Curtin University of Technology
School of Computer Science

Computing Project
Semester 2, 1999

Realtime Walkthroughs of Massive Architectural Models

Name: Bernhard Tschirren
Student ID: 970623I
Supervisor: Andrew Marriott

Declaration: Except as clearly indicated in the text, this project is solely my work and has not been submitted for assessment in any other unit or by any other student.

Signed:

Date:

CONTENTS

1 ABSTRACT

Clients are expected to visualise large complex architectural models given only a two-dimensional floor plan. A better approach would be to provide a three-dimensional view of the building, as seen from a hypothetical human observer. The client should be able to interactively control this observer, so that it is possible for them to explore the architecture before it is built.

The architectural model must be completely dynamic. That is, the designer must be able to make changes to the model without having to perform an expensive pre-processing phase. The only acceptable exception is pre-calculated lighting. Radiosity and raytracing schemes are far too slow to be performed in realtime. As a compromise, the system should support both dynamic lights, and more realistic pre-computed light maps.

The architectural walkthrough system described in this paper does not require any specialty hardware. It simply requires a desktop PC equipped with a recent graphics accelerator. To achieve interactive frame rates, the system uses a model that is partitioned into cells and portals. These cells and portals are stored in a cyclic scene graph. A depth-first-search starting at the viewer's cell calculates the visible cells. Hidden cells are quickly culled by checking portals for visibility. If a portal is not visible, the cell behind it is also not visible. In this manner, a large amount of geometry can be culled away.

The results show that this cell and portal approach is very effective. In wire-frame mode, it is very clear that hidden triangles are quickly culled away. Because the system is very scalable, this statement will hold true for architectural models composed of millions of triangles.

2 INTRODUCTION

A two-dimensional floor plan is both used to design and visualise an architectural model. When talking to a client, the architect must assume that they can visual what the building will look like when it's built. However, a typical client who has commissioned a building may have no experience with visualising a building given just a blueprint, and maybe a cardboard model.

A better approach would be to provide a three-dimensional view of the building, as seen from a hypothetical human observer. The client should be able to interactively control this observer, so that it is possible for them to explore the architecture before it is built. It should then be possible to make changes to the model to see how they affect the final building.

Architectural models are well suited to realtime exploration because they are densely occluded. That is, at any one time, only a small percentage of the entire model is visible to the observer. Most of the geometry is blocked by floors, ceilings and walls. Algorithms can exploit this feature of architecture to provide interactive visualisation of even large and complex models.

This project proposes several goals and objectives to describe such a walkthrough system. Algorithmic solutions are then presented that achieve these ambitious goals. This paper describes an implementation of a system that allows the user to view massive architectural models in real-time. The system uses the densely occluded nature of architectural models to cull large areas that are not visible to the observer. It supports both dynamic and pre-computed lights to provide a very realistic environment. Finally, the system does not require any specialised hardware to operate. It simply requires a cheap consumer graphics accelerator.

3 BACKGROUND

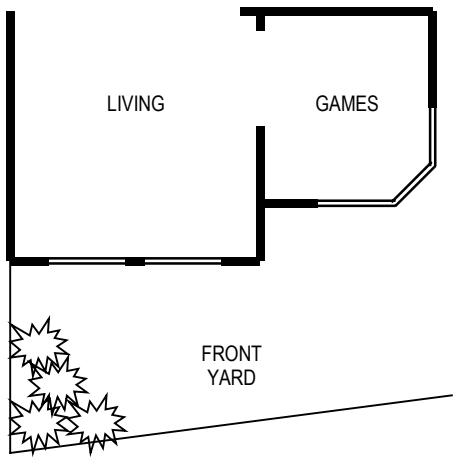
Realtime walkthroughs of architectural models were first investigated at UNC by (Brooks, 1986). Brooks started the (Walkthrough Project), which has since become the primary research body studying realtime architectural walkthroughs. Brooks' system used a BSP-tree (Fuchs, 1980) to provide visibility culling, and achieve near realtime performance. Unfortunately, this BSP-tree approach had two major limitations. Generating even a sub-optimal BSP-tree was a very time consuming operation, and once the BSP-tree was created the model could not be modified.

An alternative to BSP-tree culling was presented by (Airey, 1990). Potentially visible sets (PVS) of the model were pre-computed for different viewpoints. Rendering the scene was as simple as selecting the correct viewpoint, and sending the associated PVS to the graphics pipeline. This approach suffered from the same problems as the BSP-tree method used by (Brooks, 1986), namely long pre-processing phases and a static model. However, it was a major step in the right direction.

(Teller, 1991) subdivided the model into rectangular cells. Cell-to-cell visibility was pre-computed by linking cells that had an unobstructed sightline between them. This created an adjacency graph that identified which cells were directly visible by each cell. At each frame, the cell containing the camera was identified and the potentially visible cells were loaded from secondary storage. Tests revealed substantially reduced rendering loads. However, this approach assumed that all opaque walls and portals were axis-aligned (to accommodate the rectangular cells). This approach also suffered from excessive pre-computation.

Run-time evaluation of PVS was introduced by (Luebke, 1995). His system was heavily based on work by (Jones, 1971) who initially pioneered the concepts of cells and portals as a means of solving the 'Hidden Line' problem. Jones' approach was to subdivide the entire scene into convex polyhedral cells that were connected by convex 2D regions called portals. Cells were just logical groupings of geometry, and portals were just special kinds of polygons. Walls and portals were rendered starting at the cell that contained the viewer. As each portal was encountered, the cell behind that portal was recursively drawn (clipped into the portal). Because each cell was convex, rendering exhibited zero overdraw.

Luebke simply applied the system proposed by Jones in an architectural context. Cells corresponded to rooms bounded by walls and partitions. Portals corresponded to doors and windows through which neighbouring rooms could be seen (see Figure 1). In addition, the limitation that each cell had to be convex was removed. Hardware rendering devices could eliminate the problem of overdraw using a Z-buffer much more efficiently than the clipping approach proposed by Jones.



The living room and the games room each correspond to a cell. These two cells are connected by a single two-way portal.

If the viewer was standing in the living room looking at the solid wall on the left, the portal would not be visible, and hence, the geometry behind the portal (the games room) would not be visible. However, if the viewer was facing to the right, the portal would be visible, and rendering would recurse into the games room.

The front yard also constitutes a cell that is visible through the windows of both the living room and games room. Note that the living room contains two portals to the front yard, and the games room contains three portals to the front yard.

Figure 1: Excerpt of a house plan divided into cells and portals

To determine whether a portal was visible, the system described by (Luebke, 1995) maintained a rectangular 'current viewport'. Initially, the current viewport was set to the entire screen. Portals were then projected into screen-space and their rectangular bounding-box tested for intersection with the viewport. If the portal was visible, the current viewport was set to the intersection of the viewport and the projected portal's bounding-box (thus limiting the view through both the viewport and the portal, see Figure 2).

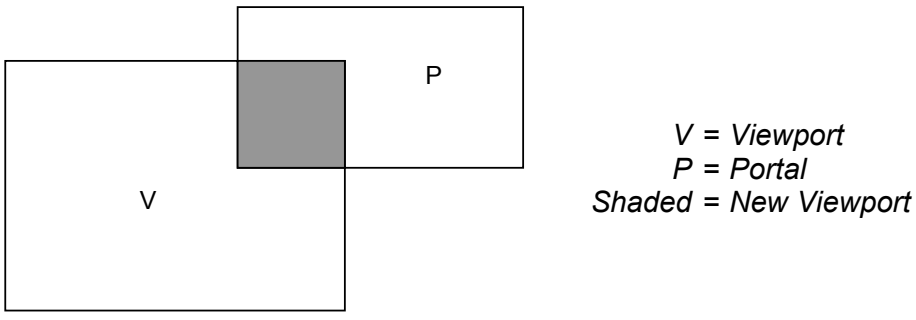


Figure 2: Intersection between Viewport and Portal

4 OBJECTIVES

4.1 Dynamic Environment

The primary goal of this walkthrough system is to provide a truly dynamic environment. Walkthrough systems exist to let architects, interior designers and clients visualise a large model. In addition, questions like “what would happen if I extended this room by one metre” must be quickly answerable. Therefore, the model must be modifiable in real time.

The ideal architectural walkthrough system should allow the architect to design a multi-level building and get an immediate 3D preview. It should then be possible to walk through this model in realtime, as if walking through the real building. The architect should be able to make changes to the model while observing the 3D preview. It should be possible to interactively add, remove and modify rooms, doors, windows and objects (tables, chairs, etc) in realtime.

The system must also support dynamic lights. Adding lights “that look right” to a scene can take a huge amount of time. If the system provides instant feedback as lighting parameters are changed, the architect can be much more productive.

To provide this dynamic environment, no visibility pre-processing can be performed. This is a very ambitious goal, because all current walkthrough systems perform a huge amount of pre-computation. For example, the system described by (Funkhouser, 1996) spends five minutes dividing the scene into cells and portals. Then, a further three hours and 31 minutes to compute cell-to-cell and cell-to-object visibility lists. The time it takes to apply the radiosity lighting is not even mentioned!

This amount of pre-computation means that the architectural model is very static once it is “compiled”. An animator can place a spline path through the model, and render an animation that follows this path in less than three hours. What’s the point of a walk-through, if it takes half a day to prepare the model!

4.2 Realistic Environment

The UCB system described by (Funkhouser, 1996) is restricted to axis aligned, rectangular cells. Unfortunately, not many architectural models lend themselves to these limitations. The system must be able to handle any kind of model thrown at it.

The Curtin University computer science building contains a window in the shape of a triangle. Most current architectural systems can only handle rectangular portals. Placing such a restriction on the shape of a portal is simply ridiculous. The system must be able to handle portals in any shape.

Probably the biggest contributor to realism is lighting (and shadows). A scene without shadows simply does not look right. Unfortunately, generating realistic lighting effects can take a very long time. This clashes with the objective that the environment must be dynamic. A solution is to provide support for both static and dynamic lights. Therefore, the user can decide whether they wish to sacrifice a dynamic environment to get higher levels of realism.

4.3 COST EFFECTIVE

Only a few years ago, graphics accelerators were not very common (and certainly very expensive). Nowdays, even cheap desktop PC’s come with a graphics card that supports 3D acceleration (via OpenGL and Direct3D).

Currently available walkthrough systems (Walkthrough Project; Funkhouser, 1996) were designed before the graphics accelerator was commonplace. They use complex CPU intensive algorithms to try to minimise the number of triangles sent to the graphics card. (Funkhouser, 1996) states that their Silicon Graphics Power series 320 workstation with Reality Engine graphics can render 50-100K Gouraud shaded, texture-mapped polygons per second. A PC equipped with an nVidia Riva TNT2 can draw 9 million Gouraud shaded, texture-mapped, Z-buffered triangles per second (nVidia, 1999). The latest offering by nVidia, the geForce-256, can handle up to 15 million triangles per second. This card also supports hardware transformation, clipping and lighting.

The architectural model used by (Funkhouser, 1996) (a fully furnished copy of their seven floor computer science building Soda Hall) contains 1,418,807 polygons. If on average a polygon can be split into 5-triangles, this comes to 7,094,035 triangles. On the Riva TNT2 card, this can theoretically be rendered in under a second. In reality, it will take a bit longer due to CPU and bus bandwidth limitations.

However, despite these impressive results, the walkthrough would still not be interactive. Interactive rates require at least 15 frame updates per second. The appearance of “smooth” movement requires at least 30fps. This target frame rate limits the number of triangles that can be rendered to a theoretical maximum of 300,000 on a Riva TNT2.

These statistics are important, because they indicate that a graphics engine should not worry about eliminating individual triangles. In the past, algorithms concentrated on exact culling (for example the hidden line problem). Today, the problem is reduced to eliminate invisible geometry as quickly as possible. Partially visible objects need not be considered because the graphics card’s depth buffer will take care of that.

5 CELLS AND PORTALS

A simple solution that meets all the stated objectives partitions the model into a number of cells. As described in the Background section, a cell is simply a collection of objects that should logically be bundled together. In an architectural environment, a cell maps nicely onto a room. The room’s floor, ceiling, walls and furniture can all be grouped together as a single cell. The other ingredient, the portal, links these cells together. A portal corresponds to doors and windows. It is a convex polygonal region through which the neighbouring cell can bee seen. The magic property of portal is this: if the portal is not visible, the cell the behind the portal is also not visible. This property allows large areas to be culled very quickly and efficiently.

The main advantage gained by using cells and portals, rather than a precalculated PVS or BSP-tree, is a dynamic environment. The UC Berkley System described by (Funkhouser, 1996) precomputes cell-to-cell and cell-to-object visibility. This information requires over three hours to compute. This is unacceptable because it does not allow the model to be later modified. Every time a single piece of furniture is moved, the visibility information needs to be recomputed! In a cell and portal environment, no pre-processing needs to be performed. Furniture can be moved around inside a cell without any restrictions. Moving objects from one cell to another is also possible, although a little more difficult (described later). Therefore, the portal solution fully meets the objective to have a dynamic environment.

A cell is just a collection of objects that are closely grouped together in space. The walkthrough engine can utilise this information when performing collision detection. Objects only need to be tested for collision against other objects in the same cell. This significantly reduces the number of object to object collision tests that need to be performed. However, this approach introduces some problems when an object spans across multiple cells. This happens whenever an object is about to cross from one cell to another via a portal. Because an object usually does not occupy a single point in space, it can reside in both cells.

Another advantage of using cells and portals is that lighting can be constrained to individual cells. When a dynamic light is added, for example an OpenGL point light, it illuminates all subsequent geometry. This can be a problem, because a wall will usually occlude the light for following geometry. A cell can effectively constrain the light to closely grouped objects. In an architectural context, this means a light can be constrained to individual rooms. Of course, this approach suffers from the problem that light does not spill into neighbouring cells through portals. Therefore, a light in one room will not illuminate an adjacent room even if the door is open. However, testing shows that this is not really a problem, and image quality is only minimally affected.

Cells also provide an advantage when pre-computing (or rather re-computing) static lighting (usually supplied by a Radiosity algorithm). Whenever the geometry in a cell is changed, the lighting computation can be restricted to that cell. This is not currently implemented, however it seems a feasible topic for future research. Interactive design can then be achieved in conjunction with realistic looking previews.

There is one major disadvantage to using a cell a portal system. No current modelling tools can handle cells and portals. That means, generating a model is hard work that involves quite a bit of manual labour. It also means that there are no file formats that can store cell and portal information. All other walkthrough systems in existence use either proprietary file formats, or heavily modified standard formats. For example, the Walkthough Project at UNC uses the FLY format, which is actually a complex scripting language. The UC Berkeley system uses the proprietary UNIGRAFIX file format (Funkhouser, 1996).

There are no tools available that automatically partition an arbitrary architectural model into cells and portals. This is unfortunate, because it means that the architect should design the model with cells and portals in mind. Luckily, it is not a difficult task to break up an existing model into a set of cells and portals.

6 SYSTEM DESIGN AND IMPLEMENTATION

6.1 Organisation

The system is constructed as shown in Figure 3. The scene is described in a text file that contains the data for the entire architectural model. This data is then loaded into a cyclic scene graph. This scene graph stores all the objects as graph nodes. This includes the cells, portal, tables, chairs, doors, cameras, etc.

The render process traverses the scene graph and sends all visible geometry to the render device. Currently only a cell and portal render device is implemented. However, any kind of rendering scheme could be substituted here.

The render device is responsible for displaying the results on the screen. This component is also very modular. Currently, only an OpenGL render device is implemented. However, a driver could be written to support other devices (such as DirectX, an image file or a printer).

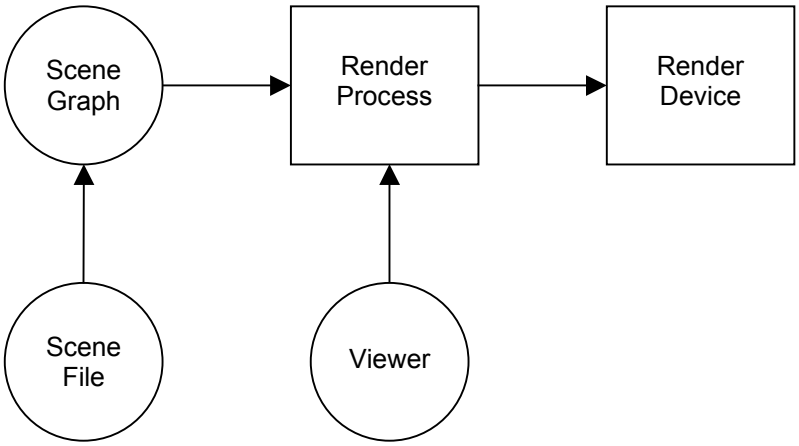


Figure 3: System Overview

6.2 Scene Graph

The entire architectural model is stored as a cyclic scene graph. Do not get this confused with traditional scene graph systems such as SGI Performer and Java3D. Those systems restrict the graph to be acyclic. That is, the graph cannot contain any loops (see Figure 4).

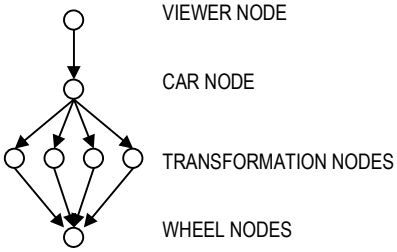


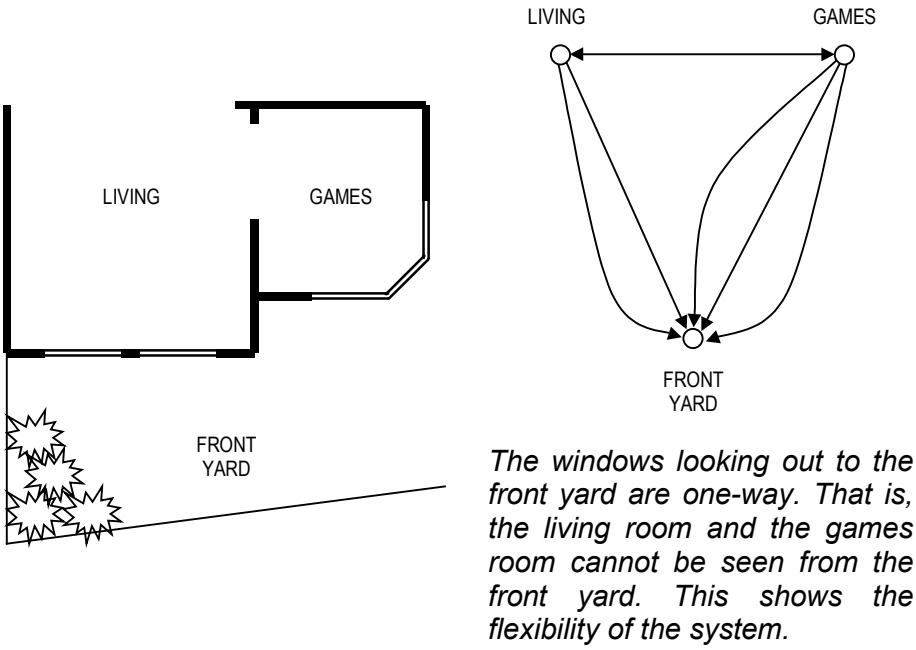
Figure 4: A traditional scene graph

The cyclic scene graph used by this system is much less restrictive because it allows loops in the structure. The presence of these loops is very important, otherwise a cell and portal system could not be mapped onto it. It is also important to note that other visibility schemes such as BSP-trees can also be mapped onto this scene graph system (Sweeney, 1998).

Much like a traditional scene graph, nodes represent objects, actions, transformations, etc. However, there is one major difference between this cyclic graph and a traditional acyclic scene graph: each node will only be rendered once. The cyclic scene graph can contain loops, and to avoid infinite recursions around these loops, each node can only be visited once.

In Figure 4, the four wheels are represented by four transformation nodes that link to a single node containing the wheel geometry. Therefore, the wheel will be rendered four times, each time at a different position. However, this cannot be done with a cyclic scene graph, because the node containing the wheel geometry will only be rendered once. Therefore, the cyclic scene graph would have to contain four separate wheel nodes. Unfortunately, there are many other side effects of cyclic scene graphs, and they will be described as they come up.

The project proposal (Tschirren, 1999) describes one method of mapping cells and portals onto this type of scene graph. Each node in the graph represents a cell. Each link (or arc) in the graph represents a portal that points to the next cell. Because each link is directed, a two-way portal (such as a door or a window) will require two links, one in each direction (see Figure 5). It is also possible for a link to loop from a cell back to the same cell. This loop represents a portal through which the current cell can be seen - a mirror.



The windows looking out to the front yard are one-way. That is, the living room and the games room cannot be seen from the front yard. This shows the flexibility of the system.

Figure 5: Cells and portals expressed as a scene graph

While developing this cell and portal implementation, it quickly became evident that it would not work. After some careful consideration, it became clear that a portal is not really an arc. A portal is another object in the scene, and much like a cell or a piece of furniture, it should be represented by a node. This has a very significant impact on the engine design. It follows that arcs are no longer special (ie: they no longer contain any data). Therefore, an arc reverts to its traditional function: to simply represent a relationship between two nodes.

The new scene graph for the above floor plan is shown in Figure 6. It may look more complex, however the data structures are much simpler because *everything* is now a node.

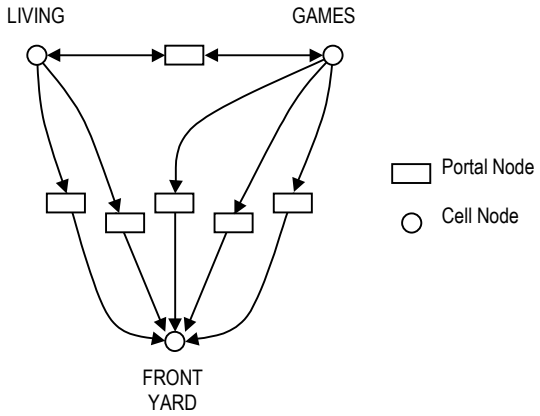


Figure 6: Portals are simply nodes in the scene graph

It is interesting to note that representing a portal as just another node in the scene has never been attempted before. Previous cell and portal systems have always treated a portal as something 'special'. Most systems, such those used by the Walkthrough Project, represent portals as 'special' polygons in the scene. However, this approach is flawed by design because it couples portals together with geometry. A portal should have nothing to do with the geometry.

By decoupling the portals from geometry, it is possible to store the geometry in any representation. For example, the implemented walkthrough engine uses triangles to store all geometry, while all portals are described by a convex polygonal outline. It is very easy to modify either the portal or the geometry representation without them affecting each other.

The project proposal also mentions special portals that link back to the same cell - mirrors. However, mirrors are not possible because each node is rendered only once. Therefore, a mirror would never reflect the scene because the scene has already been drawn in front of the mirror. However, this behaviour makes sense, because two mirrors could otherwise infinitely reflect each other. This is the second side effect caused by using a cyclic scene graph.

There are several ways to represent a graph structure in memory. Initially, the scene graph was modelled after the design specified by (Goldschmidt, 1998). This design uses two classes to represent a graph: nodes and arcs. An arc simply connects two nodes. This structure was ideal because it allowed portals to be mapped onto arcs. However, once the idea of mapping portals onto arcs was dropped, this structure became too clumsy and complex.

A better approach was inspired by the C++ Run Time Type Information (RTTI) library (Telea, 1997). Because this library fully supports multiple inheritance, it maintains a graph of all classes. This graph is stored using a form of adjacency lists (Aho & Ullman, 1992). Each class contains a list of parent classes (base classes) and a list of child classes (sub classes). This idea was adapted to the scene graph structure, such that each node of an incoming arc is a parent node, and each node of an outgoing arc is a child node. This completely removes the concept of an arc, and thus makes the scene graph structure very simple.

The scene graph must preserve the order of rendering. More specifically, it must maintain the order of all child nodes. The reason for this is certain nodes must always be rendered before or after other nodes. For example, lighting nodes must be traversed before geometry nodes, and nodes containing transparent geometry must always be rendered last.

The engine is heavily based on Object Oriented Design (OOD). Every node in the scene graph is derived from the RenderNode class. This class provides translation, rotation and scaling services to all sub-classes. New node classes can be added without affecting any existing code. In that respect, the scene graph system is highly modular and extendible. The current node class hierarchy is included in the program documentation in Appendix B: Source Code and Documentation.

6.3 Geometry

The display database for the model used by the UC Berkeley walkthrough system contains 14,478 objects, of which only 8,037 are unique. Their system supports object instances, such that each instance contains a transformation matrix describing the instance's position, direction and scale, and a pointer to the geometry. If it did not support this object instancing mechanism, the display database would have grown in size from 21.5MB to 349.5MB (Funkhouser, 1996). Therefore, supporting instances of objects is essential.

Unfortunately, as described earlier using the car wheel example, geometry sharing does not come naturally to a cyclic scene graph. Therefore, the system must explicitly maintain a global list of geometry, and each model node then has a pointer into this list. The final implementation of the walkthrough engine fully supports this, although the example scenes do not make use of it.

6.3.1 Triangles

The engine was initially designed to use polygons as the basic primitive to describe all geometry. However, several issues (related to data-structure simplicity and performance) caused the humble triangle to become the basic primitive.

Compared to polygons and bezier/nurb surfaces, triangles are very simple to store. They require a list of vertices and a list of indices (where every group of three indices describes a discrete triangle). All other surface descriptions can be converted into triangles (to varying degrees of accuracy). Therefore, triangles can approximate any surface.

However, the main reason for using triangles is that current consumer graphics hardware can only render triangles. That is, all polygons are eventually converted to triangles. This conversion usually happens in software by the graphics API. Therefore, it is quicker to just store the triangles in the first place.

Current graphics hardware is bandwidth limited, rather than fill-rate limited. That is, the hardware can render the primitives quicker than it can receive them via the graphics bus. The solution to this problem is to organise the geometry into triangle strips, where each subsequent triangle shares two of its vertices with the previous triangle (see Figure 7). Storing (n) discrete triangles requires (3n) vertices, whereas storing them as strips only requires (n+2) vertices. If each vertex contains XYZ coordinates (3 floats), a normal (3 floats), texture-coordinates (2 floats) and an RGBA colour (4 floats), the total size of each vertex is 12*sizeof(float) bytes. It is apparent that significant bandwidth savings can be made by storing the data as triangle strips.

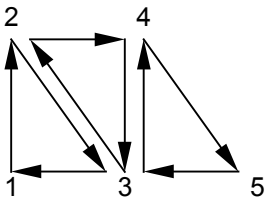


Figure 7: A triangle strip

The best representation of arbitrary geometry would be a list of triangle strips precompiled into a display list. Of course, this assumes that the geometry never changes once it is compiled. This may not be true for a dynamic environment, and re-compiling a list is very slow.

There is a simpler alternative available, which is actually quicker than display lists on current hardware. This alternative uses Compiled Vertex Arrays (CVAs) and exploits a well known “fast-path” through current OpenGL implementations. This mechanism is highly optimised because it is used by the upcoming first-person shooter Quake3 (ID software, 1999). The basic idea is to specify all triangle vertices and indices, and then explicitly tell the OpenGL driver that the given values will not change. This allows the driver to perform its own strip finding, which is very simple if the given discrete triangles are already in strip order. Therefore, the simple data-structure of discrete triangles can be kept, while reaping the benefits of triangle strips. This is the approach taken by the final implementation of the walkthrough engine. Display lists are also present, but are currently not used.

6.3.2 Surfaces

To present higher levels of realism, the engine must support pre-calculated lighting. This can be achieved by storing the shadow-map (also more popularly known as a light-map) in a texture. Then, this shadow-map is “modulated” (multiplied) on top of the texture (see Figure 8). Dark areas in the shadow-map will cause the final surface to become dark, while light areas in the shadow-map retain the standard light intensity (that’s why shadow-map is a better name than light-map). Most graphics hardware can perform these two passes simultaneously using two texture units.

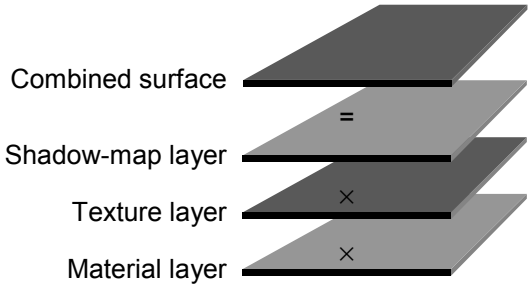


Figure 8: Multiple surface layers

To support this, the engine supports multiple surface layers. Each surface layer contains a texture map and a set of texture-coordinates. The final implementation fully supports this only if the graphics card has sufficient number of texture units. That is, it does not perform multiple passes to render all the surfaces.

6.4 File Format

As discussed earlier, there are no standard file formats for representing a scene using a cyclic graph. The only viable option was to create a simple and portable file structure similar to VRML. Initially, this was going to be a binary format because it's simpler to program. However, that made it very difficult to "hand-create" some initial test data. Therefore, the format was changed to a text file. This also has the advantage of being portable across multiple platforms.

The only difficulty pertained to storing the graph structure describing the scene. If the file format mirrored the structure in memory, each node would have to list all its neighbouring nodes. This does not fit well with the objective that the environment is truly dynamic. It should be possible to add new nodes without modifying existing nodes. Therefore, the file format needs to contain a command such as "connect these two nodes together".

The current implementation uses a simple but powerful representation. Each object in memory has an equivalent object in the file. However, the file may contain some additional "utility" objects that can perform some kind of action (for example, connecting two nodes together). These objects follow the following simple format:

```
class-name [object-name]
  {
  object-body           which is composed of
  }                     property = value pairs
```

More formally in EBNF:

```
file-format ::= object*
object      ::= class-name {object-name} '{' object-body '}'
class-name  ::= identifier
object-name ::= identifier
object-body ::= (identifier '=' value)*
identifier  ::= [a-zA-Z][a-zA-Z0-9]*
value       ::= integer | float  |
                boolean | string |
                object  | array
boolean     ::= "yes" | "on"  | "true"  |
                "no"  | "off" | "false"
array       ::= '[' value* ']'
```

The parser simply scans through the file and uses RTTI to construct an object of a class named ParseClassName (where ClassName is taken directly from the file). This parser object is then executed to scan the object-body and construct the object in memory. The object-body can theoretically contain anything, however it should follow the convention of (property = value) as set out by the EBNF rules above.

The parser maintains a list of all named objects. That's how the scene is reconstructed in memory. All node objects become part of the scene graph, and all geometry, material and texture objects get stored in a list (so they can be shared among the nodes). Any other objects are simply ignored.

It was only after developing the specification that this file format's true power was realised. For example, a feature similar to "#include" was easily added by creating an "Include" parser. Additionally, the ability to reference another object was added by creating a "Reference" parser. The problem of connecting two nodes together was solved by writing an "Attach" parser. All these utility "objects" are used by the example scene files.

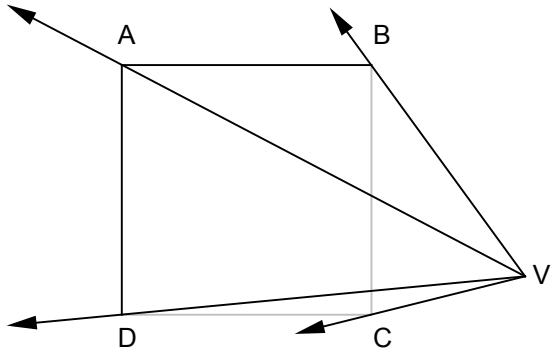
The main hassle with this text file approach is loading speed. It can take a long time to load a large scene. Time is mostly spent on constructing and destructing the parser objects. This tends to thrash memory, as a lot allocation and de-allocation is performed. Loading time would significantly reduce if this part of the parser were optimised.

In addition, the format is very verbose causing the files to be quite large. However, today's desktop computers come with huge amounts of disk space, and this is not really a problem. As a slight concession, the 3D-Studio Max ASE (Ascii Scene Export) format generates files nearly twice in size.

6.5 Render Process

To render the architectural model, the system traverses through the scene graph using a Depth First Search (DFS). This search starts at the cell node that contains the viewer. The viewer can be any node in the scene graph, however, usually a camera node is chosen.

When a portal node is encountered, the convex polygon that describes the portal outline is transformed into viewer space. Then, this transformed polygon is clipped against the current view frustum. If the polygon is completely clipped away, the portal node is not traversed any further. Otherwise, a new view frustum is cast through the clipped polygon (see Figure 9), and rendering recurses through the portal node. Figure 10 shows that the view frustum narrows as it traverses through the scene.



Viewer = *V*
Portal = *(A,B,C,D)*
Top Plane = *(A,B,V)*
Left Plane = *(D,A,V)*
Bottom Plane = *(C,D,V)*
Right Plane = *(B,C,V)*

Figure 9: Casting a view frustum through a portal polygon

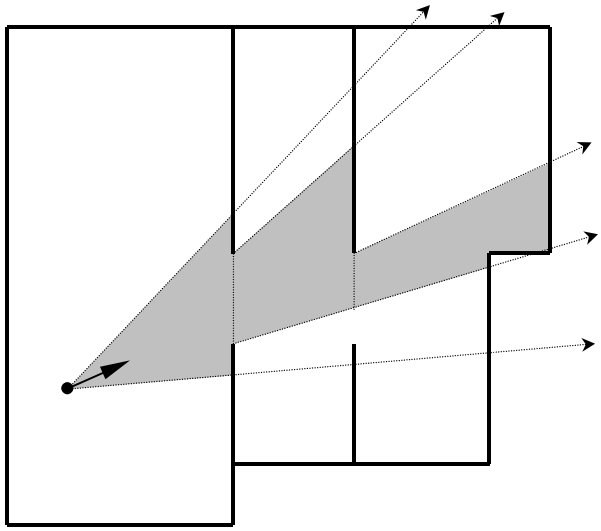


Figure 10: The view frustum narrows as it traverses through portals

The system maintains a stack of view frustums. Initially this stack contains a frustum calculated using the viewer's vertical field of view, the horizontal aspect ratio and the size of the window. Therefore, this initial view frustum bounds the entire visible area. Every time a portal node is reached, the current view frustum is pushed onto the stack, and the new frustum is calculated. When rendering behind the portal completes, the frustum is restored from the stack.

The original intent was to use OpenGL clipping planes to describe the frustum's planes. This could have a very positive effect on rendering performance, especially on next generation graphics cards that can perform the clipping in hardware. However, this is another situation that is affected by the "a node cannot be rendered more than once" problem. If all the geometry behind a portal is clipped, only the geometry visible through that portal will be drawn. A second portal leading to the same cell may expose more geometry. However, this geometry will not be drawn because the cell containing that geometry has already been visited.

Dynamic lights are also saved on a stack. Whenever rendering traverses through a portal node, all lights are saved on the stack. All lighting parameters are then cleared before rendering recurses through the portal. When rendering resumes, the lights are restored from the stack. This has the effect of localising all lights to a single cell.

6.6 Render Device

The render device is responsible for rasterising the geometry. Currently, this class is a sub-class of a window (as specified by the window manager). As its name suggests, this device should not be associated with a window. A render device should be able to represent any device capable of producing a final image.

The decision to make the render device a window sub-class was made due to code simplicity. However, the result of this is complex program flow. Whenever the window needs to be redrawn, the window manager sends an event to the render device. The render device then needs to call the render process, which in turn uses the render device for most of its work. This complex interaction between these two classes became clear only late in the development. Unfortunately, there was not enough time to fix it.

6.7 Viewer

The viewer can be any node that is ultimately attached to a cell node. It does not need to be directly attached to a cell node. It only needs to be a descendant of a node that is attached to a cell.

This has the useful property that any node in the scene (other than a cell) can be used as a viewer. In other words, the scene can be viewed from the point of view of any object. For example, the viewer can be set to a spotlight node, and the user can instantly see what this light illuminates.

The difficulty comes when the viewer node starts moving around the scene. Rendering starts at the cell that contains the viewer. Therefore, if the viewer moves into another cell, rendering must start from that other cell. The movement of the viewer from cell to cell must be tracked. In fact, the movement of any node must be tracked.

Ideally, this should be implemented as some sort of physics or animation system. That way, any node's movement could be tracked as it animates around the scene. Such a system can become very complex. Therefore, the current implementation only performs tracking of the viewer at the application level. It simply checks whether the viewer has "stepped" through a portal, and if so, attaches the viewer to the next cell.

As mentioned earlier, there is the problem that a node may occupy more than one cell. This problem can occur because an object is not simply a point in space - it occupies a volume of space. The simple solution to this is to attach the node to more than one cell. The scene graph can easily handle this, as long as the correct nodes are attached and detached as the object moves around.

7 RESULTS

It has been proven by prior research that breaking up a model into cells and portals can reduce rendering loads by factors of 20% to 50% (Luebke, 1995). However, this is highly dependent on the model and viewing position. The results observed usually exceed this culling factor. Backface culling removes approximately 50% of all geometry before rendering even starts. Then, any geometry behind the viewer is instantly culled, which usually removes a further 50% of all visible geometry. Based on the field of view, even more geometry is culled by the view frustum. Because this view frustum narrows as it traverses through the portals, a large amount of geometry is quickly removed.

The example scene (ucb.sce) is based on the sixth floor of the UCB computer science building. It is one floor of the model used by (Funkhouser, 1996). However, the model was completely re-generated given only a rough estimation of the two-dimensional floor plan. This model was chosen because it contains many rooms.

The model is composed of the six cells listed in Table 1. Ideally, the model should be partitioned further. Each room should be a separate cell. This would provide significant savings if the rooms were highly furnished.

	Triangles	Vertices
Centre Hallway	2406	2334
Centre Room	5684	9246
Left Room	4160	6690
Right Room:	4168	6690
Top Hall:	840	838
Bottom Hall:	1188	1169
Total	18446	26967
Average	3074	4494

Table 1: The six cells

The entire furnished computer science building, as used by the UCB walkthrough team, is composed of 1.4 million polygons. That roughly equates to 7 million triangles. This model is therefore only 0.26% the size of their model. However, the cell and portal algorithm is highly scalable, and quickly culls large areas.

You may notice that some of the walls are paper-thin. This was due to my mediocre architectural skills, and a lack of time. The most important walls, however, do contain some depth. In wireframe mode, you might also notice some geometry off to the side of the model. It's just a doorframe.

Figure 11: The shaded UCB scene

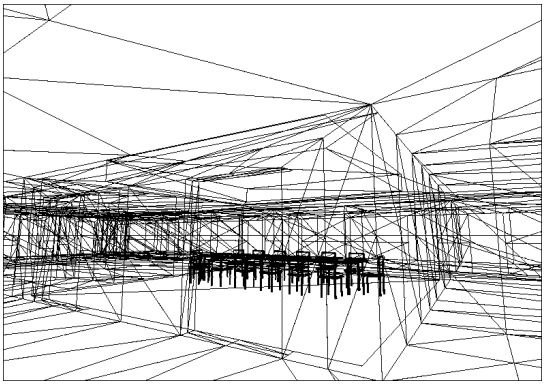


Figure 12: The wire-frame UCB scene

Both the model and all the furniture were created using 3D Studio Max, and exported as an ASE (Ascii Scene Export) file. A custom program then converted this data to the scene format used by this architectural walkthrough system. All portals were then manually added to the scene.

In wire-frame mode, the culling process is clearly visible. The centre room provides the most striking example of this. The first image (Figure 13) shows the room when the portal to the hallway is not visible. The second image (Figure 14) shows what happens when the portal becomes visible. Both the room and the hall geometry are now rendered.

Figure 13: A lot of the geometry has been culled away

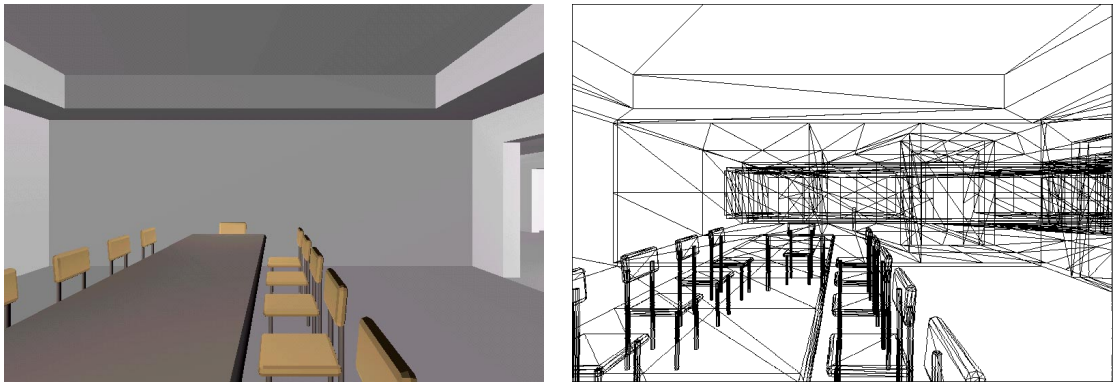


Figure 14: Once the doorway is visible, the hidden geometry becomes visible

It is also interesting to note that there is still some hidden geometry in this second image. The third image taken from further to the right of the same room (Figure 15) reveals that there is more furniture in another room down the hall. The reason that this furniture was never visible earlier was due to the view frustum narrowing as it was cast through the portal leading out into the hall. This proves that the cell and portal system does an excellent job of culling out invisible geometry.

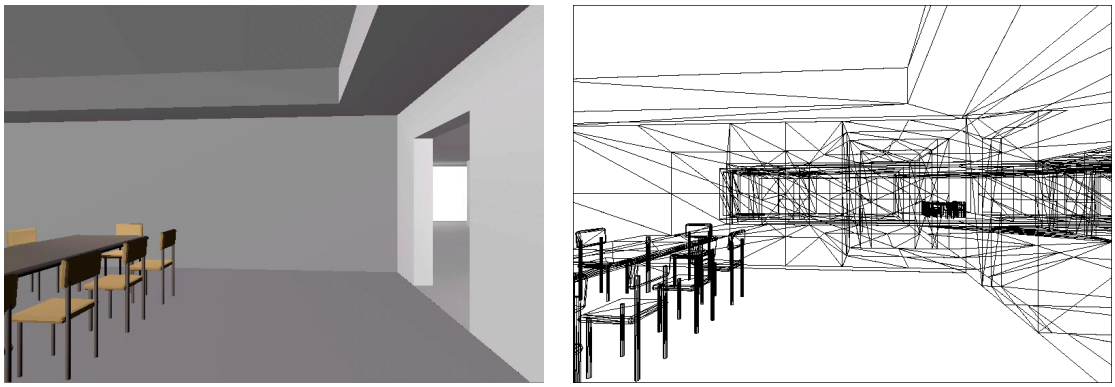


Figure 15: More hidden geometry is revealed

The UCB scene also contains an example of pre-calculated lighting. The following image (Figure 16) shows the shadow on the floor below the table and chairs. This shadow was pre-rendered using 3D Studio Max, and is being applied here as a second surface layer on top of the carpet. This shadow is only visible if the graphics accelerator supports multi-texturing.

Figure 16: Notice the pre-calculated shadows in this scene

8 CONCLUSION

The architectural walkthrough system described in this paper allows a hypothetical observer to interactively explore large architectural models. In addition, both the geometry and lights can be interactively modified at run-time. This is achieved by performing no visibility pre-processing. All visibility culling is performed at run-time and in real-time.

The system supports a realistic environment by allowing the user pre-compute expensive lights. Each surface is composed of multiple layers that contain textures and shadow-maps. Radiosity and raytracing methods can be used to compute these shadow-maps ahead of time. This significantly increases the realism of the environment.

No specialised hardware is required to make this system work. However, a cheap consumer graphics card that accelerates a graphics API such as OpenGL is a necessity.

To make this interactive walkthrough possible, the scene is broken up into cells and portals. A cyclic graph is used to maintain all the objects in the scene (cells, portals, lights, furniture, etc). The system also allows instances of objects to be placed in the scene, which significantly reduces the size of the model database.

The implementation proves that this cell and portal approach is very effective. In wire-frame mode, it is very clear that large amounts of geometry are simply culled away. Because the system is very scalable, this statement will hold true for architectural models composed of millions of triangles.

9 FUTURE WORK

9.1 Physics and Collision Detection

The most significant issue in the current implementation is the lack of collision detection. The viewer can simply march through a wall. This can have adverse effects on visibility culling. The system expects the viewer to be in one cell, when they have actually jumped through a wall into a nearby cell.

This collision detection problem directly leads to the viewer tracking issue. The system needs to track the viewer's movement so it knows which cell the viewer resides in. This is true for all nodes in the scene graph. The system needs to constantly know in which cell each node is located.

To implement this, the system needs to contain some sort of animation and physics process. This physics process needs to keep track of the movement of all nodes. When a node leaves a cell via a portal, the node needs to be detached from the cell it is leaving. When a node enters a cell via a portal, the node needs to be attached to the cell it is entering. This movement needs to be updated independently of the current frame rate. That way, an object's velocity does not depend on the speed of the machine.

9.2 Interactive Modelling

The current system fully supports a dynamic environment. However, it does not make use of this feature. An interesting idea would be to allow the user to change the model in the 3D view. The user could simply click the object that they wish to change, and manipulate it in three dimensions. Materials and textures could be applied interactively. Geometry could be changed by dragging vertices around. New rooms, doors and windows could be interactively stuck together. Lighting parameters could be changed while getting instant feedback.

Interactive modelling is a new area, and its potential has not yet been realised. However, with graphics hardware becoming a consumer product, many 3D modelling programs, such as 3D Studio Max, have already jumped on the realtime editing bandwagon.

10 REFERENCES

Aho A. V. and Ullman J. D (1992)
Fondations of Computer Science.
W. H. Freeman and Company, New York, NY.

Airey J. M. and Rohlf J. H. and Brooks F. P. Jr. (1990)
Towards image realism with interactive update rates in complex virtual building environments.
Computer Graphics (Symposium on Interactive 3D graphics), vol 24, no 2, pp 41-50.

Brooks F. (1986)
Walkthrough: A dynamic graphics system for simulating virtual buildings.
ACM Symposium on Interactive 3D graphics, Chapel Hill, NC.

Fuchs H. and Kedem Z. M. and Naylor B. F. (1980)
On visible surface generation by a priori tree structures.
Computer Graphics (ACM SIGGRAPH '80 Proceedings), vol 14, no 3, pp 124-133.

Funkhouser, T. and Teller S. and Sequin C. and Khorramabadi D. (1996)
The UC Berkeley System for Interactive Visualization of Large Architectural Models.
Presence, vol 5, no 1, pp 13-44.

Goldschmidt, D. E. (1998)
Design and Implementation of a Generic Graph Container in Java.
Masters Thesis, available at http://www.cs.rpi.edu/projects/pb/jgb/.

ID Software. (1999)
Optimizing OpenGL drivers for Quake3.
Available at http://www.quake3arena.com/news/glopt.html.

Jones C. B. (1971)
A new approach to the 'hidden line' problem.
The Computer Journal, vol 14, no 3, pp 232.

Luebke D. and Georges C. (1995)
Portals and mirrors: simple, fast evaluation of potentially visible sets.
ACM Symposium on Interactive 3D Graphics, ACM, New York, NY, pp 105-106.

nVidia. (1999)
Riva TNT2: High-Performance 128-bit TwiN Texel 3D Processor.
Available at http://www.nvidia.com/Products.nsf/htmlmedia/riva_tnt2.html.

Sweeney T. (1998)
Unreal technology features.
Unreal Technology, available at http://unreal.epicgames.com/.

Telea, Alexandru (1997)
A General-Purpose Run-Time Type Information System for C++.
Available at http://www.win.tue.nl/math/an/alext/ALEX/C++/RTTI_DOC/rtti_doc.html.

Teller S. and Séquin C. H. (1991)
Visibility processing for interactive walkthroughs.
Computer Graphics (ACM SIGGRAPH '91 Proceedings), vol 25, no 4, pp 61-69.

Tschirren, B. (1999)
Project Proposal: Realtime Walkthroughs of Massive Architectural Models.
Not published.

Walkthrough Project, The.
Available at http://www.cs.unc.edu/~walk/.

11 APPENDICES

11.1 Appendix A: Table of Figures

11.2 Appendix B: Source Code and Documentation

A pre-compiled win32 executable, source code and HTML documentation is available as ZIP archives from the following URL:

http://www.computing.edu.au/~tschirre/pub/project/

The HTML documentation is also available on-line at the following URL:

http://www.computing.edu.au/~tschirre/pub/project/doc/html/

11.3 Appendix C: Program Keyboard Control

The architectural walkthrough program has the following keyboard controls:

Left Arraw:	Rotate Left
Right Arrow:	Rotate Right
Up Arrow:	Move Forwards
Down Arraow:	Move backwards

11.4 Appendix D: Program Initialisation File

The program requres an initialisation file called “project.ini” to reside in the current directory. This file contains the following optional settings:

Settings (Defaults Shown)	Description
Application.width = 800 Application.height = 600	The size of the main application window.
Viewer.node = "viewer"	The name of the viewer node. This usually points to a camera.

Viewer.velocity = 5.0	The number of units the viewer moves when the up and down arrows are pressed.
Viewer.light = false	Attach an omnidirectional light to the viewer. As the viewer walks around, surrounding geometry is illuminated.
Render.fovY = 60.0	The vertical field of view.
Render.aspect = 1.0	The horizontal aspect ratio.
Render.zNear = 1.0	The near clipping plane.
Render.zFar = 1000.0	The far clipping plane.
OpenglRenderDevice.texturing = true	Enable texture mapping. If a software implementation is used to render the scenes, this setting should be changed to false.
OpenglRenderDevice.multitexturing = true	Enable multi-texturing if the hardware supports it. This enables multiple surface layers.
OpenglRenderDevice.mipmap = true	Enable mipmapping.
OpenglRenderDevice.minFilter = true	Perform linear filtering on the texture.
OpenglRenderDevice.minFilterMipmap = true	Perform linear filtering on the mipmaped texture. You might have to set this to false.
OpenglRenderDevice.magFilter = true	Enable bilinear filtering.