Institutionen för systemteknik
Department of Electrical Engineering

Examensarbete

A portal based system for indoor environs

Examensarbete utfört i datorgrafik
vid Tekniska högskolan i Linköping
av

Lars Abrahamsson

LiTH-ISY-EX--06/3703--SE

Linköping 2006

Linköpings universitet
TEKNISKA HÖGSKOLAN

Department of Electrical Engineering
Linköpings universitet
SE-581 83 Linköping, Sweden

Linköpings tekniska högskola
Linköpings universitet
581 83 Linköping

A portal based system for indoor environs

Examensarbete utfört i datorgrafik
vid Tekniska högskolan i Linköping
av

Lars Abrahamsson

LiTH-ISY-EX--06/3703--SE

Handledare: **Ingemar Ragnemalm**
ISY, Linköpings universitet

Examinator: **Imgemar Ragnemalm**
ISY, Linköpings universitet

Linköping, 19 June, 2006

Avdelning, Institution
Division, Department

Division of Computer Engineering
Department of Electrical Engineering
Linköpings universitet
SE-581 83 Linköping, Sweden

Datum
Date

2006-06-19

Språk
Language

☐ Svenska/Swedish
☒ Engelska/English
☐ _____

Rapporttyp
Report category

☐ Licentiatavhandling
☒ Examensarbete
☐ C-uppsats
☐ D-uppsats
☐ Övrig rapport
☐ _____

URL för elektronisk version

http://www.da.isy.liu.se
http://www.ep.liu.se/2006/3703

ISBN
—

ISRN
LiTH-ISY-EX--06/3703--SE

Serietitel och serienummer **ISSN**
Title of series, numbering —

Titel
Title

Ett portalbaserat system för inomhusmiljöer

A portal based system for indoor environs

Författare
Author

Lars Abrahamsson

Sammanfattning
Abstract

The purpose of this thesis is to document the development of the graphics part of an extremely pluggable game engine/lab environment for a course in advanced game programming. This thesis is one out of five, and concerns indoor, realtime computer 3D graphics. It covers state-of-the-art techniques such as GLSL – the OpenGL Shading Language - and more well known techniques such as portal based rendering.

Nyckelord
Keywords

GLSL, portal rendering

Abstract

The purpose of this thesis is to document the development of the graphics part of an extremely pluggable game engine/lab environment for a course in advanced game programming. This thesis is one out of five, and concerns indoor, realtime computer 3D graphics. It covers state-of-the-art techniques such as GLSL – the OpenGL Shading Language - and more well known techniques such as portal based rendering.

Acknowledgments

I would like to thank Ingemar Ragnemalm, for providing useful input during the course of this thesis, and my coworkers Per, Björn, Johan and Jolt.

Contents

Chapter 1

Preface

This thesis assumes the reader to have some knowledge in graphics programming. The thesis focuses on OpenGL, but anyone with some experience in 3D graphics programming should be able to comprehend the topics discussed. Since 3D graphics programming is rather mathematical, some understanding of linear algebra is assumed, foremost the concept of and operations performed on vectors and vertices.

Chapter 2

Introduction

Designing a computer game is a task that has many facets. One is to decide what kind of game is to be made, what game mechanics it will sport, its purpose and goals. If the game is to be played by many players at once, one part is to decide how clients communicate, and how relevant data is to be distributed between clients and – if applicable – the server. Yet another is to decide what kind of physics engine is needed. Is the goal to achieve physics as close as possible to real life, or can a simpler model be used? In case the game supports network playing, a way for the physics engine to decide what is right and wrong – although there is no absolute truth to this – is needed. If AI is to be used, what mechanics should be available? Do bots run as regular clients, or do they reside on the server (given that there is one)? And then there is the graphics engine. What kind of scenes are to be rendered? Will the game mainly be first or third person? What kind of hardware is available?

When developing a game from scratch, much time is spent on things like how to load and render models, implementing a network communication layer, writing code to intersect various objects etc. This is not necessarily bad, because those things are needed. However, if the purpose is to make a game, it would be good if those things were already taken care of. Such a collection of utilities designed to ease the process of developing a game is usually called a game engine. If we roll back a couple of years, most game developers made their own engines when creating a game. In a way this is good, because it will probably lead to an engine streamlined for its intended purposes, and the creators of the game will feel at home with the engine. But as mentioned, it also takes a lot of time. Sometimes in recent years, games have been developed upon existing engines. Heavy Metal: F.A.K.K.[2], using the Quake III engine, is an example thereof. (It would seem – perhaps – that using a game engine designed for a first-person game would be a poor choice for a third-person game, but the game was loads of fun, and played well in this author's opinion.) However, an engine like this has to be a lot more general and customizable than an engine written for one single target game.

This report covers the development of such an engine, or rather the parts of the engine that concern graphics. While perhaps not as competent as the Quake III engine, the goal when developing it – to be used as a lab environment in a course in advanced game programming - has been to achieve a high degree of freedom as well as to ease the burden on the programmers taking the course. Over a hundred man weeks have been spent on creating this engine, to be compared with the few weeks the entire lab series is to take.

Chapter 3

Introduction to the whole system

This section will serve to give the reader an understanding of what the system as a whole looks like, how it works and how the respective parts are connected. For a more in-depth view of the other parts of the system, please refer to the actual theses.

The system is divided into five parts:

- AI
- Graphics
- Networking
- Physics
- Server

The reason for this division is that these are the five areas that a typical server-based multi-player game engine has. For single play the network and server parts can be skipped, but for network games they are paramount.

The original grouping was as can be seen in the above figure. However, the actual design turned out to be somewhat different.

3.1 Data representation

When designing a multi-user system, there are a lot of things to keep in mind. Maybe foremost so the data representation, and manipulation of said data. Consider the intended usage of the system, a multitude of users all connected, acting and interacting on the information the server sends them. Who is in the right? If

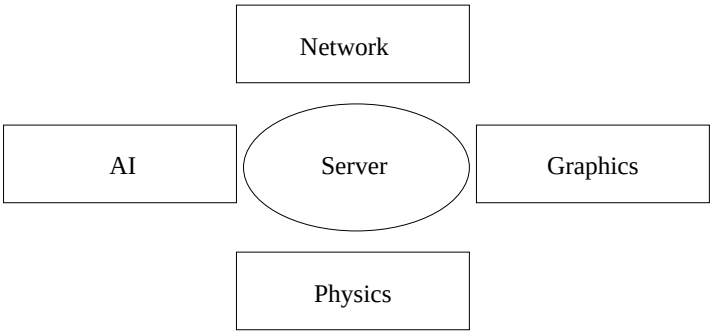

Figure 3.1. The five parts of the system.

agent A throws a marshmallow pie at agent B in the same instant that agent B strafes, does the pie hit, or did agent B successfully dodge the attack?

From A's point of view, the pie probably hit the target full on. The agent has to assume that B is either still or use some sort of heuristic to decide where B is at. A thrown pie usually hits its target quite fast, so it's quite probable that the pie reaches B in A's local state before the strafe message can reach from agent B to the server and on to agent A.

In B's point of view though, the strafe action is performed immediately after running in to the armed and dangerous agent A. When the message that A throws a pie reaches B, it's likely B has already hidden behind a pillar, or moved out of the doorway.

This problem has no *right* or *correct* solution. Every agent connected to the system is acting on old data. There are several workarounds for the above problem (as well as several more related problems). If things are to be done in real-time, there will always be errors. The approaches to handle those problems differ with the intended functionality of the system. In an FPS, for example, it might be wiser to use a network of interconnected clients, and no server at all. In some MMORPG's – such as World of Warcraft – the data handling is simplified for example by ignoring collisions between interactive objects such as player-player or player-beast, and thus let the physics reside on the client, since all clients will have the same world representation anyways. If it is not crucial that actions be performed immediately – such as in Starcraft – they can be tagged to be performed some time in the future, and the server can confirm or deny the action all in good time.

Since it was decided from the start that we were to create a server-based system, some decisions had already been made for us. We decided to opt for a solution where the server has complete and absolute rule over the data, and to let all manipulation of the data be done on the server. The clients simply send suggestions to the server – though of course the actual clients are free to use prediction not to

look like complete fools all of the time.

3.1.1 Actual system overview

Since the AI and physics parts of the system are highly dependent on a correct world state we decided to put them on the server. The networking part allows the server to communicate with the clients, whose sole responsibility is to display their views of the world - possibly with some sort of prediction heuristic – and to communicate whatever actions the agent wants to perform to the server.

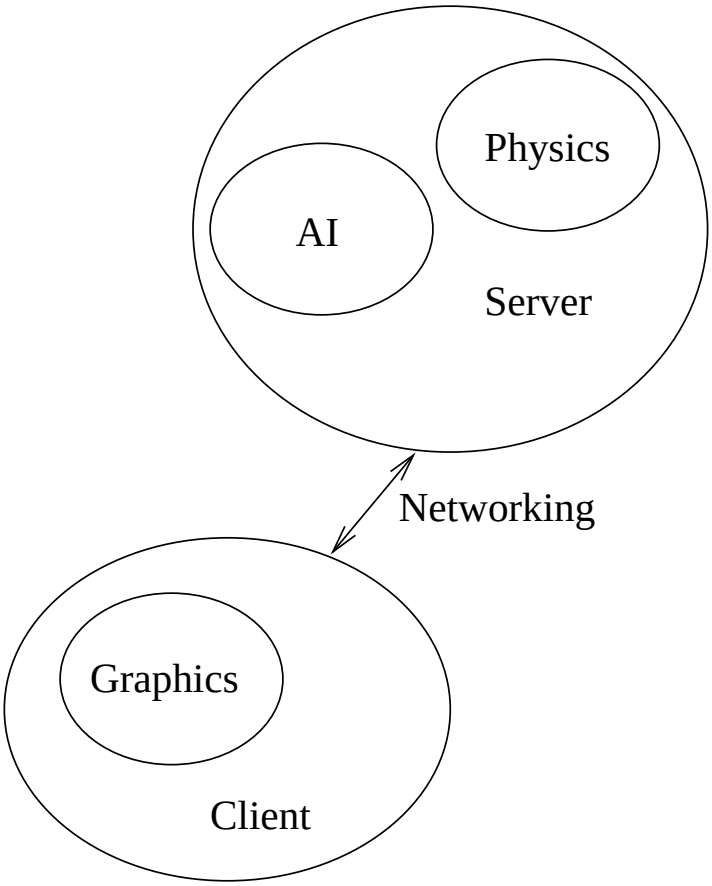

Figure 3.2. A schematic overview of the actual system

3.1.2 Object types

We have chosen to divide the objects making up our realm into two types: static and dynamic objects. Static objects are completely rigid. They never move, change shape or disappear, and are not affected by physics. Dynamic objects can be moved and/or reshaped, removed and added. Rooms – the basic building blocks of our world - are static objects. Agents – both human and AI controlled – are dynamic objects.

3.2 Physics

With the previous section in mind, it should be rather natural that the physics run on the server. After all, the server is the one with the complete and correct world state. One could do it the WoW way, and let the clients handle physics between the agent and the static world [6], but since we handle client-to-client physics as well this would mean that the physics would be run on two places – both server and client - and the roundtime would still be a factor, albeit less often. Collision handling between dynamic objects is done using bounding volumes, while actual per-face physics is used for object-world collisions.

3.3 AI

The AI part of the system resides on the server machine. Bots are written using Python, and the server can provide a complete world state. Since the actual implementation of bots is a task left for the laborants, the AI part has focused mainly on providing useful utilities. In addition, some time has been spent on creating monitoring clients, which are good for debugging.

One important task is to fully exploit the characteristics of the world representation. Since the world is portal based, the AI module can use the interconnections of the rooms of the world as a graph, where the portals become edges. Agents can then use this graph for example to navigate the world.

3.4 Networking

In order for the server to communicate with the clients, a networking layer is needed. This layer is responsible for transmitting and receiving data between the server and a client. Thus, the networking part reside on both client and server. The networking layer is also responsible for encoding the data in a way that can be easily comprehended, and to make sure that no messages are lost. It is also possible to add prediction heuristics to the client part of the networking layer, in order to reduce lag due to network delays.

Chapter 4

Background

When implementing a real-time rendering engine, the main problem lies in finding a working compromise between performance and detail. On one hand, we want to produce as snazzy and breathtaking scenes as possible, while on the other we want to maintain an acceptable frame rate. This – of course – is a conflict in interest, since producing the stunning views we strive for usually requires a high level of detail – i.e. lots and lots of polygons – and probably even fancy per-fragment shading, which invariably leads to a decrease in frame rate. To deal with this problem, a plethora of cunning ways to reduce the burden of rendering – usually by trying to decrease the number of rendered polygons per frame – has been thought up. A subset of these techniques are covered here.

There is also the issue of how to produce the fancy features more or less required in today's games. Some have been around for a long time, others are newer. Covered here is shader programming, a relatively new player on the field, but one that leaves us with new exhilarating ways to manipulate what used to be fixed functionality in the rendering pipeline. Also covered is animation, which has been around for quite some time, but is absolutely crucial for any but the most simple of applications.

4.1 Animation and skinning

The visual part of a computer game usually boils down to a set of polygons. These polygons can be divided into units. The actual units differ with the type of game. An outdoor scenery would probably have a terrain mesh, while indoor scenes consist of rooms. In addition to these, props such as chairs, lamps, rocks, palm trees and characters are common objects. While for some types of objects, remaining completely fixed as originally modeled is fine – rooms and rocks are examples of these – others would look very unrealistic if not animated. Thus, a

means for animating these objects is needed, be it a character running, jumping and pulling levers or the leaves of a palm tree gently swaying in the evening breeze.

Animation is crucial for the feel of the game. With poor animation, the game will look cheap, and the overall impression will be spoiled. There are numerous methods to perform animation, some of them covered in this section.

4.1.1 Keyframe animation

An efficient way to perform animation in real-time is to use keyframe animation. The model is broken down into components – bones – and each keyframe is a snapshot, storing the position and orientation for every (affected) bone. An animation then consists of a set of keyframes, all of which hold a relative time representing the progress of the animation. These keyframes are usually interpolated between, so that the animation appears seamless. The simplest way of doing this is using linear interpolation. However, sometimes this is not good enough to achieve smooth animation, and more advanced interpolation techniques such as spline interpolation have to be used, with an increased cost in computation time. [2], [9]

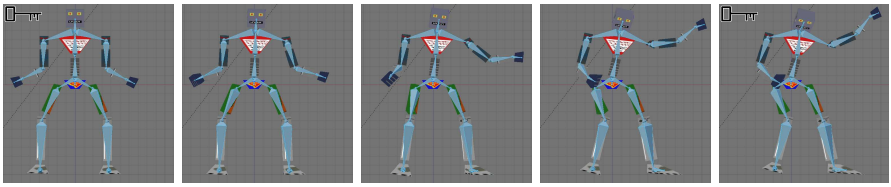

Figure 4.1. Keyframe animation: Interpolation between two keyframes (leftmost and rightmost)

An alternative to using complex interpolation algorithms can be increasing the number of keyframes. An idea mentioned in Sánches [9] is to create a lot of keyframes, and then use some sort of analyzing tool to determine which ones might be discarded with little or no impact on the result.

4.1.2 Tagged animation

Tagged animation is a technique thought up by the people at *id Software* for their game *Quake III*. It was introduced to solve the problem of combining animations affecting different body parts, such as running and shooting with a specific weapon at the same time, but have other benefits as well. The idea is to split down the model into several parts – such as the body parts head, torso and legs on a humanoid model. These parts have their own animation cycles, which means they can perform actions regardless of each other. The term *tagged* stems from the need of pivot points – *tags* – that determine where the body parts are to be attached. [9]

4.1.3 Skinning

One problem when dealing with keyframe animation is what to do with the joints. With rigid body transforms and joints modeled by connecting separate body parts – for example the elbow of an arm modeled as a forearm and an upper arm overlapping – the joints will not look realistic because they consist of overlapping parts. We need a way to model the joint as one mesh, that is somehow affected by the skeleton. One technique to do just this is called skinning. It consists of letting every vertex in the model – the skin – be affected by one or several transformations at once – these transformations corresponding to the bones of the model. [1], [2]

While skinning looks good, it is computationally expensive. However, the advent of hardware vertex shaders have given us a means to do skinning on the GPU, something that – according to Futuremark Corporation [10] – is several times faster than on the CPU.

4.2 Clipping, culling and occlusion detection

Clipping, culling and occlusion detection techniques are used to discard as much data as possible in as early a stage as possible in the graphics rendering pipeline, in order to increase performance. A few of these techniques are described in this section.

4.2.1 Back-face culling

Back-face culling is one of the most common, most effective and overall cheapest ways to reduce unnecessary drawing. The principle is as follows. Most – if not all – faces of any graphic primitive make sense only if viewed from one of its two sides. In average, only half of the total amount of faces will have this side turned towards the viewpoint at any given time, and thus drawing only the faces facing the *correct* way will reduce the amount of drawn faces with roughly 50%. In order for this approach to work, a way to determine for any given face which way is correct is needed. This is generally done by defining the faces' vertices in a special order, usually counter-clockwise. With this method, the (signed) normal of any given face can be computed cheaply, and the angle between this vector and the one emanating from the viewpoint will yield whether the face is to be drawn or not. This method is widely supported by graphics APIs, and need usually not be implemented, but simply enabled. [9]

4.2.2 Object-level viewport clipping

Viewport clipping is the process of not drawing what falls completely outside the viewport. Most 3D APIs provide per-face viewport clipping, so at first using this

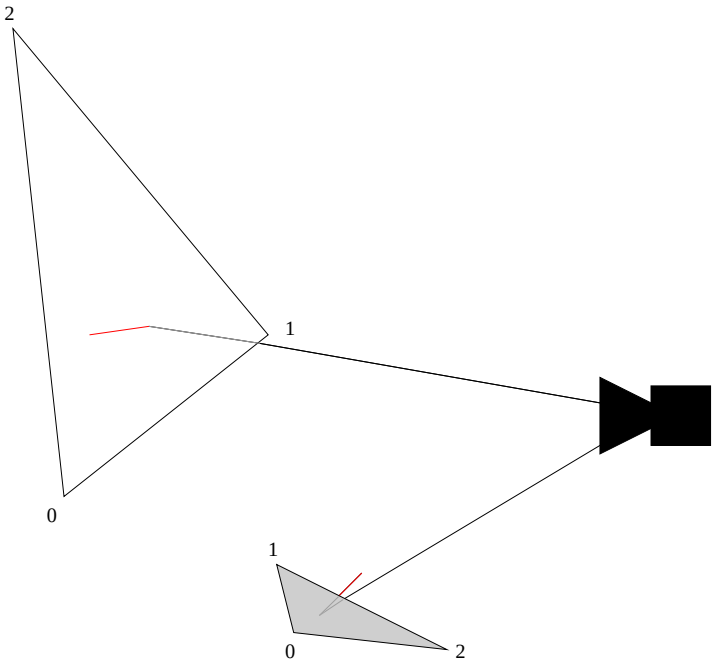

Figure 4.2. Back-face culling: a triangle is drawn if the angle between the view vector and the (signed) face normal is less than 90^o. The shaded triangle is drawn, while the unshaded triangle is culled.

same approach on a per-object level might seem redundant. This, however, is not the case. Passing geometry data to the GPU is not free, since it goes over a slow data bus. Also, there is no point in doing transformations on faces that are not to be drawn. Thus, if viewport clipping could be done cheapish for complex objects, it would prove beneficial to performance.

Several approaches to do just this exist. One that is commonly used clips the object's bounding box against the viewport. If the bounding box intersects or falls inside the viewport, the object is drawn. It is otherwise discarded. Clipping against a bounding box is cheap, but might sometimes lead to drawing an object that is in fact invisible from the viewpoint – a trade-off that is usually accepted. A similar approach is to use the object's bounding sphere, which will result in a cheaper test that is not affected by object rotation, but one that will (generally) yield more false positives, since boxes can usually be fitted tighter around an object. [9]

4.2.3 Hardware occlusion testing

Since complete ordering of objects is usually too complex and/or expensive to achieve in real-time, a depth buffer is normally used when rendering, so that when

two or more objects occupy the same screen-space, for each pixel only the closest object is seen. While this is a good solution, there is one underlying problem. In some cases the rendering algorithm does not know which object will be visible in the end, and thus processes several objects occupying the same screen-space. In effect, this means drawing something that is later occluded by something closer to the viewing point, and is called overdrawing – a deadly sin [9]. The remedy to this problem is occlusion detection. While different viewport clipping algorithms exist that cull objects that cannot possibly be seen from the camera because they end up behind or outside the viewing cone, this kind of occlusion testing tries to decide which objects inside the viewing frustum can be safely removed because something closer to the camera completely covers them.

Many modern video cards support occlusion detection in hardware, by checking shapes against the aforementioned depth buffer. By activating occlusion query mode and sending whatever geometry is to be tested to the GPU, the geometry itself will not be drawn, but instead the number of modified pixels – if the geometry *was* to be drawn – will be returned. This approach will not pay off when testing for occlusion on a per-face level. It will, however, be useful if lots of triangles can be grouped in a container consisting of relatively few faces, such as a bounding box. Just as when performing viewport clipping against a bounding box, the result of an occlusion query using an object's bounding box might yield a false positive, but the test will usually be able to prune complex objects with a cost less than drawing a box, which makes it highly useful. [9]

4.2.4 BSP trees

The BSP Tree FAQ at GameDev.net [3] states that

> A Binary Space Partitioning (BSP) tree is a data structure that represents a recursive, hierarchical subdivision of n-dimensional space into convex subspaces. BSP tree construction is a process which takes a subspace and partitions it by any "hyperplane" that intersects the interior of that subspace. The result is two new subspaces that can be further partitioned by recursive application of the method.

[3]

In three dimensions this corresponds to storing polygons separated by planes. BSP trees are good because they are extremely versatile, and can be used in many aspects of graphics programming, such as hidden surface removal. The drawback is that BSP trees in its original form cannot handle dynamic (i.e. changing) scenes.

A BSP tree can be constructed using the following steps:

- Select a partition plane.
- Partition the set of polygons with the plane.
- Recurse with each of the two new sets.

Rendering using a BSP tree is either done using *Painter's algorithm* – by starting at the back and drawing polygons in order later drawn objects cover the previously drawn, much like when painting in oil – or a front-to-back scanline approach. Either way, the key to rendering using a BSP tree is tree traversal. By starting at the root node and doing a depth-first tree traversal the polygons are visited in order of distance from the camera. By including the viewing direction and not just the view point in the traversal, entire subtrees behind the camera can be pruned.

The versatility of BSP trees is not limited strictly to graphics. A BSP tree can for example be used to perform collision detection, by sampling the motion in a series of Euler steps.

The worst case spatial complexity of an auto partitioned BSP tree is $O(n^2)$. The expected case however is $O(n)$. The rendering time complexity of a BSP follows space complexity – i.e. $O(n^2)$ in worst case, but $O(n)$ expected.

[3]

4.3 Portals

Another solution to the problem of overdraw is to introduce the concept of portals. This method is mainly used on indoor scenes, and a variety of approaches exist.

An indoor scene can be seen as a set of rooms, furnished with objects, and connected to each other by openings in the walls. We will call these openings portals. A natural approach to drawing such a scene would be starting in the room containing the active viewpoint, and then traverse the connected rooms through the portals, just as we would in real life. This method is called portal rendering. Given a scene, we start by drawing the room holding the camera, along with any objects residing in that room visible from the camera (decided for example by viewport clipping). When this is done, for each visible portal connected to the room, we draw the room it links to, and its objects. This traversal is recursed upon, until all rooms (and their corresponding objects) visible from the viewpoint are drawn. [9], [7], [1]

One important thing when doing portal based rendering is that whatever falls outside of the portal frustum (the frustum emanating from the camera, passing through the portal) will not be seen. Thus, rendering such objects would be a waste of valuable time. Several methods exist to use portals as clipping volumes. The one described here was introduced by Luebke and Georges [7], and is a very efficient yet simple method.

For each portal in the first room, we project its vertices onto the camera (or screen) plane. From these, we compute the screen-aligned bounding rectangle, which is used to create our portal frustum. The diligent reader will notice that the resulting portal frustum will be potentially bigger than the actual visible volume. While

this is true, the intersection of consequent portal frustums – created in the same manner – will be extremely cheap, and this simplification will pay off.

It should be noted that this approach will work for portals of any shape, as opposed to many other approaches, where restrictions such as convexity or four-sidedness might exist. [7]

4.4 Shader programming

Rost states that *the recent trend in graphics hardware has been to replace fixed functionality with programmability in areas that have grown exceedingly complex.* [5] This is the purpose of shader programming. What used to be fixed functionality, controllable to some extent by calls that alter the OpenGL state, has been replaced with programmable units called shaders. There are two blends – vertex and fragment shaders – that each replace a specific part of the OpenGL rendering pipeline – the *per-vertex operations* and *fragment processing* parts respectively. With this extra flexibility, the programmer is given the ability to perform operations previously not available through the OpenGL rendering pipeline, as well as a means to fine tune "standard" functionality better to fit the current need. Shader programming gives us the ability to do things that could not previously be done – at least not in real time – and moves crucial parts of the processing from the general purpose CPU to dedicated processing units on the graphics hardware. Increased performance and functionality for the win. [5]

4.4.1 T&L – vertex processing in OpenGL

Transformation and lighting – T&L for short – is the bulk of the vertex processing part of the fixed rendering pipeline in OpenGL. Its mission is to transform incoming vertices – along with complementary data such as normals, light specs and texture coordinates – according to the modelview and projection matrices. As mentioned this process has fixed functionality, although it can be modified to some extent by calls that modify the OpenGL state, such as turning on and off lighting, changing lighting and material properties and of course by modifying the modelview matrix. [5]

4.4.2 Vertex shaders

A custom vertex shader is supposed to replace the fixed-functionality vertex processing present in the OpenGL rendering pipeline, and may alter and extend this functionality according to current needs. However, the basic functionality still remains. A vertex passes through vertex processing and is translated and colored according to the desired functionality. To summarize, the vertex processor is intended to perform (among other) the following: [5]

- Vertex transformation
- Normal transformation and normalization
- Texture coordinate generation
- Texture coordinate transformation
- Lighting
- Color material application

So what are the gains of implementing a custom vertex processor? First off, we are now able to closer control the transformed position of single vertices. This can be used for instance to perform tweening of geomipmapped terrain, do displacement mapping or keyframe interpolation. For example, new versions of Cal3d allow the programmer to replace the mixer with one that runs in the vertex shader, something that can reduce the workload of the CPU tremendously.

Secondly, we have direct control of the light and color parameters of each vertex, which gives us increased flexibility in lighting and coloring.

4.4.3 What is a fragment?

After vertices have passed through T&L they pass primitive assembly, where they are combined into primitives. After that comes primitive processing, where the primitives are clipped against the view volume and any clipping planes, as well as being perspectively transformed and culled. The next step is called rasterization, and this is where fragments occur. The rasterization stage is responsible for decomposing the primitives into smaller units that correspond to pixels in the destination frame buffer – *fragments*. Each fragment holds a window coordinate, and associated values such as color and depth information and texture coordinates. The values of each fragment are interpolated from the values of the vertices from which they stem. To cite Rost, *fragments are per-pixel data structures that are created by the rasterization of graphics primitives.* [5]

4.4.4 Fragment processing in OpenGL

The fixed-functionality fragment processing stage of the OpenGL rendering pipeline is responsible for providing each fragment with a color. The main part of this is done via texturing. The fixed functionality texturing of OpenGL is way complicated. There are four basic types of texture maps: one-dimensional, two-dimensional, three-dimensional and cube maps. The first three should be obvious, but the cube map may require some explanation. It consists of six 2D textures, one for each major axis direction. In conjunction with some OpenGL extensions, cube maps can be used for example to normalize vectors, by storing the normalized vector value as RGB data and use the original vector as texture coordinate input, clamped to the corresponding side of the cube map. Recent versions of OpenGL

support multitexturing, where several textures are combined serially to give the value for the fragment.

4.4.5 Fragment shaders

A fragment shader replaces the fixed fragment processing part of the OpenGL rendering pipeline. It can be written to mimic the fixed functionality, but also to behave in different ways. It does not replace any parts of the rendering pipeline that deals with several fragments at a time, and may not alter the position of the fragment. Nor does it replace operations such as coverage, pixel ownership test, scissor, stipple, alpha test, depth test, stencil test, alpha blending, logical operations, dithering and plane masking. [5] An ordinary fragment processor may perform the following tasks: [5]

- Texture access
- Texture application
- Fog
- Color sum

One of the key advantages in fragment shaders is the ability to access arbitrary texture memory, and act on the result. This allows *dependent texture read*, which in turn allows ray-casting algorithms to be implemented in a fragment shader. [5]

Fragment shaders allow us to do per-fragment lighting. This is beneficial for example when rendering objects consisting of few polygons and wanting to achieve specular lighting. While the fixed OpenGL lighting would interpolate the light parameters between vertices – often yielding extremely poor specular lighting – a custom fragment shader can perform the specular light equation for each fragment, resulting in a much more realistic specular effect.

Procedural textures can be created using a fragment shader. A procedural texture is a functional representation of a texture, to be compared to a regular static texture, which is just a buffer holding color values. There are several advantages in using procedural textures. Procedural textures don't have a "scale", which means that they will look good whatever the zoom and angle of the surface on which they are applied, thus no mipmapping is needed. A competently written procedural texture can often be parameterized, allowing the same code to output different results depending on the parameters passed to the shader. This is hard to achieve with regular textures. Memory usage is also much lower when using procedural textures. But what about the drawbacks? It's usually harder to write a procedural texture than to generate a regular texture image. Since the color values are computed, it may result in an overhead in time compared to the lookups of static texture data. Sometimes – mainly when the algorithm is relying on noise or random values – the results may differ on different driver implementation and hardware. [5]

4.4.6 A minimal GLSL shader

The following code is a minimal diffuse shader, written in GLSL. It uses the first OpenGL light source for positional lighting calculations, and a hardcoded material color (red).

```
varying vec3 normal, lightDir;
void main()
{
  normal = normalize(gl_NormalMatrix*gl_Normal);
  vec4 p = ftransform();
  lightDir = normalize(vec3(gl_LightSource[0].position) - p.xyz);
  gl_Position = p;
}
```

Source code listing 1: Minimal vertex shader – passes vertex normal and light direction to fragment shader, and transforms vertex in the same manner the fixed-functionality OpenGL rendering pipeline would.

```
varying vec3 normal, lightDir;
void main()
{
  const vec3 color = vec3(1,0,0);
  float intensity = clamp(dot(lightDir,normal), 0.0, 1.0);
  gl_FragColor = vec4(intensity*color,1);
}
```

Source code listing 2: Minimal fragment shader – calculates the diffuse lighting per fragment using the (interpolated) normalized face normal and light vector passed from the vertex shader with hardcoded (red) material color.

To summarize, in the most general sense shader programming has two purposes. One is to perform operations one cannot readily achieve with the fixed OpenGL rendering pipeline. The other is to ease the burden on the CPU by delegating appropriate tasks to dedicated GPUs.

4.5 Orientation

All objects in the world (possibly barring objects that will remain forever completely static) will need to store its position and orientation relative to the global origin in some way. There are several approaches to this, all with their own perks and drawbacks. I will briefly describe three ways of doing this, giving the interested reader a first glance, as well as an entry point for further reading. Mathworld – http://mathworld.wolfram.com – has lots of information on these topics, and can be warmly recommended.

4.5.1 Euler angles

This is probably the most straightforward representation. Three angles are stored, that represent axis-wise rotation around the respective base vectors of a coordinate system. There are however two problems with this representation. For a start, the order of rotation is important. Not a big deal perhaps, since it can be solved by always using the same order of rotation. More cumbersome is the possibility of gimbal lock, where two axes line up and result in a loss of a degree of freedom.

4.5.2 Local base

Hackman defines a base for the vector space V as a linearly independent subset $M \subset V$ that spans V. [4] A base can be used to create a transformation matrix – in fact, what we have *is* the transformation matrix. This is good, since whatever the representation, the transformation matrix will always be needed during rendering. To boot, this representation have the base vectors, that often come in handy when manipulating the object. This makes the local base a good candidate for storing orientation. By letting each object store its own local base, objects are manipulated through manipulating their respective bases.

4.5.3 Quaternions

Quaternions are quite mathematical, so let it just suffice to say that a quaternion can be represented as a vector and a scalar, and that a unit quaternion represents an orientation. It should also be mentioned that rotary interpolation between quaternions is less cumbersome than interpolating between euler angles and bases.

4.6 Conclusion

Those are just a few ways to achieve faster and better looking graphics. The more ways one knows of how to improve the rendering process, the better the game will turn out. But it is also a matter of knowing when to use a certain technique. Advanced animation techniques such as skinning will probably only slow down a first-person flight simulator, and the processing power would be better spent on doing geomorphing – preferably in the vertex shader. The same goes for portal rendering, which is good for indoor scenes but more or less pointless outdoors. Know the techniques, and know when to use them.

Chapter 5

System overview

This section will focus on some different approaches to implementing an extendible graphics system, and explain why I have chosen one over the other. It will also show relevant parts of the system's class hierarchy, and explain some of its concepts.

Designing a system of this size is not an easy task. A decision that at first looked good might later turn out to have been poor, with a considerable impact on extensibility and generality.

In an effort to make the object hierarchies as usable and functional as possible, I have made extensive use of polymorphism and multiple inheritance. The main purpose is to introduce roles, aspects, facets or mixins (depending on terminology). One could even think of it as identity. To clarify: instead of creating a completely hierarchic inheritance structure, I have tried to identify aspects that can be of use for different object types. The main aspects I have used for my objects are:

- Placement – the aspect of having both a position and orientation
- Mesh – the aspect of having a mesh

This might not look like an impressive amount of aspects, but in fact just by recognizing these two roles, the burden of implementation is eased tremendously. In retrospect, the separation into aspects could have been done to a greater degree, which would have eased development further. Animation for example, would have been a good aspect. In the current design only dynamic objects can be animated, and in truth this is enough for our system, but with a separation of animatory properties into an aspect, the aspect of animation could be given to any kind of object.

One could argue that the current implementation does not need use the concept of aspects, since everything in it could probably be solved sufficiently with single inheritance. I however find it much easier to separate roles as aspects conceptually. I also claim that there are situations – probably not in the current system – where extending a system using aspects will prove to be a lot less cumbersome.

5.1 Placements

As mentioned one of my key aspects is that of placement, the purpose of said placement being to represent a position and orientation in three dimensions. Position is more or less straightforward, a vector holding the coordinates. More care is needed to achieve a sound representation of orientation, as mentioned earlier.

I chose to represent orientation using a local base. This is good because it doesn't suffer the drawbacks of Euler angles, makes the local axes always available to the object and makes it easy to provide pitch/yaw/roll functionality, which is a very intuitive way to manipulate objects. It is also good for transformations between bases, be it local to global, global to local or one local to another. The main reason though, is that the concept of a local base is much more clear to me than the funky mathematics of quaternions, though in truth they are probably the way to go if absolute optimal calculations of rotations are to be performed.

5.1.1 Usage

All objects that have both position and orientation use the placement aspect. This includes all kinds of graphic objects, as well as cameras. Spawn points and lamps do not keep an orientation (although spotlights could arguably do so, though I found it more logical to represent them as a subclass of lamps; spawn points ought really hold orientation so that objects could be spawned in an arbitrary way rather than global axis-aligned) and thus do not use the placement aspect. By letting the cameras and graphical objects use the same aspect for placement, they can be orientationally and positionally manipulated in exactly the same way, and even substitute each other. A camera is as much a placement as is a graphic object.

5.1.2 Functionality

A placement can be moved – either by arbitrary vectors or along any of its base vectors. It can be rotated – around its local origin, around an arbitrary point or through pitch/yaw/roll calls. In addition, placements can transform vertices and vectors from and to placement-local coordinates, and provide its base matrix.

5.2 Meshes

Meshes are stored in the Cal3d format. Cal3d uses the concept of submeshes to store different parts of a mesh that cannot from the renderers point of view be considered one mesh, either because the parts use different textures or because double vertices need be introduced to achieve different normals and/or texture coordinates. While this may sound a bit counterintuitive at first, it is actually a good thing. It gives the modeller the ability to model objects that use several

textures and normal schemes without having to split them down into several objects, which would soon give the programmer a headache. Ergo, meshes consist of a number of submeshes.

Internally, the mesh class aggregates another class, submesh, that represent a single submesh.

5.2.1 Usage

All objects that keep a mesh use the mesh class. In our world, this corresponds to all kinds of objects, rooms and portals.

5.2.2 Functionality

The only thing a mesh can do is draw itself, which internally consists of drawing all its submeshes.

5.3 Meta-objects

Because we have a client-server system, objects need exist on the server as well as on all clients. I am not going to describe in length how these representations are kept consistent: see the server thesis for that. I will however mention how we describe the additional attributes along with the Cal3d model in a way that is relevant for both server and client.

Because the server and client side of the system are interested in quite different parts of objects, they internally use their own hierarchies to represent them. If both server and client would have their own code to parse the files describing objects and worlds, both would need to change whenever the format does, which unfortunately is quite often, at least before things have stabilized. To solve this problem, we have introduced a concept we like to call meta-objects.

As we see it, a meta-object is an object that holds all descriptive data that can be read from file, as well as a means to read and write this data. Thus, when parsing a world file, a bunch of meta objects instantiate themselves, and are happy to provide anything of interest about themselves to anyone that is interested.

5.4 Object structure

There are mainly two kinds of objects the graphics part of the system is interested in. One is camera objects, and the other is objects that hold a mesh. There are of course other objects – such as light sources - that are of interest. However,

since they are hopefully quite straightforward to place in the system, they are not described in this part.

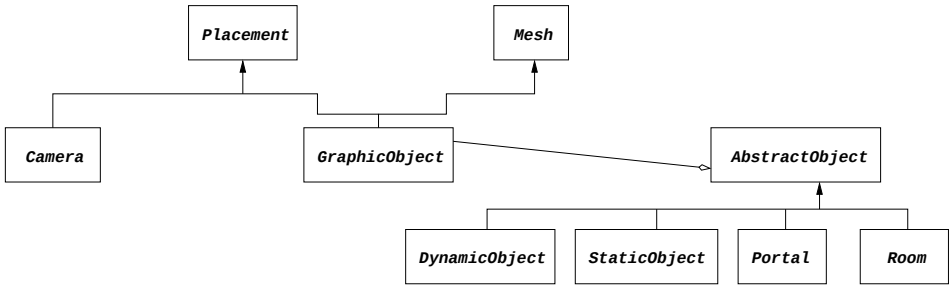

Figure 5.1. Class diagram: The relevant parts of the graphics object structure.

As can be seen in the class diagram, AbstractObject aggregates GraphicObject. This aggregation was introduced to support non-fixed rendering, which is needed for example when doing multi-pass rendering. The system does not currently support this for security reasons, but the object structure does not impair such support. In order to provide custom rendering, all one need to do is inherit the GraphicObject class and overload its draw method.

5.5 Cal3d

Cal3d is an open source skeletal based 3D character animation library. It is platform and graphic API independent, which makes it a good choice for any application. It has its own XML-based format, and exporters exist for Maya, 3D Studio and Blender. See http://cal3d.sourceforge.net for more information.

We have chosen to use Cal3d for animation support in our application, and are also using its data format to store rigid models.

5.5.1 How does it work?

Cal3d uses its own internal data format to achieve animation. This means that all modifications of the mesh need be done through the Cal3d API. This is somewhat limiting, but the API gives you the ability to manipulate models on a per-bone basis if needed. Animations can be blended through a mixer, which makes playing animations easy. Rendering is done by the implementer.

Cal3d distinguishes between core and instance classes. Each model type consists of a set of core classes. When creating the actual model instances, the core classes are the basis of creation of the instance classes that make up the instance model's state. There are – in the most general sense – four different classes: skeletons,

animations, meshes and materials. Skeletons are the basis of movement, and all manipulation of the model – direct or through animations – is done to the skeleton, which in turn modifies the meshes through the Cal3d pipeline. Animations are pre-modelled sequences of movements of the bones of a skeleton. Meshes make up the actual models and are bound to the bones of a skeleton, and the materials specify what the meshes will look like.

Cal3d can play any number of animations at the same time, and blend these together using a mixer. It distinguishes between two types of animations: cycles and actions. Cycles are animations designed to loop. They can be blended to the set of active animations, and given a weight of impact. Cycles continue to loop until cleared. Actions are one-time animations, that play for their entire and then stop.

Typical animation is done using these two animation types. However, Cal3d also allows the user to directly manipulate the bones of a skeleton, which will in turn affect the mesh. This is done by setting the rotation and translation of specific bones. For everyday usage, pre-modelled animation is to be preferred, but if one for example wants to allow the physics engine to manipulate models, this is the way to do it.

5.5.2 Caveats

Although Cal3d has been mostly pleasant to work with, there have been some problems. For instance, we have used Blender to model and create animations, which has sometimes been a bother because Blender and Cal3d does not work in the exact same way. Thus, a fair understanding of what is supported by Cal3d is needed to create working models. Another problem has been the exporter, which to our understanding is not created by the creators of Cal3d at all. While it has worked as expected most of the time, sometimes very odd things have happened. Whether this is due to bugs or just incompatible models is hard to know. Yet another problem has been the Cal3d API. Since Cal3d was in a state of development, sometimes changes were done that forced us to change our implementation. This will probably cease to be a problem once the API has matured. All in all, Cal3d is a good, easy to use straight-forward character animation library, and many applications could probably benefit from using it.

5.6 Portals and rooms

Presented previously in this chapter are the basic building blocks used to represent just about any kind of object that will ever exist in a world. Two types of objects however are more important than others, because they are a basic building block in the rendering process as well – portals and rooms. Both behave like static objects – i.e. they are rigid and not directly affected by physics – but they are also the actual world representation. The world consists of rooms interconnected through

portals, and everything else belongs to a room. The next chapter will cover how they are represented, and how worlds are generated using a custom world modeling tool.

5.7 Conclusion

By now the reader ought have some knowledge of the most important design decisions made in the process of implementing this system. While I feel it landed almost right from the start, some problems had to be solved quite late in the development. Most notably I guess, is the aggregation between the GraphicObject and AbstractObject classes, that did not exist from the beginning. This was introduced to support non-fixed rendering, and should have been there from the start. Luckily, it didn't cause too much problems, but with a bit more forethought it would never have been an issue at all. The gist is, think before you act.

Chapter 6

Implementation

This part will focus more specifically on key parts of the implementation of our system. It will show some of the problems that have occurred, and how they were solved. Here I will show how rooms and portals are connected into worlds, and will illustrate in detail how our portal rendering system works.

6.1 Building worlds

Since our approach is portal based, a mechanism is needed to interconnect rooms through portals. In addition to those, we have introduced two other kinds of objects: static and dynamic. Static objects are completely rigid, and are not affected by physics (although dynamic objects can of course collide with static ones). Dynamic objects can be manipulated in several ways, play animations and have physics simulators bound to them. An off-the-shelf 3D modeller would not be able to provide us with all this in a down-to-earth way – although most of them provide lots of other nifty features – so in order to create worlds out of objects, a customized modeller was needed. Since a full-fledged modeller on object-level as well as world-level would probably take more time to create than would be readily available to us, we decided to model our objects in a regular 3D modeller, and then create our own to connect objects into worlds.

This – of course – requires the models to be stored in a format that both the object modeller and the world modeller can understand. Since we use Cal3d for animations, we opted to use the Cal3d format for all our models. Object modeling has been done in Blender, a free and potent 3D object modeller. Blender allows Python scripting, and a more or less working Python script for exporting Blender objects to the Cal3d format exists. Other modelers also allow exporting to Cal3d. 3D Studio Max is said to work, and a beta exporter for Maya was recently released. This is a good thing, since it allows the artist to choose modeling tools.

Since the modeller is only a small part of this thesis, its use is somewhat limited. It can however be used to create worlds by adding, linking and connecting objects using an interface that is intuitive to me. (After all, I made it.) Other features include handling of lights and animations, and adding spawn points.

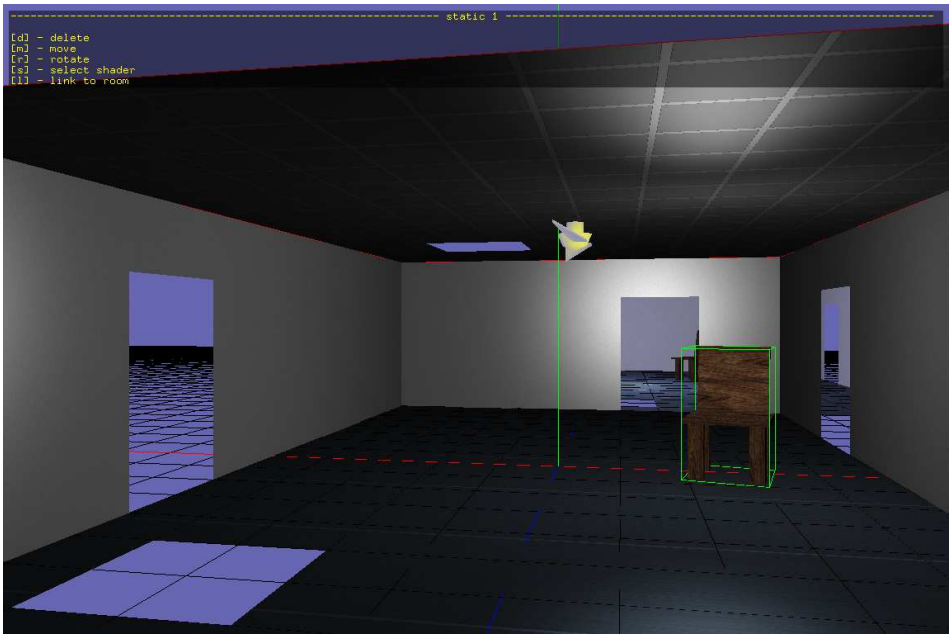

Figure 6.1. The modeller: A small world consisting of two rooms,a portal, a fan and two chairs. The fan (yes, it's moving) is using a single-pass version of the gooch shader found in [8], the closest chair a simple cell shader, and the closest room a specular shader.

6.1.1 World representation

There are of course several ways to represent the rooms and portals, as well as several ways to connect them. Our approach is simple, yet useful enough to provide us with the required functionality. We have chosen to keep rooms and portals as building blocks – think classes - and put out instances of these blocks as the actual rooms in the world. Both are static objects. We store the room instances in global coordinates, which somewhat simplifies rendering, but also limits us from creating warp holes or mirrors.

Ideally, a clip rect would be used for room-to-room clipping, but it seems that this functionality (although allegedly present on virtually all GPUs) is missing in OpenGL, so four clipping planes, arranged to mimic exactly the behavior of a clip rect is used.

6.2 Portals

6.2.1 A first approach

When I first started implementing a portal based system, I had a good deal of general knowledge and ... hunches of how it ought be done. Or at least that's what I thought. After quickly glancing through some theory I opted for static potentially visible volumes – henceforth called PVVs – pregenerated at startup, with the vague intention of creating some sort of representation suitable for storing these on file later on. The PVVs were to be generated using the following rules, where each room would hold PVVs for all portals possibly visible from the room.

For each room:

- The PVVs of portals linking the active room to its neighbors is the halfplane of the portal (i.e. full visibility)
- The PVV for each portal of level ≥ 2 is the intersection of the (prolonged) PVV of the preceding portal and the volume resulting from calculating the convex hull of all combinations of rays emanating from vertices of the first portal to vertices of the active portal

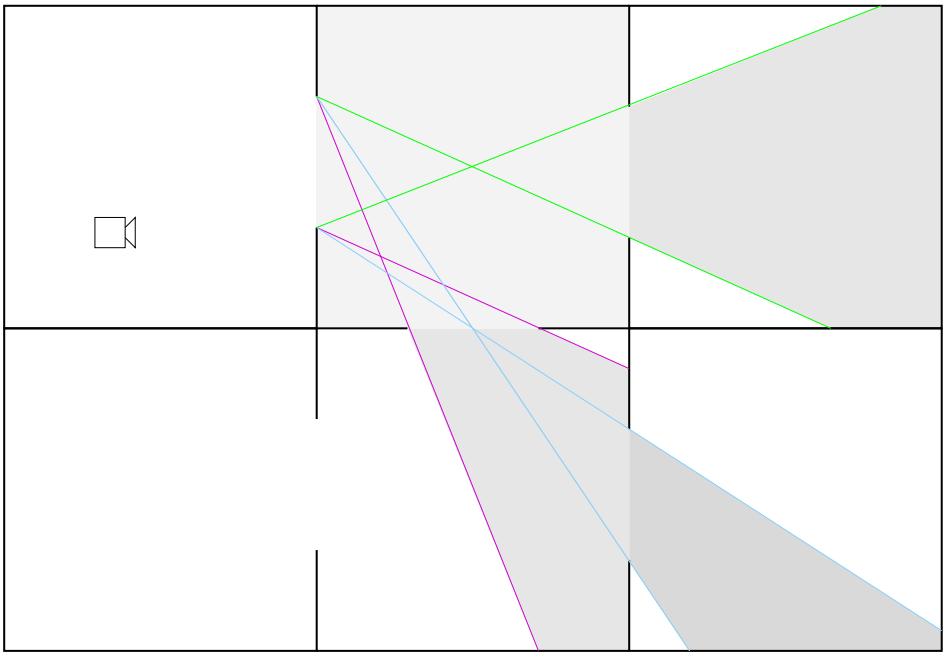

Figure 6.2. Static PVVs: the PVVs are precomputed to encapsulate everything that can be possibly seen from a room. (Example in 2D)

Generation would be recursive, and stop whenever the PVV of a portal was the

empty set. Since the intersection of two convex sets is always convex, all PVVs would be convex. Always. However, generating these sets showed to be quite a bundle, for several reasons. For example, the direction of the portals had to be remembered, and the portals needed be clipped to each other in the correct way for PVV generation to work. Also, since the volumes extended to infinity, a lot of headache was induced from on how – if at all – this would effect computation of sets and visibility.

To boot, there are at least three major drawbacks with this approach.

One: The PVVs have to be pregenerated. This will either take time at startup or require some sort of format that can store the PVVs. They need also be updated whenever a portal moves, which puts restrictions on how dynamic a scene may be. The regeneration would have to emanate in a strange way, and could scarcely be done in real-time.

Two: The PVVs will always be much bigger than the actual visible volumes, since they effectively are generated to encompass everything that can possibly be visible from a room, all extreme points accounted for ... simultaneously.

Three: The set of halfplanes – if that approach is to be used - defining the PVV will grow with the number of chained portals, and because of the prolongation towards infinity, removing the ones that have no effect on the resulting volume can be cumbersome. This leads to a lot of computation for each query.

6.2.2 Starting over – the final design

After fully realizing the shortcomings of the forementioned approach, and reading some theory, I opted a complete do over, this time aiming to follow in Luebke and Georges' [7] footsteps, as described previously in this document.

In our approach portals can have any shape, and consist of as many polygons as needed. As this representation might prove troublesome to project and bound, the convex hull of the portal mesh is precomputed at startup, using Graham's scan. This – of course – means that it will need be recomputed if the portal were to change during runtime. An alternative might be using a bounding box. However, although recomputing the bounding box for a mesh might be cheaper than finding the convex hull, it is still recomputation, and the convex hull will give a tighter fitting screen-aligned bounding rectangle than the box in some cases. Another problem with the convex hull is that its complexity usually is relative to the complexity of the object. This can be a really bad thing, since computing the portal frustum should optimally be done in time invariant of the portal shape. If portals are allowed to be extremely complex – for example round windows with lots of edges, resulting in a convex hull with as many vertices as the original portal outline – a bounding box-approach will probably be much better than the convex hull.

Drawing is performed on a per-room basis, in that whenever a room is visible

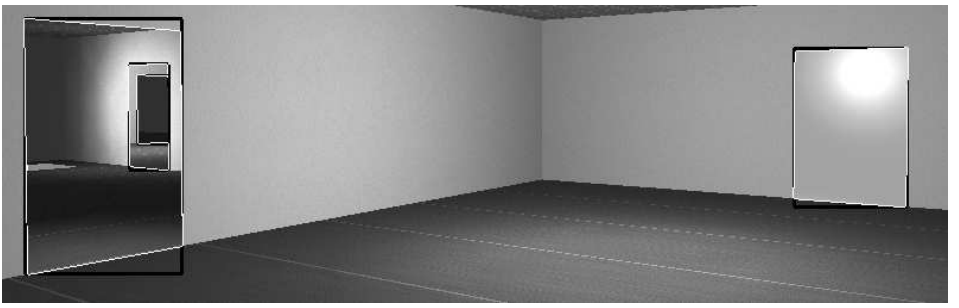

Figure 6.3. Portal rendering: an image rendered with our portal rendering system. The portal convex hulls are drawn in white, and the screen-aligned, projected portal frustums are drawn in black.

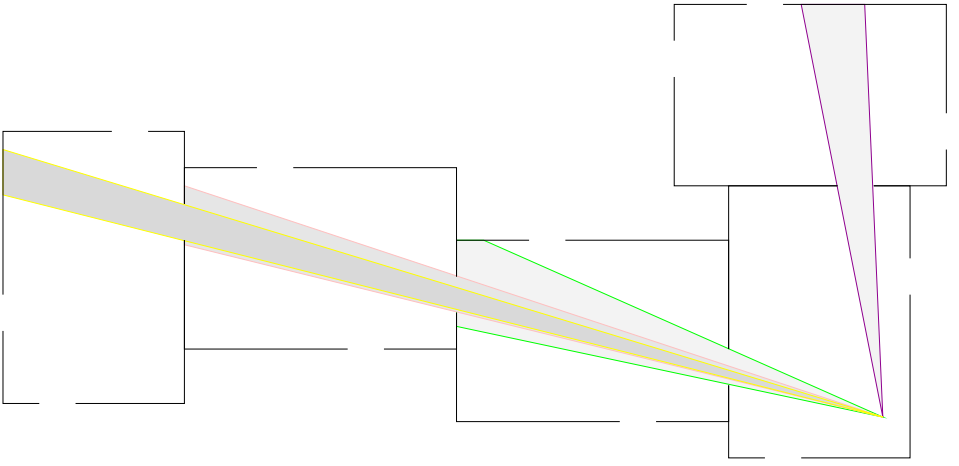

Figure 6.4. Portal rendering frustum: 2D abstraction of what the actual portal frustums might look like for the above scene

through a portal the entire room is drawn. Arguably, this is way faster than figuring out which faces are visible through the portal frustum and then send them one at a time to the GPU.

Whenever the portal frustum is updated, four clipping planes are introduced (or moved, depending on the recursion depth) that correspond to the four sides of the projected portal frustum. Thus, polygons falling outside the frustum will not be culled by the z-buffer on a per-fragment basis, but rather per-face.

After a room is drawn, all objects belonging to that room that overlap or fall inside the portal frustum are drawn in their complete. Again, this should be much faster than drawing them on a per-face basis.

Drawing is done in camera space. The main reason for this is that any portal can be projected to the camera plane and represented as a rectangle. The intersection of

two rectangles is very easy to compute, which leads to really cheap portal frustum calculations. For each room, no matter how many recursions down, the portal frustum is nothing but a rectangle.

6.2.3 The portal rendering algorithm simplified

- Find out in which room the camera resides
- Transform this room into camera space and render
- Set initial portal frustum rect to the (projected) view frustum
- For each portal in the room:
  - Transform the portal into camera space
  - Project the portal (or its convex hull, or bounding box) to the camera plane
  - Intersect the camera plane aligned bounding rectangle with existing portal frustum rectangle, or continue with next portal if empty
  - Set up clipping planes from the four sides of the clipping rectangle
  - Render the room on the other side of the portal
  - Recurse over all portals in this room (barring the one through which we came)

The same approach can be followed with any objects belonging to each room. In the name of efficiency, these should be rendered after the room, and culled against the portal frustum.

6.2.4 Problems and anomalies

In our representation, all objects belong to exactly one room, always. This is usually not a source of problems, but there are situations where extra care is needed to avoid peculiar behavior.

Consider, for example, the camera positioned as in Figure 6.5. An object is positioned so that it belongs to room 2, but is partially inside room 1, and overlapping the camera frustum. Without extra precaution, this would lead to the object not showing up on screen, since the portal linking room 1 and 2 does not overlap the camera frustum, and thus all objects in room 2 are culled. The same problem will occur whenever an object belonging to a room not drawn overlaps the camera frustum.

My solution to this is to detect all objects in non-visible neighboring rooms that overlap the portal linking the current room to the neighboring, and draw them if they overlap the camera frustum. This will work to some extent, but is not a

perfect solution. Consider, for example, a train going through a tunnel. The train is extremely long, and the tunnel quite short. Now, in this situation there would probably be three "rooms": one being the tunnel, and one on each side. If the train is long enough so as to be located in the room on one side of the tunnel, while on the same time extrude to the room on the other side, it would not show up in Figure 6.6. To completely solve this problem, one would either have to check all objects against non-visible portals in all visible rooms, or figure out some sort of complex traversal heuristic. Or – probably the simplest, easiest and best solution - allow objects to be in several rooms simultaneously.

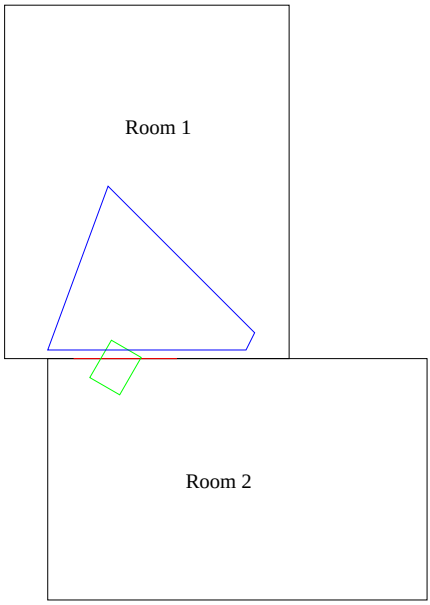

Figure 6.5. Partial overlap 1: Without extra care, the object in room 2 intersecting the view frustum will not be drawn

6.3 Multipass rendering

Sometimes when rendering objects, a simple onepass rendering approach is not enough – mostly when rendering complex objects that have reflection or outlining, or when applying interactive changes such as scratches, bumps, footsteps or bullet holes to objects in the world. While onepass rendering is somewhat fixed functionality – it's usually enough to change material specs, or use different shaders - any subsequent pass will need to be groomed to the porpose it is to fulfill. This causes problems with distributed systems, in conjunction with the vision we have of letting the users add objects in runtime. The multipass rendering algorithm that is to be used has to be distrbuted to all other clients, and moreover this rendering has to be safe – i.e. it cannot be allowed to do other things than render an

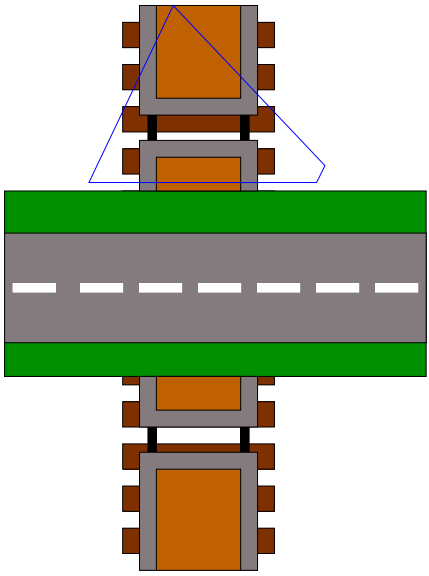

Figure 6.6. Partial overlap 2: Where is the train? This figure illustrates a possible problem with portal rendering algorithms

object. This is a very hard problem to solve while maintaining high performance – allowing users to send dynamically linked libraries written in C++ to other users, load them and execute is not acceptable, since there is no end to what nasties a malicious hacker can achieve. We have not came up to a solution to this problem, since the problem in itself is so complex that it would probably be a good starting point for a stand-alone thesis.

Chapter 7

Conclusions

When this project started, some goals were set up. Condensed, they were the following:

- render objects (i.e. sets of polygons)
- allow shading and texturing of said objects
- give a means to manipulate the position and rotation of said objects
- provide some sort of format to represent said objects
- support multipass rendering such as shadows and reflections (i.e. objects need to know about each other, at least to some extent)
- animate objects
- tell the server to distribute code to other clients
- compile, link and use code sent from the server
- decide which objects to consider for rendering

So did we reach this goal? Well, sort of. Rendering, texturing and shading is very much supported, as is manipulation and animation. The data format we use is that of Cal3d, that allows us to describe very general models, and also animatory properties via skinning. Opting to use an existing format, and one that supports animation, has saved lots of time, and resulted in much better performance than would have been reached if implementing an animations system from scratch - something that would have probably taken more time than the project itself. Multipass rendering is not currently supported due to the low level of security distributing this would result in – which also means that compiling and linking of distributed code is not done – but the implementation supports it to almost all ends. The only thing left is to store what (inherited) renderer is wanted in the world files, which means the modeller will need be extended to support it.

Distribution is not done by the graphics part of the engine, but it was designed with ease of distribution in mind. The burden of rending is reduced via portals.

So what have I learned? One thing is to think, gather information and read a bit of theory before rushing mindlessly into implementation – a lesson I have learned quite a few times before, but one that doesn't seem to stick no matter what. The time spent on developing the portal system could be reduced with a couple of weeks if I had done this.

Another is that it is hard to achieve complete flexibility while obtaining high security, at least when dealing with compiled languages. User-customizable rendering is a goal not achieved, and one that we still do not know how to solve in a good way.

Finally, developing this kind of system takes a long time. And even if you know that, it still takes longer than you think. The development of this system has taken me more than 20 weeks, which is a lot more than it should have, since the thesis need also be written. The overall size of the graphics part – including the lab, modeller, comments and blank lines – is around 17000 lines.

Chapter 8

Future outlook

Now that we have the background information necessary to understand the key concepts, along with a general understanding of how the actual system works, it's time to ask the hard question: how can the system be made better? A system like this is of course never completed. New features will always be wanted, and optimizations can be done eternally. With the ongoing advance of graphics hardware, things not previously feasible in real time will become more or less required. Barring this, there are a few things that could have been done better.

One is to rely more on modern GPU abilities. On modern hardware, using hardware occlusion testing instead of portal clipping might prove to be beneficial. Because of how the portal rendering code works, z-buffer lookups done after the portal rendering stage would probably not be that far from ideal. After all, the walls are all there, so the only thing left is the objects.

The portal code really should use the bounding box rather than the convex hull of portals to calculate the portal frustum. This would lead to some increase in overdraw, but on the other hand the portal frustum calculations would be independent of portal complexity. It is also much easier to keep the bounding box up to date if dynamically changing portals are to be used.

Since Cal3d allows us to write a custom mixer in the vertex shader, this should really be done on modern hardware. As Futuremark Corporation states, this can be several times faster than letting the CPU do the work.

And then of course are the tools required to create worlds. The modeller implemented during the progress of this thesis is rudimentary at best. Though it works surprisingly well given the limited amount of time available for development, there are several desirable functionalities missing, for example scaling of portals, rooms and objects. Maybe the absolute best way to deal with connecting rooms would be to skip the modeled portals entirely, relying on the user to select a set of vertices from the rooms to represent the portal. Another nice feature would be to be able to modify the actual vertices of rooms, so the modeller can model a room without

doors, and then just break up part of a wall, and create both the hole and portal in the modeller. This would probably not be easy to implement using the Cal3d data format, but maybe a dedicated format for rooms would be in place anyways. The decision to stick to Cal3d exclusively was made largely given the time span of the project.

Thus ends this thesis. It has been an interesting – but time consuming - project. I hope that others following in my footsteps may benefit from my mistakes rather than endlessly repeat them, as is the norm. I also hope that this thesis will spur interest in graphics programming. It is a very interesting and challenging field of programming. *Over and out.*

Bibliography

[1] Akenine-Möller, Tomas & Haines, Eric (2002), Real-Time Rendering, Second Edition, A K Peters

[2] DeLoura, Mark (2000), Game Programming Gems, Charles River Media

[3] GameDev.net – BSP Tree FAQ (1998-06-02) (2005-05-24), `http://www.gamedev.net/reference/articles/article657.asp` (2006-03-12)

[4] Hackman, Peter (2001), Boken med Kossan på, Tekniska Högskolan i Linköping, Matematiska instutitionen

[5] Kessenich, John, Baldwind, Dave & Rost, Randi (2004), The OpenGL®Shading Language

[6] about.com – Life in World of Warcraft 1.4 (2005-05-01), `http://internetgames.about.com/od/worldofwarcraft/a/wowhonor_2.htm` (2006-06-19)

[7] Luebke, D. and Georges, C. 1995. Portals and mirrors: Simple, Fast Evaluation of Potentially Visible Sets. In Procedings of the 1995 Symposium on Interactive 3D Graphics, P. Hanrahan and J. Winget, eds. (ACM Siggraph, April 1995, 105-106)

[8] Rost, Randi J. (2004), OpenGL®Shading Language, Addison-Wesley

[9] Sánches-Crespo Dalmau, Daniel (2004), Core Techniques and Algorithms in Game Programming, New Riders

[10] Sarkkinen, Tero (2003-02-14), Futuremark's Response to 3dMark®03 Discussion, `http://www.futuremark.com/companyinfo/Response_to_3DMark03_discussion.pdf` (2005-05-30)