School of Electronics and Computer Science
Faculty of Engineering, Science and Mathematics
University of Southampton

Philip Richard Boulain

3rd February 2006

Mandala: A Portal Engine

ELEC6025 Advanced Computer Graphics

Abstract

Mandala is a project to implement a practical portal engine—one where worlds are created out of 'sectors' of isolated space, linked by 'portals'—and demonstrate some of the advantages this approach has over conventional BSP-based engines. Zone portals, and derivatives such as antiportals, have been, and still are, often used for visibility determination in large sections of geometry; Mandala, however, is a 'full' portal engine, which allows for non-Euclidean geometry and 'folded space'. It also provides an architecture upon which to build applications (such as games) using the engine.

This report highlights the difficulties in designing and implementing such an engine.

Contents

Contents ii

Acknowledgements iii

1 Introduction and Background 1

2 Analysis 2
- 2.1 Existing systems . . . 2
- 2.2 Goals . . . 3

3 Design 4
- 3.1 Architecture . . . 4
  - 3.1.1 Overview . . . 4
  - 3.1.2 Control system . . . 5
  - 3.1.3 Entities . . . 5
- 3.2 Relevant light set . . . 6

4 Implementation 6
- 4.1 Technologies . . . 6
- 4.2 Portal mapping . . . 6
  - 4.2.1 Graphical . . . 6
  - 4.2.2 Point . . . 7
- 4.3 Results . . . 8

5 Testing 10

6 Evaluation 10
- 6.1 Reflection . . . 10
- 6.2 Future work . . . 11

References 12

A CD Contents 13
- A.1 Building Mandala from source . . . 13
  - A.1.1 Requirements . . . 13
  - A.1.2 Build options . . . 13
  - A.1.3 Building under UNIX®-like platforms . . . 14
  - A.1.4 Building under Windows® . . . 14
- A.2 Controls . . . 14

Acknowledgements

Thanks to Harry Mason for his initial support for the idea of attempting a full portal engine, and for suggesting some of the applications. Great thanks to Jim Gerrard for taking an interest in the problems of dealing with multiple co-ordinate systems and deriving the matrix transforms which map points into an aligned, polygon-relative co-ordinate system and back.

1 Introduction and Background

Visibility culling is an important part of real-time 3D graphics engines. It is often worthwhile to perform some simplistic (and thus fast) checks to determine if coarse sections of the environment and its contents are outside of the possible visible area; thus preventing time from being spent, either by the CPU or GPU, further clipping them against the view frustrum and performing Z-buffer tests.

Portals are one method to achieve this. The environment is carved into sections, usually termed ‘sectors’, which are ideally concave, at least to a basic approximation. Objects may be placed within these sectors, with their visibility tied to the sector containing them. Between the sectors are placed ‘portals’, which are invisible polygons covering all the joins. Optimal places for these sectors are usually in doorways, and from corners of major obstructions such as pillars to the corners of rooms.

The portal-based visibility test is a recursive algorithm. Starting with the sector containing the viewpoint, all the portals of the current sector are clipped against the view frustrum. If any of them are visible (i.e. not clipped away completely), the sector on the other side is rendered recursively: that is, it is drawn (along with any objects in it), becomes the current sector, and has its portals tested. This technique can efficiently clip away large sections of a complex environment by simply ignoring anything which cannot be seen through the appropriate sector.

Portal engines can, however, be made to achieve more than simple visibility testing. If each sector is treated as a separate piece of independent space, and portals are consider unidirectional ‘links’ to another sector, then it becomes possible to construct environments which are physically impossible.

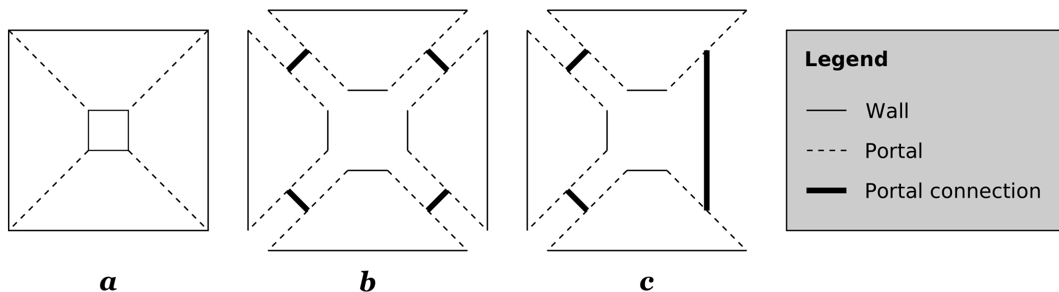

Figure 1: How to create a non-Euclidean triangular room

Figure 1 shows the derivation of one such environment, which was considered in a Usenet discussion [1]. In this figure, each pair of portals is mutually linked, effectively making the portal connections bidirectional.

1a shows a square room containing a pillar, with portals positioned so as to provide effective visibility culling around the pillar without any subdivision of the polygons of which the room consists. 1b shows an ‘exploded view’ of how this room can be constructed from separate, yet linked, sectors. In 1c, however, one of the sides of the room has been removed, and the portals either side linked to eachother. The room now has three sides, and is

thus a triangle; however, each corner is now clearly a right angle. Instead of summing to 180°, they sum to 270°; this is not a triangle possible in 'normal' space.

This also highlights one of the main complications of such a 'full' portal engine: it is no longer sufficient to simply pass a sector's geometry to the rendering system (e.g. OpenGL) when it is visible. It is also potentially necessary to perform transforms upon the sector such that the portals on either side align correctly. In the case of figure 1c, from the topmost side in the diagram, to render the bottommost side through the portal, it must be rotated 90°anticlockwise and translated such that it 'connects' to the topmost side.

A Mandala is a ritualistic geometric design used in meditation in some religions as a symbol for the universe. Discordianism's Mandala is a set of five interlocking rings in an impossible configuration, which inspired the name.

2 Analysis

2.1 Existing systems

My introduction to the idea of using linked polygonial surfaces to create 'impossible' geometry came from the Unreal engine, which uses a conventional BSP[1] approach to visibility, along with a few others, such as 'zone portals': simple portals which define how to cut the static world into zones (sectors). Unreal allows zones with a single portal to be declared 'WarpZones' [2], which may then link to one other WarpZone, causing the portals to lead to eachother. These zones need to exist as an area of physical 'overlap' space, which is theoretically never seen; in reality, however, this technology is barely used in Unreal engine games, and thus the implementation is not well-maintained. There are many, many limitations (not least that physics-controlled objects will pass straight through the portal into the overlap space), and it is very obviously a 'stuck on' feature.

Prey, an unreleased game with a troubled history, recently revived, originally used a full portal engine, as described by William Scarboro a few years after he left the original development team [1]. A video interview by Jess Holderbaum of gaming site 'Infinite MHz' with developer Paul Schuytema in 1998 showed some of the effects achieved with the technology, including the dynamic creation of portals floating in midair by the player and an allusion to the multiplayer gameplay implications of this ability. The engine was capable of rendering complex scenes, by the standards of the time, with impressive speed thanks to the portal culling; it was also capable of handling 'impossible' space, as demonstrated by a spinning gyroscope whose centre was a portal to a different part of the level.

The new version uses the DOOM 3 engine, again BSP-based, but little detail is availible yet; a video trailer released at the gaming conference E3

[1]Binary Split Partitioning, a commonly used visibility culling approach

showed that portal effects were in use, yet presumably these have also been shimmed on top of the licensed engine.

2.2 Goals

The goals, as defined in the project proposal, were:

1. Implement a basic portal-based engine.
2. Implement a basic editor or editing mode (the latter is a better demonstration of the lack of a need for precompilation with portal engines, as opposed to BSP) for the creation of environments.
3. Allow object navigation through portals (e.g. being able to walk/fly the camera through them).
4. Allow OpenGL lighting through portals (should be later extendable to positional sound).
5. Ensure that engine can deal with infinitely recursive portals (i.e. does not hang).
6. Ensure that engine deals with nontrivial cases such as mirrors.
7. Implement decals and 'spraycan' so as to better demonstrate non-Euclidean geometry.
8. Allow engine to cope with portals where the two 'sides' do not match in size by scaling objects that cross through. Demonstrate with a hallway of infinite growth/shrinking.
9. Improve handling of infinitely recursive portals by using high-recursion-depth startup texture generation to produce 'termination' textures, similar to aspects of "Images for Accelerating Architectural Walkthroughs" [3].
10. Allow point distance calculations through portals (e.g. for trivial, sphere-based collision detection).
11. Allow tracing of rays (for projectile collision detection) through portals; possibly add a 'laser pointer' to demonstrate.
12. Allow/demonstrate graph-based pathfinding (e.g. trivial A* implementation) through portals (requires a combination of the above two points to autogenerate a graph from a set of nodes: must connect nodes if in range and path clear).
13. Per-polygon collision detection of objects partway through a portal (again depends on tracing of rays).

These goals formed an approximate 'sliding scale' of scope, and it was not expected to complete all, or even most, of them in the time availible; they instead formed a 'direction' for the engine to take.

3 Design

3.1 Architecture

3.1.1 Overview

Mandala has an object-oriented design, and is designed to be decoupled from the application using it. In the demo application, the source `mandalademo.c` provides the entry point and command-line parsing, then instanciates and uses a mandala object and a world subclass instance which sets up a sample environment, demoworld.

A quick summary of the classes comprising Mandala:

mandala The root object of the system, handing operations such as changing resolution and simulation rate control.

controls A set of controls: dispatches SDL events and inputs to action objects.

action An abstraction action (such as ‘move forward’) and a binding to a concrete input.

world An environment containing entities.

entity An item in the world; this may be a sector, or an object in a sector.

mesh A collection of polygons which may be rendered as one, and automatically cached into an OpenGL display list.

polygon A single, planar polygon with any number of points exceeding three. May have a target polygon and entity, in which case it is a portal. May have a texture. Has utility methods such as automatic normal generation.

vector A translation—its usual role—with an optional rotation component applied afterwards, used to position entities. Has many utility methods, such as normalisation, dot and cross products.

matrix4 A 4×4 matrix, with methods for vector/matrix and scalar matrix multiplication, including on its transposition.

texture A handle to an OpenGL texture; handles the loading and deallocation of such.

bag High-performance dynamic array which does not guarantee ordering on removal; most operations are O(1).

tricks A set of utility functions, including a memory-tracing system to detect leaks and corruptions, and various common definitions.

3.1.2 Control system

A detailed description of the control system is beyond the scope of this report, but the basic operation is that actions are created and registered with the controls object for each action which can be taken. These actions are then bound to SDL inputs, with a sensitivity scale: e.g. bind the action for looking left and right to the X mouse axis, sensitivity 0.9. Additional bindings can be made, which will create a chain of actions, all of which report back to the head action. This system is sufficiently generic to allow, for example, the look action to also be bound to a pair of keyboard keys.

When the mandala object runs through the main loop, it dispatches input events to the controls object, which then finds any appropriate actions. All inputs map to a 'state' of the action on the scale -1 to 1, which provides input device agnosticism. There is also a state machine on each action which can be used to determine if it has been 'pressed'—has become nonzero since last tested. This allows keypresses to be reliably trapped even at low framerates.

3.1.3 Entities

An entity is a generalisation of both a sector an an object. An entity is simply an item in the world: how it looks and what it does are defined by five 'behaviours':

act Act independent of any other entities; this behaviour is theoretically parallelisable.

interact Interact with another entity; the set of entities this is called with can be controlled by flags such as '`onlycolliding`'.

display Render the entity. The world positions the OpenGL coordinate system and handles the matrix stack.

collide Detect a collision between a line segment and this entity, and return information such as the point of impact, and the normal at this point (for bouncing).

destroy Deallocate all memory and release all resources used by this object.

Each behaviour is controlled by a field, such as `acttype`, which accepts constants for basic behaviours, such as `A_NONE`, or `A_SIMPLEMOTION`. All behaviours also have a `_FUNCTION` constant, which allows a function pointer such as `fact` to be set, allowing completely custom behaviour.

When an entity wants to move, it sets a `desiredmovement` vector of itself, rather than directly modifying its position. This allows the world to perform collision detection during the movement. Collisions are stored by the world in a bag, and the entity can then respond to them the next time it `acts`.

Entities form a tree structure; top-level entities are effectively sectors, and the entities they contain are objects. Children of those 'object' entities can be used to create hierarchical objects, such as robotic arms.

Entities may have meshes, which may be used for display or collision detection. These meshes may contain portal polygons, effectively laying a directed graph of portals over the tree structure of entities. While this complicates the rendering process, this was considered worthwhile for the potential advantages.

3.2 Relevant light set

Lights cannot be simply rendered recursively like geometry; the way OpenGL works requires that they are set up prior to the polygons which need to be lit are drawn. As such, a trivial solution would be to recurse through the visible set of portals, as if rendering, before the actual rendering stage, setting up lights. While this would work, it would be inefficient.

Instead, a relevant light set can be calculated for each entity when lights change, which records which lights are in range of that entity (allowing for portal traversals), and what their relative positions are. When that entity is to be rendered, the appropriate lights can be read from this set and told to set up first (because lights are actually entities with a `displaytype` of `D_LIGHT`, this is a simple `entity_display()` call). Afterwards, these lights can be deactivated again. This also effectively reduces the number of hardware lights required at any one point, which should greatly improve performance in heavily-lit environments.

4 Implementation

4.1 Technologies

Mandala is written in C (C99 standard), as it is the language for which bindings for many libraries exist, and is suitable for applications which require good performance. It uses OpenGL and SDL for graphics and windowing/input, as both are well-designed, stable, cross-platform APIs. The SDL_image library is used for texture loading.

4.2 Portal mapping

4.2.1 Graphical

The transforms required for changing the co-ordinate system to that of an entity on the other side of a portal are left to OpenGL, both for simplicity and speed.

Figure 2 shows the required transforms to move a disconnected sector (show in grey), stage a, to a correctly positioned sector for the portal, stage b. The thick lines represent the normals of the sides they are attached to, which are portals. First, a translation is made from the centre of the entity (i.e. the origin of the current co-ordinate system) to the polygon's centre. Now a rotation is required, such that the normals of the two portal polygons

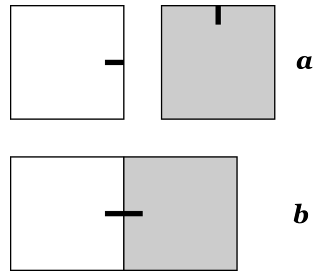

Figure 2: Transforming a sector to correctly align portals

point in exactly opposite directions. This is performed around a unit vector which is the result of the cross product between the two normals: this gives a suitable perpendicular. The angle to rotate by is the inverse cosine of of the dot product of the normals: i.e. the angle between them. If the cross product results in a zero-length vector, then the normals are parallel. This means that the normals are either already in opposite directions, in which case no action is required, or identical, in which case a half rotation is required.

Finally, a translation must be made from the centre of the target polygon to the origin of the target entity. Because this is now in the co-ordinate system of the target entity, this is simply a translation by the inverse of the target polygon's centre.

4.2.2 Point

Much of the mathematics in this section was determined with the great assistance of Jim Gerrard.

Mapping points across portals is important for many purposes, not least that of object navigation. Other applications include tracing rays for collision detection, and calculating relative light positions. This is achieved by a pair of methods of polygon which map points to and from a polygon-relative co-ordinate system, where the normal is the positive Z axis. By mapping a point into the co-ordinate system of the source portal's polygon, then unmapping it from that of the *target* polygon, the point will be transformed into the co-ordinate system of the entity on the other side of the portal. This mapping is also useful for collision detection, as the Z value of the point in polygon co-ordinates is its distance from the polygon's plane; if two points representing the start and end of a movement have Z-values of a different sign, then they have cross the plane and collision detection should continue to see if the intersection point is within the boundaries of the polygon.

If we define x, y and z as the polygon normal, and let:

$$\alpha = \sqrt{1 - z^2}$$

then the matrix to rotate a point by aligning the normal to the positive Z axis is:

$$\begin{pmatrix} \frac{xz}{\alpha} & \frac{yz}{\alpha} & -\alpha \\ \frac{-y}{\alpha} & \frac{x}{\alpha} & 0 \\ x & y & z \end{pmatrix}$$

and the inverse for this matrix is simply itself transposed. There is a singularity if $|z| = 1$, as α will be zero, and will cause divisions by zero. Thinking geometrically, this is the case where the normal is already aligned, although possibly flipped. The solution is to instead use the identity matrix multiplied by z; this will be the identity matrix (i.e. do nothing) if the normal is already the positive Z axis, and will be a simple flip if it is the negative axis.

Next, a rotation around the Z axis needs to be performed to lock down an unwanted degree of freedom: rotation around the normals. This is achieved by specifying a point on the polygon as the 'anchor point', and aligning it to the positive X axis. If a_x and a_y are the transformed offset of this anchor from the centre of the polygon, then the rotation is a simple 2D matrix (which can be padded to 3D with the identity matrix):

$$\begin{pmatrix} a_x & a_y \\ -a_y & a_x \end{pmatrix}$$

The derivation of these matricies is too lengthy for this report, but is based on representing the normal as an *azimuth* (rotation around Z axis) and *declination* (rotation around X axis transformed by azimuth), then using trigonometric identities to simplify.

4.3 Results

The implemented engine has a largely complete rendering and entity system, with collision detection; however, there are some unresolved issues. Crucially, portal traversal by the camera is nonfunctional, which has limited the number of effects demonstratable. Camera/portal collisions are detected successfully, but the rotation is not mapped across portals, and the position currently tends to get 'stuck' in the plane of a portal.

Nevertheless, some portal effects are demonstratable. In the demonstration environment, there are three top-level entities (i.e. sectors): the starting sector, which is textured with debugging textures showing which face is which, and providing alignment information on the floor; a second sector, which has grey walls; and a diagonal corridor section, with wooden and metallic texturing. All textures were generated in The GIMP: using the seamless tiling filter, plasma-generation filter, and stock pattern fills.

Most portals in the demonstration environment are bidirectional: that is, they are in matched, mutually linked, pairs. The starting sector is connected by a single portal to the second sector. The second sector connects on both the left and right sides to different ends of the corridor sector: this creates an infinite loop of corridor sections and 'second' rooms. The second sector's roof also connects to the single portal of the start room: this is a unidirectional connection, else the start room's portal would have two destinations, which is not possible. The bright green lines in all screenshots are a rendering option (which can be changed in the world object) showing the portal outlines.

Figure 3 shows the view from the first sector, looking through the second sector, up to its roof, where the first sector is repeated, rotated. The dark,

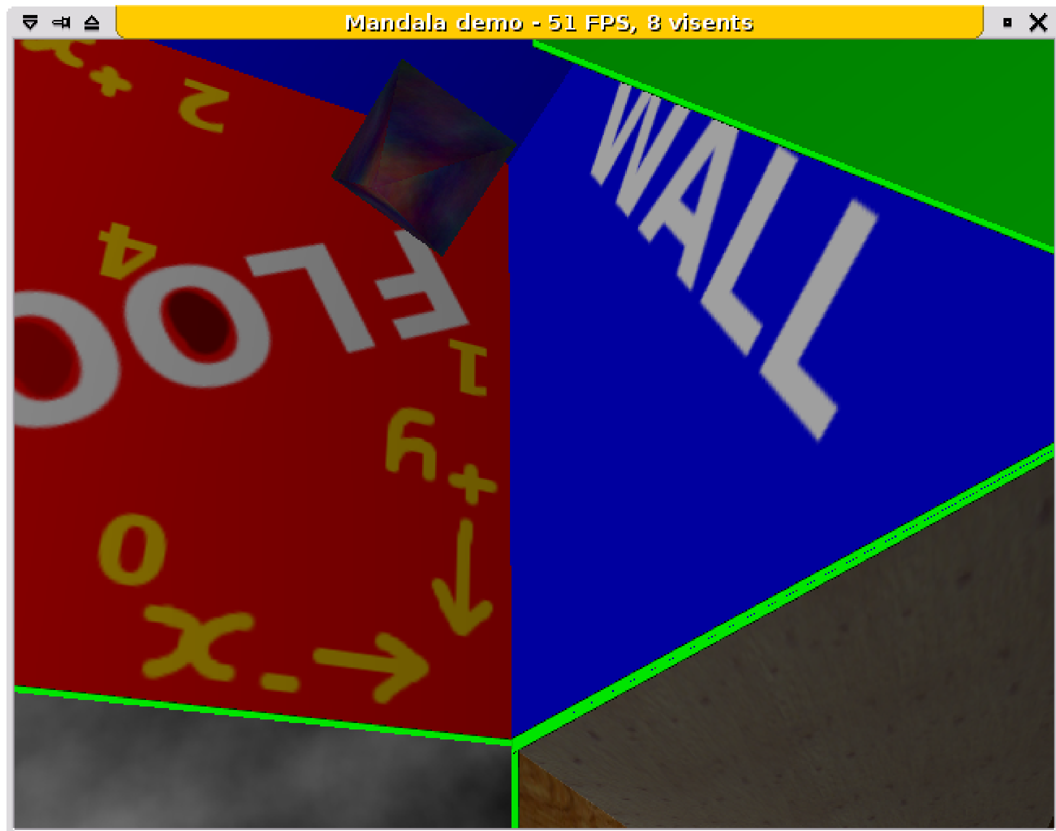

Figure 3: Screenshot: the camera sees itself in a 'copy' of the current sector

multicoloured pyramid is the representation of the camera: it is looking at itself.

In figure 4, the camera has passed backwards through the wall of the first sector, and been risen to show the extent of the environment. Note that the sectors form an apparent loop, yet all corridors are the same. This also demonstrates the engine's ability to limit recursion in cases where the portal graph is cyclic.

Figure 5 is from a viewpoint slightly to the left, such that the leftmost portal of the second room is now not visible: while it is inside the view frustrum, it is facing the wrong way and has been backface culled. As such, the corridors and other structures leading from this are not rendered: note the lower visible entity ('visent') count in the window title.

Figure 6 is from much the same viewpoint as figure 4; the sector visible at the bottom is the start sector. However, the destination of its portal has been changed, at runtime (via a custom `act` method which detects that an associated action has been pressed), to one end of the corridor. Hence, a different level structure appears. This demonstrates the lack of a requirement for precomputation.

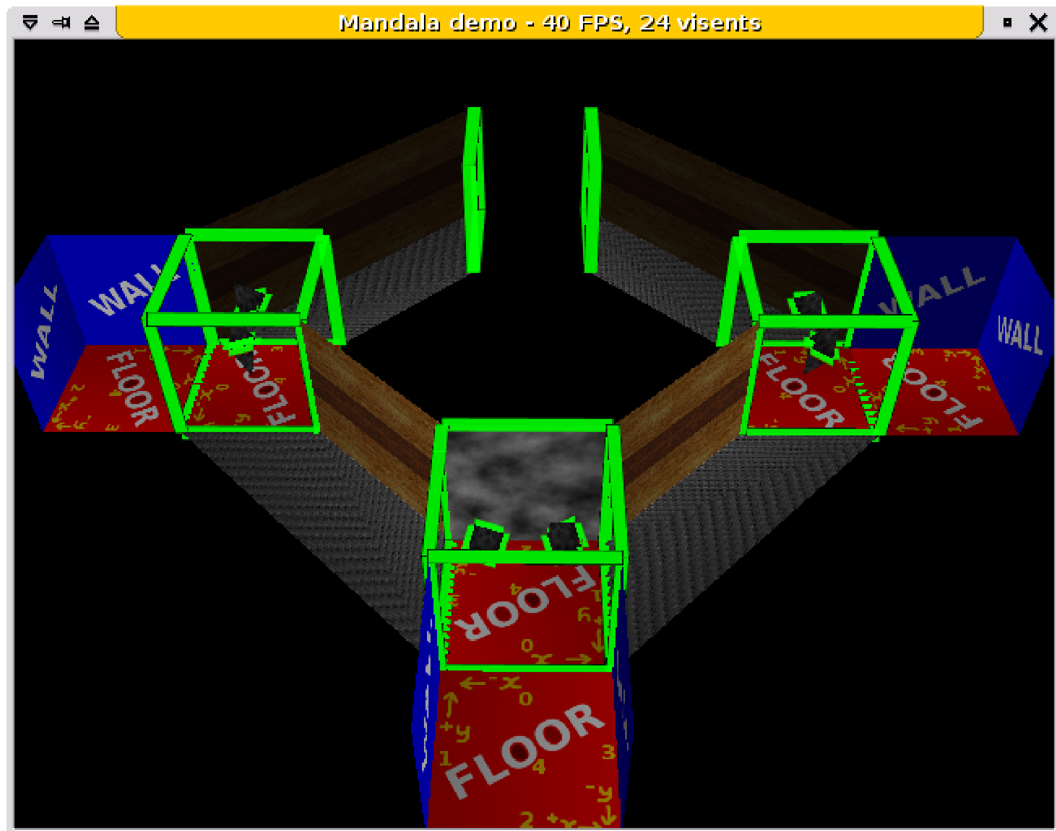

Figure 4: Screenshot: the full 'ring' created by the test data

5 Testing

Testing was performed continually during development; formal testing was not a practical option given the time constraints and difficult with formalising tests for a graphical project. Despite a modular design, these projects do not lend themselves easily to unit testing, so the majority of testing was white-box; verifying that a change was functioning as intended.

An exception was the mapping of points to polygon-relative coordinates: this was informally, individually tested on a set of data, using test harness code which printed the results out to the console. This harness code was also used to test the reverse-transform code, by applying it to the transformed co-ordinates and checking for discrepancies between the original point and the result of the unmapping.

Testing was performed on a small range of hardware.

6 Evaluation

6.1 Reflection

It is disappointing that portal traversal was not fixed in time for the final presentation; however, Mandala *does* support rendering of portals, handling

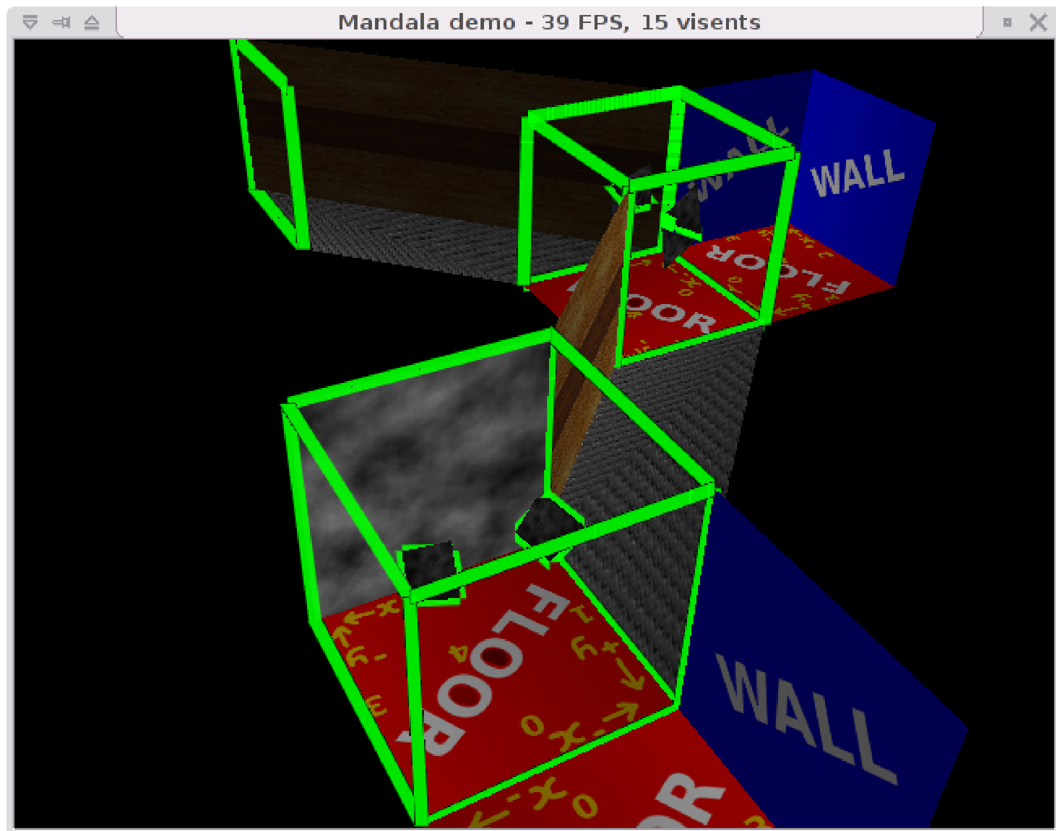

Figure 5: Screenshot: part of the ring is culled

cases such as infinite recursion (cycles in the portal graph), and has much of the architecture required to complete the goals with time.

Attempting to design and implement and entire, extensible *engine*, as opposed to a one-off technical demo on top of a simplistic framework, was an excessive target for a single course assignment; this situation was not helped by the need/desire to develop large sections of code (such as the entity system, and the control system to move the camera) before work on actual portal rendering and mapping could begin. If repeating this project in the scope of the Advanced Computer Graphics course, the 'engine' components would probably be dropped as goals.

6.2 Future work

Evidently, there are many goals left to implement, including editing (and serialisation) of entity structures, as entering world data in the form of C function invocations is neither user-friendly nor scalable. Lighting is designed, and the methods are in place to implement it correctly; they merely need the designed algorithm to be programmed in. Collision detection in the form of raytracing across portals is in the same state, with much of the code implemented.

The entity system could potentially be improved: defining new behaviours,

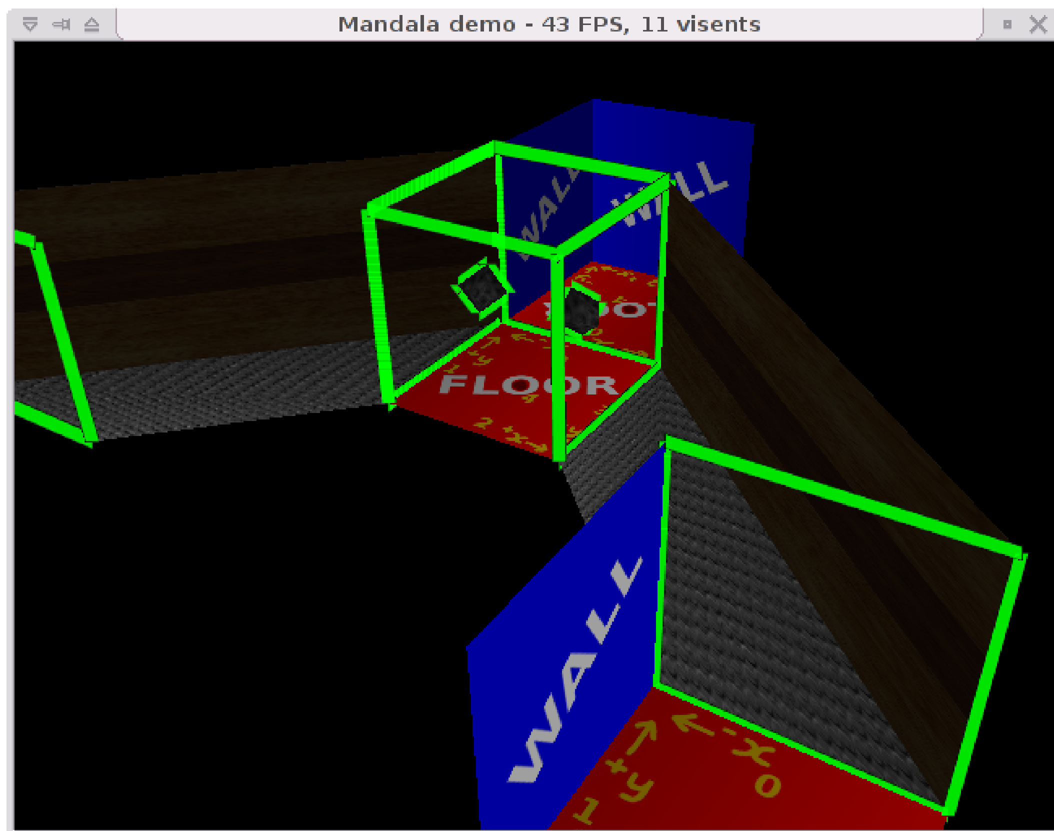

Figure 6: Screenshot: the same initial room, with its portal target changed to the corridor

mechanisms for them to reference any required state, and arranging proper deallocation is awkward in C. Ultimately, the optimum approach would be to adopt a high-level scripting system with garbage collection, as the Unreal engine does with UnrealScript [4]. Rather than developing a whole new language, possible candidates for embedding are PLT-Scheme, Perl, Lua, and JavaScript. Languages which support closures would greatly simplify behaviour writing and make pointers such as `actdata` redundant. Failing this, allowing multiple behaviour types to be chained would be greatly useful, allowing entities with `A_SIMPLEMOTION` actions to also have a custom, function-based action which operates on, for example, an `action`. The destruction behaviour would especially benefit from being able to 'inherit' the default behaviour and add additional, user functions to deallocate extra data.

Once these changes are made, I believe that Mandala is a seriously feasible engine to use for experimenting with the gameplay implications of impossible geometries.

References

[1] Various, "Usenet thread 'Portal Engines'," 1999. Viewed online at http://groups.google.com/group/comp.graphics.algorithms/

browse_thread/thread/d8e6be3323c9f5f8/?fwc=1.

[2] Various, “WarpZoneInfo documentation on UnrealWiki.” Viewed online at http://wiki.beyondunreal.com/wiki/WarpZoneInfo.

[3] M. M. Rafferty, D. G. Aliaga, V. Popescu, and A. A. Lastra, “Images for Accelerating Architectural Walkthroughs,” *IEEE Computer Graphics & Applications*, vol. 18, no. 6, pp. 38–45, 1998.

[4] T. Sweeney and M. Hendriks, “UnrealScript Language Reference.” Viewed online at http://udn.epicgames.com/Two/UnrealScriptReference.

A CD Contents

prb102-mandala.zip Mandala source code, resources and build system.

advcompgraphrep.pdf This report in PDF format.

A.1 Building Mandala from source

A.1.1 Requirements

To build Mandala from source, you will need:

- OpenGL 1.1 or higher
- SDL (Simple Directmedia Library)
- SDL_image
- GNU C library (libc)—required for `argp` command line parser

A.1.2 Build options

There are five preprocessor defines which control the features of the resultant binary.

NDEBUG Standard C define which will disable some pedantic sanity checks (via `assert()`). Can be used for release build.

DEBUG Enables some additional sanity checks.

DEBUGMEMORY Enables memory allocation (heap) tracing. This will find memory leaks and other errors (such as attempts to deallocate unallocated memory). Requires DEBUG.

DEBUGMEMORYQUIETLY By default, DEBUGMEMORY is very verbose, which can slow down startup and shutdown. This disables all non-error memory trace output.

STENCILCLIP Enables stencil clipping. This allows the renderer to better handle overlapping geometry, but may affect performance on low-end hardware.

The Makefile can accept these arguments in the `CFLAGSEX` variable; some default values (debug and release builds) are provided in comments.

A.1.3 Building under UNIX®-like platforms

Mandala has been compiled successfully on Linux systems with GCC and GNU make, and should work on other, similar platforms. If using a compiler other than `gcc`, change `CC` and `LD` in the Makefile to your C99 compiler and linker respectively, and check the values of `CFLAGS` and `LDFLAGS`. To build, change to the `mandala` directory (above `src`) and run `make`. This should generate a `mandalademo` binary.

A.1.4 Building under Windows®

Mandala has not been compiled under Windows®, although it has been implemented using cross-platform technologies. Simply compiling all the `.c` files together into a binary, linked with the appropriate libraries, should work.

A.2 Controls

The following keys affect the demonstration application:

Escape Quit immediately.

Backtick Toggle fullscreen mode.

Mouse motion Rotate camera (look around).

W, A, S, D Standard FPS movement: W and S move forwards and backwards; A and D strafe.

Space, Ctrl Move directly up and down.

9 Toggle the sector target of the start room between the second room and the corridor.