MASTER

Portal processing in 3D-environments: visibility determination and other applications

Dirkse, P.

Award date:
2002

TECHNISCHE UNIVERSITEIT EINDHOVEN

Department of Mathematics and Computer Science

MASTER'S THESIS

Portal Processing in 3D-Environments: Visibility Determination and Other Applications

by
P. Dirkse

Supervisor :	dr. ir. H .v.d. Wetering
Advisor:	L. Lunesu

Eindhoven, April 2002

Chapter 1: Introduction

Even using today's fast hardware graphics accelerators, displaying large and complex virtual worlds is still a compromise between maximizing detail and maintaining an interactive frame rate. Even though graphics accelerators have improved scene processing and rendering speeds tremendously (from the Silicon Graphics GT, the first system to break the 100,000 polygons/second mark in 1988 to the Nvidia GeForce4 chipset achieving 136,000,000 polygons/second[1] of 2002) and will continue to do so in the future, this delicate balance will exist for many years to come: designers/ modelers will always find new ways of adding detail to a world that will bring the rendering hardware/software on its knees. Because more detail adds to a more realistic experience, a lot of research into speeding up scene processing and rendering has been done in the past 30 years.
One of the areas of research is the field of *visibility determination*: which polygons are visible to a viewer and which aren't? This has yielded various techniques ranging in complexity from simple frustum culling to the use of complex tree structures like BSP-trees (these will be discussed later in this chapter).

A relative new technique gathering much attention these days is visibility determination using *portals*, which are openings in the major opaque features of a virtual world that narrow a viewers field of view. By identifying these portals, a (usually) small superset of the actual visible polygons can be determined at (pre-) run-time allowing for faster processing and more detailed and complex worlds. This technique is called *portal processing*, and as this thesis will show introduces some properties that can be used for a variety of purposes, like dynamic memory management and 'special effects' (*special FX*) features like mirrors.

This thesis was written as a master thesis for the department of Mathematics and Computer Science at the Technische Universiteit Eindhoven, and presents the results of an in-depth study into the applications of portal processing in *vrSS*, a proprietary virtual reality toolkit created by Eindhoven based company Mondo Bizzarro B.V. Their goal was to have a portal processing algorithm implemented in their engine, but also to examine how this technique could be used in other areas to help improve performance.

Chapter 1 continues with a short introduction of some of the better known visibility determination techniques developed over the past years and provides the reader with some information about pseudo-code syntax used throughout the thesis. Chapter 2 introduces vrSS, the C++ virtual reality toolkit used that will be extended with code for portal processing and other applications of portals. Some of the functionality and classes proprietary to the toolkit are explained briefly, as to familiarize the reader with the toolkit before it is expanded to allow for portal processing.
Chapter 3 explains the details of real-time visibility determination using portal processing and its implications on the construction of a virtual reality model. By example of the implementation of the portal processing algorithm, the reader is provided with an insight in the intricacies and pitfalls of implementing a portal processing algorithm. Chapter 4 discusses dynamic cell management: due to the partitioning of a world into cells, the basis has been provided for loading and displaying a large and complex virtual world using several more manageable chunks (the cells) that can be loaded/unloaded into/from program memory when required. Many situations, each requiring a different approach to loading/unloading of cells, exist therefore making it impossible to give one memory management solution that is optimal in all situations. Chapter 4 discusses some commonplace situations in which memory management is required and gives some hints on how to solve the problem in these situations.
Until now only static scenes have been considered, but most virtual reality applications/games will also have many dynamic objects. Chapter 5 explains how dynamic objects can take advantage of the improved visibility determination introduced by portal processing, and solves the problem introduced by objects whose influence extends beyond the boundaries of their physical parent cell (e.g. lights).
Chapter 6 discusses how portals can be used to add visually interesting effects, like mirrors, monitors, reflecting windows etc. to a scene. Chapter 7 discussed pre-run-time visibility determination using portals: Using pre-computed visibility sets, processing speeds can be increased even further, but they can also be used to implement smarter memory management or as heuristics on scene complexity.
Finally, chapter 8 summarizes the findings of this thesis and discusses the measured results of some test applications. The thesis ends with a discussion on further possible expansions and applications of portals and the incorporation of other techniques in a portal based environment.

[1] Thus greatly exceeding the level of improvement predicted by Moore's law, which predicted the number of polygons/second to be around the 52,000,000 mark in 2002.

1.1 Visibility determination

Before introducing the portal processing technique, a short history of other techniques used for approximate and exact visibility determination is discussed. Most of these techniques are used by vrSS for visibility determination (*view frustum culling, back-face culling, Z-buffering* and *bounding volumes*) while BSP-trees come to mind when thinking of viable alternatives to portal processing (chapter 8 will briefly discuss the advantages/disadvantages of BSP-trees vs. portal processing).

View frustum culling (see Fig. 1) is one of the oldest and simplest visibility determination techniques and is used in probably every 3D engine in existence today. Using a pyramid (sometimes a cone or tetrahedron) called the *frustum* to represent the non-occluded field of view of the viewer, all polygons not intersecting the frustum can be ignored as they are not visible. View frustum culling is an effective way to select a superset of visible polygons, which must then be processed further to determine exact visibility (e.g., the dark gray area represents an area occluded from the viewer by polygon *p* but inside the frustum).

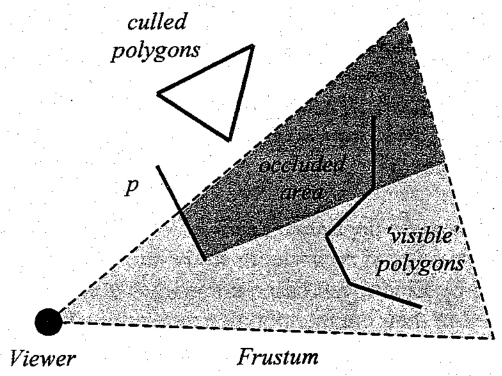


Fig. 1 View frustum culling.

Another common technique is *back-face culling*, where polygon faces are treated as one-sided entities that can be culled when facing away from the camera.
Bounding volumes are used to quickly determine the visibility of the set of polygons encompassed by it: if the bounding volume does not intersect the frustum, neither will any of the polygons it encompasses.

The techniques presented so far require the use of an additional algorithm for determining exact visibility. Today the most commonly used technique for this is the *Z-buffer* [10] where, if the distance between a pixel to be drawn and the viewers position is no further than the distance stored in the Z-buffer, the pixel is drawn, otherwise rejected. Today even the simplest of 3D graphics accelerators support Z-buffering, but back in 1974 when Z-buffering was invented (and up until only a few years ago), the cost of the memory for storing the z-information was considerable.
Unfortunately, Z-buffering cannot handle complex and heavily occluded scenes alone, for this extra visibility information is required.

An approach providing this extra information AND exact visibility is the use of BSP-trees[2], first introduced by Fuchs et. al. [9] in 1980. BSP-trees spatially partition a static scene by choosing separating planes that split any polygon intersecting them into two separate parts. This allows producing an exact back-to-front (or front-to-back) ordering of polygons from any viewpoint. The major advantage of BSP-trees is that exact visibility of *n* polygons from any viewpoint can be determined in $O(n)$ time, which can even be improved using bounding volumes that encompass all splitting planes in a sub tree: if the frustum does not intersect with this volume, then neither will a polygon in the sub tree.
The disadvantages of BSP-trees are that they are time-consuming to build (they usually cannot be computed at run-time) and that the splitting operations needed to construct the tree may generate $O(n^2)$ [12] polygons, severely affect its rendering performance. Finding the optimally balanced BSP-tree is an NP-complete problem so approximations must be made, and incorporating dynamic objects into the scene requires maintaining Z-information. Regardless of these

[2] Please note that octrees [11] were left out of this summary because they are a special case of BSP-trees and thus have equal properties.

drawbacks, BSP-trees have been (and still are) very popular in 3D graphics and found some of their first practical commercial uses in the Id Software games DOOM and Quake.

The subject of this thesis, portal processing, is in effect nothing more than using *dynamic frustum narrowing* when rendering a scene. By spatially subdividing a static scene along its major opaque features into cells, and then converting the transparent portions of the shared boundaries into *portals*, these portals can be used to exclude the volume occluded by the opaque cell boundaries from the view frustum (See Fig. 2, where the dotted lines represent the boundaries of the initial frustum and the gray area represents the narrowed frustum for each cell visible through a portal).

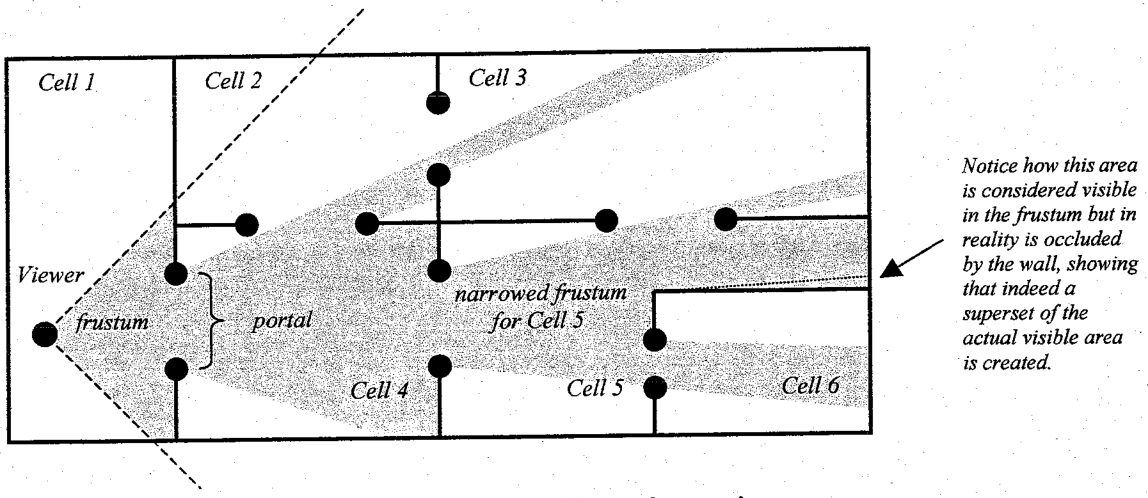

Cell 1
Cell 2
Cell 3
Viewer
frustum
portal
narrowed frustum for Cell 5
Cell 4
Cell 5
Cell 6
Notice how this area is considered visible in the frustum but in reality is occluded by the wall, showing that indeed a superset of the actual visible area is created.

Fig. 2 Frustum narrowing with portal processing.

Even though portal processing provides a better estimate of the visible polygons than regular frustum culling, it still produces a superset, and any superset generator will require a Z-buffer or another technique for computing exact visibility.

The concept of using portals to estimate visibility was first introduced in 1991 by Teller and Séquin [3], who used them to approximate the visibility information in a scene offline for later use in an interactive rendering phase. Although Teller and Séquin automated the process of partitioning the scene into cells and portals, this task can also be left to the modeler when designing the virtual world. When automating the task, usually more cells and portals will be created then when the task is left to the modeler, resulting in better performance. However, a modeler can also add some insight into the model not detected by the automated process, and he can add special portals like mirrors and monitors.

For the remainder of this thesis, how the world has been partitioned into cells and portals is left undetermined, it is just assumed that it has been done. All example code fragments in this thesis are given in *pseudo* C++ code, even when discussing the implementation in vrSS. This was done as not to confuse the reader with passing arguments by value, reference, pointer, or a mixture of these.

Chapter 2: Introducing the vrSS Toolkit

This chapter introduces the vrSS (*virtual reality Solutions System*) toolkit, which will be used for implementing the portal processing environment. Some features of the toolkit that are expanded on in the later chapters are introduced, as well as a short introduction in scene rendering using vrSS.

vrSS provides an extensive and highly modular C++ framework that facilitates the various aspects of designing and manipulating a virtual reality environment. Except for those parts of the toolkit that require interaction with system devices and API's (i.e. Head-Mounted-Displays and trackers, DirectX) the toolkit is completely system independent. System dependencies are integrated using plug-in modules but currently the only module available is for use on IBM-PC's running Microsoft Windows 98/2000 and DirectX versions 7.0 and up[3].

For a smooth performance of a simple application created with vrSS a minimum PC configuration with an Intel Pentium II-300, 64 Mb of RAM and at least a Nvidia GeForce2 graphics accelerator is recommended.

2.1 *Creating a VR-world with vrSS*

Like many other 3D-engines, vrSS stores and represents the virtual world in a *world tree* (*single parent constraint*) that holds all the objects that make up the world (see Fig. 3). Using a tree structure allows using a hierarchical *parent-child relation* where the child position and orientation in the world is dependent on the position and orientation of its parent.

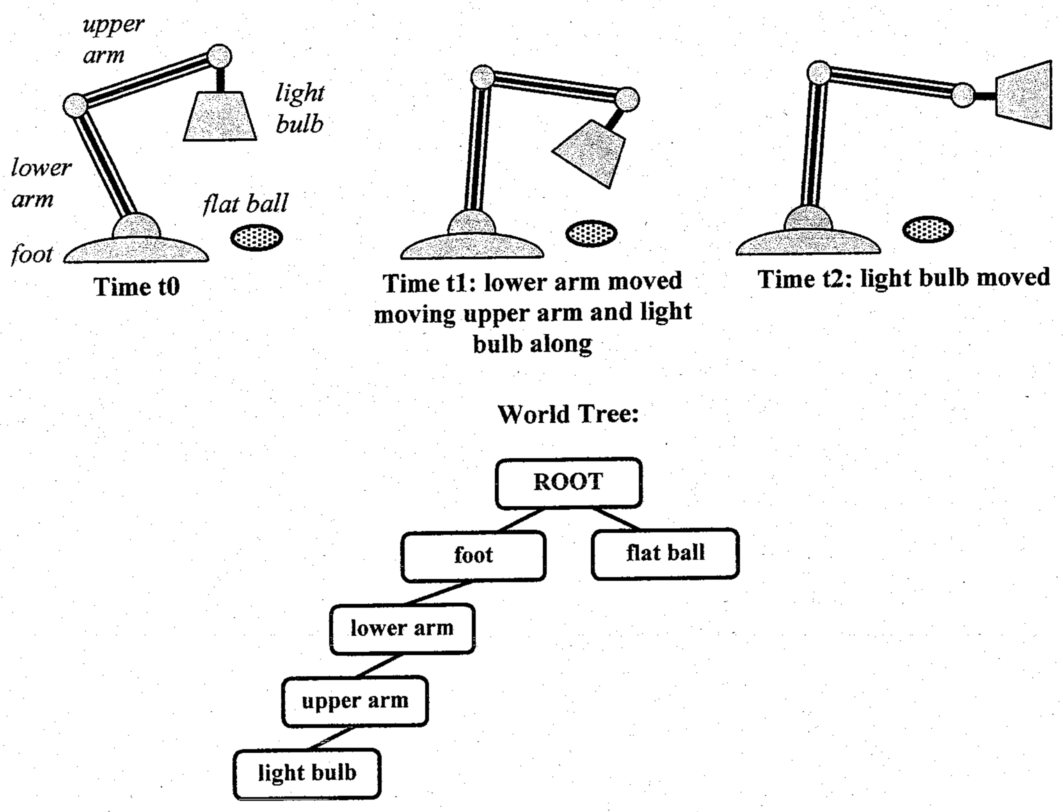


Fig. 3 The world tree and three different animation frames for a lamp looking at a flat ball.

[3] An OpenGL version is being developed at the time of writing this document.

Every tree node in vrSS has a *local transform*, defining the position and orientation of the *root* of the node's object in the coordinate system of its immediate tree parent node, and a *world transform* that holds the root position and orientation of the node in the world coordinate system (see Fig. 4). The world transform of every node can be computed by combining its local transform with the world transform of its parent.

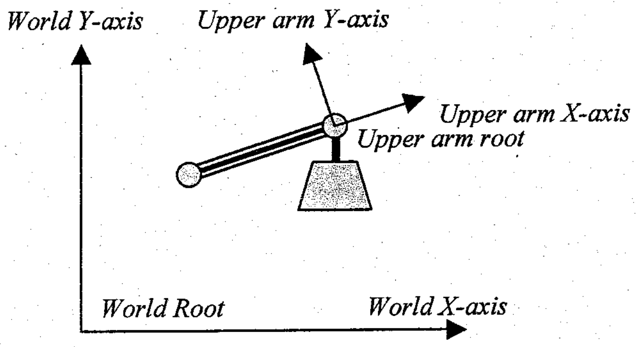


Fig. 4 Local and world coordinate systems.

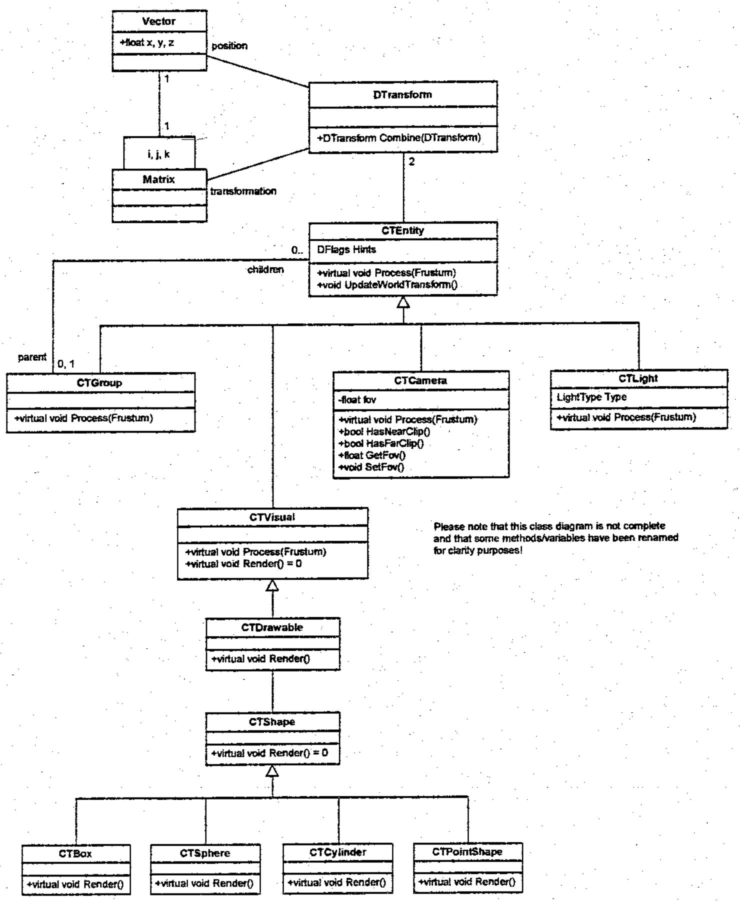


Fig. 5 UML diagram of vrSS tree components.

The base class for every node in the tree is *CTEntity* (see Fig. 5) which stores the local and world transforms and which updates the world transforms of the nodes each time a new frame is rendered.

The tree is built up using *group nodes* and *leaf nodes*. Group nodes are non-visible objects that have one or more (coherent) children whose position and orientation as a group can easily be manipulated by changing the position and orientation of the group node. Group nodes are implemented in vrSS by the *CTGroup* class from which other group nodes can be derived (e.g. the *CTLODGroup* class which only processes one of its children based on the distance of the group to the camera allowing a *level-of-detail* scheme to be used).

Leaf nodes have no children and usually represent the visible or audible objects like geometry (object shapes like buildings, cars, people etc.), lights and sound.

2.2 *Visibility determination in vrSS*

To render a scene the world tree is processed and the objects are checked for visibility using a camera (representing the position and orientation of the eyes of the viewer) and its frustum (see Fig. 6).

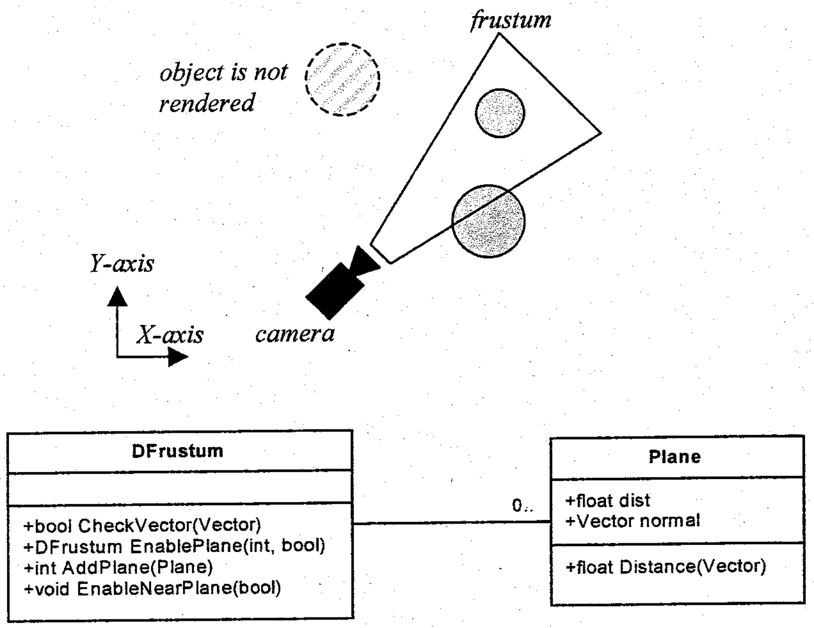


Fig. 6 The view frustum + UML Diagram of Dfrustum class.

The frustum in vrSS is a convex polyhedron defined by the intersection of the negative half space of a set of planes. The negative half space of a plane with normal *n* and distance to the origin *d* is defined by the set of points *p* with $\{ p : p.n - d < 0 \}$. If an object's shape does not intersect with the frustum then the object is not visible and will not be processed/ rendered any further. To speed up the intersection test even more, usually not the object's actual shape but its *bounding volume* is used. A bounding volume is a simple shape like a box or a sphere that can be tested for intersection with the frustum much faster thus speeding up visibility determination (in the case of Fig. 6 for example the circles could represent the bounding volumes of complex shapes).

vrSS uses a *bounding volume hierarchy* where every parent node in the tree has a bounding volume that includes the bounding volumes of its children. If the bounding volume of the parent node does not intersect the frustum then neither will the bounding volume of any of its children so processing that tree branch can be stopped.

The vrSS engine does not clip objects against the frustum before sending them to the render API (i.e. OpenGL or DirectX) but does flag the API to do so if it supports this. vrSS uses back-face culling to reduce the amount of polygons send to the render API. Exact visibility determination is left to the render API's Z-buffer.

2.3 Rendering a scene in vrSS

Processing a tree in vrSS starts by calling the root node's *Process* method, which is a virtual method of *CTEntity*. Every tree node must implement this method to define what must be processed (geometry., lights, sounds or child nodes). The root node, which is an instance of *CTGroup*, then calls the *Process* method for each of its children recursively traversing the tree.

When rendering a scene in vrSS the world tree is traversed several times: first a *lighting pass* is done to find and process all visible lights which are then used in the *draw pass* that actually draws the visible geometry on the output device. Finally, a *sound pass* finds and processes all audible sounds in the tree. Just like a bounding box hierarchy, vrSS also has a *kind flag hierarchy* in the world tree: if a node does not have one of the kind flags set then neither will any of its children. For instance a light node will only have the KIND_LIGHT flag set signaling the engine that it only requires processing during a lighting pass and not in any of the other passes.

For a more thorough overview of the vrSS engine the reader is referred to the vrSS documentation [8] and to appendix A, showing a complete vrSS class diagram.

Chapter 3: Cells and Portals

This chapter discusses the details of implementing a portal processing algorithm. Although the portal processing algorithm in itself is very simple, there are some consequences associated with the use of it that must be handled carefully, one of the most important being that portal processing is not an exact visibility determination algorithm but creates a superset the visible objects in a scene. Therefore, additional exact visibility determination algorithms are necessary to further process and render this superset. However, it will be shown that by careful integration of the portal processing algorithm into the vrSS framework, the visibility determination algorithms already incorporated into vrSS can be used for this task.

This chapter starts by a detailed explanation of how the portal processing algorithm works, why it creates a superset and what the consequences to storing the world are. Next the algorithm is implemented in vrSS, discussing the classes created to integrate the portal processing algorithm into the vrSS engine. Optimizations and solutions to problems that arise (some general and some vrSS specific) are discussed and finally a summary of the findings is given.

3.1 Portal processing

When thinking of the interior of a building it is easy to think of this as a series of inter-connected rooms. This intuitive approach is the basis for *portal processing* where the model of the world is partitioned into a number of *cells* connected to each other by *portals*. A *cell* defines the appearance and volume of a coherent part of the world (e.g. an office room which has desks and computers in it and whose volume is defined by the walls). A *portal* is an opening between two adjacent cells that connects these cells so that the *only way* one can look from one cell into the other is through this portal (i.e. a door connecting two office rooms).

As can be seen in Fig. 7 portals can narrow the field-of-view (frustum) for each cell seen through a (sequence of) portal(s) from the *(physical) camera cell* (the *physical cell* of an object is the cell in which the object is positioned physically) reducing the number of overdrawn objects when rendering a scene.

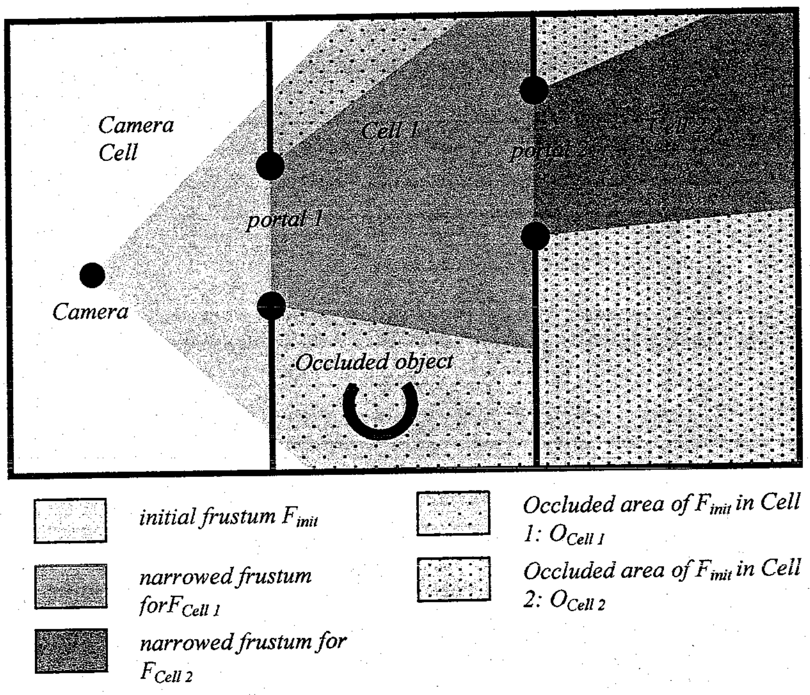

Camera Cell
Cell 1
portal 2
Cell 2
portal 1
Camera
Occluded object
initial frustum F_{init}
Occluded area of F_{init} in Cell 1: $O_{Cell\ 1}$
narrowed frustum for $F_{Cell\ 1}$
Occluded area of F_{init} in Cell 2: $O_{Cell\ 2}$
narrowed frustum for $F_{Cell\ 2}$

Fig. 7 Frustum narrowing and occlusion culling due to portals.

Another way of looking at a portal is as an infinite wall with an opening in it. A viewer can only see those objects on the other side of the portal that are not occluded by the wall, so a portal narrows the view (or leaves it intact if the viewer is positioned close to the portal) which can be simulated by narrowing the camera frustum.

Let the frustum F be thought of as the set of points visible to the camera and let F_{init} denote the initial camera frustum. Let the number of portals in the sequence through which destination cell *cd* is visible from a source cell *cs* be given by $PS_{cs \to cd}$. Due to potential frustum narrowing by the portal, it is clear that the following property holds:

Property I :

$$\{ \forall dc : PS_{sc \to dc} = 1 : F_{dc} \subseteq F_{sc} \}$$

The frustum for the connected cell *dc* of cell *sc* will be a subset of the frustum for cell *sc*. If *sc* is the cell the camera resides in, then using property I the following property can be formulated:

Property II :

$$\{ \forall dc : PS_{camera\ cell \to dc} = 1 : F_{dc} \subseteq F_{init} \}$$

Then, by induction, the following property can be formulated:

Property III :

$$\{ \forall dc : PS_{camera\ cell \to dc}\ exists : F_{dc} \subseteq F_{init} \}$$

So a series of consecutive visible portals can (and, most importantly, usually will) keep on narrowing the frustum causing more and more objects to be rejected by the frustum intersection test, resulting in a smaller superset of visible objects than would have been created using the initial frustum. This allows a scene to be rendered faster resulting in higher frame rates, or makes rendering of more complex scenes at the same frame rate possible. The speed improvement gained by portal processing depends highly on consecutive portals narrowing the frustum, therefore limiting its use to mainly interior scenes because the notion of a cell has little meaning in outdoor scenes[4].

Rendering a scene using portals can be implemented using the simple recursive algorithm shown in Fig. 8:

```
void ProcessPortals( Frustum frustum, Cell currentcell )
{
  // check all portals of this cell
  for( int f = 0 ; f < currentcell.Portals.Count() ; f ++ )
  {
    if( Portals[ f ].VisibleInFrustum( frustum ) )
    {
      Frustum narrowedfrustum = CreateFrustum( frustum, Portals[ f ] ) ;
      ProcessPortals( narrowedfrustum, Portals[ f ].ConnectedCell ) ;
    }
  }
}
```

Fig. 8 The portal processing algorithm.

When first called with the initial camera frustum and camera cell, each portal of the camera cell is evaluated for visibility after which the (narrowed) frustum for the connected cell of that portal is created and the function is recursively called for processing the connected cell with the narrowed frustum. This process is continued until no more visible portals are encountered.

One of the consequences of portal processing is that a scene is no longer rendered by traversing a world tree top-down visiting all the leaves (as is the case in vrSS), but instead a scene is rendered jumping from visible cell to visible cell via

[4] Although cells could be used as sort of a sub-division (or limited octree-like) scheme in outdoor scenes.

portals. This implies a new internal representation of a world where each cell has its own object tree called the *cell tree* which defines the appearance of that cell, and a graph connecting these *cell tree* nodes through their portals called the *world graph* (see Fig. 9).

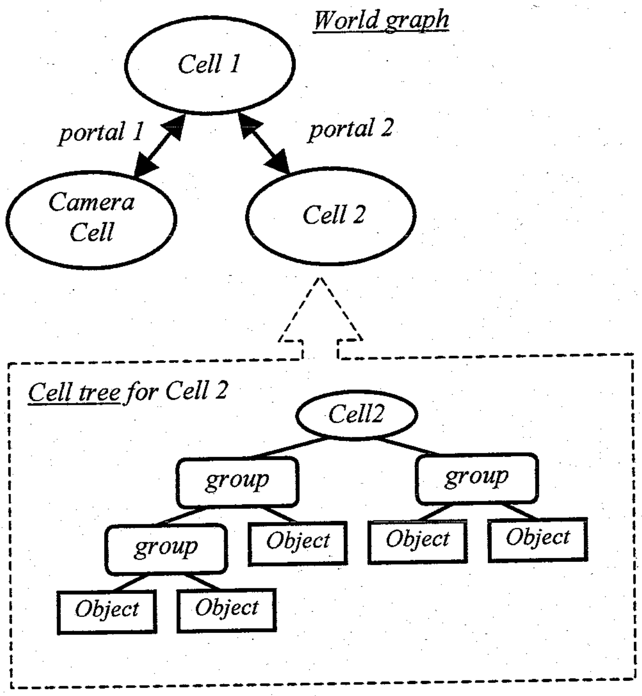


Fig. 9 The world graph and cell trees.

Using the graph for scene processing, another potential speed up can be gained: In many VR-engines, the world is represented by one tree. When rendering the world the tree is traversed top-down and every object node is checked against the frustum to determine visibility. When using portals every cell has its own cell tree whose object nodes are only checked for visibility if the cell itself is visible, or more precisely, if a portal connecting to that cell is visible (unless the cell is the physical camera cell which will always be processed). So if the cell is not visible then the objects in its tree will not be checked therefore again saving processing time.

Some VR-engines, like vrSS, use a bounding box hierarchy in their world tree where if the bounding box of a node is culled by the frustum the sub tree is classified as not visible and therefore not further processed. In this case, the actual speed up gained by the graph processing of the world might not be very significant.

3.2 Implementing Cells and Portals in vrSS

In order to implement a portal rendering system for the vrSS engine, three major new classes are introduced into the vrSS framework: *FCCell*, *FCPortal* and *FCRootCell*. An UML-diagram of the framework extension for implementing portal processing is shown in Fig. 10:

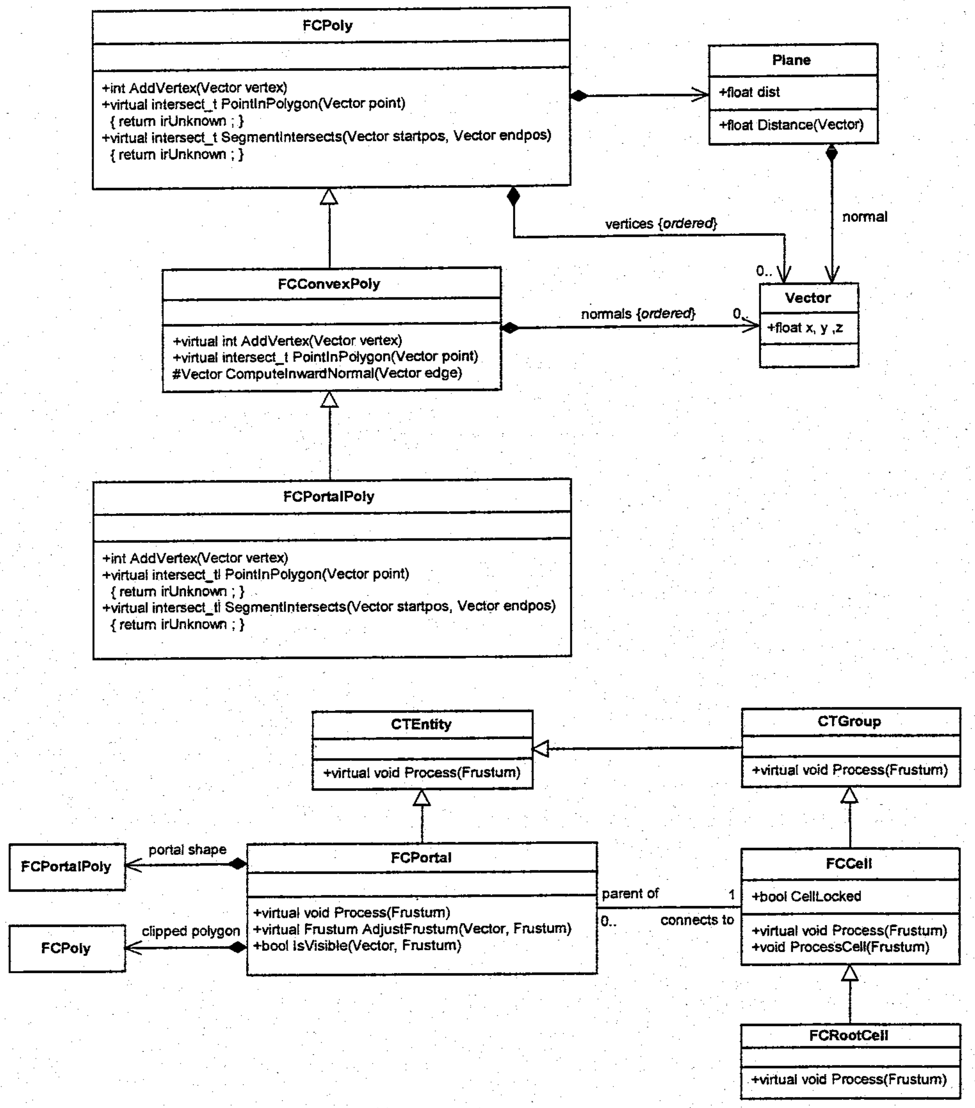

FCPoly
+int AddVertex(Vector vertex)
+virtual intersect_t PointInPolygon(Vector point) { return irUnknown ; }
+virtual intersect_t SegmentIntersects(Vector startpos, Vector endpos) { return irUnknown ; }
Plane
+float dist
+float Distance(Vector)
vertices {ordered}
normal
0..
FCConvexPoly
+virtual int AddVertex(Vector vertex)
+virtual intersect_t PointInPolygon(Vector point)
#Vector ComputeInwardNormal(Vector edge)
normals {ordered}
0..
Vector
+float x, y ,z
FCPortalPoly
+int AddVertex(Vector vertex)
+virtual intersect_tl PointInPolygon(Vector point) { return irUnknown ; }
+virtual intersect_tl SegmentIntersects(Vector startpos, Vector endpos) { return irUnknown ; }
CTEntity
+virtual void Process(Frustum)
CTGroup
+virtual void Process(Frustum)
FCPortalPoly
portal shape
FCPortal
+virtual void Process(Frustum)
+virtual Frustum AdjustFrustum(Vector, Frustum)
+bool IsVisible(Vector, Frustum)
FCPoly
clipped polygon
parent of
1
0..
connects to
FCCell
+bool CellLocked
+virtual void Process(Frustum)
+void ProcessCell(Frustum)
FCRootCell
+virtual void Process(Frustum)

Fig. 10 UML diagram of framework extension for portal processing.

The *FCCell* class uses an instance of a *FCPortalPoly* class to store the portal polygon. *FCPortalPoly* is a descendant of *FCConvexPoly* which stores for each edge *i*, a normal to that edge lying in the portal plane pointing inwards of the polygon. This allows for a very fast *point-in-polygon* check (at the expense of extra memory overhead) requiring only *n+2* dot products, with *n* the number of edges of the polygon. This functionality will be used in chapter 5 when determining the intersection of a line with the portal.

3.2.1 The FCPortal class

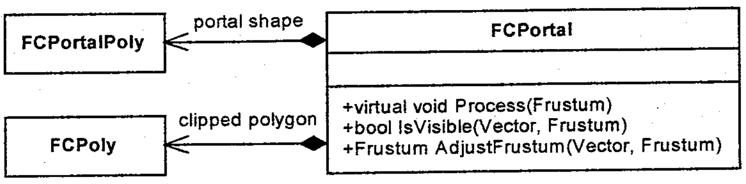

FCPortalPoly
portal shape
FCPortal
FCPoly
clipped polygon
+virtual void Process(Frustum)
+bool IsVisible(Vector, Frustum)
+Frustum AdjustFrustum(Vector, Frustum)

Fig. 11 The *FCPortal* class.

The class *FCPortal* is derived from class *CTEntity* (see chapter 2) and uses an instance of *FCPortalPolygon* to define a convex polygon that uni-directionally connects its *parent cell* (the root of the cell tree of which the portal is a child) to the connected cell. A polygon with *k* edges is defined as $\{v_0, v_1, ..., v_{k-1}\}$ with v_i and v_{i+1} the vertices for edge *i* and v_{k-1} and v_0 the vertices of edge *k*-1, defined in clockwise order and all vertices v_i, $0 <= i < k$ on the same plane.

The portal polygon is required to be convex for two reasons: it guarantees that the frustum created for a connected cell will be convex and it allows for a fast point-in-polygon check (both will be discussed later).

The *portal plane* is defined by the plane of the portal polygon with the plane normal pointing in the *see-through direction* of the portal. The choice for uni-directional portals was made because this makes separating the world graph into individual cell trees possible which can e.g. be used to load large worlds cell by cell (see Fig. 12 where a world consisting of two cells has been separated into two individual cells and their cell trees).

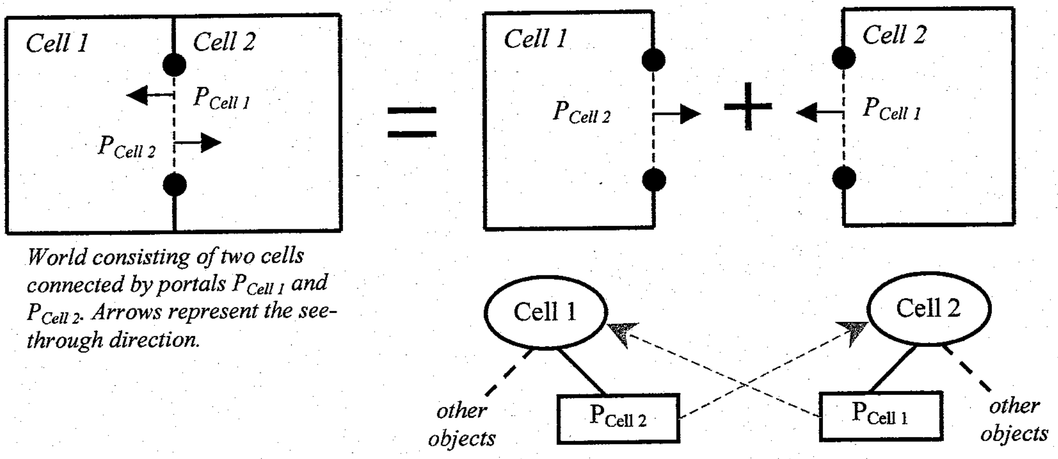


Fig. 12 Separating the world into individual cell trees.

In order to preserve the correlation of the cell trees (shown in Fig. 12 by the dotted arrows between the two cell trees) the name of its connected cell is stored along with a portal allowing the connected cell to be looked up when the portal is visible[5]. This requires each cell to have a unique name.

So a portal is implemented as an infinite opaque wall with a one-way opening in it: when looking through this opening in the see-through direction one can see everything in the connected cell that is not occluded by the wall, and when looking in the other direction the portal is not visible and culls nothing.
Although most portals will usually be rectangular shaped (i.e. doors or windows) they can be any convex polygon desired.

[5] The actual implementation will only look up the connected cell by name once. After that a pointer to the cell tree root is stored in the portal object.

3.2.2 The FCCell class

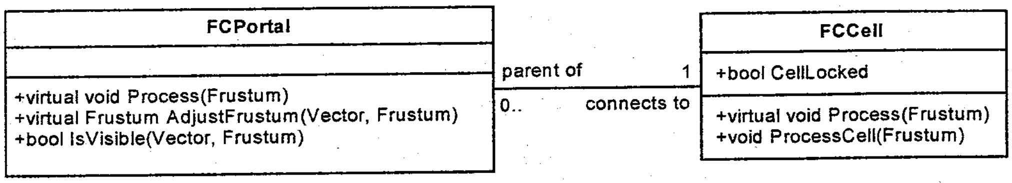

FCPortal
+virtual void Process(Frustum)
+virtual Frustum AdjustFrustum(Vector, Frustum)
+bool IsVisible(Vector, Frustum)
parent of
1
0..
connects to
FCCell
+bool CellLocked
+virtual void Process(Frustum)
+void ProcessCell(Frustum)

Fig. 13 The *FCCell* class.

The class *FCCell* is derived from class *CTGroup*, and represents the root of the cell tree. Often a cell will have some geometry objects representing its physical boundaries due to the nature of the environment in which portal processing will be used (e.g. the walls of a room) but no restrictions are put on the type, amount or shape of the objects in the cell tree. Cells are required to have a unique name for identification.

Although an *FCCell* object is a cell tree root, it is still desirable to be able to store an entire world with multiple cells in one world tree where cells can be children of other cells (see Fig. 14). This way the advantages of grouping objects and the parent-child relationship (see chapter 2) can also be applied to cells. However, care must be taken that these child cells are not processed using their parent cell's frustum since that would negate the advantage of portal processing for these child cells.

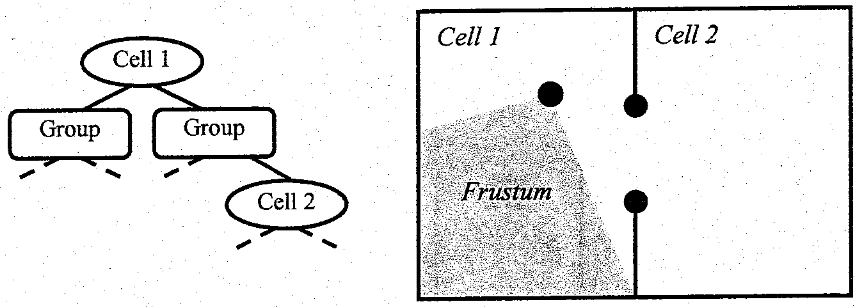
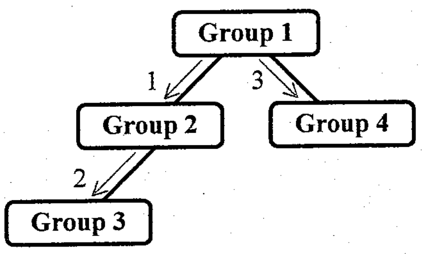


Fig. 14 Parent and child cells (Processing *Cell 2* as a child of *Cell 1* is useless since it is not visible in the frustum).

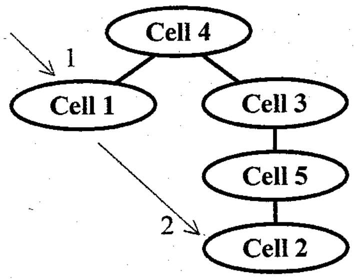
3.2.3 Portal processing vs. vrSS tree processing

It is desirable to (re) use as much of the functionality as possible provided by the vrSS toolkit for rendering scenes/processing trees/exact visibility determination, since it has already been implemented and is highly optimized. It is therefore a good idea to examine where the portal processing algorithm differs from the standard vrSS tree processing algorithm (see chapter 2): The portal processing algorithm is different in that the world tree is no longer processed top-down and that the frustum is adjusted before processing of a cell tree starts. However, once the frustum is adjusted, the cell tree can be processed using the vrSS algorithm which also performs the necessary exact visibility determination of the cell tree's objects. There are two catches to using the vrSS algorithm: The vrSS algorithm processes a tree by calling the *Process* method of the root and then recursively calling this method for each of its children that require processing. This implies that when processing a cell tree, should a child cell be encountered the *Process* method for this child cell will also be called. Unless the cell distinguishes between being processed as the connected cell of a visible portal and being processed as the child of another cell that is being processed, implementing the *Process* method would result in a child cell being processed as if it were a (physical) part of its parent cell's cell tree (see also Fig. 14).

The other catch is that the vrSS algorithm assumes that as a pre-condition to calling the *Process* method of a node, the world transform of that node's parent is updated. When using the vrSS algorithm this is taken care of automatically as shown in Fig. 15: starting at the root node the tree is processed top-down and at every node its local transform and parent world transform are combined to get the world transform for the node before it or any of its children are further processed.



vrSS tree processing: processing starts at the root whose parent transform is the same as its local transform and then traverses down the tree updating each nodes world transform. The numbers and arrows denote the order and direction in which the child nodes are processed.



portal processing: processing starts at cell 1. Its parent's (Cell 4) world transform has not been updated yet so do this first. Processing continues with Cell 2. Again its parent's world transform must be updated resulting in the world transform of Cell 3 to be updated as well.

Fig. 15 updating the world transform.

Portal processing on the other hand starts at the physical camera cell by processing its cell tree and then jumps through the world tree, processing other cell trees based on the visible portals. In the case shown in Fig. 15 the algorithm starts in *Cell 1* and then follows a visible portal connecting to *Cell 2*. For both of these cells the world transform of their parent cells (respectively *Cell 4* and *Cell 3*) are not up-to-date resulting in erroneous world transforms for the objects in their cell trees. Therefore a cell has to traverse up the tree to compute its world transform, in the process updating the world transforms of other cells that may not be processed at all (*Cell 3*, *Cell 4* and *Cell 5*) which will result in some negligible but extra overhead.

When processing a cell tree using the vrSS algorithm, portals should be not processed, so the *FCPortal::Process* method is also empty. Instead, the portals are processed after the vrSS algorithm finishes processing the cell tree. By doing so a *near-to-far* rendering scheme at the cell level is created, which will benefit Z-buffer performance[6], particularly when there are significant lighting computations per pixel.

When implemented carelessly portal processing can introduce some processing inefficiencies, which are shown in Fig. 16 with two situations in which a cell tree is processed twice.

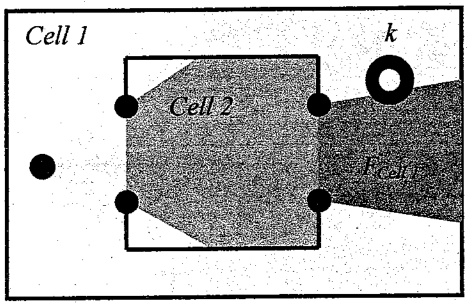


Fig. 16-a

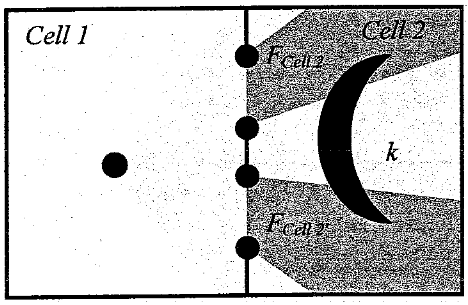


Fig. 16-b

Fig. 16 portal processing inefficiencies.

In the situation of Fig. 16-a, *Cell 1* is processed first, rendering object *k* which lies in the initial frustum. The portal to *Cell 2* is encountered and *Cell 2* is processed where a portal connecting the cell back to *Cell 1* is encountered. Due to property III, the new frustum $F_{Cell\ 1'}$ will always be smaller or equal than the initial frustum so processing *Cell 1* again with $F_{Cell\ 1'}$ will provide no new information, but will cost valuable processing time since object *k* would be rendered

[6] When using Z-buffers, visibility of a pixel is determined before more costly computations like lighting are performed. However, today's hardware accelerators make the performance gain noticeable only in very complex scene.

again. To solve this problem cells should be *locked* until all their portals have been processed. Once a cell is locked and a portal connecting to that cell is encountered the cell cannot be processed again.

In the situation of Fig. 16-b, *Cell 2* is processed twice causing object *k* to be processed twice also. If no clipping of any sort is used then the entire object will unnecessarily be send through the render pipeline a second time. The only remedy for this situation is to introduce a *time stamping* mechanism in the vrSS engine where once an object is rendered it will not be processed again until rendering of a new frame has started[7].

Note that even though the time stamping mechanism would also solve the problem of Fig. 16-a this would still result in the unnecessary reprocessing of the cell tree of *Cell 1*.

Having discussed the intricacies of the portal processing algorithm and the interconnection with the vrSS algorithm, a blueprint for the portal processing algorithm implemented in the *FCCell::ProcessCell* method will look like Fig. 17.

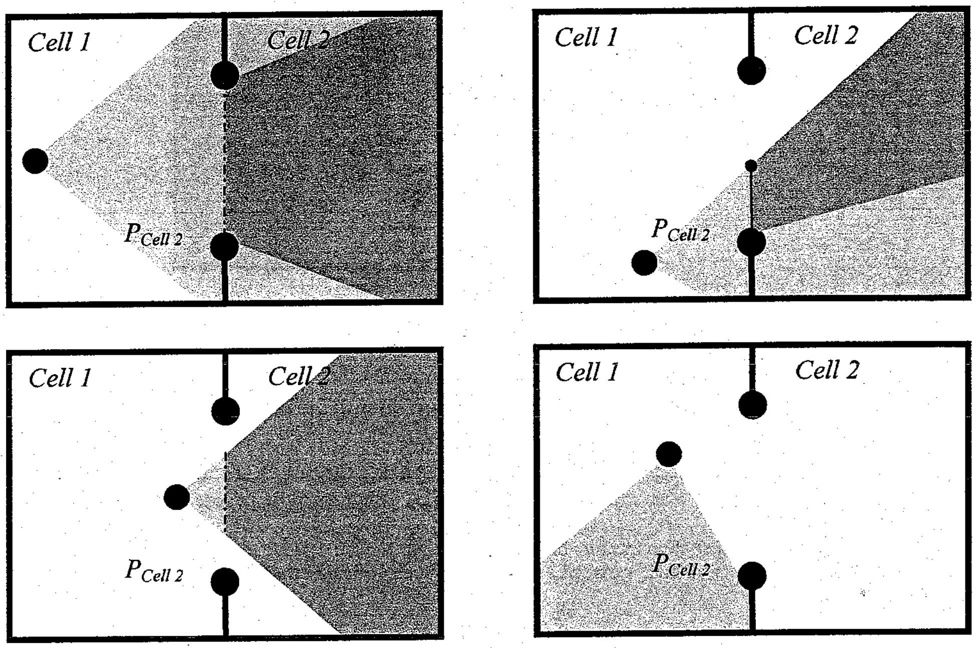
```
// CellFr is the frustum constructed for current cell
void FCCell::ProcessCell( Frustum Fr )
{
  // lock the cell
  CellLocked = true ;

  // make sure the parents world transform is up-to-date
  // and then process the cell tree using vrSS Process()
  GetParent().UpdateWorldSpace() ;
  CTGroup::Process( Fr ) ;  // call parent class's Process()

  // now process the portals
  for( int f = 0 ; f < Portals.Count() ; f ++ )
  {
    // is the connected cell of the portal locked?
    if( Portals[ f ].ConnectedCell.CellLocked )
      continue ;

    // is the connected cell visible (check the portal
    // polygon against the current cells frustum) (to be defined)
    if( !Portals[ f ].IsVisible( GetCameraPosition(), Fr ) )
      continue ;

    // Construct a new frustum for the connected cell (to be defined)
    Frustum ConCellFr = Portals[ f ].AdjustFrustum( GetCameraPosition(), Fr ) ;

    // process the connected cell with the new frustum
    Portals[ f ].ConnectedCell.ProcessCell( ConCellFr );
  }

  // and finally unlock the cell
  CellLocked = false ;
}
```

Fig. 17 The portal processing algorithm in vrSS.

[7] Based on the complexity of *Cell 2*, it might even be faster to render it using the initial camera frustum, instead of processing it twice using the portals. This option however has not been investigated further.

3.2.4 Determining portal visibility

Determining the visibility of a portal is done by first determining if the camera position is in the negative half space (see chapter 2) of the portal plane (since a portal is uni-directional it is impossible to look through it if the camera position is in the positive half space) and then clipping the portal polygon against the frustum (see Fig. 18). If the clipped polygon has zero vertices then the portal does not intersect the frustum and is not visible. The clipped polygon is stored to help create the adjusted frustum discussed later in this chapter.



The clipped portal polygon in 4 different situations: The gray area represents the camera frustum. Clipping the portal polygon against the frustum results in the clipped portal polygon which is represented by the dashed line. The new frustum for the connected cell is represented by the dotted area.

Fig. 18 Clipping the portal polygon against the frustum.

A special case occurs when the camera position in *Cell 1* is on the portal plane but outside of the polygon (see Fig. 19). Clearly, even though the portal will be classified as visible and the clipped polygon will be a copy of the portal polygon, it is impossible to look through it and the created frustum does reflect this: it will have two coinciding planes with opposite plane normals. To avoid creating an 'empty' frustum and processing the connected cell with it, a special check for this situation is added to the portal visibility check classifying the portal as not visible before the polygon is clipped.

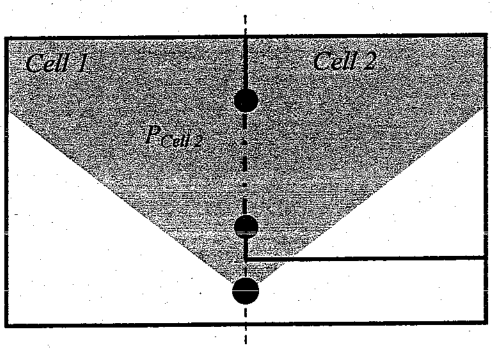


Fig. 19 A special case of the clipped polygon and its frustum.

If the frustum has a near plane the following situation can occur (see Fig. 20): if the camera in *Cell 1* is positioned close to the portal then the near plane of the frustum lies in the positive half space of the portal plane classifying the portal polygon as not visible therefore not processing the connected cell. Therefore, when checking portals against the frustum for visibility the near plane must always be ignored.

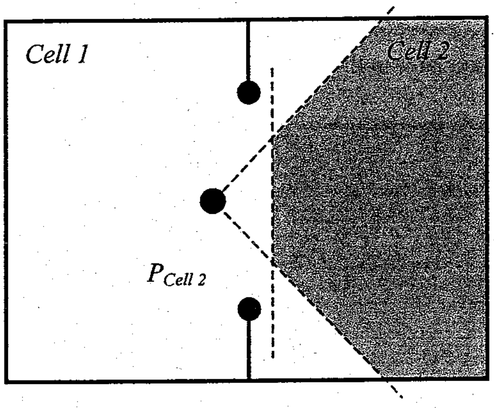


Fig. 20 The frustum near plane trap.

Now the algorithm for determining portal visibility can be given and is shown in Fig. 21:

```
bool FCPortal::IsVisible( Vector cameraposition, Frustum frustum )
{
  // is the camera in the positive halfspace?
  double distance = portalpolygon.plane.Distance( cameraposition ) ;
  if( distance > 0 )
    return false ;

  // is the camera on the portal plane but outside of the portal poly?
  if( distance == 0 && portalpolygon.PointInPolygon( cameraposition ) )
    return false ;

  // disable frustum near plane
  frustum.DisableNearPlane() ;

  // clip the portal (the clipped polygon is stored for later use)
  clippedpolygon = portalpolygon.ClipAgainstFrustum( frustum ) ;

  // enable frustum near plane
  frustum.EnableNearPlane() ;

  // is the portal visible?
  return ( clippedpolygon.Vertices.Count() != 0 ) ;
}
```

Fig. 21 Portal visibility algorithm.

3.2.5 Creating the adjusted frustum

In vrSS, the negative half spaces of the frustum planes define a convex polyhedron that is used for visibility culling/polygon clipping. Because of the convexity of the polyhedron and the constraint that a portal polygon must be convex, the resulting clipped polygon will also be convex allowing for the easy construction of a new frustum for the connected cell: Except for the near and far clipping planes every plane *i* in the frustum can be created using the camera position and the vertices v_i and v_{i+1} of edge *i* of the clipped polygon resulting in a convex polyhedron stretching from the camera position into infinity (see Fig. 22)

To limit the volume of this polyhedron a far plane is added for which the far plane of the original frustum is simply copied. If the original frustum does not have a far plane, than neither will the new one.

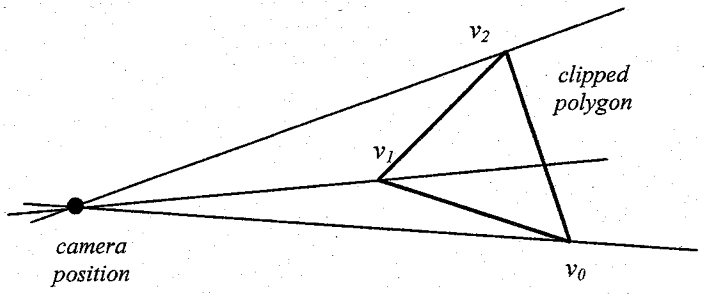


Fig. 22 Creating the new frustum.

The volume of the polyhedron can be limited even further by using the (inverted) portal plane as the near clipping plane: light cannot bend around corners so only objects in the positive half space of the portal plane can be visible through the portal.

Now the algorithm for creating the adjusted frustum can be given and is shown in Fig. 23.

```
Frustum Portal::AdjustFrustum( Vector CameraPosition, Frustum CurFrustum )
{
  Frustum AdjustedFrustum ;

  // using the clipped polygon stored in method IsVisible now create
  // the visibility frustum for the connected cell of the portal
  int VertexCount = clippedpolygon.Vertices.Count() ;

  for( int f = 0 ; f < VertexCount ; f ++ )
  {
    Plane p ;
    p.Set( CameraPosition, clippedpolygon.Vertices[ f ],
           clippedpolygon.Vertices[ ( f + 1 ) % VertexCount ) ;

    AdjustedFrustum.AddPlane( p ) ;
  }

  // add the far plane and portal plane as extra visibility limiters
  AdjustedFrustum.AddPlane( CurFrustum.GetFarPlane() ) ;
  AdjustedFrustum.AddPlane( portalpolygon.plane ) ;

  return AdjustedFrustum ;
}
```

Fig. 23 The *AdjustFrustum* algorithm.

3.2.6 The FCRootCell class

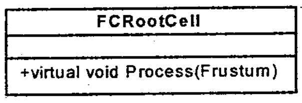


Fig. 24 The *FCRootCell* class.

The *FCRootCell* class is introduced to solve the following problem: the vrSS engine always starts the rendering process in the root of the world tree by calling that node's *Process* method. This however presents two problems:

1. Portal processing requires that the rendering process starts in the camera parent cell. In a world tree with multiple cells this would require the camera parent cell to be the root of the world tree. This makes moving the camera into another cell impossible without changing the tree structure.

2. Since the *Process* method of an *FCCell* object is empty rendering would immediately stop.

To solve these two problems the *FCRootCell* class was introduced. This class derived from *FCCell* has implemented a *Process* method that finds the camera parent cell and then calls that cell's *ProcessCell* method to start the portal processing algorithm. To use portal processing it is required that the root of every world tree is of type *FCRootCell*.

```
void FCRootCell::Process( Camera* cam, Frustum* Fr )
{
  // process all global objects
  inherited::Process( Fr ) ;

  // get the cell the camera currently resides in
  FCCell cpc = FindPhysicalCell( cam ) ;

  // start the portal processing algorithm
  cpc.ProcessCell( Fr ) ;
}
```

Fig. 25 The *FCRootCell::Process* method.

By implementing the *FCRootCell::Process* method another useful feature is introduced: now *global objects* can easily be introduced into the world graph. Global objects are objects that are processed every frame (i.e. an ambient light source used to globally light the world) that are not a part of any cell tree.

3.3 Summary

Portal processing is a simple yet effective tool in speeding up real-time visibility determination for (building) interior-type scenes. This is done using the narrowed frustum to cull many objects that would be classified as visible in the initial frustum. Portal processing is a visibility estimation algorithm that produces a superset of visible objects, so it must be used in conjunction with an exact visibility determination algorithm.
Usually this algorithm will be a Z-buffer, whose performance will also benefit from portal processing as this allows *near-to-far* processing at the cell level: by first rendering the cell closest to the camera, many pixels in the Z-buffer will already have been drawn that will not be overwritten again. This saves on the expensive lighting computations that would otherwise have been performed for pixels created by objects in cells positioned further away, because Z-buffering first checks whether a pixel will be visible before performing other operations on it.

The main disadvantages of portal processing are that it is only suitable for interior-type scenes, and that the world must be designed with cells and portals already defined. This however requires very little effort from a modeler, and has the advantage over strategies that perform this spatial subdividing automatically (like BSP-trees or the portal implementation used by Funkhouser and Teller [3, 4]) that little changes in the model don't require a time consuming pre-computation task before the model can be used.

Due to its object oriented design, vrSS lends itself ideally for integrating a portal processing algorithm into its standard tree processing algorithm, provided some problems introduced by optimizations for processing a world tree are properly dealt with. By using the standard tree-processing algorithm of vrSS to process the interior of a cell, all visibility determination functionality already incorporated into vrSS (frustum culling, back-face culling, bounding volumes and Z-buffering) can be used to determine exact visibility of each processed object.

Once a world partitioned into cells and portals has been created, portal processing is done completely transparent to the programmer. This allows programmers skilled in vrSS to easily adapt the advantages provided by portal processing into their own programs.

Chapter 4: Dynamic Cell Management

When using portal processing the world is partitioned into separate cell trees that are only processed when visible through a portal. These cell trees are usually stored on a large inexpensive and slow storage medium (like a hard disk or CD-Rom) and are loaded into the limited program memory when the program is running. However, if a cell is not visible then there is no need for its cell tree to be stored in memory.

This property can be used when trying to use memory resources effectively when working with large worlds build from many cells. Cells can be loaded into memory when required for rendering a scene and cells that are no longer visible can be removed from memory. This process is called *dynamic cell management.*

Dynamic cell management is not limited to just memory management, it also has other applications like serving cells over a network (i.e. the internet). In this case, a user can already view a partial world while more cells are being sent to his system to enhance scene detail.

This chapter starts with an introduction of the many different approaches to creating a dynamic cell management system and why these different situations require different types of cell management. The chapter then continues with a description of the framework designed to allow programmers to integrate their specific dynamic cell management system into the vrSS engine.

4.1 Strategies for dynamic cell management

The *dynamic cell manager* or *d.c.m.* is responsible for loading cells into memory and removing them from memory. The portal processing algorithm requests a cell tree from the d.c.m, and the d.c.m. then retrieves this either directly from memory or first loads it from storage (see Fig. 26).

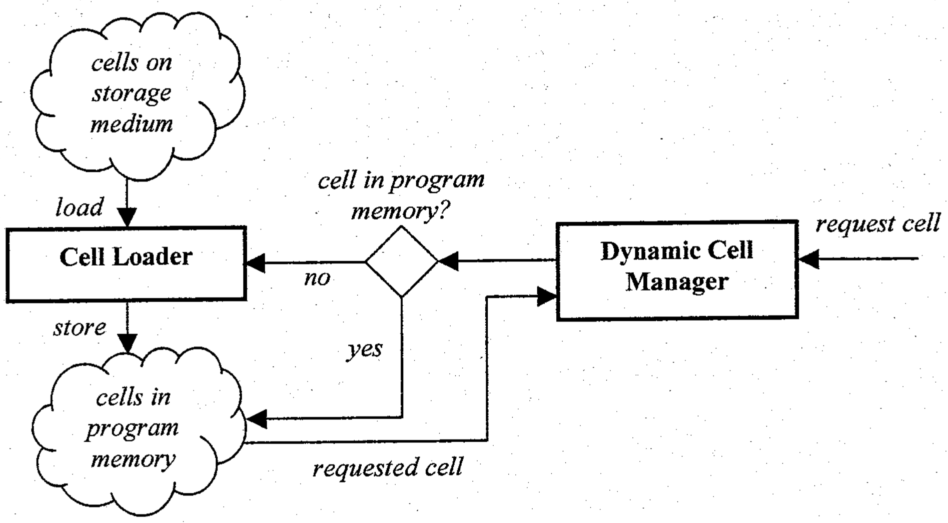


Fig. 26 Requesting cells from the dynamic cell manager.

So implementing a simple dynamic cell manager is straightforward: if a cell is requested that is not yet in memory load it from storage and return the cell (tree), and when a loaded cell is not visible then it can be removed from memory.

This implementation would perform just fine if loading cells did not take up processing time resulting in dropping frame rates or even the temporal freezing of the program when (multiple) large cells must be loaded while rendering a frame. Unfortunately, this problem cannot be avoided if not all cells are loaded into memory so now the problem to solve becomes minimizing the frequency and the delay caused by the loading of cells.

In the case of the example above indiscriminately removing cells from memory that are currently invisible might not be a good idea if there is a good chance that they can become visible again during the rendering of one of the next frames.

Solving this problem with a d.c.m. is a non-trivial task for which many approaches exist each with their own advantages and disadvantages and whose performance is highly dependent on the situation in which they are used. A d.c.m. that performs very well in one situation might be completely inappropriate for another.
It is therefore not possible to give just a single implementation of a d.c.m. that will perform optimally in every situation so a few pointers and suggestions will be given that can be used to implement a d.c.m. for certain situations.

4.1.1 When memory resources are at a minimum

When memory resources are at an absolute minimum, a d.c.m. can be implemented that immediately removes a cell from memory after having been processed. So for instance processing starts in the physical camera cell which is loaded and whose cell tree is processed. After the cell tree has been processed, the portal processing algorithm starts with processing the connected cells of the visible portals. To process the cell tree of a connected cell, the camera cell tree is now replaced by the cell tree of the connected cell. Although this approach requires that the unprocessed portals of the camera cell (and every consecutive cell that has not completely been processed) remain available, the rest of the cell tree can safely be removed from memory.

The delay for processing each frame f is bounded by the number of portals $N_{p,f}$ that is being processed when rendering the frame, so: $delay_f = O(N_{p,f})$. An upper-bound for memory resources used is $O(M) + O(N)$, with M the memory needed to load the largest cell and N the maximum number of unprocessed portals that must be stored when rendering the scene[8].

The performance of this algorithm will be poor due to the many loading delays and, even if care has been taken to keep the cell trees small, there still might be many cells that have to be processed during the rendering of a single frame (as can be seen in Fig. 27 where all dark gray cells have to be loaded as a result of an unfavorable layout of the world). The sum of these loading times can cause considerable freezing.

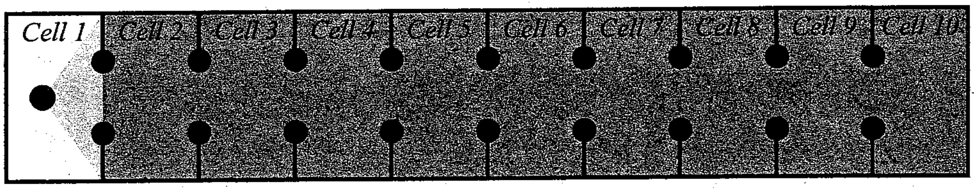


Fig. 27 A problematic world layout.

In order to get a performance gain the d.c.m. can implement a *level-of-detail* scheme where a requested cell is not loaded but instead an empty cell tree is returned if the visible part of the portal (in pixels) is small. In the case of Fig. 27, the d.c.m. might decide to return an empty cell tree when the tree for *Cell 6* is requested based on the visible portal area. Now cells *Cell 6* till *Cell 10* don't have to be processed anymore reducing the processing time by 50% (assuming the complexity of the cells is about the same).

Of course, this performance gain is not limited to use in this situation only, any d.c.m. can decide to stop processing visible cells based on a number of criteria, like visible portal area (in pixels), distance to the camera, portal sequence length or any other criterion a programmer sees fit to use.

4.1.2 When loading cells over a slow connection

Even with fast network and internet connections, loading a cell from a remote location can be a time consuming task causing severe freezing of the program. A d.c.m. could decide to first copy (*pre-fetch*) the cells from the remote location onto a local storage facility like a hard disk (or if enough memory is available load the cells directly into memory) but this could require the user to wait a long time until all cells are loaded from the remote location.

If these loading times becomes unacceptably long, the d.c.m. can also take a progressive approach (*delayed load*) where once the first cell has been loaded the user can already see the partial world while the d.c.m. copies additional cells (in a separate thread as not to interrupt the program) from the remote location to the local storage for faster access.

[8] This number can be computed using the potentially visible cells set described in chapter 7.

4.1.3 Removing cells from memory

The decision by the d.c.m. to remove a cell from memory will usually be instigated when the memory is full and a new cell has to be loaded. The d.c.m. then has to decide which cell(s) to remove and must do so carefully as not to remove those cells that will require processing soon. How should the d.c.m. decide which cells can be removed from memory?

This problem can be described in the terms of a *caching problem* as is common in computer architecture (see Fig. 28): the *cache* is a small and very fast memory located close to the processor that holds the most recently accessed data. If the processor finds a requested data item in this cache memory it is called a *cache hit*, otherwise it is called a *cache miss* and a fixed-size data block holding the data item is retrieved from main memory and stored in the cache. Due to the *spatial locality* property of data, it is likely that the other data items in this block will also be needed soon by the processor.

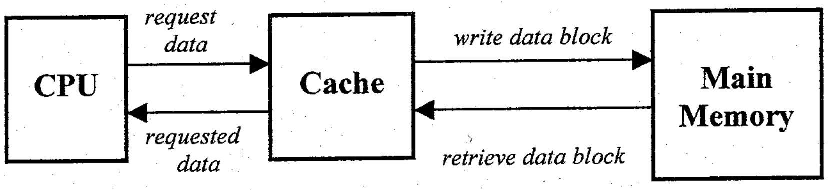


Fig. 28 CPU's and Data Caching .

Cells that are stored in memory (the cache) are called *cached cells* and every time a cell is required by the portal processing algorithm that is not in memory, a cache miss occurs requiring the d.c.m. to load the cell from storage and possibly to remove (an)other cell(s) before doing so. The goal is to reduce the number of cache misses to a minimum, which will result in a minimum of loading delays and therefore better program performance.

A lot of research has been done into this problem in the field of computer architecture yielding techniques like *victim lists* and *second level caching* [2] which can also be applied when implementing a d.c.m.
These techniques all share a common problem though: it is impossible to predict exactly when a freeze will occur at run-time.

To get a better handle on this problem the *spatial coherence* of cells can be used as was the spatial locality of data in a computer system: when a cell is currently being processed it is likely that its connected cells will also require processing soon. The d.c.m. can use this property to pre-fetch the connected cells most likely to be processed soon.

When combined with a (possibly offline computed) potentially *visible cells set* (see chapter 7), the spatial coherence property can be used to implement a clever cache management system for use with cells: A potentially visible cells set for a cell is the set of all cells that can be visible to a viewer who can be positioned anywhere inside the cell. For every cell in the world such a set is computed (see Fig. 29). When the camera enters a cell, the potentially visible cells set for that cell is examined: cells in the set that are not in the cache are loaded into memory and those cells that are in the cache but not in the set can be safely removed with the knowledge they will not be required for rendering as long as the camera resides in the current cell. Due to spatial coherence, the potentially visible cells sets for two connected cells will often share many cells, reducing the amount of cells that must be loaded and thus reducing delay times.
Another advantage is that it is possible to exactly predict when a delay will occur (i.e. only when the viewer moves into another cell) and how long this will take (the loading time of each cell can be measured, so the delay time is the sum of the loading times of all cells that are loaded).

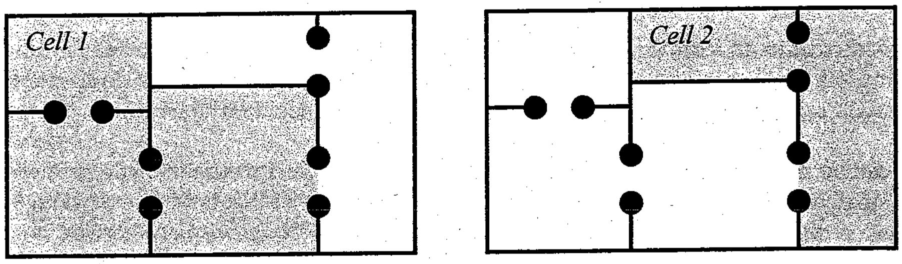


Fig. 29 Potentially visible cells for two different cells in the world.

When combined with a separate thread for loading cells and with enough memory available, the potentially visible cells sets can be used to pre-fetch the potentially visible cells for each of the connected cells of the current camera cell. This will further reduce program freezes and can even eliminate them when the loading time of these cells is less than the time it takes for the camera to leave the current cell through a portal different from the one through which it entered.

4.2 Implementing a dynamic cell manager in vrSS

In order to implement dynamic cell management for the vrSS engine three new classes are introduced into the vrSS framework: *FCCellManager*, *FCCellData* and *FCCellLoader*. An UML-diagram of the framework extension for implementing dynamic cell management is shown in Fig. 30, the classes themselves are explained in the next sections.

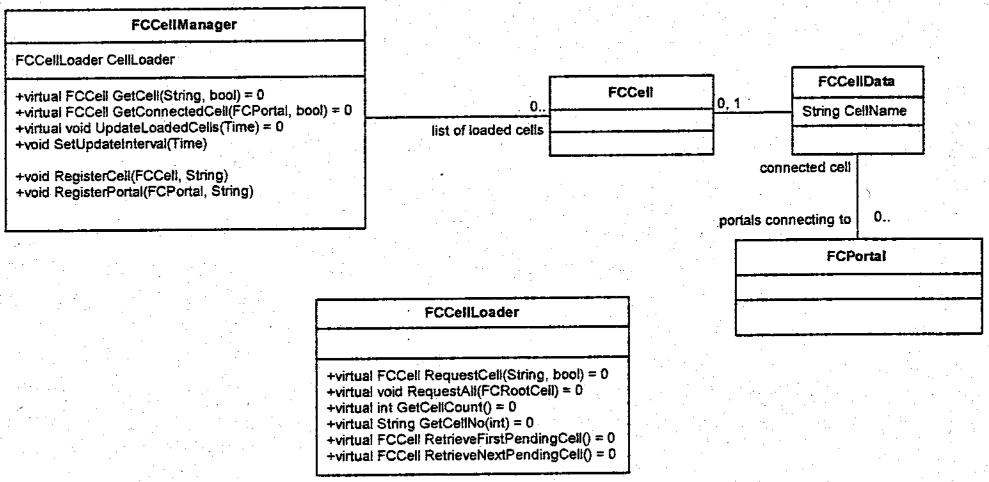

FCCellManager
FCCellLoader CellLoader
+virtual FCCell GetCell(String, bool) = 0
+virtual FCCell GetConnectedCell(FCPortal, bool) = 0
+virtual void UpdateLoadedCells(Time) = 0
+void SetUpdateInterval(Time)
+void RegisterCell(FCCell, String)
+void RegisterPortal(FCPortal, String)
0..
list of loaded cells
FCCell
0, 1
FCCellData
String CellName
connected cell
portals connecting to
0..
FCPortal
FCCellLoader
+virtual FCCell RequestCell(String, bool) = 0
+virtual void RequestAll(FCRootCell) = 0
+virtual int GetCellCount() = 0
+virtual String GetCellNo(int) = 0
+virtual FCCell RetrieveFirstPendingCell() = 0
+virtual FCCell RetrieveNextPendingCell() = 0

Fig. 30 UML diagram of framework extension for dynamic cell management.

4.2.1 The FCCellManager class

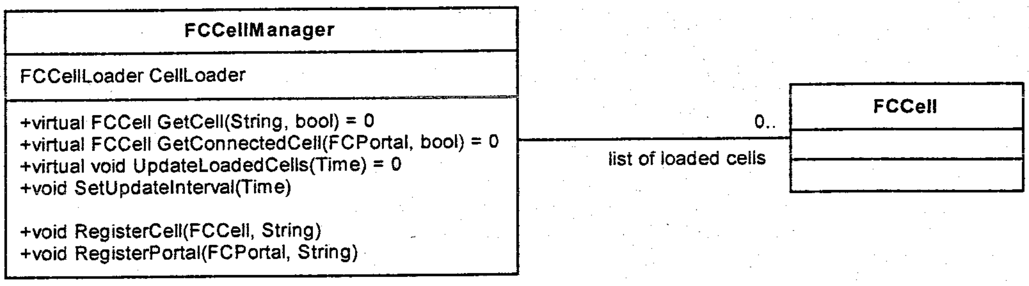

FCCellManager
FCCellLoader CellLoader
+virtual FCCell GetCell(String, bool) = 0
+virtual FCCell GetConnectedCell(FCPortal, bool) = 0
+virtual void UpdateLoadedCells(Time) = 0
+void SetUpdateInterval(Time)
+void RegisterCell(FCCell, String)
+void RegisterPortal(FCPortal, String)
0..
list of loaded cells
FCCell

Fig. 31 The *FCCellManager* class.

The task of the *FCCellManager* class is to make the numerous possible ways of loading and removing cells completely transparent to the rest of the system. To do so the *FCCellManager* class defines a few abstract methods to be implemented by a derived class that must be used by the rest of the system when requesting access to a cell tree. These methods are:

- *FCCell GetCell(String cellname, **bool** immediately)* to get (a pointer to) the root node of the cell tree, and;
- *FCCell GetConnectedCell(FCPortal p, **bool** immediately)* to get (a pointer to) the root node of the cell tree of the connected cell of portal p.

When set to true the Boolean *immediately* is used to signal the method not to return before the cell tree of cell *cellname* is loaded into memory, when set to false the manager can decide to load the cell at a later point in time and in the mean time e.g. return an empty cell tree.

The cell manager also defines the abstract *UpdateLoadedCells* method that is called every *interval* seconds to evaluate which cells (if any) can/will be removed from memory. The interval can be set using the *SetUpdateInterval(Time interval)* method.

4.2.2 The FCCellData class

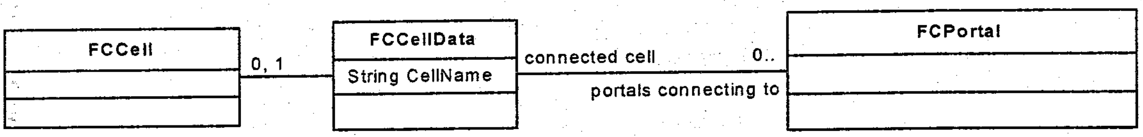


Fig. 32 the *FCCellData* class.

Due to the dynamic nature of the set of cells currently loaded into memory, a portal cannot simply store a pointer to its connected cell's cell tree: At the time of creating the portal the connected cell may not exist, and once the connected cell is created it cannot set the pointer for the portal connecting to it because it has no knowledge of its existence due to the separation of the cell trees (see chapter 3).

So whenever the connected cell of a portal is needed, the portal processing algorithm has to search the world graph for that cell using the only piece of information the portal has of it, the cell name. This is a slow process especially since it is performed multiple times during the rendering of a frame. To speed up this search, the cell manager could keep a list of all cells currently loaded allowing a look-up of the cell in this list by name. If the cell is not in the list, the cell manager will load it into the cache.

This string look-up is still wasteful considering that once a cell has been loaded into memory its pointer will never change and can safely be stored by portals to speed up access to the connected cell. Only when the cell is removed again a problem arises because now the pointer becomes invalid.

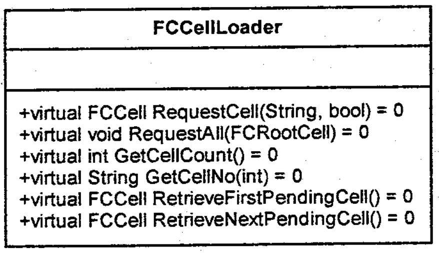
To solve this problem the *FCCellData* class is introduced which restores a bi-directional link by providing the connected cell of a portal with information of that portal connecting to it. For each cell, one instance of an *FCCellData* object is created once the cell, or a portal connecting to it, is created. The *FCCellData* objects holds a list of portals connecting to that cell, and holds a pointer to the cell tree if it exists and is NULL otherwise. Both the cell and the portals connecting to it keep a pointer to the *FCCellData* object.
Now when the connected cell of a portal is required this no longer has to be looked up in a list of loaded cells, but instead the pointer to the cell tree in its *FCCellData* object is evaluated: if it is NULL the cell tree is not loaded so the cell manager must retrieve it.

In order to set this pointer when creating a cell or portal, cells and portals must register themselves with the cell manager which performs this task. Requiring them to do so also has another advantage: due to the syntax of the registration methods (see Fig. 31) the programmer is forced to provide the engine with all the information required for portal processing. In the case of registering a cell both a pointer to the cell tree and the name of that cell must be passed as parameters and in the case of a portal both a pointer to the portal and the connected cell name must be passed.

Once a portal or a cell is deleted, the *FCCellData* object is updated so the information it contains about the world is always correct. If a cell tree no longer exists and no more portals connecting to that cell exist then the object deletes itself.

The only problem that remains is setting the *FCCellData* pointer of the cell and portals by the cell manager. This is done using a string look-up function as was described above[9]. This lookup has to be done only once when a cell or portal is created after which the *FCCellData* object can be used to get the cell pointer.

4.2.3 The FCCellLoader class



```
+virtual FCCell RequestCell(String, bool) = 0
+virtual void RequestAll(FCRootCell) = 0
+virtual int GetCellCount() = 0
+virtual String GetCellNo(int) = 0
+virtual FCCell RetrieveFirstPendingCell() = 0
+virtual FCCell RetrieveNextPendingCell() = 0
```

Fig. 33 The *FCCellLoader* class.

Because cells can be stored on and retrieved via many different mediums which may require substantially different accessing methods, and also to keep the cell manager as flexible as possible, the loading of cells is done using an instance of the abstract *FCCellLoader* class which defines a set of methods that can be used by an implementation of the abstract *FCCellManager* class to load cells into memory.

Again, an *immediately* Boolean is used by method *RequestCell* to signal the loader whether or not it has to return the cell tree immediately. If a cell tree is retrieved later, it can be retrieved by the cell manager using the *RetrieveFirstPendingCell* and *RetrieveNextPendingCell* methods.

4.3 Summary

There are as many different approaches to dynamic cell management as there are situations in which to use it, and each approach and situation has its own specific set of advantages and problems. Therefore creating a cell manager for a certain situation will require a thorough examination of the situation and the problems that must be overcome. Although in this chapter some hints were given on how one could create a cell manager for some common situations, these are not the only possible approaches to solving the problems and there may be other conditions making the hints given difficult to use.

Because of this the addition of dynamic cell management to vrSS is less an actual implementation then it is an interface a programmer can use to integrate his cell manager into the portal processing framework designed for vrSS.

[9] Nevertheless, searching the tree is also possible if memory resources are of concern.

Chapter 5: Inter-Cell Objects and Dynamic Portals

Until now, it was assumed that a VR-world could be completely partitioned into separate cells whose only inter-relation was the portals connecting them. However objects like light- and sound sources can extend their influence beyond their physical cell, and a moving (*dynamic*) object can be partially inside two cells at the same time when moving through a portal. These objects cause extra inter-relations between cells and are therefore called *inter-cell objects*.

Inter-cell and especially dynamic inter-cell objects are often a problem when it comes to static visibility determination schemes like BSP-trees [2], and even most research papers [3], [4], [7] on portal processing (which is a dynamic visibility determination solution) never mention them. Often when a world has dynamic objects they are simply processed every frame (using the camera frustum for clipping) and visibility determination is completely left to the Z-buffer. This approach always works correctly but sacrifices the frustum narrowing properties of portals leading to the unnecessary rendering of invisible dynamic objects, thus reducing performance.

This chapter explores how processing inter-cell objects can also profit from frustum narrowing using portal processing and solves the problems that arise. The problems introduced by adding inter-cell objects to the world are identified and, when possible, solved.

In addition, there is no reason why portals should remain static. Imagine a sliding door where a portal is used for determining what is visible on the other side of the door. Using a static portal, this would have to be the size of the maximum door opening resulting in a lot of redundant rendering of objects only to be overdrawn by a half open door, but using a dynamic portal that changes along with the door opening this can be prevented.

Dynamic portals and their consequences are also discussed in this chapter. The chapter then continues with the framework extension required for using inter-cell objects in the vrSS portal processing environment created in the previous chapter and ends giving a summary of the results.

5.1 The problem with objects influencing the appearance of more than one cell

With portal processing the world tree is partitioned into cell trees where each cell tree holds the objects defining the appearance of that cell. The only way these objects will be processed using the current portal processing algorithm is when a visible portal connecting to that cell is processed (or if the cell is the physical camera cell). However, consider the situations depicted in Fig. 34:

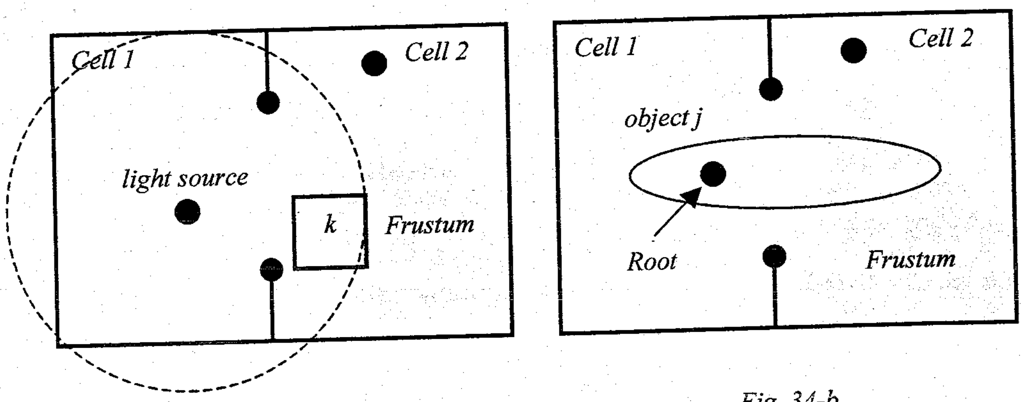


Fig. 34-a *Fig. 34-b*

Fig. 34 Scene errors in the portal processing environment.

Both situations will result in an erroneous rendering of the final scene! In the case of Fig. 34-a the light source that is a child of *Cell 1* shines it light through the portal into *Cell 2* lighting object *k* (the sphere of light is depicted by the dotted line). However, since the portal is not visible (it is not in the frustum) the cell tree of *Cell 1* will not be processed and therefore neither will the light source leaving the object unlit.

In the case of Fig. 34-b the object *j* is a child of *Cell 1* (its root is in *Cell 1*) but sticks through the portal into *Cell 2* where it intersects the frustum and is therefore visible. Again, since the portal is not visible *Cell 1* will not be processed and thus *j* is not rendered.

The problem is caused by the fact that objects belonging to the cell tree of one cell can influence the appearance of neighboring cells through portals. However due to the division of the world into separate cell trees and the properties of a tree, if the parent cell of the object is never processed, then neither will the object be even if its influence is visible in another cell.

5.1.1 Solving the problem using global objects

The easiest approach to solve this problem is to make these objects *global* (see chapter 3): global objects have no cell parent and are processed every frame using the initial (camera) frustum, so the problems described above cannot occur. However, there are disadvantages:.

- Global lights/sounds are not restricted by cell boundaries, so a light source in one cell could shed light on an object in another cell through an opaque wall;
- Global objects are processed every frame using the initial frustum, so the advantage of frustum narrowing is sacrificed (see Fig. 35);

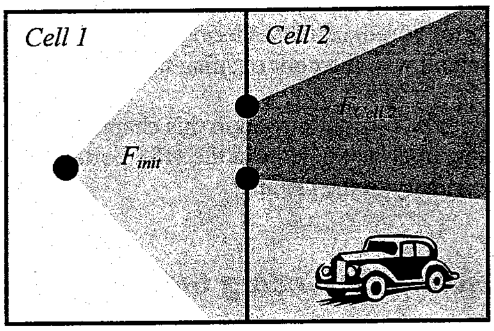


Fig. 35 Global object 'car' is visible in F_{init} but not in F_{Cell2}.

- The parent-child relation is lost, forcing the programmer to move the global object along 'manually' with the cell it is in, if that cell is moved.

Due to these disadvantages, global objects should only be used for objects that must be processed every time a frame is rendered, e.g. a global (ambient) light source or a heads-up display.

5.1.2 Solving the problem using object referencing

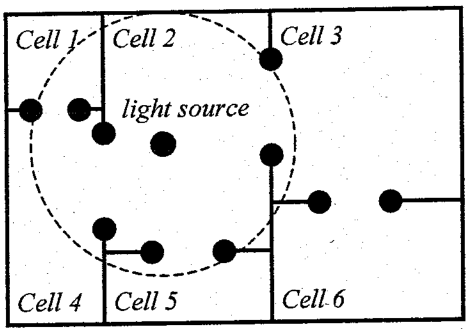
A better solution is to have all cells whose appearance can be influenced by the object keep a reference to that object (which in effect makes the object a child of multiple trees). Now, when a cell is processed not only will its cell tree be processed, but also any object references the cell has to objects in other cell trees.

Using this approach all cells influenced by the object must be determined so they can get a reference to the object. This can be done using the recursive algorithm shown in Fig. 36:

```
void SetObjectRefs( Object o, FCCell c = NULL )
{
  // add an object reference to the current cell
  if( c )
    c.AddRef( o ) ;
  else
    c = o.GetParentCell() ; // only on initial call

  // prevent infinite recursion
  c.CellLocked = true ;

  // now check whether the portals of the current cell
  // intersect the objects shape
  for( int f = 0 ; f < c.Portals.Count() ; f ++ )
    if( !c.Portals[ f ].ConnectedCell.CellLocked )
      if( c.Portals[ f ].IntersectsWith( o.Shape ) )
        SetObjectRefs( o, c.Portals[ f ].ConnectedCell ) ;

  c.CellLocked = false ;
}
```

Fig. 36 The recursive object-referencing algorithm.

When using this algorithm all cells that have a portal connecting to them that intersects with the object's shape[10] get a reference to the object (except for the actual parent cell of the object who already has an explicit reference to the object in its cell tree). This technique is called *object referencing* and an example of the references found for a light source is shown in Fig. 37.



*Refs(light source) = { Cell 1, Cell 3,
Cell 4, Cell 5 }*

Fig. 37 Object referencing (dotted line is the shape of the object).

[10] For light and sound sources this is usually a sphere of influence, for geometry it is either the actual render shape of a bounding volume.

5.1.3 Inter-cell objects and occlusion culling

When using light sources (and to a lesser degree, sound sources) in a portal processing environment a strange phenomenon can be observed in certain situations as is shown in Fig. 38:

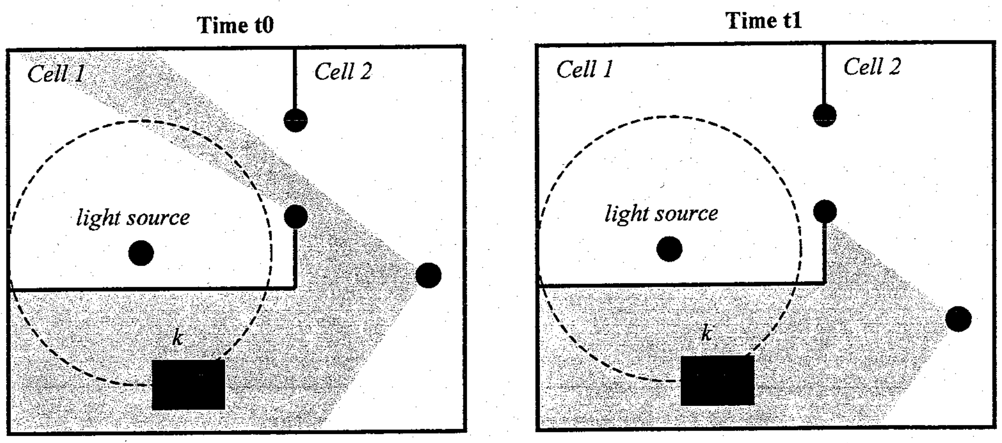


Fig. 38 Flickering light problem.

At time *t0* the portal is visible and therefore the light source in *Cell 1* will be processed. This is usually done by sending the light source to the render API which then uses it to light the other objects it renders. When lighting objects most render API's (e.g. DirectX, OpenGL) do not check for occlusion of that object by other objects. In Fig. 38, due to the visibility of *Cell 1* at time *t0*, the light source will be processed and the render API will erroneously light object *k*, which is in the sphere of influence of the light source (dotted line), through the opaque wall. When the camera moves to its new position at time *t1*, *Cell 1* is no longer visible so object *k* is now no longer lit by the light source. This results in a *'flickering light'* that can be very annoying to a viewer.

The source of this problem is not the portal processing algorithm, but the way in which render API's handle lighting. To make real-time rendering of complex scenes possible the render API does not check for occlusion culling when lighting objects. Therefore, when rendering the scene using the tree-processing algorithm the exact same problem is present, however it is less noticeable because the scene will be lit erroneously all the time.

To make the problem less noticeable when using portal processing, the occlusion culling properties of portals can be used for light as well as is shown in Fig. 39:

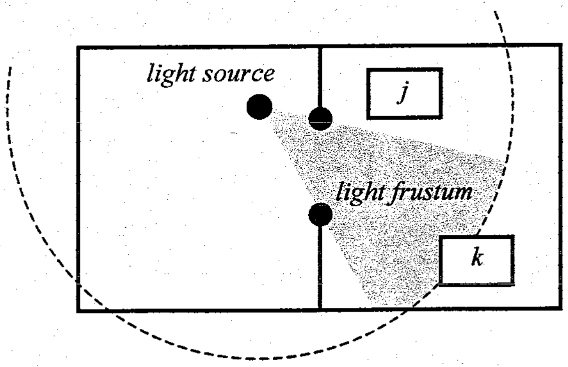


Fig. 39 Light Frustums.

As can be seen from this figure using the light source and the portal a 'light frustum' can be constructed that can be used to evaluate which objects are lit by the light source in other cells and which aren't. Using this property requires that for each object the sources lighting it can be set in the render API[11]. Selecting the light sources for each object can be done at pre-run-time for static objects and light sources, or run-time but then it requires an extra processing pass where the light sources for each object are selected.

5.2 The problem with dynamic objects

Dynamic objects in a portal processing environment introduce two problems that have to be solved.

The first problem is caused by the fact that most dynamic objects have a shape requiring the objects to be referenced by the cells in which they are visible (for which the object referencing technique was introduced in the previous section). However, since the objects can change, the cells requiring a reference to the object can also change. So i.e. whenever the object moves the cells requiring a reference to the object must be re-evaluated. This is called *dynamic object referencing* and an example showing the references of an object *k* at three different times is shown in Fig. 40:

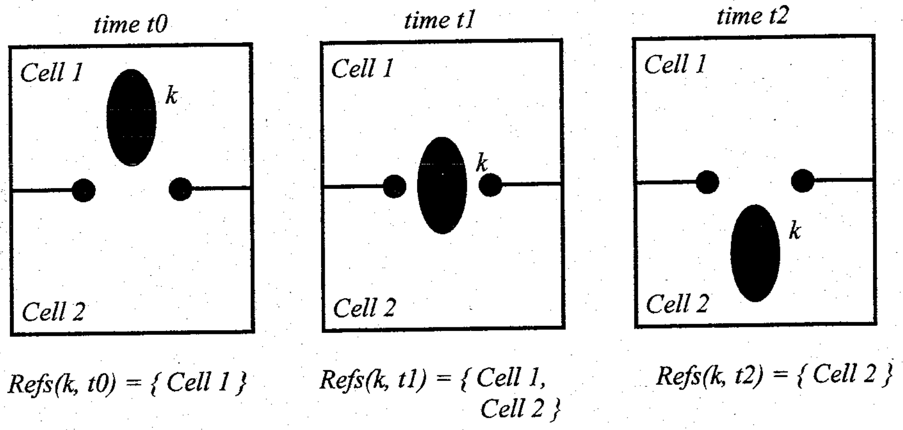


Fig. 40 Object references for *k* at 3 different times.

The second problem is caused by the question whether or not an object passing through a portal should change parent cells and become a child of the connected cell's cell tree. Both situations may occur based on the properties of the object:

In the case of a *Man* walking through a portal connecting a *Tunnel* cell to a *Boat* cell, as long as the *Man* in inside the tunnel he is not affected by the rocking of the boat, but once he passes through the portal on to the boat he will rock along with the boat.
So the *Man* should change parent cells from the *Tunnel* cell to the *Boat* cell once he passes through the portal so that because of the parent-child relation he will move along with any movement of the *Boat* cell. This type of moving object is called a *slave object* because it will always be a child of the cell tree of the cell in which it physically resides.

In the case of a *Ufo* object hovering through a series of *Tunnel* cells, if the cell in which the u.f.o. is hovering moves, the u.f.o. should not automatically move along since the two have no physical connection. In this case, the *Ufo* object should not change parent cells when moving through a portal. This type of moving object is called a *free object.*

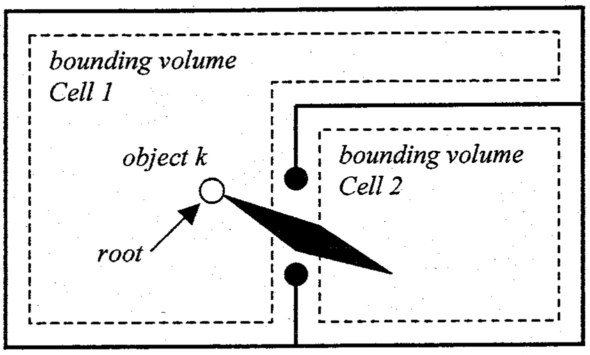
Implementing free objects can be done by making them global but *only* processing them through object references found using dynamic object referencing. This way, the object has no parent cell, and is only processed using the frustum(s) for the cell(s) in which it physically resides.

Implementing slave objects requires that the object changes parent cells once it moves through a portal to keep the parent-child relations correct. This brings up two questions of which the first is: Most objects have a shape and size so when moving through a portal it will temporarily be physically in two (or more) cells at once. At what point should the

[11] Although DirectX and OpenGL allow switching lights on and off while rendering a frame, unfortunately vrSS has no provisions for this yet. It's an all-lit or all-unlit deal due to the separate lighting pass used by vrSS when processing the tree.

object now change its parent cell? To solve this problem the design decision is made that once the root of the slave object tree (which is a single point with a position but no size) is physically inside a cell the slave object should be a child of that cell. This requires that a modeler chooses the position of the root of an object wisely (e.g. NOT two meters removed from the object!).

The second question is how to detect when the root of an object has moved into another cell. There are basically two different approaches to detecting whether an object has moved into another cell: the *point-in-cell* check and the *line-intersects-portal* check. Fig. 41 shows the point-in-cell check:



```
FCCell PointInCellCheck( Object o )
{
  // get the position of the root of the object
  Vector p = o.GetWorldPosition() ;

  // find the cell in whose bounding volume this root lies by searching a list
  // of all cells in the world
  for( int f = 0 ; f < Cells.Count() ; f ++ )
    if( Cells[ f ].BoundingVolume.PointInBoundingVolume( p ) )
      return Cells[ f ] ;

  // root is not inside any bounding volume->ERROR!
  return NULL ;
}
```

Fig. 41 Point-in-cell check.

When using the point-in-cell check every cell must have a bounding volume (whose outline is shown using a dotted line which, for clarity purposes, is drawn within the actual bounding volume which would coincide with the cell walls) By finding the physical cell for the root of the object whenever it has moved, and comparing this cell to the physical cell of the old root position, it is easy to determine whether the root has changed physical cells.

The biggest advantage of this approach is that an object can be randomly positioned in the world and the algorithm will make sure the object becomes part of the correct cell tree. The disadvantages are that a bounding volume must be created for every cell and that these bounding volumes may not overlap to keep determining the physical cell of a point using the point-in-cell check simple (see Fig. 42).
Another disadvantage is that it is never checked whether the object has actually passed through a portal or not.

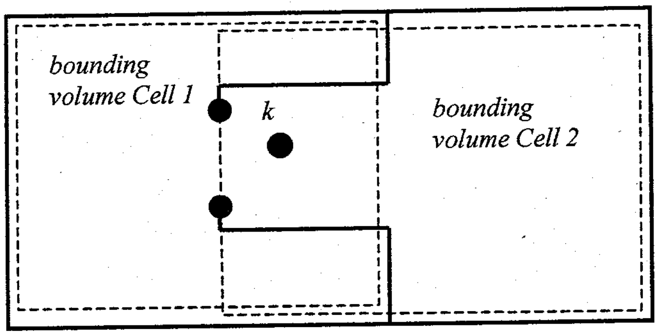
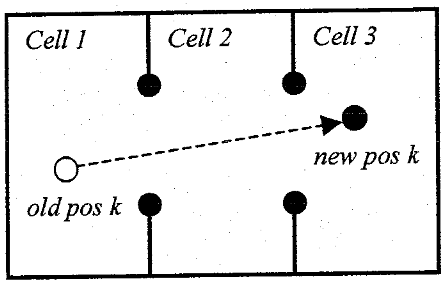


Fig. 42 *k* lies in two bounding volumes, which one should be the physical cell for *k*?

The line-intersects-portal check is shown in Fig. 43:



```
FCCell LineIntersectsPortalCheck( Object o )
{
  // start in the current parent cell of the object
  FCCell cpc = o.GetParentCell() ;

  // create the line from the old position of the root to the new one
  Line l = Line( o.LastWorldPosition, o.GetWorldPosition() ) ;

  bool exit = false ;
  while( !exit )
  {
    exit = true ;  // for ending loop

    // check the portals of the current parent cell
    for( int f = 0 ; f < cpc.Portals.Count() ; f ++ )

      // on intersect update the parent cell pointer
      if( cpc.Portals[ f ].LineIntersects( &l ) )
      {
        cpc = cpc.Portals[ f ].ConnectedCell ;
        exit = false ;
        break ;
      }
  }

  return cpc ;
}
```

Fig. 43 line-intersects-portal check.

If an object moves, this method checks whether the line between the old and the new root position intersects with one of the portals of its current physical cell. If so then the search is continued in the connected cell of the portal and now for this cell the line is checked against its portals[12]. This process continues until no new portals are found that intersect the line. The last cell whose portals were checked for intersection with this line is now the new physical cell of the object.

[12] Making sure that the (bi-directional) portal through which the line entered is not checked again for intersection as this would lead to an infinite recursion!

As can be seen most of the (dis)advantages given for the point-in-cell check are the exact opposite with the line-intersects-portal check: no bounding volumes are required and the object must actually pass through the portal in order to enter another cell, but care must be taken that the object is initially positioned within its physical cell.

5.3 Dynamic portals

One last problem that must be taken into account is *dynamic-* or *moving portals*. When a world has moving portals, not only can a moving object pass through a (static) portal, but a moving portal can also 'pass over' a (moving or static) object. This is shown in Fig. 44 where due to the moving of *Cell 2* to the left, even though object *k* did not move between time *t0* and *t1*, at *t1* the object is now inside *Cell 2* because the portal passed over the cell.

If the portal processing algorithm only checks for physical cell changes of objects when the objects move then this situation is not detected. Therefore not only must the algorithm check for physical cell changes when an object moves, but also whenever a portal of the objects physical cell has moved. Moving portals also require that the object references by cells are updated after a move of the portal as can be seen in Fig. 45.

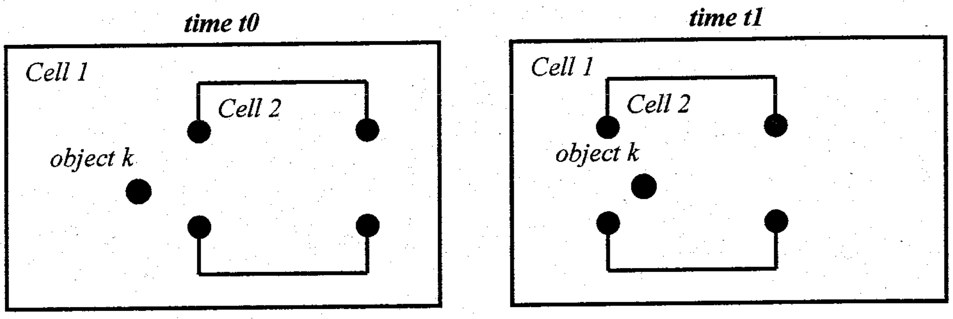


Fig. 44 A moving portal.

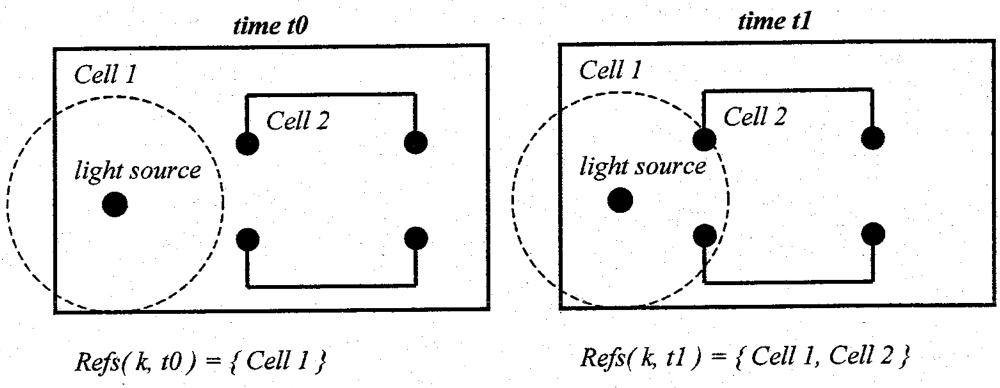


Fig. 45 Object referencing when moving portals (the light source is static).

5.4 *Inter-cell objects and dynamic cell management*

Inter-cell objects requiring object referencing make dynamic cell management more difficult. When a visible cell *c* is under the influence of a light source in a neighboring cell *n* that has not yet been loaded, the appearance of *c* is abruptly changed once *n* is loaded into memory. To prevent this for static inter-cell objects, a pre-run-time evaluation of the cells under the influence of an inter-cell object can be made so that if one cell under the influence of the inter-cell object becomes visible, all other cells under the influence of that object are also loaded by the cell manager.

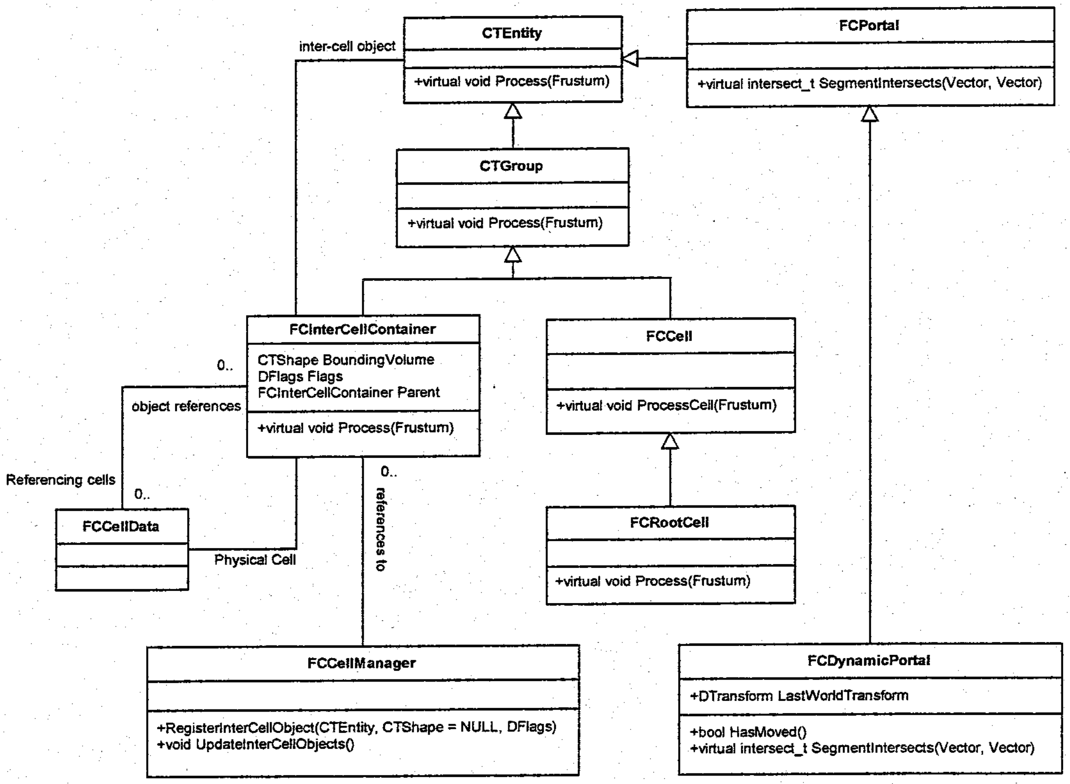
Should the inter-cell object also be able to move, then this evaluation must be done at run-time after every move of the object. Because this requires every cell under the influence of the object to be loaded in order to evaluate their portals, this can require many invisible cells to be loaded which could be an expensive task. In the case of a dynamic light source, it would probably be cheaper to make the light source a global object.

The same is true for dynamic portals: re-evaluation of object references due to changes of the portal must be done at run-time, requiring every cell under the influence of the object to be loaded in order to evaluate their portals.

When removing cells from memory, the cell manager must take into account that some cells may not be visible but are under the influence of inter-cell objects so they should not be removed indiscriminately.

5.5 *Implementing inter-cell objects in vrSS*

In order to implement inter-cell objects and dynamic portals for the vrSS engine two new classes are introduced into the vrSS framework: *FCInterCellContainer* and *FCDynamicPortal.* Most classes already defined in the previous chapters will also require some additional code: The *FCCellManager* class (defined in chapter 4) is expanded with new functionality for managing and updating inter-cell objects, the *FCPortal* class (defined in chapter 3) must check if an inter-cell object has passed through it in order to update that object's physical cell and has to signal the cell manager whenever the portal has changed/moved in order to update the object references and finally the *FCCell::ProcessCell* and the *FCRootCell::Process* methods (also defined in chapter 3) also require some additional functionality.
An UML-diagram of the framework extension for implementing inter-cell objects in vrSS is shown in Fig. 46:



CTEntity
+virtual void Process(Frustum)
inter-cell object
FCPortal
+virtual intersect_t SegmentIntersects(Vector, Vector)
CTGroup
+virtual void Process(Frustum)
FCInterCellContainer
CTShape BoundingVolume
DFlags Flags
FCInterCellContainer Parent
+virtual void Process(Frustum)
0..
object references
Referencing cells
0..
FCCellData
Physical Cell
0..
references to
FCCell
+virtual void ProcessCell(Frustum)
FCRootCell
+virtual void Process(Frustum)
FCCellManager
+RegisterInterCellObject(CTEntity, CTShape = NULL, DFlags)
+void UpdateInterCellObjects()
FCDynamicPortal
+DTransform LastWorldTransform
+bool HasMoved()
+virtual intersect_t SegmentIntersects(Vector, Vector)

Fig. 46 UML-diagram of framework extension for inter-cell objects.

The line-intersect-portal check was chosen for detecting physical cell changes of inter-cell objects due to the fact that using this approach modelers/programmers don't have to provide/update bounding volumes for every cell and it allows the greatest degree of freedom when designing VR-worlds.

5.5.1 The FCInterCellContainer class

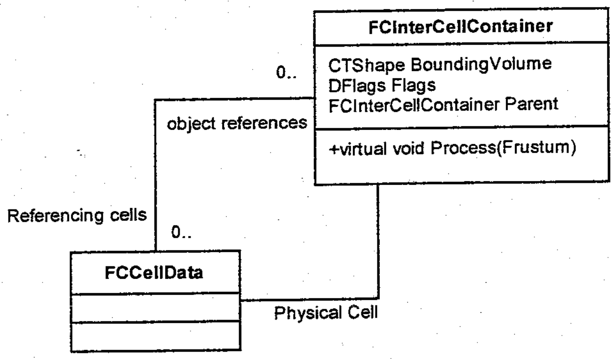


Fig. 47 The *FCInterCellContainer* class

The *FCInterCellContainer* class is derived from *CTGroup* but only contains one child, the inter-cell object. The *FCInterCellContainer* object stores a pointer to the *FCCellData* object of the cell in which the root of the inter-cell object currently physically resides (recall that in the case of a slave object this will always be the slave object's parent cell), and is provided with a shape (currently only a bounding box or sphere are supported) that is used for (dynamic) object referencing of its inter-cell object.
The *FCInterCellContainer* class has a list of *FCCellData* objects that reference the object, so that when the object is removed from the world this list can be used to quickly remove the object reference from these cells, and also the last transform of the inter-cell object is cached so that any movement of the object requiring an update of the object references can be detected.

5.5.1.1 Creating an inter-cell object

To create an inter-cell object the container for that object must be set using the method *FCInterCellContainer::SetEntity(CTEntity entity, CTShape shape, DFlags flags, FCCell physicalcell).*

The first parameter *entity* is the object that becomes the inter-cell object. Any type of vrSS tree node can be an inter-cell object, both leaf nodes (like lights, sounds and geometry) as well as group nodes. The container positions itself in between the entity and that entities parent in the cell tree.

The second parameter *shape* provides the *FCInterCellContainer* with the shape to be used for (dynamic) object referencing. There are currently two types of shapes allowed: a sphere or a box. A sphere will mostly be used for light and sound objects while a box is more common for geometry objects. Using a box instead of the actual geometry has the advantage that intersection testing is performed much faster, but has the disadvantage that it can result in erroneous object references as can be seen in Fig. 48: the bounding box intersects the portal however the object does not, so when *Cell 2* is processed object *k* is being processed unnecessary since it will not be visible in *Cell 2*.

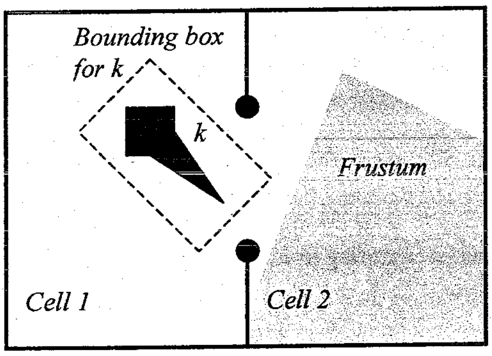


Fig. 48 Bounding box intersects portal.

Usually a hybrid form is adapted: first use the cheap bounding volume check to determine whether the object *might* be visible, and then use the exact geometry check to determine if the object *is* visible. The current implementation only does the bounding volume check to maximize the performance. It allows for the erroneous object reference inefficiency based on the observation that inter-cell objects usually move resulting in this reference to exist only for a short time, while using the expensive geometry-portal intersection test will usually result in a much higher performance loss.

If the *shape* parameter is NULL the inter-cell object is considered to have no size. The camera is the most common example of such an inter-cell object. A camera has no size, but because it is usually a moving object whose physical cell must be kept track of (recall that rendering in the portal processing environment always starts in the physical camera cell) this is easiest done using an inter-cell object. Obviously, object referencing can be skipped for inter-cell objects with no shape since they can only be in one cell at a time.

5.5.1.2 Inter-cell object flags

The third parameter *flags* of the *SetEntity* method is used to configure how the inter-cell object will be processed. The flags are:

- **ICCF_DYNAMIC**: by setting this flag dynamic object referencing will be used to update the physical cell of the object and its referencing cells whenever the object moves. By default an inter-cell object is assumed to be static, so object referencing is only required when the object is created and whenever a portal is added[13].

- **ICCF_DONT_UPDATE_REFS** and **ICCF_REFS_SET**: the ICCF_REFS_SET flag is a status flag that signals when the object references for the object have been set. The ICCF_DONT_UPDATE_REFS flag is used in situations where the code will unnecessarily update the object references. Imagine the situation depicted in Fig. 49:

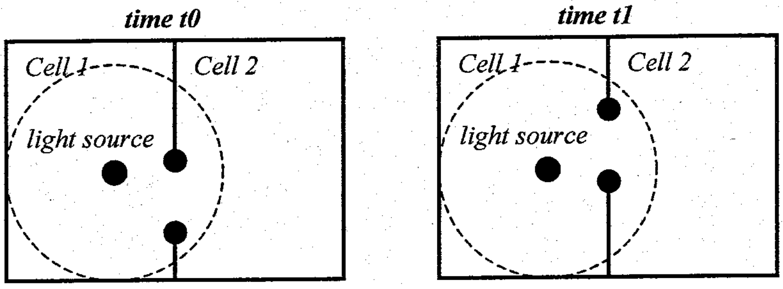


Fig. 49 Object referencing with moving portals.

The portal will only move in between the two extremes shown at time *t0* and *t1* so the object references to the light source never change. A programmer can make use of this observation by setting the ICCF_ DONT_UPDATE_REFS flag. Now object referencing is only performed when the ICCF_REFS_SET flag is not set which is when the inter-cell object is first created.
When at some point during program execution an update of the object referencing cells is required the ICCF_REFS_SET flag can simply be reset again resulting in a new single update.

- **ICCF_FREE_OBJECT**: By default an inter-cell object is assumed to be a slave object resulting in the object changing parent cells when passing through a portal. When the ICCF_FREE_OBJECT flag is set only the physical cell pointer of the *FCInterCellContainer* is changed when moving through a portal, but not the parent cell.

- **ICCF_RESET**: This is an internal flag used to signal the cell manager that the object references of the inter-cell object must be updated.

[13] The new portal can intersect with the shape of the inter-cell object thus requiring an update of the object references.

The fourth and final parameter is optional and in the case of free objects sets the initial physical cell for the object. If this parameter is not provided *SetEntity* will search the tree from the entity upwards for the parent cell that will become the initial inter-cell objects physical cell.

When a slave object passes through a portal the cell container (and thus also the inter-cell object) will change parent cells. This requires that in the case of slave objects the physical cell of the object must always be loaded into memory but also in the case of global objects the physical cell because otherwise the object can never again leave the cell because no portals exist. Therefore, the cell manager should never remove cells from memory that have inter-cell objects in them.

5.5.1.3 Child Inter-cell objects

An inter-cell object can be a child of another inter-cell object and in that case should never change parent cells when moving through a portal, unless the parent inter-cell object also moves through this portal. Fig. 50 clarifies this showing the cell trees of *Cell 1* and *Cell 2* at three different times:

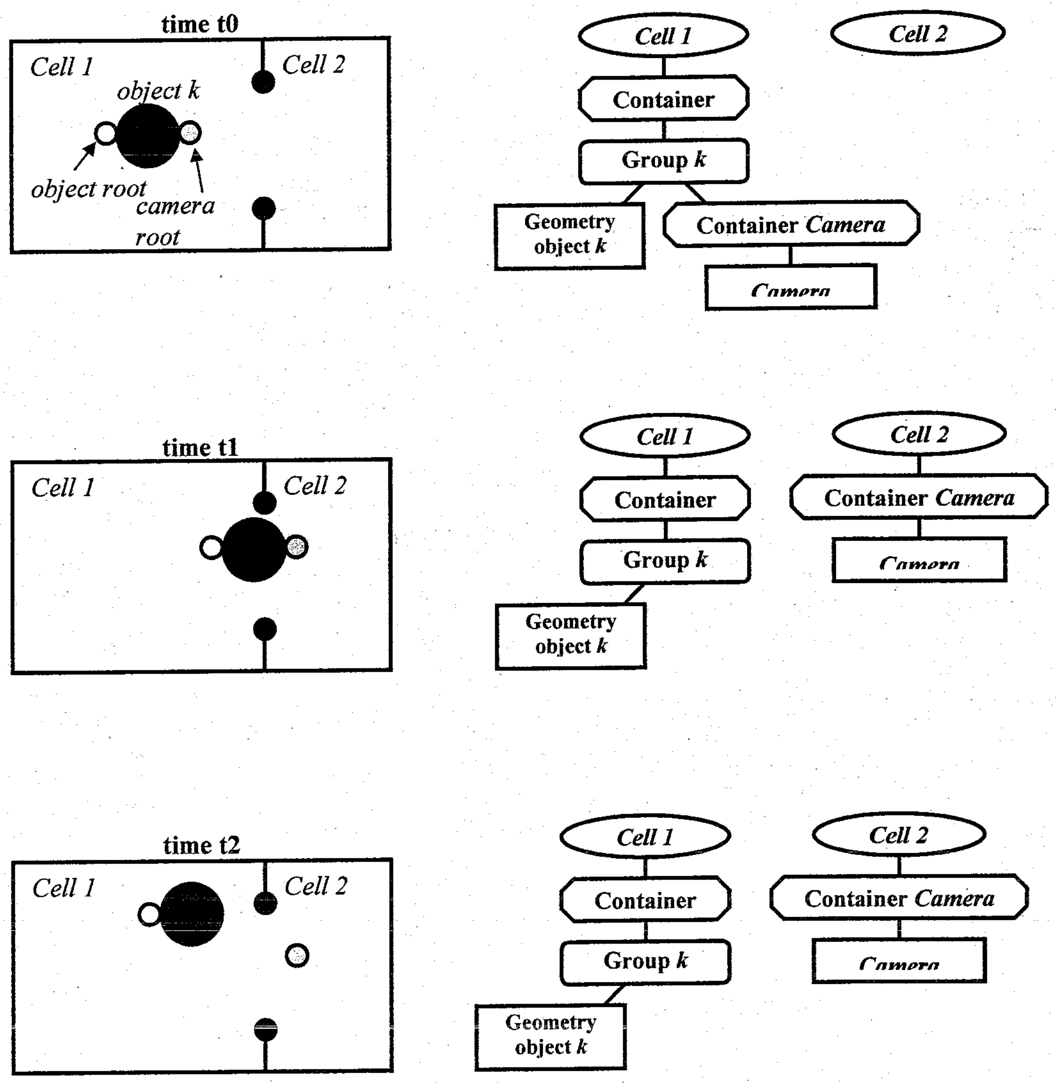


Fig. 50 Child inter-cell objects.

The camera is attached to object *k* and because a camera is automatically converted into an inter-cell object the tree for *k* will look like the tree in Fig. 50 at time *t0*. At time *t1* the camera has moved physically into *Cell 2* but the root of *k* is still in *Cell 1*. If the camera container were to change parent cells now it would detach itself from object *k* and would no longer move along as is shown at time *t2*.

5.5.2 The updated FCRootCell::Process method

As a pre-condition to rendering a new frame all object references must be updated. Rendering a frame using portal processing always starts in the root cell and this cell is processed only once per frame, so updating the inter-cell objects is done in the *FCRootCell::Process* method as is shown in Fig. 51 (lines that are printed in italic represent the changes):

```
void FCRootCell::Process( Frustum Fr )
{
  UpdateAllInterCellObjects() ;

  // process all global objects
  FCCell::Process( Fr ) ;

  // get the cell the camera currently resides in
  FCCell cpc = ( ( FCInterCellContainer )( GetCamera().GetParent() ) ).GetPhysicalCell() ;

  // start the portal processing algorithm
  cpc.ProcessCell( Fr ) ;
}
```

Fig. 51 The updated *FCRootCell::Process* method.

After the inter-cell objects (including the camera) have been updated and the global objects have been processed the physical cell of the camera is retrieved. Now the portal processing algorithm is started by calling that cell's *ProcessCell* method.

The *FCRootCell* class also automatically converts a camera into an inter-cell object to make this process transparent to the programmer (this is not shown in Fig. 51).

5.5.3 The updated FCCellManager class

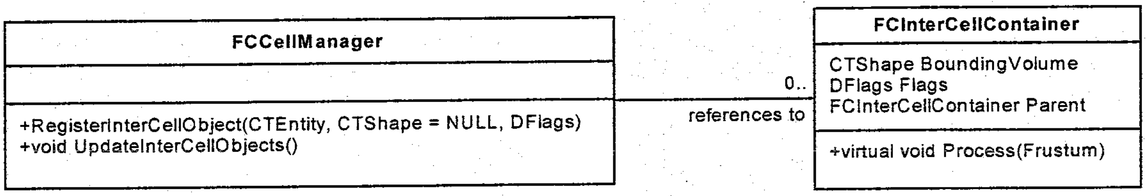

FCCellManager
+RegisterInterCellObject(CTEntity, CTShape = NULL, DFlags)
+void UpdateInterCellObjects()
0..
references to
FCInterCellContainer
CTShape BoundingVolume
DFlags Flags
FCInterCellContainer Parent
+virtual void Process(Frustum)

Fig. 52 The updated *FCCellManager* class.

Until now the task of the cell manager was to manage the loading of cells into memory and removing them from memory. The cell manager is now extended to also be responsible for managing and updating inter-cell objects. This choice has been made because managing and updating inter-cell objects works together closely with (dynamic) cell management.

In order to keep a uniform programming interface for working within the portal processing framework, inter-cell objects are created by registering them with the cell manager just as were cells and portals. When registering the *FCInterCellContainer::SetEntity* method is called resulting in the container being added to the cell tree.

Inter-cell objects are updated at the start of every frame by a call from the root cell (see Fig. 51) to the cell manager's *UpdateInterCellObjects* method, which is shown in Fig. 53:

```
void FCCellManager::UpdateInterCellObjects()
{
  // Pre-condition: if any portal has changed vertices the PortalsChanged flag will
  // be set to true.

  // update the dynamic portals
  for( int f = 0 ; f < DynamicPortals.Count() ; f ++ )
    PortalsChanged |= DynamicPortals[ f ].HasMoved() ;

  // the inter-cell object references must be updated
  if( UpdateReferences )
    for( int f = 0 ; f < InterCellObjects.Count() ; f ++ )
        InterCellObjects[ f ]->SetFlag( ICCF_RESET ) ;

  // update the inter-cell objects
  for( int f = 0 ; f < InterCellObjects.Count() ; f ++ )
  {
    // is this a dynamic object?
    if( InterCellObjects[ f ].GetFlag( ICCF_DYNAMIC ) )
    {
      // has the object moved?
      if( InterCellObjects[ f ].HasMoved() )
      {
        // find the physical cell for the object's root
        GetPhysicalCell( InterCellObjects[ f ] )
        UpdateObjectReferences( InterCellObjects[ f ] ) ;
      }
      else if( InterCellObjects[ f ].GetFlag( ICCF_RESET ) )
        UpdateObjectReferences( InterCellObjects[ f ] ) ;
    }

    // must the object references for the static inter-cell object be updated?
    else if( InterCellObjects[ f ].GetFlag( ICCF_RESET ) )
      UpdateObjectReferences( InterCellObjects[ f ] ) ;
  }

  // reset the PortalsChanged flag for the next frame
  PortalsChanged = false ;
}
```

Fig. 53 The FCCellManager's *UpdateInterCellObjects* method.

Whenever the vertices of a portal's polygon change (e.g. when creating a portal new vertices are added), the portal sets the cell manager's *ChangedPortals* flag to true. The *UpdateInterCellObjects* method then uses this flag, combined with a separate test of whether a dynamic portal has moved (see next section), to determine whether the cell references for all inter-cell objects should be updated and if so it sets the ICCF_RESET flag for those objects.
The method then continues with updating the inter-cell objects: for dynamic inter-cell objects it checks whether they have moved and if so whether the root has passed through a portal into a new physical cell using the *UpdatePhysicalCell* method. This method also changes the parent cell for slave objects that have passed through a portal. If the object has moved then also a call to *UpdateObjectReferences* is performed. This method checks whether or not the ICCF_DONT_UPDATE_REFS flag for the inter-cell object is set before the object references are updated. If the dynamic inter-cell object has not moved or in the case of static inter-cell objects the *UpdateObjectReferences* method is only called when the ICCF_RESET flag is set.

5.5.4 The FCPortal and FCDynamicPortal class

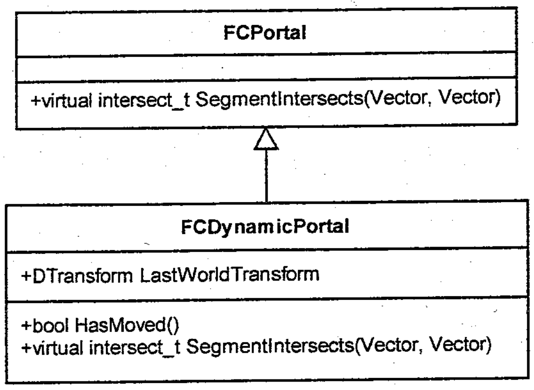

FCPortal
+virtual intersect_t SegmentIntersects(Vector, Vector)
FCDynamicPortal
+DTransform LastWorldTransform
+bool HasMoved()
+virtual intersect_t SegmentIntersects(Vector, Vector)

Fig. 54 The *FCDynamicPortal* class.

The portal class signals the cell manager whenever the vertices of its portal polygon change by setting the cell managers *PortalsChanged* flag. The portal class also implements the line-intersects-portal check in a virtual *SegmentIntersects* method. The default implementation (used for static portals) performs this check in world space. However, when using dynamic portals this implementation of the check can fail as can be seen in Fig. 55:

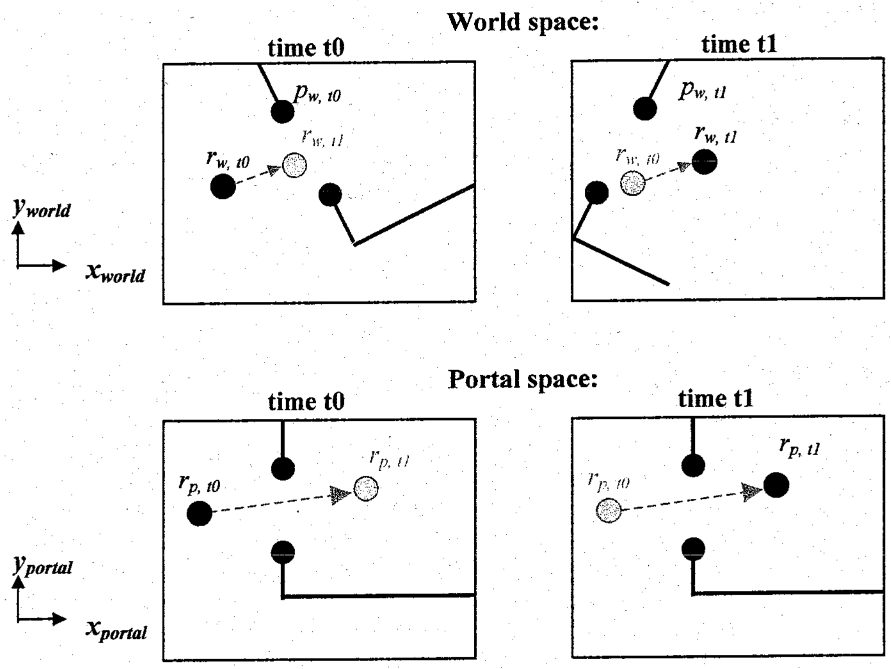


Fig. 55 Line-intersects-portal check in portal space.

Between time *t0* and *t1* object *r* moves from position $r_{w,\,t0}$ to $r_{w,\,t1}$ (at *t0* the position at *t1* is indicated using the gray dot and vice versa) in world space, and the portal (world transform) moves from $p_{w,\,t0}$ to $p_{w,\,t1}$. Using the line-intersects-portal check in world space will not detect the passing of *r* through the portal in this situation (both at *t0* and *t1* the line between $r_{w,\,t0}$ and $r_{w,\,t1}$ does not intersect the portal). However by transforming $r_{w,\,t0}$ into the local portal space for $p_{w,\,t0}$, resulting in $r_{p,\,t0}$, and transforming $r_{w,\,t1}$ into the local portal space for $p_{w,\,t1}$, resulting in $r_{p,\,t1}$, and then performing the line-intersects-portal check in the local space of the portal the passing through of *r* is detected.

Because this check is slightly more expensive (due to the transformation of the start- and end position of *r* into the local space), and also because in order to use this check the last world transform of the portal must be cached (in order to compute $r_{p,\,t0}$ at time *t1* which is the time when the check is performed!), a special *FCDynamicPortal* class has been derived from *FCPortal*. This class overrides the virtual *LineIntersects* method with the check described above and has an extra *HasMoved* method used by the cell manager when deciding whether to update all inter-cell object references.

The updated FCCell::ProcessCell method

The updated *ProcessCell* method is shown in Fig. 56 (changes are again shown in italic print). It should be noted that in the actual implementation the object references for a cell are stored in its *FCCellData* object.

```
// CellFr is the frustum constructed for current cell
void FCCell::ProcessCell( Frustum Fr )
{
  // lock the cell
  CellLocked = true ;

  // make sure the parents world transform is up-to-date
  // and then process the cell tree using vrSS process()
  GetParent().UpdateWorldSpace() ;
  CTGroup::Process( Fr ) ;

  // process all references to inter-cell objects
  for( int f = 0 ; f < InterCellObjects.Count() ; f ++ )
    InterCellObjects[ f ].Process( Fr ) ;

  // now process the portals
  for( int f = 0 ; f < Portals.Count() ; f ++ )
  {
    // is the connected cell of the portal locked?
    if( Portals[ f ].ConnectedCell.CellLocked )
      continue ;

    // is the connected cell visible (check the portal polygon against
    // the current cells frustum)
    if( !Portals[ f ].ConnectedCell.IsVisible( Fr ) )
      continue ;

    // Construct a new frustum for the connected cell
    Frustum ConnCellFr = Portals[ f ].AdjustFrustum( GetCameraPosition(), Fr ) ;

    // process the connected cell with the new frustum
    Portals[ f ].ConnectedCell.ProcessCell( ConnCellFr );
  }

  // and finally unlock the cell
  CellLocked = false ;
}
```

Fig. 56 The updated *FCCell::ProcessCell* method.

Before processing its portals the cell now first processes all object references it has before continuing processing the portals.

5.6 Summary

Because most render API's today use (amongst others) a Z-buffer for exact visibility determination, any kind of object, including (dynamic) inter-cell objects, can be added to a scene by just rendering them every frame and let the Z-buffer decide whether they are visible or not. However using the few simple adjustments that were shown in this chapter, (dynamic) inter-cell objects can be made to fully profit from the visibility determination speed-up provided by portal processing. Since often these dynamic objects (i.e. people, cars, space ships) benefit from having a high polygon count that make their appearance more realistic, this reduction in processing time can be highly beneficial when there is a good change the dynamic object will often be inside the initial view frustum without being visible (e.g. the characters in a multiplayer shoot-em-up game who are hunting each other through a maze).

However, also some problems with using inter-cell objects have been identified, like the 'flickering light' problem and unnecessary object (re-) referencing. Avoiding these problems are mainly a task of the modeler who designs the virtual world. Therefore, he should adhere to the following standards:

- When placing light sources in a world that are not meant to be global, see to it that their influence on objects in other cells through opaque walls is either none or not noticeable to a viewer exploring the world;
- Make the position of the root of a dynamic object relevant to that object. In the case of the *Man* boarding the *Boat* from a *Tunnel* example, if the root of the man is positioned ten meters behind the man, to the viewer the man is already on the boat and should be rocking, but since the root is still in the tunnel the *Man* object has not yet changed parent cells and therefore does not rock along with the boat;
- When using a dynamic portal, check if dynamic updating of an inter-cell object's references with every change of the portal is necessary. If not, flag the object with the ICCF_DONT_UPDATE_REFS flag.

When following these rules the advantages of portal processing and inter-cell objects are optimally used allowing for more complex worlds and objects to be designed and used.

The implementation of inter-cell objects in vrSS only requires the programmer to register the inter-cell object with the cell manager and set its shape and flags as an extra task. After this all updating is done transparent to the programmer making the use of inter-cell objects very easy to a programmer experienced in vrSS.

Chapter 6: Special FX portals

Special FX ('effects') portals allow the addition of interesting visual effects to virtual reality worlds. Some examples are: mirrors, tinted glass windows, reflective windows, etc. As this chapter will show, portals can be used to implement these effects but many of them require changes to the core portal processing classes (derived in the previous chapters). By implementing some special FX portals and making these changes to the core classes, the basis for incorporating many other special FX portals is created.

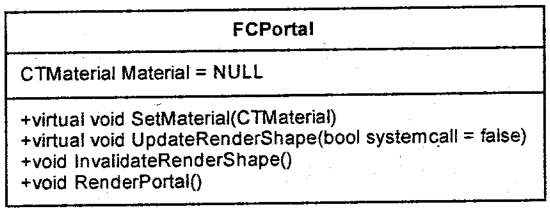
The layout of this chapter differs with the previous chapters in that it immediately starts with the '*implementation in vrSS*' section. This was done because for each of the discussed special portal types, their behavior is often easiest explained by showing the code or class changes.

6.1 Implementing portals in vrSS

In each of the next sections, a special FX portal type is discussed and (when possible[14]) implemented in vrSS. The sections have been arranged so that each section builds on the changes made in the previous ones.

6.1.1 Translucent portals

Translucent portals give the impression that the viewer is looking through some translucent material when looking through the portal. Actually, there is nothing special about this type of portal; the only difference with a regular portal is that an *alpha texture* (a transparent texture) is drawn on the portal polygon once it has been processed to simulate looking through glass or some other translucent material. To create even more interesting effects an animated texture can be used, e.g. to simulate a force field of some sort.
Because almost every other special portal type described in this chapter can also require the portal polygon itself to be rendered, this functionality is added to the *FCPortal* base class as shown in Fig. 57:



```
FCPortal
CTMaterial Material = NULL
+virtual void SetMaterial(CTMaterial)
+virtual void UpdateRenderShape(bool systemcall = false)
+void InvalidateRenderShape()
+void RenderPortal()
```

Fig. 57 Extensions to the *FCPortal* class.

By default the *Material* variable has value NULL, which prevents the portal polygon from being rendered. The portal polygon is rendered by providing *FCPortal* with a material using the *SetMaterial* method. By default, method *UpdateRenderShape* creates a simple *rendershape*[15] used to render the polygon interior with the provided material, but this method can be overridden to allow for e.g. animated rendershapes or textures.
The (portal processing) system calls the *InvalidateRenderShape* method whenever the portal polygon has changed, which in turn calls *UpdateRenderShape* with the *systemcall* parameter set to true. This allows the programmer to identify the system updating the rendershape and one of his own methods. The actual rendering of the portal is done by the *FCCell::ProcessCell* method (see section 6.1.4) which calls the *RenderPortal* method.

[14] Due to both time constraints and unavailability of software some portals could not be implemented.
[15] A structure used by vrSS to store the information needed (vertices, faces, etc.) to render an object.

6.1.2 Mirror portals

Mirror portals mimic reflecting surfaces by mirroring the camera position and orientation when processing the portal before creating the new frustum. Mirror portals connect to their parent cells, which causes a problem with the current cell locking mechanism (introduced in chapter 3):

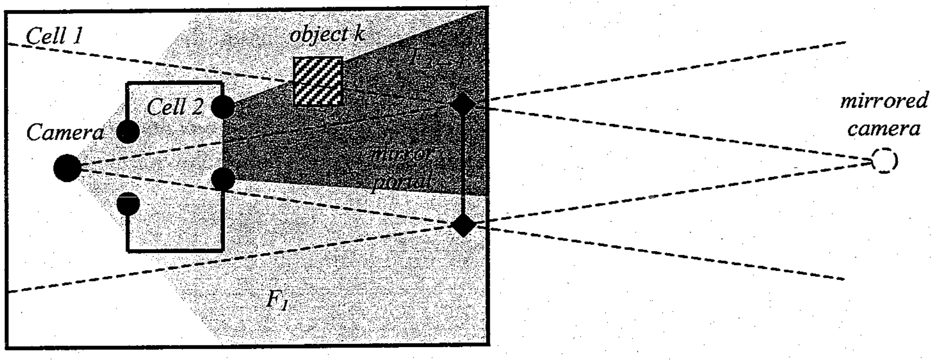


Fig. 58 Mirror portals and cell locking.

When the scene in Fig. 58 is processed without cell locking, object *k* would be rendered twice (three times actually, counting its reflection in the mirror, but identically rendered twice from the initial camera position and orientation), once when processing *Cell 1* using frustum F_1, and once when processing the portal from *Cell 2* leading back into *Cell 1* using frustum $F_{2\rightarrow 1}$. However, processing the scene with cell locking results in the mirror casting an empty reflection because it is denied access to the cell tree of *Cell 1*!
In order to solve this problem, a cell is now locked in a *camera context*: any portal that changes the camera's position and/or orientation which (potentially) requires a locked cell to be processed again creates a new camera context, in which every cell is initially unlocked again. Now 'regular' portal processing can continue, again locking cells being processed in the new camera context. When processing the special portal is finished, the old camera context, in which the portals parent cell was (and still is) locked, is restored and processing of that cells other portals continues.
In the case of Fig. 58, when processing the scene, *Cell 1* is locked in the initial camera context cc_1. When encountering the mirror portal, the camera position and orientation is mirrored and a new camera context cc_2 is created. In this context, all cells are unlocked again, so using the mirror portal frustum (shown by the dotted lines leaving the mirrored camera) now *Cell 1* and *Cell 2* can be processed again. Now the mirror portal is finished processing, so camera context cc_1 is restored in which *Cell 1* is still locked, so when processing *Cell 2* the portal leading back into *Cell 1* is again correctly ignored.

Whether or not *Cell 2* is processed before or after the mirror portal is processed doesn't matter in the final result: if *Cell 2* is processed before the mirror portal, due to *Cell 1* being locked in cc_1, the portal leading back into *Cell 1* is ignored, so when processing the next portal of *Cell 1*, the mirror portal, this again creates cc_2 in which both *Cell 1* and *Cell 2* are unlocked.

Again, due to the general nature of this problem, instead of deriving new classes, core portal processing classes are updated and the only new class introduced is *FCCameraContext* as shown in Fig. 59:

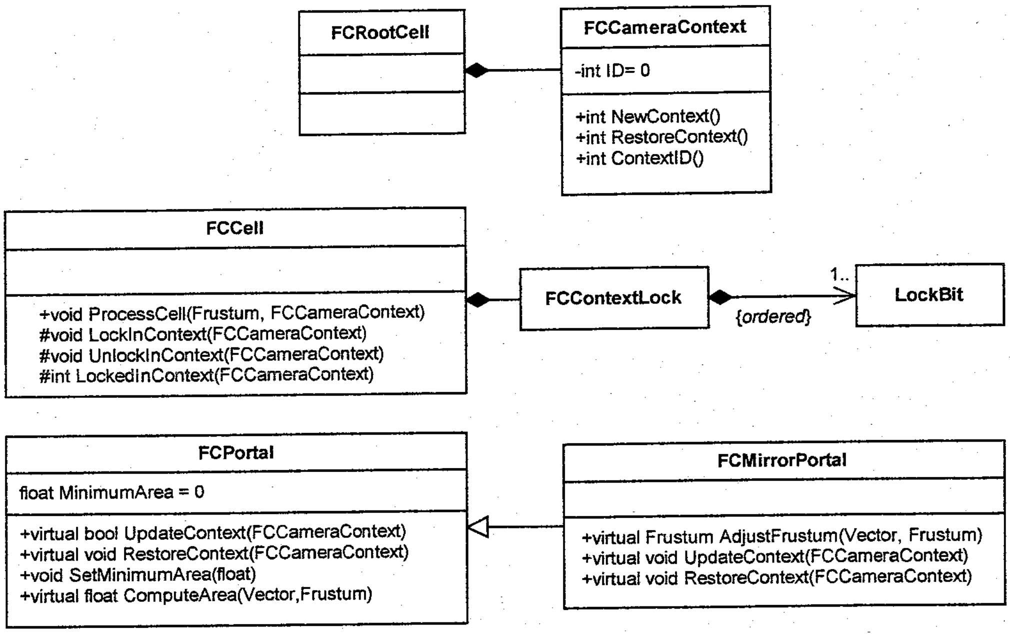

FCRootCell
FCCameraContext
-int ID= 0
+int NewContext()
+int RestoreContext()
+int ContextID()
FCCell
+void ProcessCell(Frustum, FCCameraContext)
#void LockInContext(FCCameraContext)
#void UnlockInContext(FCCameraContext)
#int LockedInContext(FCCameraContext)
FCContextLock
1..
{ordered}
LockBit
FCPortal
float MinimumArea = 0
+virtual bool UpdateContext(FCCameraContext)
+virtual void RestoreContext(FCCameraContext)
+void SetMinimumArea(float)
+virtual float ComputeArea(Vector,Frustum)
FCMirrorPortal
+virtual Frustum AdjustFrustum(Vector, Frustum)
+virtual void UpdateContext(FCCameraContext)
+virtual void RestoreContext(FCCameraContext)

Fig. 59 Implementation of mirror portals.

The *FCRootCell* class creates an instance of class *FCCameraContext*, which is then passed along with every call to *FCCell::ProcessCell* (see section 6.1.4). This method calls *FCPortal::UpdateContext* before processing a portal and calls *FCPortal:: RestoreContext* afterwards. By default these methods do not change the camera context, but derived class *FCMirrorPortal* overrides these methods to create a new camera context (in *UpdateContext*) by calling *FCCameraContext::NewContext*, which simply increases a counter, and later to restore the old context (in *RestoreContext*) by calling *FCCameraContext::RestoreContext*.
In order to keep track of all camera contexts in which a cell is locked/unlocked, each cell creates an instance of the *FCContextLock* class, which stores an array of bits that represent the locked status given a camera context ID. By checking this list before processing a cell, setting the bit when processing a cell in a camera context and resetting the bit when finished processing in a camera context, the implementation of the camera context solution is now complete.

Class *FCMirrorPortal* overrides the *AdjustFrustum* method to create the mirror frustum by first mirroring the camera position in the portal plane and then using this mirrored position to create the mirror frustum using the (clipped) portal vertices. Only one potential problem remains: infinite recursion.

If two mirror portals are placed opposite and facing each other, an infinite recursion could follow due to the mirrors reflecting into each other. A first solution to prevent this from happening is to have the portal count the number of times it has been processed in a frame and disable itself after a pre-defined maximum has been reached. This solution will not always work correctly as is shown in Fig. 60:

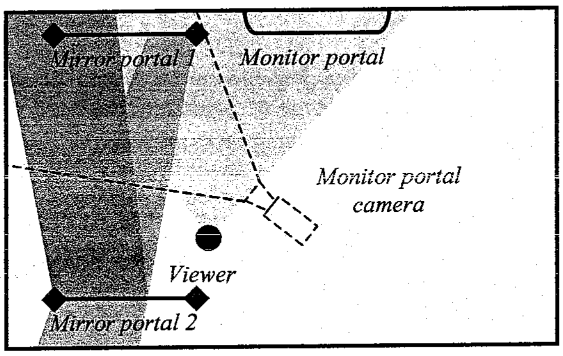


Fig. 60 Mirror portal recursion.

Here a world consisting of two mirror portals and one monitor portal (discussed in the next section) is rendered from the viewers position in initial camera context cc_1. Suppose both mirrors have their maximum recursion count set to 1: *Mirror portal 1* is being processed, resulting in *Mirrorportal 2* being processed, after which *Mirror portal 1* again becomes visible but is not processed anymore due to the maximum amount of recursions being reached. Processing *Cell 1* now continues by processing the *Monitor portal*, which renders the scene as seen through the *Monitor portal camera*. In this frustum (shown by the two dotted lines), *Mirror portal 1* is visible again but will not be processed because the maximum amount of recursions has already being reached, resulting in a 'gap' in the final rendered scene!

To fix this problem using a counter requires some way of keeping track how many times a portal has been processed as seen through some portal sequence *x* (e.g. how many times has *Mirror portal 1* been processed as seen through portal sequence *Mirror portal 1* → *Mirror portal 2* → *Monitor portal*). This is a complex and expensive solution, so another approach has been chosen to prevent infinite recursion.

Instead of counting how many times a portal has been processed, the visible area of the portal in screen space is computed. If this falls below a pre-set minimum area, further recursion of the portal is stopped. The minimum area can be set for each portal using the *FCPortal::SetMinimumArea* method (see Fig. 59) which sets the *MinumumArea* variable, and the visible area can be computed using the *FCPortal::ComputeArea* method.

When the *FCMirrorPortal::UpdateContext* method is now called, first the visible area is computed, and when this is less than *MinimumArea* the method returns false, signalling *FCCell::ProcessCell* not to process the connected cell of the mirror portal.

This solution is not perfect, and situations in which recursion continues for a very long time exist (e.g. two giant mirror opposite and facing each other at a very close distance, and the camera at the center and in the middle of both mirrors with its viewing direction perpendicular to one of the mirrors) exist, but finally recursion *will* end and in most situations this solution is adequate both in terms of performance as well as correctness of rendered scenes.

Not just mirror portals require a way to prevent infinite recursion, basically every portal type that can change the camera position and/or orientation can cause infinite recursion and must be guarded against it.

6.1.3 Monitor portals

Actually, monitor portals are not really portals in the definition used so far because they cannot narrow the frustum. A monitor portal shows what is being filmed by a camera placed elsewhere in the world. This requires that the world as seen through this camera is rendered as a bitmap and then texture-mapped on top of the portal. Displaying this rendered texture-map does not require the use of a portal, but the functionality of rendering a texture map and then mapping it on top of a portal is also required by reflecting window portals and their likes (discussed in the next section).

Monitor portals present a problem: vrSS cannot use multiple cameras while rendering a single frame, and unfortunately at the time of this writing this functionality has not yet been added to vrSS so an actual implementation does not yet exist. Therefore in the following the existence of a function *CTTexture RenderTexture(Camera)* is assumed that returns the scene, rendered using the the passed camera, as a texture. After this texture has been rendered, it can be mapped on to the portal polygon using the *RenderPortal* method that was added to *FCPortal* in section 6.1.1.

6.1.4 Multiple pass portal

An example of a *multiple pass portal* is a reflecting window portal, which mimics a window that also partially reflects the world on the viewers side of the window. This type of portal requires processing two (different) connected cells when processing the portal: one cell must be processed to render the 'outside' portion of the window, and the other to render the 'inside' portion, the reflection. Rendering the reflection is done first, as this must be transformed into a transparent texture to overlay on the outside rendering.

Multiple-pass portals require yet another update of the core portal processing classes as shown in Fig. 61:

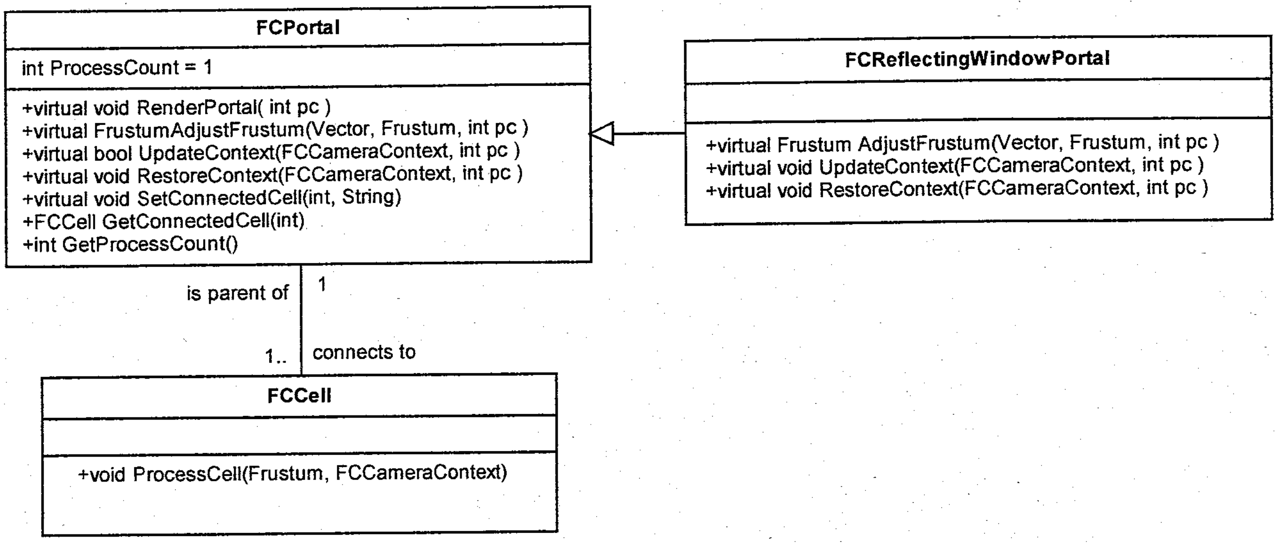

FCPortal
int ProcessCount = 1
+virtual void RenderPortal(int pc)
+virtual FrustumAdjustFrustum(Vector, Frustum, int pc)
+virtual bool UpdateContext(FCCameraContext, int pc)
+virtual void RestoreContext(FCCameraContext, int pc)
+virtual void SetConnectedCell(int, String)
+FCCell GetConnectedCell(int)
+int GetProcessCount()
FCReflectingWindowPortal
+virtual Frustum AdjustFrustum(Vector, Frustum, int pc)
+virtual void UpdateContext(FCCameraContext, int pc)
+virtual void RestoreContext(FCCameraContext, int pc)
is parent of
1
1..
connects to
FCCell
+void ProcessCell(Frustum, FCCameraContext)

Fig. 61 Implementation of reflecting window portals.

For every processing pass of the (multiple-pass) portal, the connected cell of the portal is stored in a list to which connected cells can be added using method *FCPortal::SetConnectedCell(int processingpass, String ConnectedCell-Name)*. For a 'regular' portal this list will contain only one connected cell. Now when processing its portals, *FCCell::ProcessCell* (see chapters 3 and 5 and Fig. 62) iterates through this list for each portal (added/changed code in Fig. 62 is shown in italic print, this new implementation also incorporates all changes required by the previously discussed portals).

Using multi-pass portals also requires a revision of the code for restoring the bi-directionality of portals needed for (dynamic) cell loading (see chapter 4), and requires some changes in the *FCCellManager* class: the *RegisterPortal* method must be changed to allow setting multiple connected cells for a portal and the *FCCell::GetPhysicalCell* method, used to determine the physical of a moving inter-cell object (see chapter 5), must have a way of knowing which of the multiple connected cells is the one the object moved into when it passed through the portal. The code changes for implementing these changes are not shown here.

```
void FCCell::ProcessCell( Frustum Fr, FCCameraContext CamCtx )
{
  // lock the cell in the current camera context (see section 6.1.2)
  LockInContext( CamCtx ) ;

  // make sure the parents world transform is up-to-date
  // and then process the cell tree using vrSS process()
  GetParent().UpdateWorldSpace() ;
  CTGroup::Process( Fr ) ;

  // process all references to inter-cell objects
  for( int f = 0 ; f < InterCellObjects.Count() ; f ++ )
    InterCellObjects[ f ].Process( Fr ) ;

  // now process the portals
  for( int f = 0 ; f < Portals.Count() ; f ++ )
  {
    FCPortal p = Portals[ f ] ;

    // is the portal visible in the frustum?
    if( !p.IsVisible( Fr ) )
      continue ;

    // iterate through the connected cells of the portal (see section 6.1.4)
    for( int g = 0 ; g < p.GetProcessCount() ; g ++ )
    {
      // update the camera context for the current processing pass(also prevents infinite
      // recursion, see section 6.1.2)
      if( !p.UpdateContext( CamCtx, g ) )
        continue ;

      // is the connected cell of process pass g of the portal locked in the camera context?
      if( p.GetConnectedCell( g ).LockedInContext( CamCtx ) )
        continue ;

      // Construct a new frustum for the connected cell
      Frustum ConnCellFr = p.AdjustFrustum( GetCameraPosition(), Fr, g ) ;

      // process the connected cell with the new frustum
      p.ConnectedCell.ProcessCell( ConnCellFr );

      // render the portal (e.g. texture map the portal with rendered reflection)
      p.RenderPortal( g ) ;  // (see section Fout! Verwijzingsbron niet gevonden.)

      // reset camera context
      p.RestoreContext( CamCtx, g ) ;
    }
  }

  // and finally unlock the cell
  UnlockInContext( CamCtx ) ;
}
```

Fig. 62 The FCCell::ProcessCell method, updated for special FX portals.

6.1.5 Teleporting portals

The final special portal type that is discussed in this chapter is a teleporting portal. This lets the viewer see into parts of the world as if he were standing somewhere else. In order to implement teleporting portals the camera is re-positioned just as was done with mirror portals, but now the (clipped) portal is transformed to a pre-defined position and orientation before the adjusted frustum for processing the connected cell is computed (see Fig. 63).

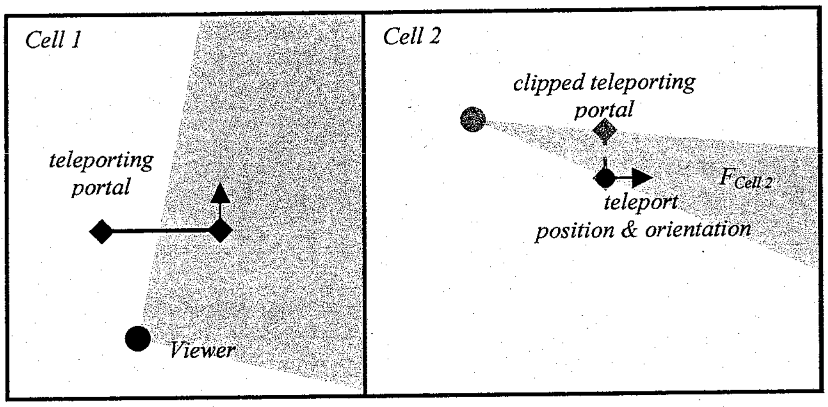


Fig. 63 A Teleporting portal.

The extra complication with teleporting portals is that when the viewer moves through them, he must also be teleported to a new position in the world. This presents some difficult problems that have unfortunately not been solved due to time constraints: the problems mentioned consist of dealing with objects physically being in two different places at the same time when sticking through a portal (recall that when solving this problem in chapter 5 at least the world position of the object in both cells was the same) and related to this, determining the position of child inter-cell objects (chapter 5).

6.2 Summary

As was shown in this chapter, portals allow the addition of a multitude of interesting and exciting features to a virtual reality world. However there is a price to be paid (of course!): processing time. Most special FX portals require extra information to be rendered (e.g. reflection, TV-screen images), a process that takes up valuable processing time, especially when very deep recursions can occur (e.g. the two mirrors opposite and facing each other example). Therefore, they must be used carefully as not to bring performance down to a crawl.

Special FX portals also carry another price: they often require extensive changes to the code of the portal processing environment. Each of the examples above required one change or another to the vrSS portal processing framework, some simple (translucent portals), others quite extensive (reflective window portals). Although the changes currently made to the framework will allow many different type of special FX portals to be added to vrSS, no doubt many other special FX portals require yet more changes. In this case the programmer should consider very carefully how the changes required to get the desired effect will affect overall performance: if this is seriously harmed maybe he should consider an alternative or forget about implementing the portal type completely (e.g. in the case of reflective window portals, due to the changes in *FCCell::ProcessCell* now the connected cells are processed in a loop, but most portals only have one connected cell making this loop expensive and unnecessary overhead for them).

Chapter 7: Potentially Visible Sets

In the previous chapters, all visibility determination optimization was done at run-time by using the portals to narrow the frustum. This chapter deals with *pre-run-time visibility determination* or in other words, computing in advance the superset of *potentially visible objects* to a viewer given a (set of) camera position(s). This information can then be used for several different purposes, like:

- Completely foregoing run-time portal processing: using the potentially visible set, a well-chosen superset of visible items is already at hand;
- Further optimizing visibility determination: objects which are never visible to the viewer don't have to be checked for intersection with the (narrowed) frustum;
- Determining which cells are potentially visible to a viewer so that a cell manager can use this information to (pre-) load these cells;
- For complex scenes, implement a *level-of-detail* hierarchy of the visibility of objects within cells: only load those objects into memory that are potentially visible to a viewer;
- Collect heuristics about a world, i.e. find an upper limit for the number of polygons that must rendered, so that this information can be used by the modeler to further optimize the model.

This chapter deals with computing *potentially visible sets*, which are sets of objects potentially visible from the viewer's position and orientation. A step by step approach to computing potentially visible sets using the occluding properties of portals is given, after which a short explanation of how to use the potentially visible sets in the examples above is given. The chapter then continues describing the framework extension to incorporate potentially visible sets in vrSS.

7.1 The potential visible volume of cells

Computing the potentially visible volume of a cell at pre-run-time is different from computing it at run-time: at run-time the viewers position and orientation in a cell is given so an exact frustum for each visible cell can be created. However, at pre-run-time, the viewers position and orientation nor its physical cell are known. However, as long as the viewer is restricted to a position inside one cell it is possible to determine which parts of other cells are potentially visible as is shown in Fig. 64 for a viewer positioned in *Cell 1*:

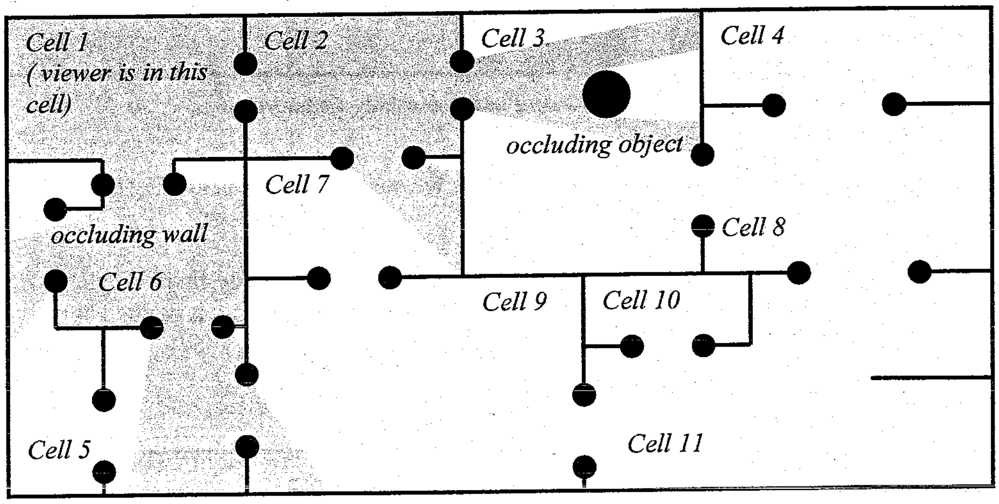


Fig. 64 Potentially visible volume (gray) for viewer in *Cell 1*.

The *actual potentially visible volume* of a cell for a viewer in *Cell 1* is defined by the union of every point in that cell that can be reached by a line of sight originating from *Cell 1*. For our purposes, taking into account all occluders in a world (like the occluding wall and object in Fig. 64), is too heavy a computational task. This would require some form of ray tracing, which can result in unacceptable computation times for large complex worlds. So instead of honoring all occluders, it is therefore faster to use only a well chosen subset of these occluders. In the following the subset of occluders used are the portals, because they can cause large parts of their connecting cells to be occluded thus potentially yielding large sets of invisible objects. The potentially visible volume for a cell found using this set of occluders is called the *potentially visible volume* (or p.v.v.) of the cell.

The p.v.v. is a superset of the actual potentially visible volume, guaranteeing that no visibility information is lost when rendering a scene.

The next section deals with finding a potentially visible volume for all cells that can be seen from the source cell.

7.2 Finding the potentially visible volume set for a cell

In the following $PVV_{sp \to p1 \to p2 \to dp}$ (see Fig. 65) denotes the set of clipping planes whose intersection of negative half spaces (see chapter 2) defines the potentially visible volume for a cell[16] (the *destination cell* or *dc*) as seen through the *portal sequence sp → p1 → p2 → dp* from the viewer's physical cell (the *source cell* or *sc*). The portal through which the viewer's line of sight exits the source cell is called the *source portal* and the portal through which it enters the destination cell is called the *destination portal.*

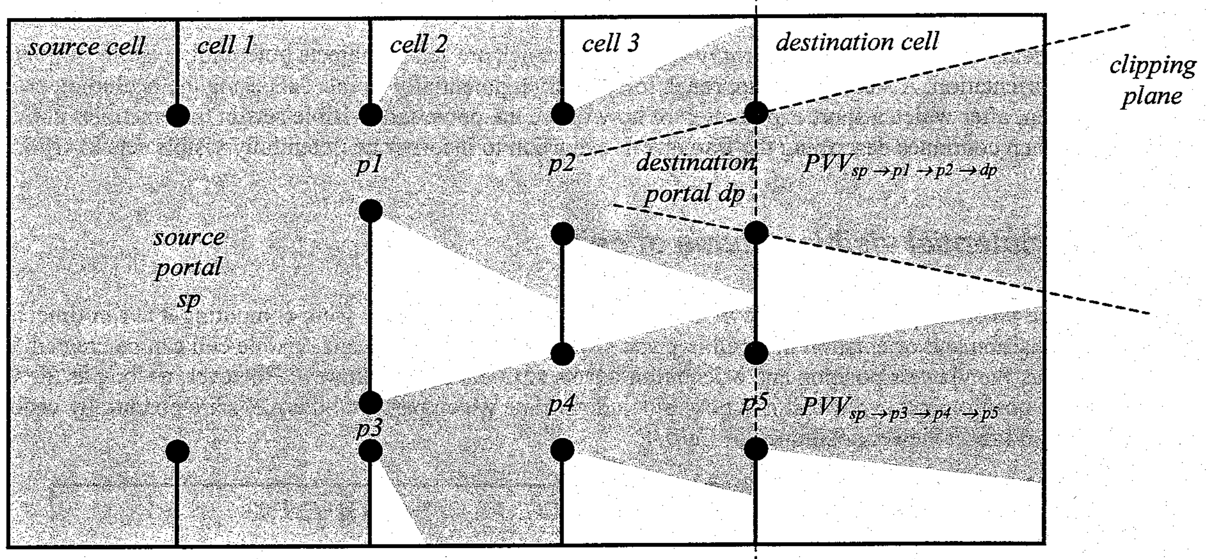


Fig. 65 Portal sequences and potentially visible volumes.

Note that the cell boundaries are implicitly a part of the p.v.v. because no objects belonging to a cell will be positioned behind that cell's boundaries.

Because there can exist several different portal sequences from a source cell to a destination cell, let $PS_{sc \to dc}$ denote the set of all these sequences. Now the *combined potentially visible volume* $CPVV_{sc \to dc}$ is defined by:

$$CPVV_{sc \to dc} = \mathrm{Y}_{ps \in PS_{sc \to dc}} PVV_{ps}.$$

Note that $CPVV_{\emptyset}$, the combined visible volume of the source cell, is completely defined by the source cell boundaries: The viewer can move anywhere in the source cell so nothing in it is invisible.

If C denotes the set of all cells in the world, and by noting that for invisible destination cells $CPVV_{sc \to dc} = \emptyset$, then the *potentially visible volume set* for the source cell is:

$$PVVS_{sc} = \mathrm{Y}_{i \in C} CPVV_{sc \to i}$$

[16] In the text it will be implicitly assumed that planes are created such that this property is true.

This set holds the potentially visible volume for any cell in the world for a viewer positioned in cell *source cell.* By computing this set for all cells in the world a complete picture for pre-run-time visibility determination has now been drawn: given a source cell, for each object in a destination cell visibility can be determined by checking the object against $PVV_{destination\ cell}$ found in $PVVS_{source\ cell}$.

7.3 Finding portal sequences

To find $PVVS_{source\ cell}$ for a given source cell, every portal sequence *ps* starting with a portal of the source cell must be found and its PVV_{ps} must be computed. This is done using the algorithm shown in Fig. 66.

```
PVVS CreatePVVSet( Cell sourcecell )
{
PVVS pvvs ;  // this holds PVVSsource cell once FindPVVS finishes

  // set the source cell for the pvvs
  pvvs.SourceCell = sourcecell ;

  for( int f = 0 ; f < sourcecell.Portals.Count() ; f ++ )
  {
    Portal sp = sourcecell.Portals[ f ] ;

    // use a stack to store the portal sequence
    PortalStack portalstack ;
    portalstack.Push( sp ) ;

    // create the p.v.v. for the connected cell of sp
    PVV pvv = CreatePVV( portalstack ) ;  // (see Fig. 68)

    // find all portal sequences starting with source portal sp leading to
    // potentially visible cells.
    FindPortalSequences( sp.ConnectedCell, portalstack, pvv, pvvs ) ;
  }
}

bool FindPortalSequences( Cell cell, PortalStack portalstack, PVV pvv, PVVS pvvs )
{
  // add PVVportal sequence to p.v.v. set
  pvvs.AddPVV( portalstack, pvv ) ;

  for( int f = 0 ; f < cell.Portals.Count() ; f ++ )
  {
    Portal p = cell.Portals[ f ] ;

    // is the current portal potentially visible given the p.v.v for cell?
    if( pvv.IsObjectVisible( p )
    {
      // create p.v.v. for connected cell of portal
      portalstack.Push( p ) ;
      PVV pvv_concell = CreatePVV( portalstack ) ;  // (see Fig. 68)

      // if no p.v.v. for the connected cell stop traversing the sequence
      if( pvv_concell == NULL )
        continue ;

      // and recurse through it
      FindPortalSequences( p.ConnectedCell, portalstack, pvv_concell, pvvs ) ;
      portalstack.Pop() ;
    }
  }
  return true ;
}
```

Fig. 66 Finding portal sequences (and p.v.v.'s).

For every source cell portal *sp*, function *CreatePVVSet* creates PVV_{sp} using function *CreatePVV*. It then calls function *FindPortalSequences* which checks the portals in the connected cell of *sp* for (potential) visibility. For every visible portal *cp* in the connected cell $PVV_{sp \rightarrow cp}$ is created and *FindPortalSequences* then recursively calls itself to check the connected cell of portal *cp*. This process is repeated until no further visible portals are encountered.

A stack is used to store the portal sequence, which is needed to determine the p.v.v. for a destination cell, and every p.v.v. that is created is stored in the p.v.v. set *pvvs* so that when *CreatePVVSet* finishes it returns $PVVS_{source\ portal}$.

The only function now left to implement is *CreatePVV*. This will be done next.

7.4 Creating the potentially visible volume

How to create the potentially visible volume for a portal sequence differs based on the length of the sequence. As was mentioned before, for a sequence of length zero the p.v.v. is defined by the source cell boundaries resulting in all objects in the source cell to be potentially visible. For a sequence of length one (source portal = destination portal), the algorithm for creating the p.v.v. is simple because only one occluding portal has to be considered. For a sequence of two portals this algorithm is expanded because an extra occluding portal must be considered and for sequences of three or more portals some extra optimizations are added to the algorithm that further reduce the p.v.v. for the destination cell due to the occluding properties of the portals in between the source- and destination portal.

7.4.1 Creating the potentially visible volume for a portal sequence of length one

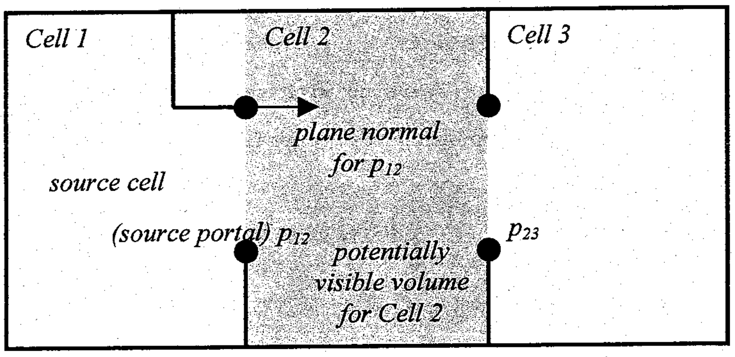

Cell 1
Cell 2
Cell 3
plane normal for p_{12}
source cell
(source portal) p_{12}
p_{23}
potentially visible volume for Cell 2

Fig. 67 Potentially visible volume for an immediate neighbour cell of the source cell.

Finding $PVV_{source\ portal}$ is easy (see Fig. 67): the only portal occluding destination cell *Cell 2* from sight from the source cell *Cell 1* is source portal p_{12}. Since the only restriction to the viewers position is that it must be inside the source cell, the viewer can be positioned anywhere on portal p_{12}. Lines of sight cannot bend around corners so the destination portal plane is added to the p.v.v. Because the portal is the only occluder, this will be the final p.v.v. for the destination cell, so $PVV_{p12} = \{ p_{12} plane \}$. Function *CreatePVV* for a portal sequence of length one will thus look like Fig. 68:

```
PVV CreatePVV( PortalStack portalstack )
{
PVV pvv ;

  // add the destination portal plane to the p.v.v.
  pvv.AddPlane( portalstack.Top().Plane ) ; // add destination portal plane

  return pvv ;
}
```

Fig. 68 Function *CreatePVV* for a portal sequence of length one.

7.4.2 Creating the potentially visible volume for a portal sequence of length two

Because now there are two portals occluding the destination cell from view from the source cell, (usually) a much larger volume will be occluded than was the case with a portal sequence of length one. To find this occluded volume (and the potentially visible volume which is the inverse of the occluded volume) the function *CreatePVV* is altered to Fig. 69 (code printed in italic is the code added to function *CreatePVV* of Fig. 68):

```
PVV CreatePVV( PortalStack portalstack )
{
PVV pvv ;

  // add the destination portal plane to the p.v.v. (always!)
  pvv.AddPlane( portalstack.Top().Plane ) ; // add destination portal plane

  if( portalstack.Size() == 1 )
    return pvv ;

  // for each edge of the destination portal (which lies on top of the stack)
  // find the separating plane
  for( int f = 0 ; f < portalstack.Top().Edges.Count() ; f ++ )
  {
    Plane m = FindSeparatingPlane( portalstack.Bottom(), portalstack.Top(), f ) ;
    if( m )
      pvv.AddPlane( m ) ;
  }
  return pvv ;
}

enum HALFSPACE { INTERSECTS, POS_HALF, NEG_HALF } ;

Plane FindSeparatingPlane( Portal sp, Portal dp, int i )
{
  for( int f = 0 ; f < sp.Vertices.Count() ; f ++ )
  {
    // create plane through ps vertex and pd edge
    Plane m = Plane( sp.Vertices[ f ], dp.Edges[ i ] ) ;

    // is sp competely on one side of this plane?
    HALFSPACE hsp = CheckPortalWithPlane( sp, m ) ;
    if( hsp == INTERSECTS ) continue ;

    // same for pd
    HALFSPACE hdp = CheckPortalWithPlane( dp, m ) ;
    if( hdp == INTERSECTS ) continue ;

    // are sp and dp on opposite sides?
    if( hsp != hdp ) return m ;
  }
  // no separating plane exists, so return NULL
  return NULL ;
}
```

Fig. 69 Algorithm for finding the potentially visible volume for a portal sequence of length two.

For every edge e_i of the destination portal *dp*, function *CreatePVV* uses function *FindSeparatingPlane* to find a *separating plane* (if it exists) through this edge and a vertex v_j of the source portal that completely separates the source- and destination portal. This separating plane divides the destination cell into an occluded volume and a potentially visible volume and is therefore added as a clipping plane to the p.v.v. of the destination cell. Why is this correct?

Due to the convexity of the portal and by recalling that a portal can be seen as an infinite opaque wall with an opening in it, e_i can be considered to be the edge of a wall (called the *edge wall*) coinciding with the portal plane and stretching into infinity that occludes part of the destination cell from view (see Fig. 70).

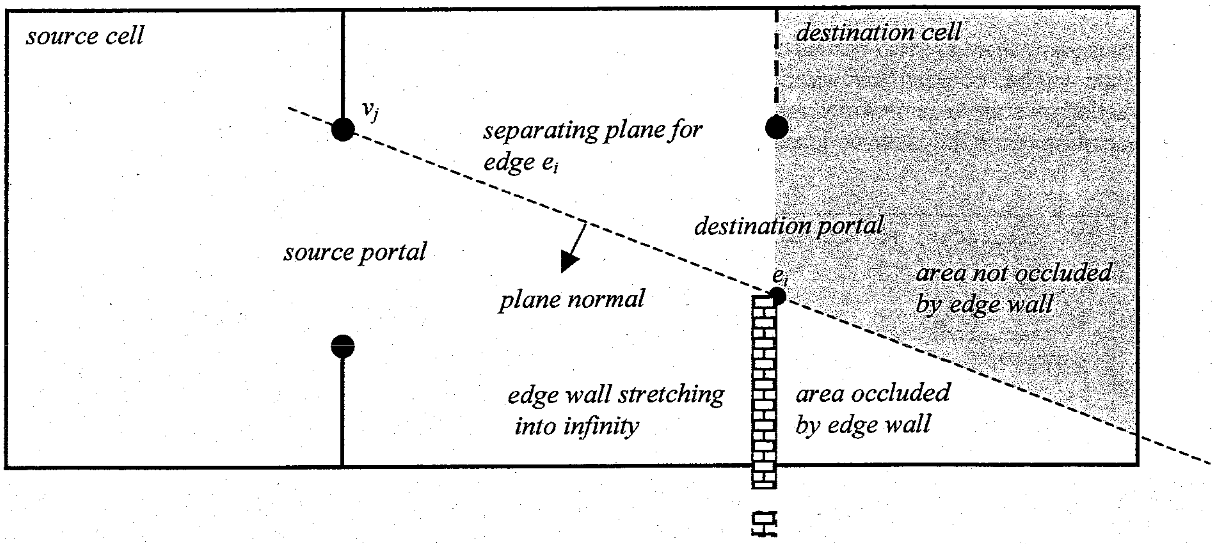

source cell
destination cell
v_j
separating plane for edge e_i
destination portal
source portal
plane normal
e_i
area not occluded by edge wall
edge wall stretching into infinity
area occluded by edge wall

Fig. 70 Finding the clipping plane for an edge e_i of the destination portal (edge e_i is perpendicular to this paper).

The separating plane now divides the destination cell into the volume completely occluded from sight from the source cell by the edge wall and the volume not occluded (and thus potentially visible) by the edge wall. The validity of this statement is easily understood when looking at Fig. 70: Every line of sight through the source portal into the destination cell will start in the positive half space of the separating plane and end in the negative half space, or the line of sight coincides with the separating plane. No line of sight through the source portal can see into the occluded volume without intersecting the edge wall and only through the source portal can a line of sight exit the source cell[17], so the volume of the destination cell that lies in the positive half space of the separating plane is completely occluded from sight. Everything in the negative half space of the separating plane is not occluded by the wall and would therefore be visible if the edge wall would be the only occluder.

By taking the intersection of the non-occluded volumes for destination portal edges, the p.v.v. for the destination cell is now $PVV_{sp \rightarrow dp}$ = { p_i : p_i = *FindSeparatingPlane('source portal sp', 'destination portal dp', 'destination portal edge i')* } $\cup$ *destination portal plane*, which is returned by *CreatePVV*.

What happens when *FindSeparatingPlane* cannot find the separating plane and returns NULL? This situation can only occur when the destination portal plane intersects with the source portal as shown in Fig. 71:

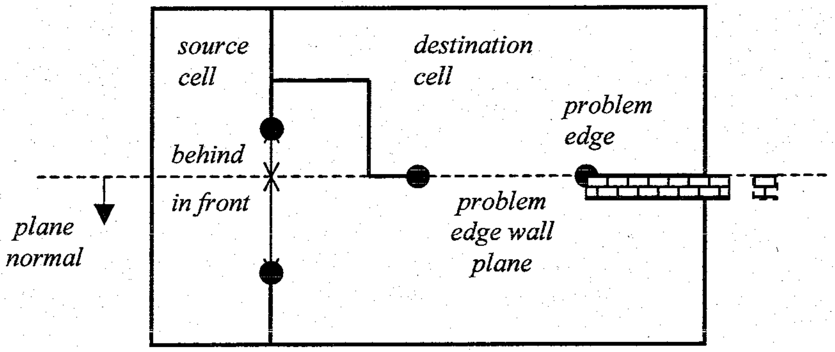


Fig. 71 Edge without separating plane.

In this case, for *problem edge* no plane completely separating the source- and destination portal exists and therefore the edge wall does not occlude part of the destination cell to a viewer in the source cell. The destination portal plane

[17] Remember, $PVV_{source\ portal \rightarrow destination\ portal}$ is being computed, so the assumption that the source cell has only one portal is valid in this case.

'separates' the source portal into a volume *behind* and a volume *in front* of the destination plane, and only lines of sight originating in the *in front* volume can see into the destination cell. So the clipping plane that this *problem edge* would add to the p.v.v. is the destination portal plane which is already a part of the p.v.v and is therefore not added a second time.

7.4.3 Creating the potentially visible volume for a portal sequence of length three or more

While using the current version of *CreatePVV* would also result in a valid p.v.v. (i.e. it is a superset of the actual potentially visible volume) for portal sequences of length three or more, usually the extra portals in between the source- and destination portal introduce extra occlusion which can reduce the p.v.v. for the destination cell significantly as shown in Fig. 72:

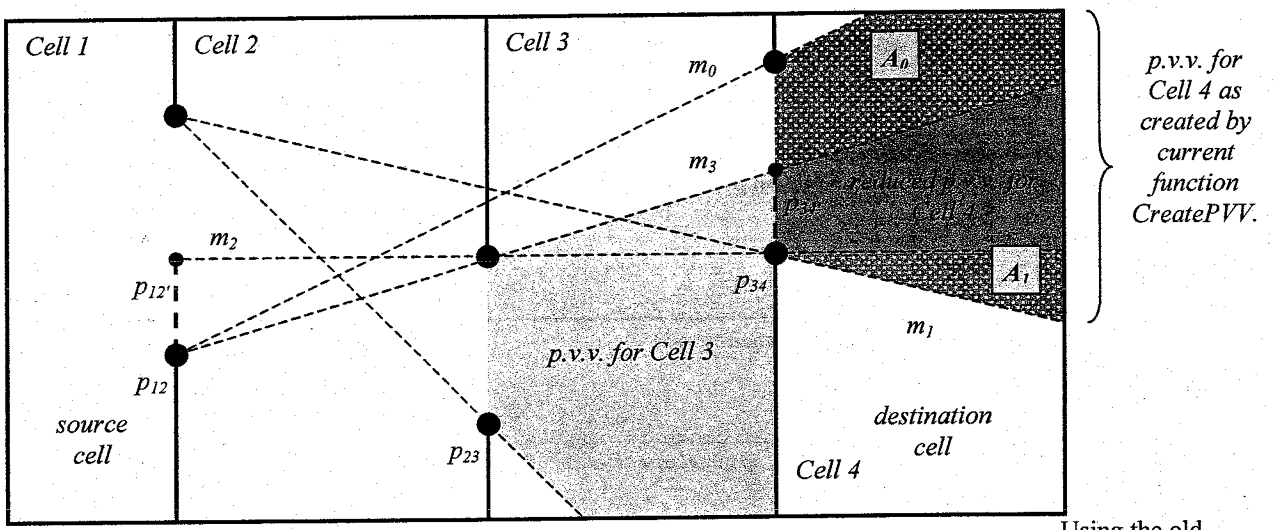


Fig. 72 Occlusion by portals and reduced potentially visible volumes.

Using the old *CreatePVV* function would result in the separating planes m_0 and m_1 of the p.v.v. for *Cell 4* as shown by the dark gray and the dotted dark gray area, however:

- Part of portal p_{34} is occluded by portal p_{23} as can be seen when looking at the p.v.v. for *Cell 3*: any part of portal p_{34} outside of the p.v.v. of *Cell 3* is invisible. Therefore volume A_0 is also invisible and can be omitted from the p.v.v. for *Cell 4*. By clipping the destination portal against the p.v.v. for that portals parent cell, and then using this clipped destination portal (in this case $p_{34'}$ shown using the dotted line) before creating the p.v.v. for the destination cell, the volume resulting from an invisible part of the destination portal in function *CreatePVV* will be omitted from the computed p.v.v;

- Portal p_{23} occludes part of portal p_{34} from sight from a viewer positioned in *Cell 1*, however portal p_{23} also occludes part of portal p_{12} from sight from a viewer positioned in *Cell 4*! A line of sight from source cell *Cell 1* into destination cell *Cell 4* cannot intersect the occluded volume of either portal, so by traversing the portal sequence back to the source portal, creating the p.v.v. for each cell as if *Cell 4* were the source cell, the non-occluded volume $p_{12'}$ of portal p_{12} can be found by clipping portal p_{12} against $PVV_{p_{34'} \to p_{23}}$ as shown in Fig. 73.

By using clipped portals $p_{12'}$ and $p_{34'}$ with the current version of *CreatePVV* now clipping planes m_2 and m_3 are found resulting in the much smaller p.v.v. for *Cell 4* shown by the dark gray area in Fig. 72. Note that in the two observations made above, no assumptions have been made about the number of portals in between the source and the destination portal. Therefore, these optimisations are applicable to any portal sequence of length greater than two. The final version of *CreatePVV* is now given in Fig. 74. Note that the implementation of functions *ClipDestinationPortal* and *ClipSourcePortal* is left to the reader.

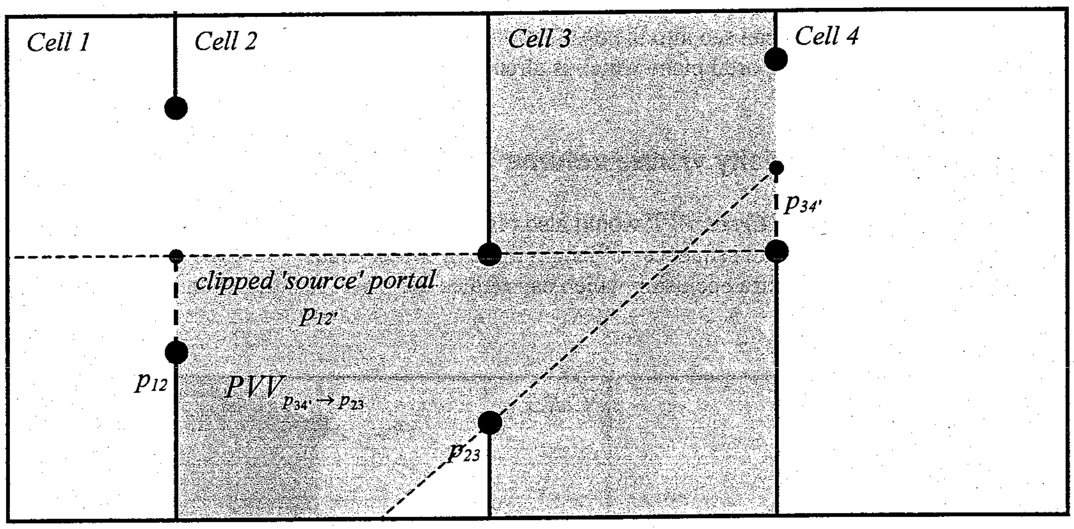


Fig. 73 Traversing the inverted portal sequence to find the non-occluded volume of the source portal.

```
PVV CreatePVV( PortalStack portalstack )
{
PVV pvv ;

  // add the destination portal plane to the p.v.v. (always!)
  pvv.AddPlane( portalstack.Top().Plane ) ; // add destination portal plane

  if( portalstack.Size() == 1 )
    return pvv ;

  // clip the destination portal against the p.v.v. for the its parent cell
  Portal cdp = ClipDestinationPortal( portalstack ) ;

  // clip the source portal (if not visible from the destination cell then
  // stop recursion)
  Portal csp = ClipSourcePortal( portalstack ) ;
  if( csp == NULL )
    return NULL ;

  // now find the separating planes for csp and cdp
  for( int f = 0 ; f < cdp.Edges.Count() ; f ++ )
  {
    Plane m = FindSeparatingPlane( csp, cdp, f ) ;
    if( m )
      pvv.AddPlane( m ) ;
  }
  return pvv ;
}
```

Fig. 74 Final implementation of function *CreatePVV*.

7.4.4 The camera far plane

Should the camera have a far plane to limit its view there is another clipping plane that can be added to the p.v.v. of a cell as can be seen in Fig. 75:

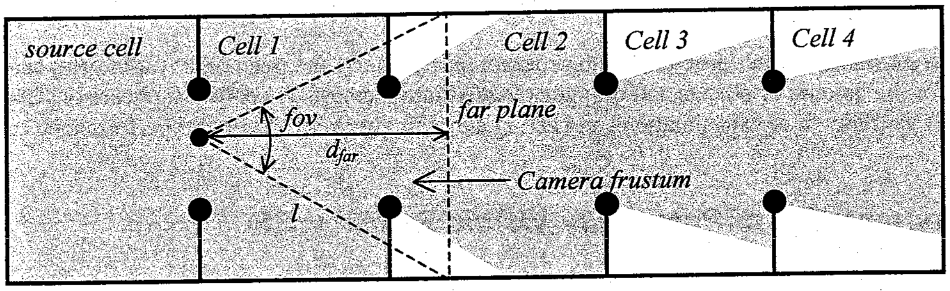


Fig. 75 Camera far plane. The camera far plane limits how far can be seen into neighboring cells by adding an extra clipping plane to the frustum. In the example above, the camera is positioned on the source portal where the far plane partly clips *Cell 2* and completely clips cells *Cell 3* and *Cell 4*. By computing $l = d / \cos(0.5 * fov)$ and adding a plane parallel to the source portal at distance l to the p.v.v, for each cell this plane[18] now adds an extra upper bound to the p.v.v.

7.5 Using potentially visible sets

Now that $PVVS_{source\ cell}$ can be computed how can this information be used to speed up visibility determination?

First, it is now possible to completely forgo run-time portal processing while still having a well-chosen superset of visible objects. For each cell the viewer can be in, a set of potentially visible objects exist, and only the objects in this set are tested for visibility against the initial camera frustum. This was the approach originally taken by Funkhouser et. al. [4] who have reported that on average almost 95% of the world could be culled using a potentially visible set.

When combining potentially visible sets with run-time portal processing, and only checking those objects for intersection with the (narrowed) frustum that are in the potentially visible set, also a slight performance gain is achieved. However, as frustum intersection is not a true bottleneck in the render pipeline, the performance gain will only be noticeable when very complex worlds, consisting of thousands of objects, are used.

A better way to get a performance gain using the combination of pre-processing the world and portal processing is to use the p.v.v. set of a cell to determine the visible area (e.g. in pixels) of a portal. If this area is smaller than a pre-defined threshold, a (pre-rendered) bitmap is used to represent the connected cell(s), otherwise the connected cells are rendered at run-time.
This *level-of-detail* scheme saves time by not processing cells visible through a small frustum, but has the disadvantage that the visible portal area must be small because otherwise the viewer will notice the 2D-nature of the bitmap.

Expanding on this thought, the p.v. set can also be used to determine which objects must be loaded in the cell tree of a cell visible from the source cell, and what the level-of-detail of the objects themselves should be [4]. This saves on both memory resources and visibility processing time, because only those objects potentially visible are loaded into program memory at an adequate level-of-detail (use a lower polygon count for objects farther away), requiring less and smaller objects to be processed for visibility.
In addition, the delay caused by a cell manager (running in the same program thread) loading previously invisible cells into program memory is shortened: when a cell becomes visible only the potentially visible objects in that cell are loaded instead of all objects belonging to that cell. Every time the viewer changes physical cells, the cell tree of each visible cell is updated: if more objects of a cell become potentially visible they are loaded into that cells cell tree, and if objects in a cell become invisible they can be removed from the cell tree and from memory.
This technique spreads loading of all objects in a cell over several short intervals instead of one long delay when loading a complex cell. The performance gain is dependent on the organization of the cells, their interiors and the portals: if by

[18] Another way of adding a 'far plane' to the p.v.v. would be to find the source portal vertex closest to the destination portal and then adding a plane perpendicular to the destination portal at distance d_{far} from this vertex.

turning around a corner all of a sudden a cell with thousands of objects in it becomes completely visible then the performance gain will be none. In addition, there are also consequences for the required amount of storage space due to storing objects at multiple levels of detail.
Unfortunately, due to time constraints, using potentially visible sets to create level-of-detail cell trees has not been examined further.

To try and prevent this and other problems with program performance, the potentially visible sets can be used to find an upper bound for the amount of objects and polygons loaded into memory. By collecting heuristics about (potential) object/polygon visibility in certain situations and acting accordingly overall smoother performance can be achieved.

7.6 Implementation of Potentially Visible Sets in vrSS

Currently three uses of p.v.v. sets have been implemented in vrSS: computing the set of visible cells from a source cell for use with a dynamic cell manager, computing the set of visible objects in those cells (for use with either run-time portal processing or just as a PVS) and gathering heuristics to help build better models for real-time applications.
For these purposes the following major classes are introduced into the vrSS framework: *FCPVSBuilder*, *FCCellIndexer*, *FCPVCSet* and *FCPVVSet*. Some changes are also made to *CTEntity*, *FCPortal* and *FCCell*. Note that this is the first time a change to a core component of the vrSS toolkit is required[19]. A UML-diagram of the framework extension for implementing potentially visible sets in vrSS is shown in Fig. 76:

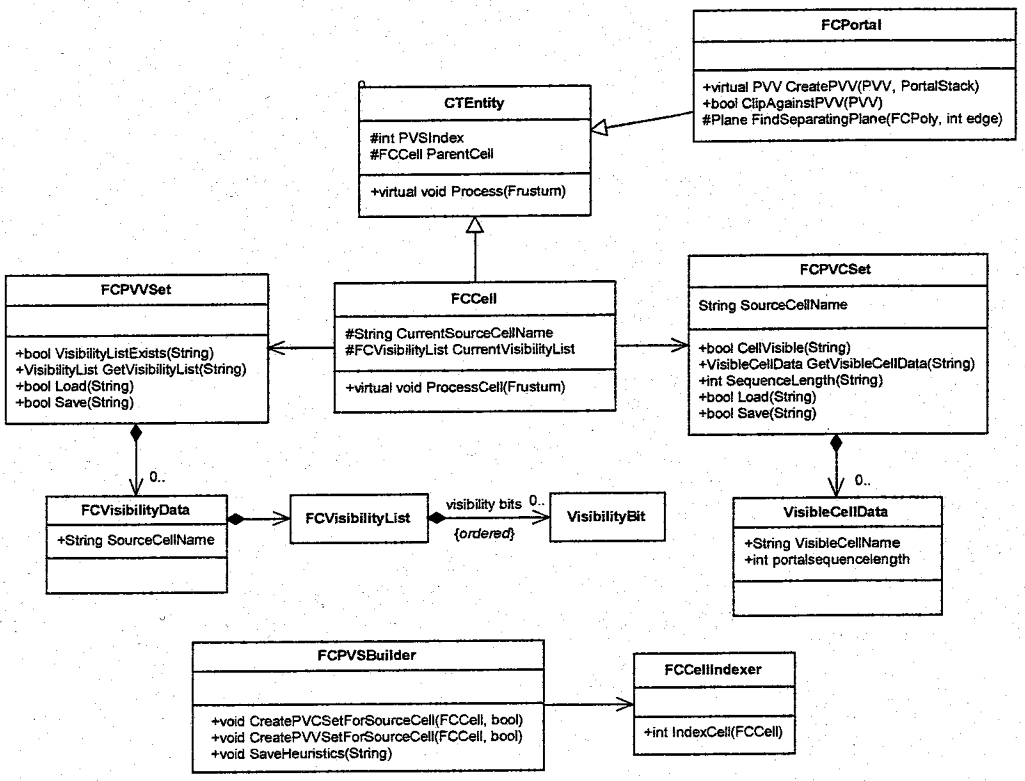

FCPortal
+virtual PVV CreatePVV(PVV, PortalStack)
+bool ClipAgainstPVV(PVV)
#Plane FindSeparatingPlane(FCPoly, int edge)
CTEntity
#int PVSIndex
#FCCell ParentCell
+virtual void Process(Frustum)
FCPVVSet
+bool VisibilityListExists(String)
+VisibilityList GetVisibilityList(String)
+bool Load(String)
+bool Save(String)
FCCell
#String CurrentSourceCellName
#FCVisibilityList CurrentVisibilityList
+virtual void ProcessCell(Frustum)
FCPVCSet
String SourceCellName
+bool CellVisible(String)
+VisibleCellData GetVisibleCellData(String)
+int SequenceLength(String)
+bool Load(String)
+bool Save(String)
0..
FCVisibilityData
+String SourceCellName
FCVisibilityList
visibility bits
0..
{ordered}
VisibilityBit
0..
VisibleCellData
+String VisibleCellName
+int portalsequencelength
FCPVSBuilder
+void CreatePVCSetForSourceCell(FCCell, bool)
+void CreatePVVSetForSourceCell(FCCell, bool)
+void SaveHeuristics(String)
FCCellIndexer
+int IndexCell(FCCell)

Fig. 76 UML-diagram of framework extension for potentially visible sets.

The functionality of these classes will be discussed in the next sections.

7.6.1 Creating a visible cells set

[19] Although by only using special classes derived from *CTEntity* this could be avoided, but could create confusion for programmers.

When a dynamic cell manager has knowledge about the set of cells potentially visible from the physical camera cell, it makes deciding which cells to remove from memory simpler: those cells not in the *potentially visible cells set* (*p.v.c. set*) are the first candidates to be removed. Computing the p.v.c. set is already implicitly performed by the algorithm for computing p.v.v. sets described in this chapter: if a potentially visible volume for a cell exists, that cell must be visible. Therefore, using a skinned-down version of this algorithm, the visible cells set and the length of the portal sequence connecting them to the source cell can easily be computed. The length of the portal sequence can be used by the cell manager to 'think ahead': which cells are visible to the physical camera cell through a portal sequence of length *x*? Should these cells also be kept in memory to get a smoother performance?

Instead of just storing the length of the portal sequence, the actual portal sequence or the distance between the source- and destination portal in the sequence or other potentially useful information for the cell manager could be stored along with the visible cells p.v.c. set. For now this has not been implemented, but doing so when required by a cell manager is a simple task.

7.6.2 Creating a potentially visible volume set

By storing the visibility information of objects in such a way that looking up this information is much faster than performing an *object-frustum* intersection test, processing and rendering complex worlds where each cell can consist of thousands of objects can be accelerated. To implement a fast look-up scheme the leaf nodes of each cell tree are provided with a unique index[20] that is used to look-up their visibility bit in an array as shown in Fig. 77:

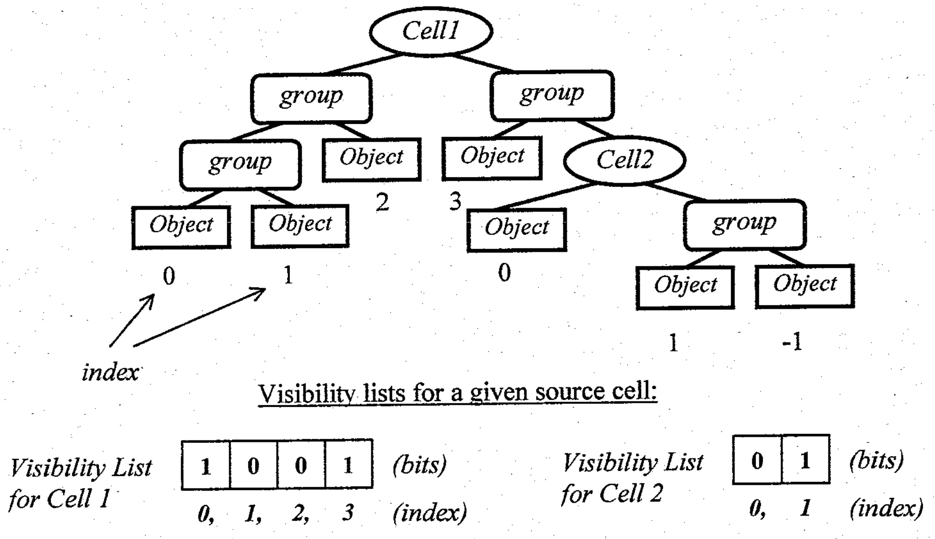


Fig. 77 Object indexing and visibility lists.

Objects, whose visibility cannot be determined by the p.v.v. (e.g. an object tagged as a dynamic object), are given an index of -1, which tags them as visible in the p.v.v. so that actual visibility determination will be done at run-time. By doing this the length of the visibility lists can be restricted. The indices are now stored with the objects, or they can be re-created at run-time but then care must be taken that each object gets the same index again (so in the case of a recursive indexing function the cell tree structure should be identical).

Initially all bits in the visibility list of a cell for a given source cell are set to 0, which tags the object as invisible. Using the computed pv.v.'s, for visible objects this bit is set to 1. Run-time visibility determination can now be done by each object first checking the visibility list stored in its parent cell[21], and only if its visibility bit is 1 continue with the object-frustum intersection test. Because the base class of every tree object in vrSS is *CTEntity* and visibility processing is done in the virtual *Process* method, the adjusted *CTEntity::Process* method now becomes (changes shown in italic print):

[20] Or going one step beyond, also provide group nodes where all children are invisible with an index.
[21] Note that storing the p.v.v. set for a source cell has been distributed over all destination cells to localize visibility information and to implement a fast look-up scheme.

```
void CTEntity::Process( Frustum Fr )
{
  // has visibility been determined pre-run-time for this object?
  if( ParentCell && PVSIndex != -1 )
  {
    // if the object is invisible, stop further processing
    if( ParentCell.VisibilityList[ PVSIndex ] == 0 )
      return ;
  }
  ... rest of Process method ...
}
```

Fig. 78 Adjusted *CTEntity::Process* method.

This requires that *CTEntity* gets an extra member variable, *PVSIndex*, which is initially set to -1. Only if the entity has been indexed and a visibility list is selected will its visibility be evaluated using the visibility list, otherwise the default visibility determination implemented in the *Process* method of its descendants will be used.

Because usually there are several source cells that can see into a destination cell, a list of source cells and their accompanied visibility lists has to be stored for a destination cell. When the camera changes physical cells, the cell should notice this and select the according visibility list. Should there exist no such list for the current source cell, an empty visibility list is selecting signalling the *CTEntity::Process* method to skip checking the visibility list. The adjusted *FCCell::ProcessCell* now becomes:

```
void FCCell::ProcessCell( Frustum Fr )
{
  // has the camera changed cells?
  if( GetPhysicalCameraCellName() != CurrentSourceCellName )
  {
    CurrentSourceCellName = GetPhysicalCameraCellName() ;

    // if there exists a visibility list for this source cell select it,
    // otherwise use the default list in which each object is visible.
    if( PVVSet.VisibilityListExists( CurrentSourceCellName ) )
      VisibilityList = PVVSet.GetVisibilityList( CurrentSourceCellName ) ;
    else
      VisibilityList = NULL ;
  }
  ... rest of ProcessCell method ...
}
```

Fig. 79 Adjusted *FCCell::ProcessCell* method.

7.6.3 Gathering heuristics

When computing the p.v.v. sets for a world, several statistics are gathered and written to a file. These statistics for a given source cell include:

- List of visible cells;
- Number of objects in a visible cell;
- Number of visible objects in a visible cell;
- Number of triangles in a visible cell (approximation);
- Number of visible triangles in a visible cell;
- Total number of triangles for all visible cells;
- Total number of visible triangles for all visible cells.

By carefully studying these statistics when creating a model, or by evaluating the weak spots in a model with weak performance, smoother overall performance can be gained.

Each of the tasks described above is performed by class *FCPVSBuilder*, which still has a very simple interface as will be shown next.

7.6.4 The FCPVSBuilder class

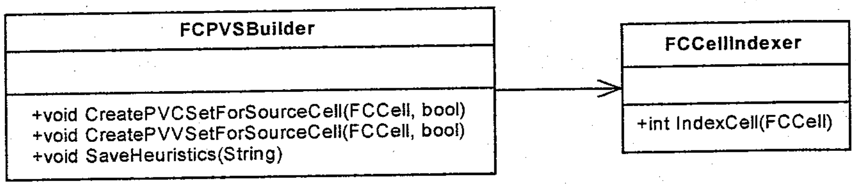

FCPVSBuilder
+void CreatePVCSetForSourceCell(FCCell, bool)
+void CreatePVVSetForSourceCell(FCCell, bool)
+void SaveHeuristics(String)
FCCellIndexer
+int IndexCell(FCCell)

Fig. 80 The *FCPVSBuilder* class.

The *FCPVSBuilder* class has a method *CreatePVCSetForSourceCell* that, when passed a source cell, computes the potentially visible cells list for that cell and stores it in the source cell's *FCPVCSet* object.
FCPVSBuilder also has a method *CreatePVVSetForSourceCell* that, when passed a source cell, computes the visibility list for each visible cell and stores this in the *FCPVVSet* object of each visible cell. In order to index the objects in the cell tree *FCPVSBuilder* uses an instance of class *FCCellIndexer*.
For both methods, if *store* is set to true, *FCPVSBuilder* will call the *FCCell::StorePVCSet* resp. the *CCell::StorePVVSet* methods to store the computed information. These sets are stored by the *FCCell* class to keep the information as local as possible.
In addition, *FCPVSBuilder* can save all the heuristics gathered after computing the p.v.v. set or p.v.c. set for a source cell to disk by calling *SaveHeuristics*.

7.6.5 The updated FCPortal class

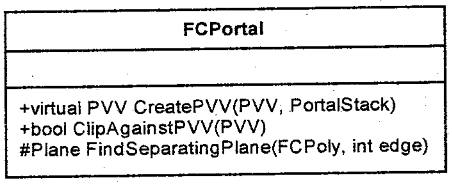

FCPortal
+virtual PVV CreatePVV(PVV, PortalStack)
+bool ClipAgainstPVV(PVV)
#Plane FindSeparatingPlane(FCPoly, int edge)

Fig. 81 The FCPortal class.

Computing the potentially visible volume for a cell is performed by the destination portal. This allows for the use of special portals like mirrors (see chapter 6) to define correct p.v.v.'s to use for pre-run-time visibility determination.

7.7 Summary

Using portals, well-chosen supersets of potentially visible objects at *cell-to-cell* level (that is, objects in one cell visible to a viewer in another cell) can be computed that can be used for more than visibility determination alone: they are also quite helpful for dynamic cell management and heuristics gathering.

Currently though, only these three applications of potentially visible sets have been implemented in vrSS but other very useful applications, like the level-of-detail cell trees discussed in section 7.5, exist and are worth studying further. Unfortunately due to time constraints, this could not be done within the time period for this project.

Although not a real performance booster when combined with real-time portal processing, by considering more occluders than portals alone when computing potentially visible sets, the resulting superset of visible objects could be reduced even further. This could make their influence noticeable even when combined with real-time portal processing in less complex worlds. However, even though potentially visible sets are computed pre-run-time, considering ALL occluders in a complex world will quickly become too heavy a computational task so a balance must be struck. This would also be worth studying further.

Chapter 8: Summary and conclusions

8.1 Thesis summary

Over the course of the previous chapters, portal processing and many of its applications have been examined. As was shown, portal processing can be used to speed up visibility processing for both static scenes (chapter 3) and dynamic objects (chapter 5). This performance gain is caused by:

- Early culling of invisible objects by the narrowed frustums, creating a smaller superset of visible objects than would have been created using the standard view frustum;
- Enhancing Z-buffer performance, used for exact visibility determination, by ordering cells *near-to-far*;
- The ability to apply the visibility determination algorithm to dynamic objects also by keeping track of the cell(s) the objects physically reside(s) in, therefore allowing the visibility of those objects to be determined using the narrowed frustums for those cells.

It was shown that, due to the nature of the algorithm, the major disadvantage of the portal processing algorithm is that it is highly dependent on the availability of large occluders in a scene, restricting its use to (building) interior type scenes. Another disadvantage is that the scene must be partitioned into cells and portals at design time, or that otherwise a scene must be partitioned (automatically) later on.

However, as was shown in chapter 4, due to the partitioning of the scene into cells and portals, a favorable subdivision of the world is created that can be used for dynamic memory management. In this chapter, some situations requiring memory management were addressed and the different *dynamic cell managers* best suitable for handling those situations were discussed.

Chapter 5 showed how to incorporate *inter-cell objects* (e.g. light- and sound sources) into scenes, and how dynamic objects can also be made to profit from visibility determination using the narrowed frustum. Chapter 5 also showed that cells and portals themselves are not required to be static structures: both cells and portals can have a dynamic nature, allowing the creation of much more interesting virtual worlds, while still benefiting from portal processing's performance gains.

The subject of creating interesting worlds was further discussed in chapter 6, which showed how, using portals, a multitude of special effects (like mirrors/monitors/reflecting windows/etc.) could be added to a world. However, these additions come at a price: extra processing time, required to process/render the extra information (e.g. the reflection of a mirror). Also often, for a certain type of effect to be achieved, changes to the core portal processing code are required which can influence the overall portal processing algorithm performance.

Finally chapter 7 concentrated on scene pre-processing in order to compute *potentially visible sets*, and it was argued that even though they don't add much to the performance gain already achieved when using (run-time) portal processing, their usefulness in other areas (e.g. dynamic cell management, level-of-detail cell trees) is undeniable.

When combining all applications discussed in this thesis, near-infinite size interior-type scenes of very high detail can be created and used interactively. Using dynamic cell management and an implementation of the *level-of-detail* scheme discussed in chapter 4, memory resources can be adequately managed (as was shown by Funkhouser [4]). Due to the portal processing algorithm, culling the majority of objects in the visible cells positioned farther away from the viewer, most of the rendering power can be concentrated on drawing detailed objects close to the viewer. Add to this the flexibility of using dynamic objects, including cells and portals themselves, and special effects portals to enhance the viewers experience, the range of applications is near infinite, ranging from architectural walkthroughs to computer games.

8.2 Empirical performance evaluation

Unfortunately, Mondo Bizzarro B.V. ceased all activities in October 2001, effectively grinding down further vrSS development to a halt[22]. This made the implementation of some of the more advanced features described in this thesis (such as special effects portals, but also in part level-of-detail cell trees) within the time window for writing this thesis, impossible. Therefore, the empirical results presented in this chapter are confined to those parts that have been implemented.

To test the performance of portal processing vs. standard vrSS tree processing, and also to test the performance of inter-cell objects, a simple 3D game of pacman was created: The maze consists of 583 cells, and the walls of the maze account for 29,988 polygons (triangles). There are 2735 energy dots for pacman to eat, each consisting of 20 polygons. The smallest cells will only contain one energy dot, whereas the largest cells will contain up to 12 energy dots.
Note how this layout is also representative for other indoor-type scenes, e.g. an office building: an office building is also made up of multiple cells, some large and some small, but for each the majority of polygons is usually determined by the objects inside the cell, not the cell boundaries (walls) themselves. Larger rooms will usually contain more geometry just as the larger cells in the maze contain more energy dots. Also, only a small subset of the total set of cells in the world will be visible.

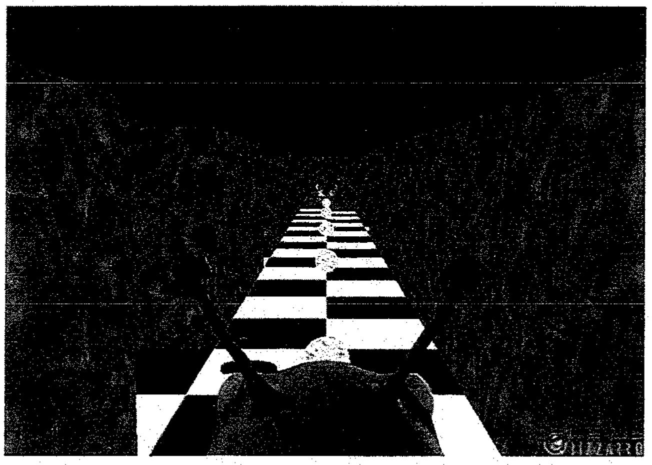


Fig. 82 Screen shot of Pacman and Inky.

Fig. 83 shows the percentage of polygons processed from 290 sample points using both standard vrSS processing and portal processing, taken from the point of view of pacman when moving along a pre-defined path through the maze. Notice the scale difference of the values along the Y-axis! While without portal processing up to 67.76% of the total amount of triangles is processed, the maximum amount with portal processing is only 0.69%, a reduction of almost a factor 100!

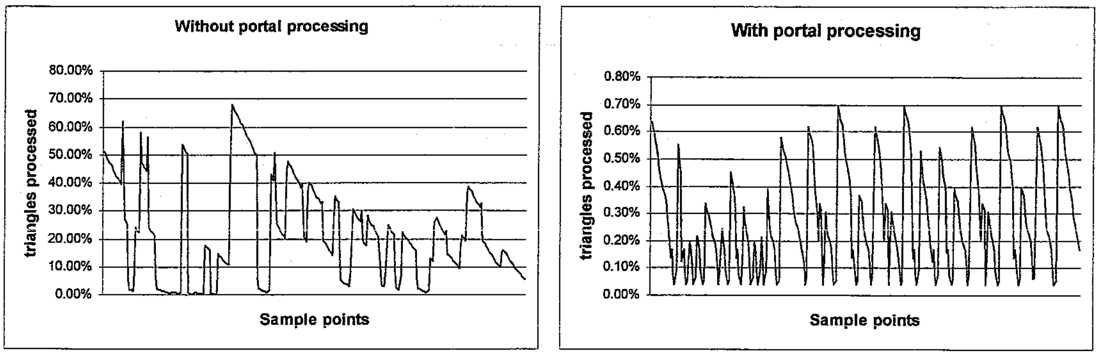


Fig. 83 % of triangles processed.

With portal processing, performance is mainly dependent on the complexity of the visible cells. When more (complex) cells are visible, performance is less than when less (complex) cells are visible. In Fig. 83, for the 'with portal processing' graph, the peaks represent the moment pacman enters a new corridor and can see the entire corridor with its power dots and all corridors connecting to it (the situation depicted in Fig. 82), while the valleys represent the moment pacman turns around a corner basically seeing only the corridor walls.

Without portal processing, there is an extra factor resulting in performance loss: the depth complexity of the scene. Here the peaks in Fig. 83 represent the camera being positioned and oriented so that most of the maze falls within the frustum. The valleys represent the camera being positioned and oriented so that most of maze falls outside of the frustum, resulting in performance equal to that of the portal processing algorithm.

As can be seen from Fig. 83, the depth complexity of the scene causes most of the performance loss, or in other words, portal processing and scene partitioning effectively reduce redundant rendering.

For the standard algorithm, moving objects can only worsen performance as they add extra depth complexity to the scene. However, for the portal processing algorithm, inter-cell objects have been created (see chapter 5) to enhance performance of moving objects. To examine the performance gain of inter-cell objects versus the alternative, global objects, four ghosts (Blinky, Pinky, Inky and Clyde, each consisting of 11,188 triangles) chasing pacman are added to the maze. In this test, both pacman and the ghosts move along a pre-defined path as to be able to create an identical set of sample points for both cases. Fig. 84 shows the difference between the amount of triangles processed in the case of using global objects for the ghosts and in the case of using inter-cell objects ($\#_{\textit{triangles processed using global objects}} - \#_{\textit{objects processed using inter-cell objects}}$):

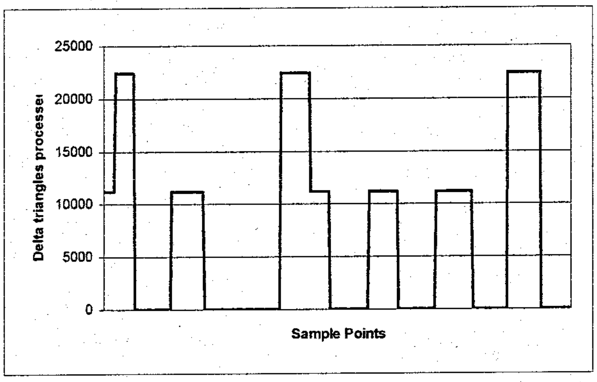


Fig. 84 Inter-cell objects versus global objects.

As Fig. 84 shows, when using global objects, unnecessary extra depth complexity is added to the scene by ghosts that intersect the frustum but are residing in an invisible cell. From this it can be concluded that inter-cell objects indeed speed-up performance, based on the amount and complexity of moving objects that can intersect the frustum without being visible to the viewer.

Unfortunately, determining the 'Z-buffer performance gain' by using near-to-far processing of cells could not be performed within the time period for writing this thesis. This was due to both time constraints and vrSS limitations: the performance gain is achieved by reducing the amount of overdrawn pixels in the Z-buffer, however vrSS has no provisions for measuring this overdraw. Measuring frame rates is not very accurate as the test programs are running on a multi-tasking system where system processes running in the background can interfere. Also, currently vrSS does not support some of the more time consuming pixel operations like bump mapping and reflection mapping, making overdraw with today's fast hardware accelerators not a limiting factor. A simple test program consisting of a world of ten consecutive cells, each containing 200.000 polygons lit by four light sources showed an estimated performance gain of only 0.2 frames per second.
However, even though the performance gain is small, it comes free with the portal processing algorithm and might improve performance of future versions of vrSS more significantly.

[22] However, chances are that development will be continued soon.

8.3 Portals vs. BSP-trees

A comparison that quickly comes into mind, and is therefore included in this thesis, is that of visibility determination using BSP-trees and visibility determination using portal processing. Although both algorithms indeed speed up visibility determination, they each do so in a very different way.

BSP-trees spatially partition a scene using separating planes, where each object intersecting one of these planes is split into two separate parts (creating more polygons than were originally in the scene). When determining the visibility of objects in the view frustum, due to an object having been split up into several parts, often some of these parts are culled.

Portals, on the other hand, don't split objects. When an object intersects the view frustum, the entire object is processed (in vrSS). The speed-up using portal processing comes from view frustum narrowing by the portals, therefore culling more objects from view.

Both algorithms suffer from the same problem: when many objects are positioned behind each other, with the nearer objects occluding (most of) the farther objects, no performance gain is achieved (even worse, performance is lessened due to the extra processing overhead). However, interior-type scenes with many occluding walls portal processing will generally perform better due to a (much) higher number of culled objects.

Therefore, from these observations made about the two algorithms, it is only logical to conclude that they complement each other very well: using portal processing for *inter-cell visibility*, and then use a BSP-tree for (static) *intra-cell visibility*, the best of both worlds is combined. Portal processing creates narrowed frustums for the visible cells that are then used to cull large parts of split objects.

Although with this combination the Z-buffer is no longer required for exact visibility determination (since portal processing can be adapted to process cells far-to-near, and BSP-trees allow exact visibility determination within the cells), it facilitates incorporating dynamic objects in the world and is therefore best kept.

8.4 Future points of research

Finally, to finish this thesis, a summary of potentially interesting subjects that were touched upon, but not examined further, is given. Unfortunately, portal processing and its many uses is too comprehensive a subject to examine completely within the time period for writing a master thesis. The following subjects are therefore open for further investigation and will hopefully be addressed in the near future:

- Determining when to use a narrowed frustum to process the connected cell of a portal, and when to use the frustum used to evaluate the parent cell of that portal or another frustum. Fig. 85 explains this:

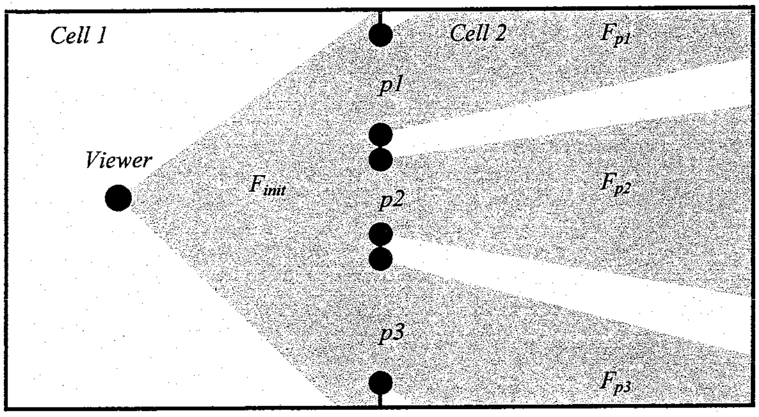


Fig. 85 When not to use frustum narrowing.

  Only a very small fraction of *Cell 2* is actually occluded from sight to the viewer positioned in *Cell 1*, resulting in having to traverse the cell tree of *Cell 2* three times when using portal processing, once for each of the frustums F_{pi} created by portal *pi*. Since in the final scene, probably all of *Cell 2*'s objects have now been rendered, it would have been faster to just process its cell tree once using frustum F_{init} or another suitable frustum;
- Automating the process of partitioning a world into cells and portals. A method for this has already been proposed by Teller and Sèquin [3] but their design can be improved on. In addition, using an automated process, usually more cells and portals are created then when leaving this task to a modeler, potentially resulting in better performance, but also potentially reducing performance: a careful balance must therefore be struck;
- Combining the inter-cell portal processing algorithm with intra-cell BSP-trees. This combines the best of both worlds: the portals narrow the view frustum, and due to the splitting of polygons in the BSP-tree usually only a part of the object intersecting the narrowed frustum is rendered. Another option is to use the pre-computed potentially visible volumes to split objects so that no more splits than required are created that can be used to speed up processing objects in the connected cells of the current camera cell;
- Adapting the portal processing technique for real-time occlusion culling: by pointing out large occluders in a scene, a *shadow frustum* [5] can be created to quickly cull away portions of the scene.

References

[1] M. Abrash, *ZEN of Graphics Programming, 2nd Ed.*, Coriolis Group Books, 1996

[2] J.L. Hennessy, D.A. Patterson, *Computer Architecture: A quantative approach, 2nd Ed.*, Morgan Kaufmann, 1996

[3] S.J. Teller, C.H. Séquin, *Visibility Preprocessing For Interactive Walkthroughs*, Proc. SIGGRAPH, 1991

[4] T. Funkhouser, S.J. Teller, C.H. Séquin, *Management of Large Amounts of Data in Interactive Building Walkthroughs*, Proc. Symposium on Interactive 3D Graphics, 1992

[5] Hudson et. al., *Accelerated Occlusion Culling using Shadow Frusta,*

[6] D. Sunday, *Intersections of Rays and Segments with Triangles in 3D*, 2001

[7] D. Luebke, C. Georges, *Portals and Mirrors: Simple, Fast Evaluation of Potentially Visible Sets*, Proc. Symposium on Interactive 3D Graphics, 1995

[8] P. Dirkse, L.Lunesu, *vrSS Documentation*, 1999 - 2001

[9] H. Fuchs, Z. Kedem and B. Naylor, *On Visible Surface Generation by A Priori tree structures*, Computer Graphics (Proc. SIGGRAPH), 1980

[10] E. Catmull, *A Subdivision Algorithm for Computer Display of Curved Surfaces*, Dept. of CS, University of Utah, 1974

[11] Doctor, J. Torborg, *Display techniques for octree-Encoded Objects*, CG & A, 1(3) , July 1981

[12] M.S. Patterson, F.Frances Yoa, *Efficient binary space partitions for hidden surface removal and solid modelling*, Discrete and computational geometry, 1990