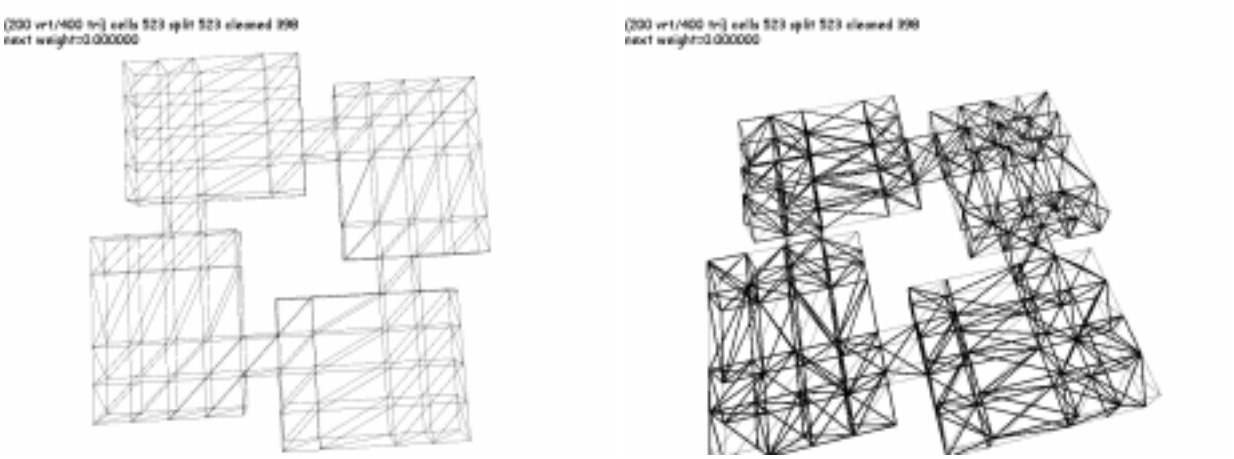


Figure 4.6

Using the original heuristic, it turned out the cell growing easily ‘slipped’ through doors and quickly collapsed all four major rooms into one single cell before pruning the smaller cells around the room boundaries.

Since all the initial cells now have roughly the same size as the cells of the corridors, it becomes inherently hard for the algorithm to distinguish the corridors from mid-room cells. Since the heuristics assumes no knowledge of occluder polygons, it finds no less reason to slip through one of the ‘real’ corridors than wandering through an ‘imaginary’ corridor made up of adjacent mid-room cells.

Based on a few experiments, the heuristic was adjusted to include a measure of the size of the portal. The cell-portal aspect ratio was now divided by the square root of the projected cell area (using the square root of the area as an indication of the ‘average’ scale of the cell). In this way, larger portals are favoured over smaller ones, allowing for growth inside large rooms to be self-accelerating until it reaches the boundary of the room.

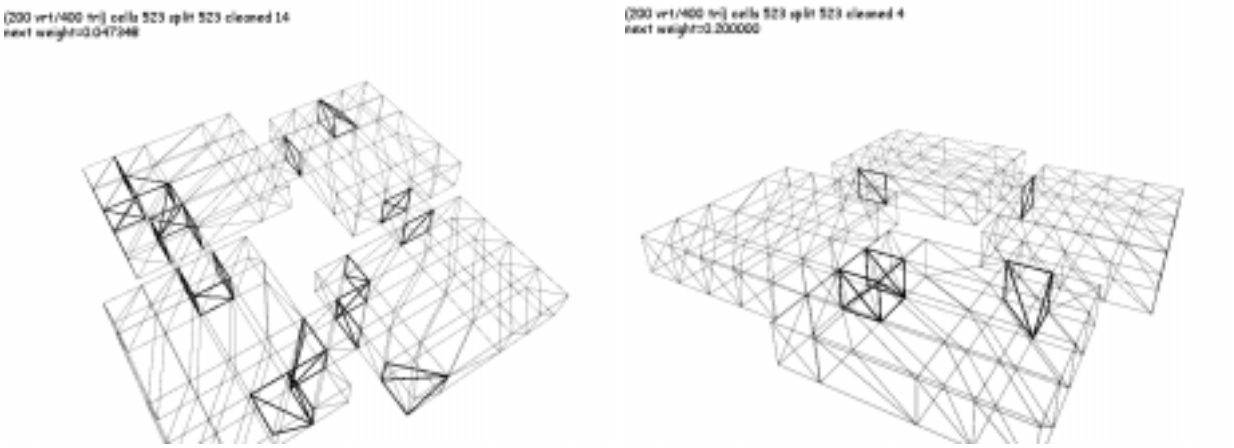


Figure 4.7

If all cells initially are of uniform size, the algorithm is statistically likely to pick a starting point inside a large room, since these have far more cells. The self-accelerating nature of the new heuristic (favouring large portals) will make the cell growing expand inside such a room. However, the algorithm may still pick a starting point inside a corridor. In this case it will eventually grow into a neighbouring room and start expanding here. Though this type of environment renders the algorithm less focused and more prone to ‘slip through’ doors, it appears from this experiment that the heuristic - on a large scale – is still stable and converges towards classifying separations between large rooms (figure 4.7).

As can be seen from figure 4.7, the algorithm finds less optimal portal positions for two of the corridors, but seen from a large-scale occlusion culling perspective, the fundamental room separation is still good.

The drawback of introducing an absolute measure of the portal scale of course is that the ‘termination threshold’ for the decimation process now becomes model scale dependent and thus, at the moment, requires some kind of interactive supervision.

4.3.3 Distorted Room Model

To examine the behaviour of the algorithm in less regular environments, the four-room model is now distorted in order to reduce the number of coplanar, orthogonal and axially aligned faces substantially. Each of the four rooms is rotated in different angles around all three axes, and furthermore some of the corner vertices of each room are displaced in arbitrary directions.

The distorted model contains 106 vertices and 96 triangles. The increase in the number of vertices over the first example is due to numerical inaccuracies, where some vertices are no longer identified as shared between multiple faces. The Delaunay triangulation generates 299 cells, of which 112 or 37% are split by model polygons. The graph colouring removes 251 exterior cells, leaving 160 ‘active’ cells, or only 54% more cells than for the first, regular model. The high number of removed cells indicates that the main increase in cells is outside the model interior or close to the boundary of the convex hull.

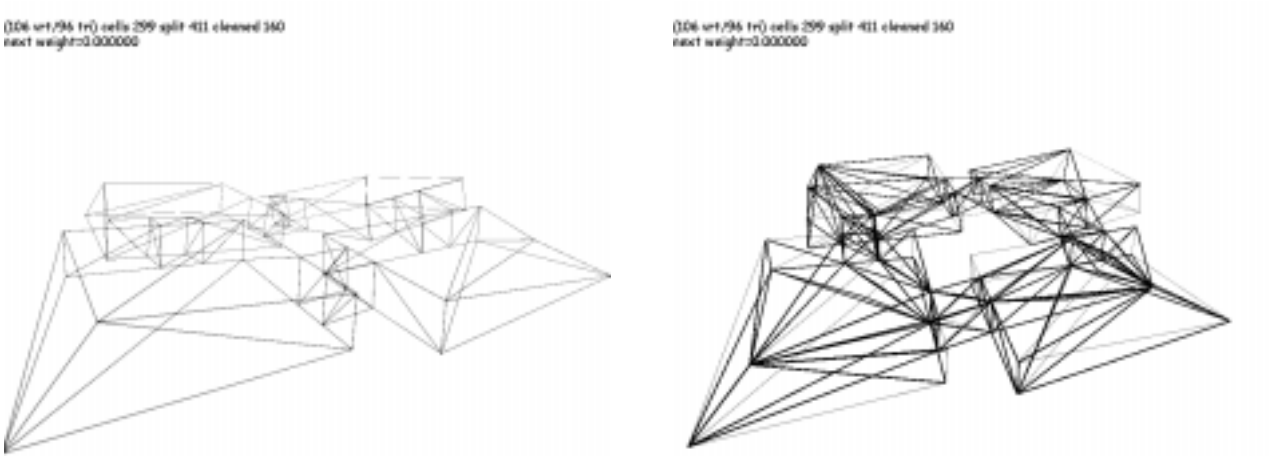


Figure 4.8

For comparison, we also implemented a method for BSP subdivision (minimising polygon splits). For the distorted model the number of cells generated by this subdivision increased by 143% (from 14 to 34) compared to the first, regular model. More important, most of the subdivisions produced by the BSP turned out to have little relevance to the ‘natural’ partitioning of the model into rooms and corridors. Some of them generated potentially ‘hard cases’ such as very thin cells or polygons.

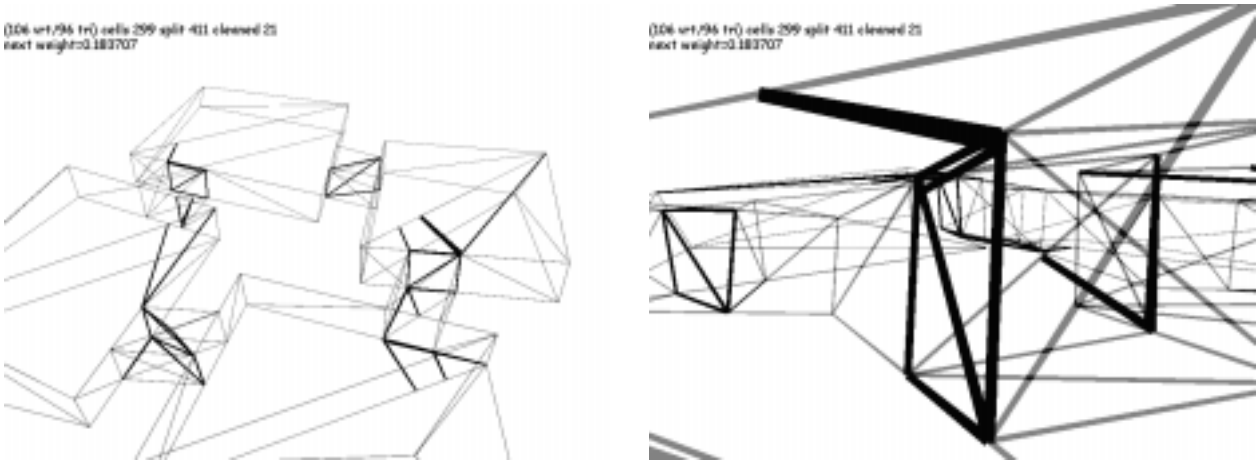


Figure 4.9

The decimation process essentially proceeds as usual, unaffected by the skewed angles. It initially removes large portals across open spaces and keeps high concentrations of portals close to the corridors. However, it runs into troubles with a set of ‘hard case’ cells stemming from the Delaunay triangulation, namely a set of thin, spike-shaped cells (figure 4.9). Due to the degenerate shape of the portals on these cells, the cell-portal aspect ratio becomes very high and the heuristic fails to remove these obviously irrelevant cells.

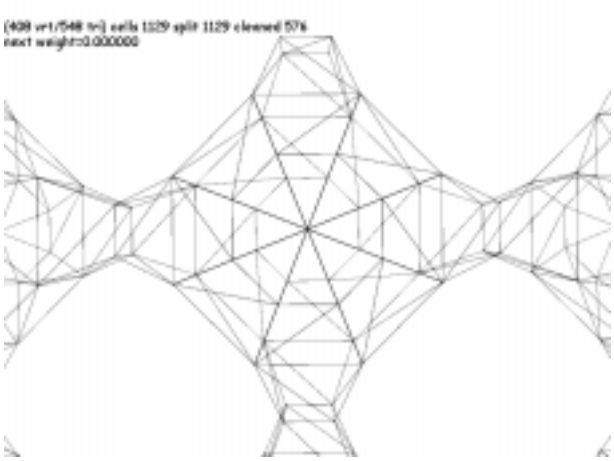
As can be seen from figure 4.9, the decimation process correctly isolates portals separating the 4 major rooms, but it has a problem with 17 ‘spikes’ that are removed as the very last cells in the process. In figure 4.9 (right) is a close-up of one of the spikes.

In order to enable the algorithm to remove spike-shaped cells earlier in the process, the heuristic was modified to include a measure of the *volume* of the cells connected by a portal. The idea was to devalue the importance of cells that were extremely small compared to their neighbours, but it turned out that this strategy made the decimation process very unstable. As the cell growing achieved cells of a certain size, *any* small, adjacent cell would be considered unimportant, and the growing would slip through corridors in a self-accelerating attempt to consume the entire model.

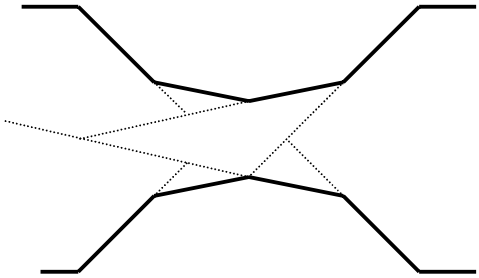
No further attempts to deal with the spikes were conducted, but we remain confident that the unique nature of the spike cells can be identified and eliminated by adding a few extensions or special checks to the heuristics. Again, the scale of the model becomes relevant; In order to decide whether a cell is degenerate or not, it is necessary to introduce a metric threshold for the validity of cells.

4.3.4 Caverns

To verify the claim that our technique in irregular environments will be able to find better portals than a BSP subdivision technique, a cavern-like six-room model with non-articulated transitions between rooms and corridors was created (figure 4.10).



The shape of each of the six rooms in the model



Subdivision (and portal candidates) obtained by a BSP approach

Figure 4.10

This cavern model contains 408 vertices and 548 triangles. The Delaunay triangulation generates 1129 cells, of which none (!) are split by model polygons. The graph colouring removes 553 exterior cells or 49%, leaving 576 ‘active’ cells.

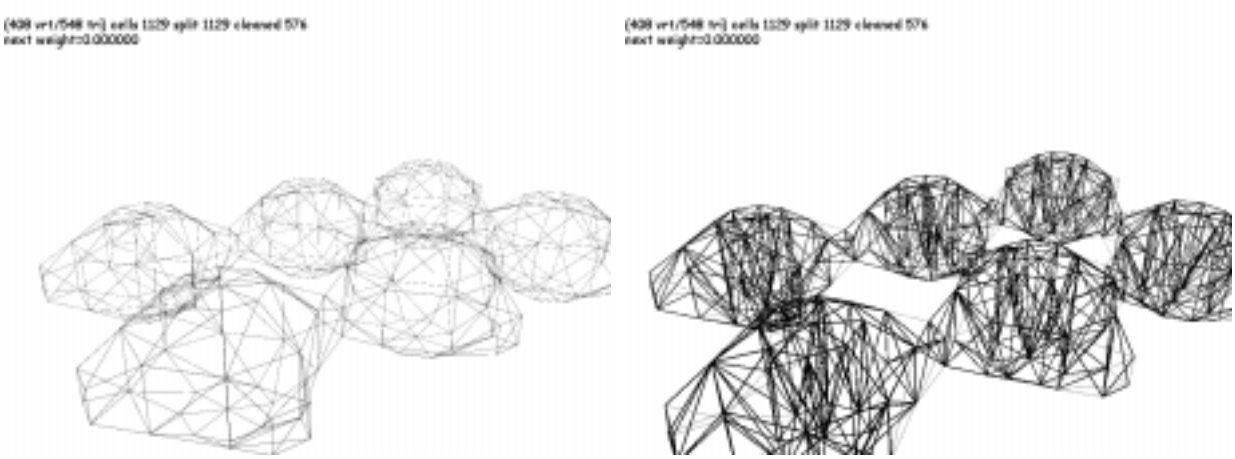


Figure 4.11

The decimation process tediously removes portals until it ultimately identifies exactly the 7 portal positions necessary to separate the 6 major rooms (figure 4.12). Furthermore, it identifies portals that are orthogonal to the main axes of the ‘corridors’, something that could not have been achieved with an ordinary BSP approach (figure 4.10).

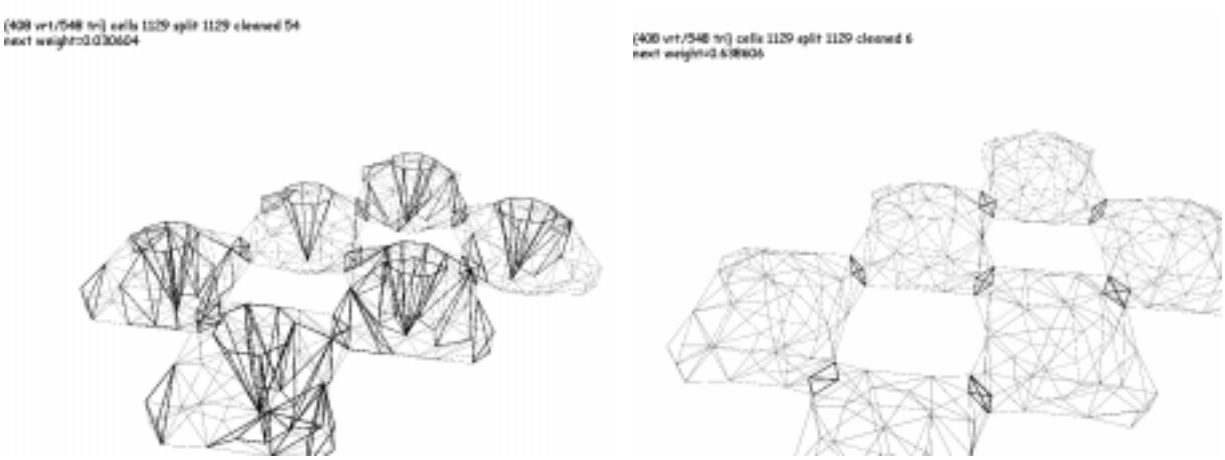


Figure 4.12

Finally, to verify the robustness of the method with respect to axial alignment, the entire model was rotated 7, 8 and 9 degrees around the x, y and z axis respectively. As for the distorted four-room model, the number of vertices increased slightly due to round-off errors, but curiously the number of cells produced by the Delaunay triangulation was slightly smaller than for the axially aligned model. After splitting and removal of exterior cells, the number of cells was equal to the original model.

The subdivision and decimation process was repeated for the rotated model, yielding exactly the same 7 portals as found for the original model.

4.3.5 Other Tests

A number of further experiments was conducted for increasingly irregular models. For several of the models, a noise-function was applied to the vertex positions. A special room was created with a highly tessellated and perturbed wall and a table and a chair was inserted in the room. For all cases, large amounts of degenerate or otherwise troublesome cells were generated.

Initially, the largest mid-room portals were correctly removed, but the substantial amount of portals connecting the degenerate cells was *not* correctly identified by the current, simple heuristics as bad portals. Similar to section 4.3.3, the 'good' portals were removed prior to the collapsing of the spike-shaped cells, concentrated around the irregularities.

We conclude from these tests, that the heuristic is not yet stable enough as irregularity grows. Further experiments, balancing of the heuristics and test for special cases must be applied to the heuristic, before it is able to handle certain classes of input models.

4.4 Evaluation

In this chapter we have developed a new method for subdividing models and shown that it is capable of achieving a sound portal subdivision for irregular environments.

Due to the 'natural neighbour' concept of the underlying Delaunay triangulation, the method achieves more natural subdivision than can be achieved with BSP trees, i.e. with less undesirable splitting across the model. Additionally, in irregular environments the method is likely to find more optimal portal positions than BSP subdivision is.

For initial research, we implemented a BSP subdivision method that performed subdivision into cells as well as portal and occluder classification (but not graph structure). Comparing the performance of this method to the Delaunay approach on our experimental models, show that the Delaunay approach is certainly not more computationally heavy than that of the BSP approach.

Our method quickly showed good results, using a very simple heuristic, that is not even optimally precise, since it uses crude measures such as bounding rectangles and boxes for area and volume calculations. Nevertheless, for relatively well-behaved rooms, it seems to be stable and converge towards intuitively sound subdivisions. This renders the technique very promising and subject to further improvements in the heuristic, based on more extensive experiments. As can be seen from the last round of tests, this is necessary in order to make the heuristic cope with the 'special cases' of more irregular models.

The major, general problem of the proposed technique is the termination criteria for the re-assembling process. Initially, we showed that for a scale-independent heuristic it was possible to select a *general threshold* for the cell-portal aspect ratio, in effect deciding the granularity for the graph description of a model. However, we were forced to introduce a size-dependent factor into the heuristics, making the threshold model-specific. When improving the heuristic, one should strive to

obtain scale-independent measures in general, possible by *normalising* absolute values relative to corresponding values at other points in the models.

5 Conclusions

5.1 Achievements

The two major achievements of this thesis has been

- to verify that a portal framework without heavy pre-computation can be successfully used to visualise a large-scale world in real-time, even on low-end platforms (chapter 3), and
- to propose a new general technique and approach towards automated portal classification (chapter 4).

A primary concern has been to describe scaleable techniques, i.e. techniques applicable to virtual worlds of constantly increasing size.

It is generically difficult to obtain useful theoretical measures of these techniques. The theoretical upper bounds on storage usage and performance may often be pessimistic, although the expected and practical performance of a technique is optimistic. In real target models, many of the involved factors such as edges per polygon, surfaces per convex hull and neighbour relations between two cells are bound by small constants. As the size of the world increases, these values remain inside a domain dictated by the detail level and general nature of the model. An example could be the maximum length of portal sequences, reported by [Tel92] to be around 10 for a specific case with thousands of cells.

This means that factors such as ‘maximum neighbour relations per cell’ can often be substituted by (model dependent) time constants in the analysis of expected execution times. Indeed, as e.g. the number of average neighbour relations per cell for a world increases, the world moves out of the domain of ‘densely occluded environments’ and becomes irrelevant for the focus of this thesis.

Comparing the performance directly to brute-force techniques may also be of little relevance. A premise for our target worlds is that the maximal length of sight lines is small compared to the world size. Therefore, it is easy to obtain an acceleration factor of 100 (or 200 by simply doubling the size of the world!). Practically, for such large worlds, no one would submit the entire world model to the graphics pipeline in a brute-force attempt, but rather introduce e.g. a fixed visibility horizon. In this case, the two approaches are not directly comparable anymore.

However, we shall argue that the techniques described in this thesis really *are* scalable. Due to our assumption of densely occluded environments, our target worlds have no *global* cell-to-cell dependencies, and the number of neighbouring relations for a cell is relatively small. Therefore, the operations performed in each step of the decimation process are of a local nature and bound by a small constant. The sorting of portal weights introduce a logarithmic factor, making the decimation process perform in expected time $O(n \log n)$ in the number of initial cells, since the number of portal

edges to be removed is bound by *n* and each step requires sorting of only a few, local edges (introducing the logarithmic factor). Furthermore, experiments show that most of the initial cells have aspect ratios equal to, or very close to 1, making it possible to limit the sorting to a small subset of the initial portals.

The crucial point for the portal classification therefore becomes the initial cell subdivision and establishing connectivity information for the cells. Our decimation framework could work on top of any subdivision, but the proposed Delaunay triangulation exhibits valuable properties to the subdivision and can be performed effectively in time $O(n \log n)$ in the number of input vertices [Dwy86]. The naive algorithm for constructing cell-to-cell connectivity runs in time $O(n^2)$ in the number of cells (since each cell pair has to be compared), but interleaving the process with the triangulation process or applying proper spatial sorting (again assuming cells are small compared to world size), should make the process run in expected time $O(n \log n)$ as described in section 4.2.1.

5.2 Application Areas

The general development in computer performance indicate that the broad base of personal computers is closing in on traditional work stations, making advanced graphics techniques generally useable in a wider area.

As common access to Virtual Reality platforms is becoming easier, the number of application areas for this type of visualisation techniques is expected to grow. As we have shown in chapter 3, the techniques are already applicable to entertainment and simulation purposes on low-end platforms.

As Virtual Reality and visualisation of synthetic worlds is slowly maturing, it can be expected that these techniques will be widely used in design processes in architecture, engineering and design of 3D structures in general. Generally, these techniques will generate shorter prototyping cycles, allowing for faster or better design processes, similar to our experience described in section 3.3.

5.3 Future Research

Our proposed technique for automated or semi-automated portal classification seems very promising, since it obtains good results even with a quite simple heuristic. It still needs adjustments and stabilising from a wide range of practical experiments, and it would be interesting to see how close one could get to high-quality, fully automated portal classification.

A possible extension could be to introduce a backtracking step, where the inferred world structure is used in analysing and readjusting portal positions and probably introducing further segmentation of e.g. tunnels.

Another approach could be to perform multiple subdivision/decimation rounds, letting only the highest quality portals survive between passes until the inferred world structure has stabilised.

Bibliography

[Air90] John M. Airey. *Increasing Update Rates in the Building Walkthrough System with Automatic Model-Space Subdivision and Potentially Visible Set Calculations*. Ph.D. thesis, UNC Chapel-Hill TR #90-027, 1990.

[ArKi89] J. Arvo and D. Kirk. *A Survey of Ray Tracing Acceleration Techniques*. In Glassner, An Introduction to Ray Tracing, Academic Press, 1989.

[Ben75] J. Bentley. *Multidimensional Binary Search Trees used for Associative Searching*. Communications of the ACM, 18, 1975.

[CoTe97] S. Coorg and S. Teller. *Real-Time Occlusion Culling for Models with Large Occluders*. In Proc. 1997 ACM Symposium on Interactive 3D Graphics, 1997.

[Dwy86] R. A. Dwyer, *A Simple Divide-and-Conquer Algorithm for Computing Delaunay Triangulations in O(n log log n) Expected Time.* Proceedings of the 2nd Annual ACM Symposium on Computational Geometry 276-284., 1986

[FKN80] H. Fuchs, Z. Kedem and B. Naylor, *Visible Surface Generation by A-Priori Tree Structures* . Conf. Proc. of SIGGRAPH '80, 14(3), 124-133, Jul 1980.

[Fol90] J. Foley, A. van Dam, S. Feiner and J. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley, 1990.

[Fun93] Thomas A. Funkhouser. *Database and Display Algorithms for Interactive Visualization of Architectural Models.* Ph.D. thesis, UC Berkeley

[Her87] John E. Hershberger. *Efficient Algorithms for Shortest Path and Visibility Problems*. Ph.D. thesis, Stanford University, 1987.

[HFGL98] J.. Hahn, H. Fouad, L. Gritz and J. W. Lee. *Integrating Sounds and Motions in Virtual Environments.* MIT Presence Journal, vol. 7 issue 1, February 1998.

[Jon71] C. B. Jones. *A New Approach to the 'Hidden Line' Problem.* The Computer Journal, vol. 14 no. 3, pp.232-237, 1971.

[Lin96] P. Lindstrom, D. Koller, W. Ribarsky, L. Hodges, N. Faust and G. Turner. *Real-Time Continuous Level of Detail Rendering of Height Fields*. TR #96-02, Visualization and Usability Center, Georgia Institute of Technology, 1996.

[Lue95] David P. Luebke and Chris Georges. *Portals and Mirrors: Simple, Fast Evaluation of Potentially Visible Sets*. In Proc. 1995 ACM Symposium on Interactive 3D graphics, 1995.

[TeHa93] S. Teller and P. Hanrahan. *Global Visibility Algorithms for Illumination Computations.* In Computer Graphics Proceedings, Annual Conference Series, 1993.

[Tel92] Seth J. Teller. *Visibility Computation in Densely Occluded Polyhedral Environments.* Ph.D. thesis, UC Berkeley TR #92-708, 1992.

[TeTe98] M. Teichmann and S. Teller, *Assisted Articulation of Closed Polygonal Models*, in Prof. 9th Eurographics Workshop on Animation and Simulation, 1998.

[Wats81] D. F. Watson, *Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes*, The Computer Journal 24(2), p. 167-172, 1981.

[YaRa96] R. Yagel and W. Ray, *Visibility Computation for Efficient Walkthrough of Complex Environments*. Presence, 5(1): 1-16, 1996.

[ZMHH97] H. Zhang, D. Manocha, T. Hudson and K. Hoff. *Visibility Culling using Hierarchical Occlusion Maps*. UNC Chapel-Hill. Proceedings of ACM SIGGRAPH, 1997.