IES Group Report
Z Fighter Forever

John-Mark Bell, Philip Boulain, Karl Doody, Jim Gerrard

14th June 2006

Contents

1 Game Design

1.1 Gameplay

Z Fighter Forever is a dog-fighting space-ship game in 3D. It is an online multi-player game, making use of a client-server architecture to pit pilots against each other in a free-for-all arena.

The game play is simple but addictive: in joining a server, you spawn within a large arena and you shoot at other space ships in the hope of killing them. Interesting choices, with non-trivial solutions, are presented to the player given the unpredictable human nature of their opponents. The result of a bad, or even fatal, choice is not permanent. If a player is killed, they are immediately re-spawned within the arena.

The game takes advantage of a new portal engine, introducing strange non-Euclidean effects into the layout of the arena. This presents strategic choices for the players, in using the environment to their advantage, and can be combined with the more immediate tactical choices of aiming or dodging.

A basic scoring mechanism, points for kills, provides a numerical method of skill comparison in the form of positive feedback.

There is only one choice of ship and it only has one weapon: guns. The guns are fast-moving finite-distance intermittent laser beams. The combat ship has four thrusters and in zero gravity a lot of different manoeuvres can be performed. Movement is inertial and frictionless. A joystick is non-essential but preferable.

The symmetrical nature of this adversarial game lends itself to balanced gameplay: every player is in the same ship, with the same characteristics, and no starting position gives an overwhelming advantage.

In the future, the gameplay could be improved further with score persistence, ranking, teams, and squads. Such changes would provide the platform from which a community may grow.

1.2 Level design

Aside from a series of test cases to verify the function of the level loader, Z Fighter Forever includes one level: `]|[space`. This has been designed to demonstrate some of the geometry made possible by the portal engine (see section 4.1.1). There are three rooms, attached to three corridors; these three corridors are connected at right angles. As a result, the corridors strictly form a triangle, as it has three sides—however, each corner of the triangle is clearly a right angle. Additionally, the rooms contain enclaves where ships may spawn with some degree of shelter from ongoing firefights; these enclaves actually protrude back beyond the entrance to the room in such a way as to overlap the space occupied by the corridor. For a top-down view of the level, from the design document, see appendix B.

1.3 Game Logic

The core game logic is concerned with ensuring that the simulation of the environment is as accurate as possible and synchronised with other participants' concept of the current state of the game. It is therefore required to receive and process messages from the network and, in the case of clients, from the user's input.

In addition to retrieving and processing messages, the game logic is responsible for stepping the Mandala engine main loop. In order to do this, the code initialises all non-transient entities within the world simulated by Mandala on player connection (the server performs this step when it starts up). This setup process involves registering handlers for the behavioural callbacks that the game logic is interested in. These will be called at the appropriate time by the Mandala engine code.

The game logic is therefore largely driven by the incoming messages. It updates the relevant data structures based upon these messages and steps the engine such that these changes take effect.

The logic uses deterministic algorithms to allow resimulation of game steps should the game detect that a client is out of sync with the server. This also has the added advantage that the implementation may be shared between the server and the client.

Messages are abstracted away from their original source such that the core logic may be able to receive messages without needing to know where they came from. This has the added advantage that, in the future, it will be possible to add AI-driven entities in the world. This will require that a method be written which generates messages which are queued for processing by the behaviour callbacks when they are invoked by Mandala.

1.3.1 Resimulation

The deterministic nature of the game logic allows for simple resimulation of client state when a lack of synchronisation with the server is detected. This is performed by retaining a history of previous states which may be used to resimulate actions in order to synchronise the client with the server. Resimulation begins from a given time and applies all stored actions from that point onwards until the most recent action has been resimulation. This process can be described by the following algorithm:

1. Find state dump immediately preceding timestamp.
2. Initialise system to state in dump.
3. Iterate list of control actions subsequent to state dump but prior to next state dump, applying each action and ticking Mandala.
4. If there is another state dump, overwrite it with the current state, set it to the current state dump and goto 2.

The history is stored such that state dumps are in chronological order and each has an associated repository of subsequent actions (such as control inputs) which is also ordered temporally.

2 Game Resources

2.1 Models

There were 3 main models created for use in the game and its development: a low-poly ship, a high-poly ship, and a bounding box for collision detection. 3D Studio Max was used for modelling, with export to the MD3 file format for use in-game.

The low-poly model was created very early on in development as the availability of a ship model sat on a critical path. It also prevented any kind of performance hit, associated with a higher-poly version, when other features were being developed and debugged. See figure 1 for a render of the low-poly ship.

The high-poly model was created after the low-poly model, and in parallel with coding. Originally, splines were used to mark out the edges of the ship with the intent of applying a mesh modifier to fill in the blanks. This proved to be flawed given the complexity of the model. Instead, after learning the intricacies of poly-modelling (also known as box-modelling), the ship was re-modelled using this technique. A work-in-progress test render, or clay render as it is sometime known, can be seen in figures 2 and 3.

The high-poly ship was based upon the StarFury found in the Babylon 5 TV series, using drawings and photos as references. Only half of it was modelled, before applying a Symmetry modifier to achieve the whole ship, and can be seen in figure 4. The bounding box used for faster collision detection with the high-poly model can be seen in figure 5.

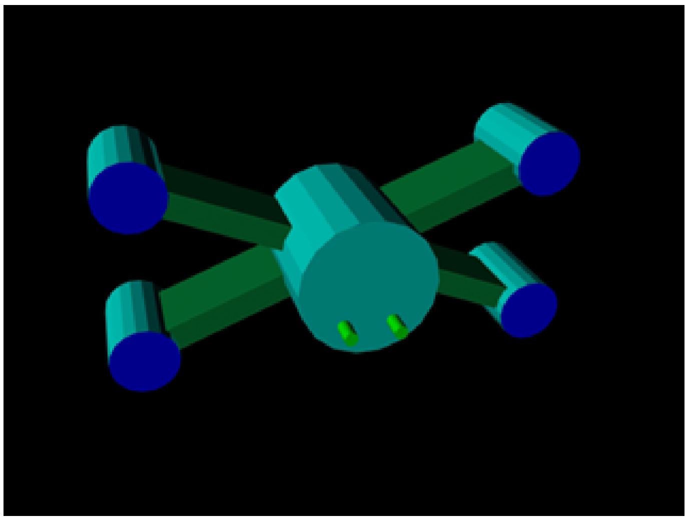


Figure 1: Low-poly spaceship used in development.

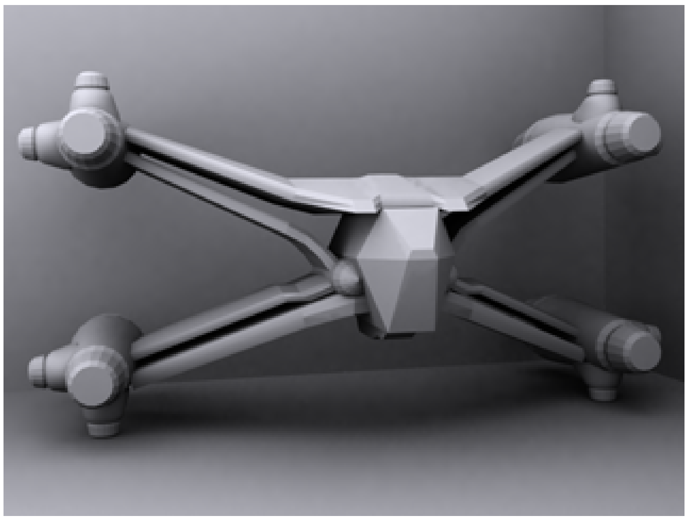


Figure 2: Test render of the high-poly spaceship.

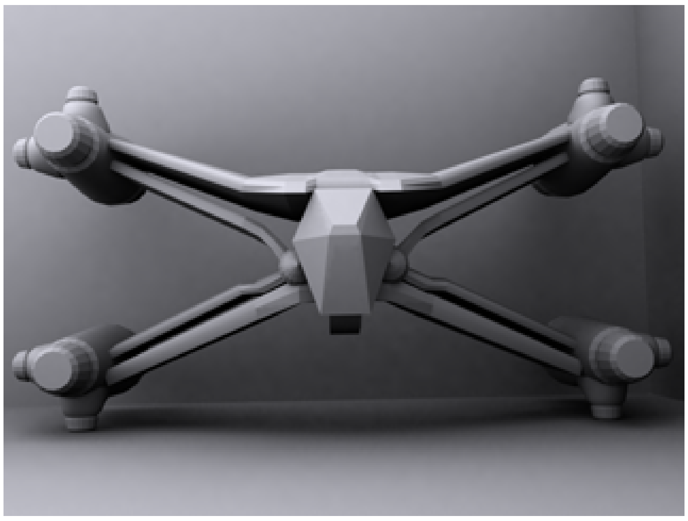


Figure 3: Test render of the high-poly spaceship.

2.2 Textures

There are many textures within the game but the most notable are the ship and level textures. The unwrapped UVs of the low-poly ship can be seen in figure 6. The high-poly ship was not UV mapped after several practice runs with simpler models highlighted the time investment that would have been needed, time that could be better spent elsewhere in the project. Instead, a stock texture was used for the high-poly ship, just as stock textures were used within the level, shown in figure 7.

3 System Architecture

3.1 Overall design

This section documents the architecture and design of the game implementation, visually represented in figure 8. The Mandala library is documented in section 4.1.1.

The system can be decomposed into two parts; the first is that handling and dispatching network and control input messages; the second is that which acts upon these messages. Both parts are driven (indirectly) from the main loop of the game. The main loop of the game is also responsible for making Mandala run.

Game objects are uniquely identifiable as they are assigned a unique ID when they are created. This, therefore, provides the ability to decompose the system

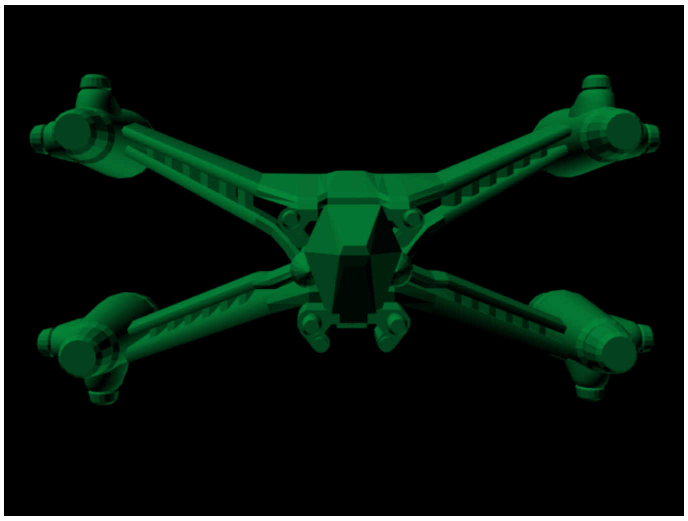


Figure 4: High-poly ship used for release.

into the two facets. Additionally, a distinction is made between objects which are local to a client and those which are referenced over the network. This is simply achieved by splitting the entity ID space in half and assigning local IDs from one half and network IDs from the other. The server is the only participant in the game that is permitted to generate network IDs; local IDs may be generated by any participant as needed.

Network and control input messages (hereafter “input messages”) are processed by reading all pending messages out of the network library and handling them appropriately. Those messages involved in the setting up of a connection are handled immediately, as are notification messages about ships taking damage and players dying. The remaining messages (state dumps and control inputs) are handled slightly differently. These are passed to a state history repository for handling. The state repository is a singleton class available on all clients. This stores a fixed-length history of game actions. This allows for resimulation if it is detected that synchronisation between client and server is lost.

In addition to being added to the history repository, these messages are passed to the action handlers for processing during the next game step.

Action handlers are integrated into the system as Mandala behavioural facets applied to the relevant entities. These are called each time Mandala steps to the next game tick. These handlers have access to a first-in first-out queue (FIFO) of input messages which apply to the entity they are attached to. Each time they are called, they process the entire pending content of the FIFO and update the appropriate state of the entity. For example, this might involve updating an object’s linear velocity based upon a player’s control inputs.

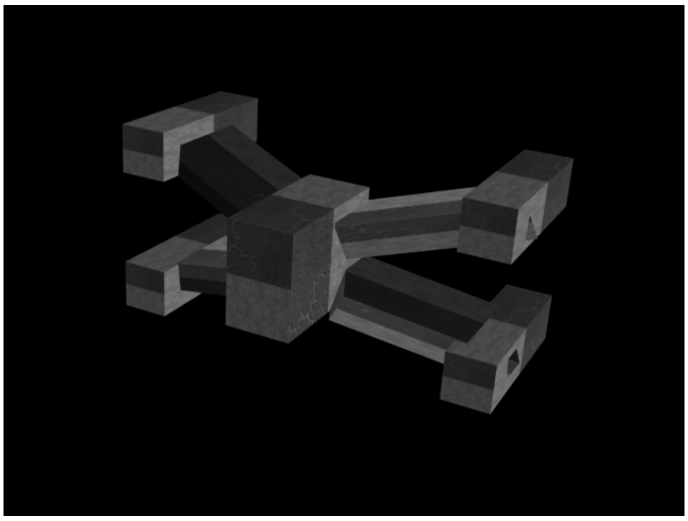


Figure 5: The bounding box used for collision detection.

In addition to the client handling of input messages, the system is designed such that the server is the authority. Therefore, whatever the server sees as the current game state is the only thing that is valid - clients' concept of states is simply an approximation, which is kept synchronised with the server through regular state updates and resimulation of actions from a known state should non-synchronisation be detected.

Due to the deterministic nature of the game, the client and server may share the vast majority of the implementation. The only difference between the two being that the server processes slightly different network input handling logic, in order to cater for messages sent by the client. Additionally, the server does not register any input bindings and does not render anything. Therefore, it is theoretically possible to run the server completely headless.

3.2 Abstraction

3.2.1 Object Management

The 3 primary objects within the game are ships, projectiles, and players. The management of these objects has been encapsulated into object manager singletons named PlayerManager and EntityManager. Each object manager is responsible for creating, deleting, and the retrieval of their encapsulated objects; in the case of the PlayerManager, this is player objects, and in the case of the EntityManager, these are the ship and projectile objects. Class diagrams of these managers can be seen in figure 9. The singleton nature of these managers allows

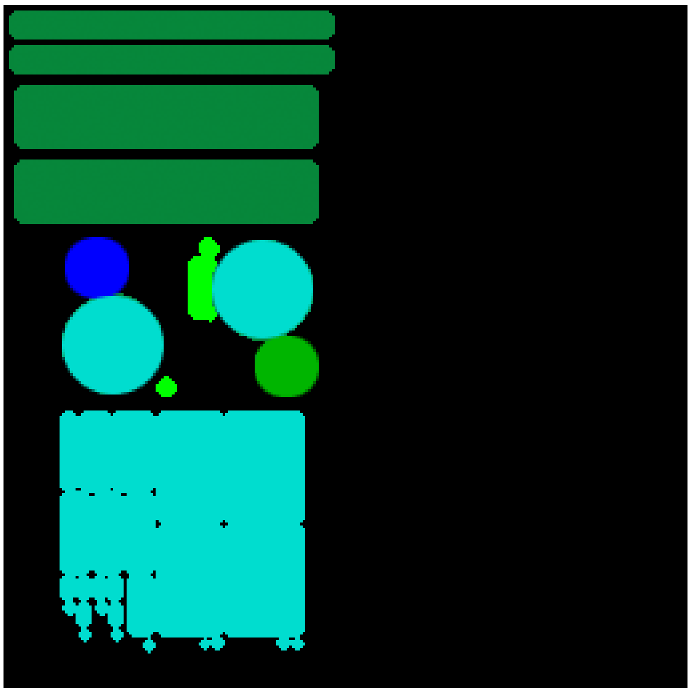


Figure 6: The UV mapped texture of the low-poly ship.

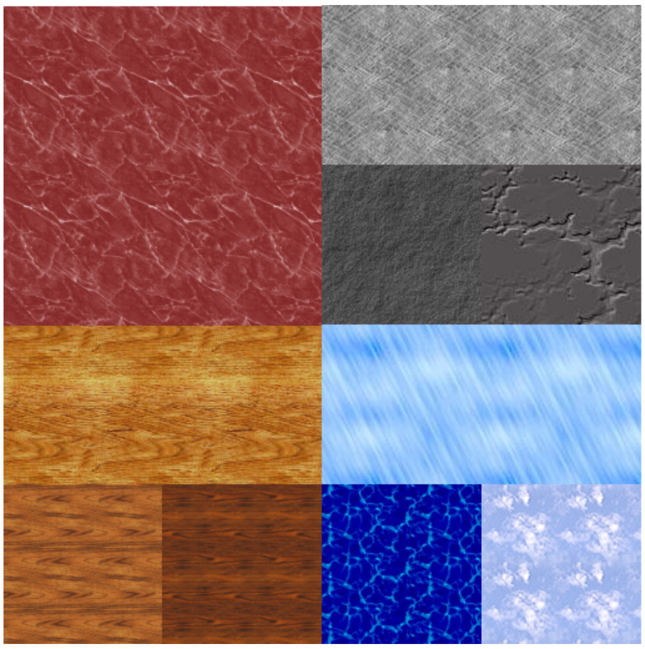


Figure 7: The stock textures used with the high-poly ship and level.

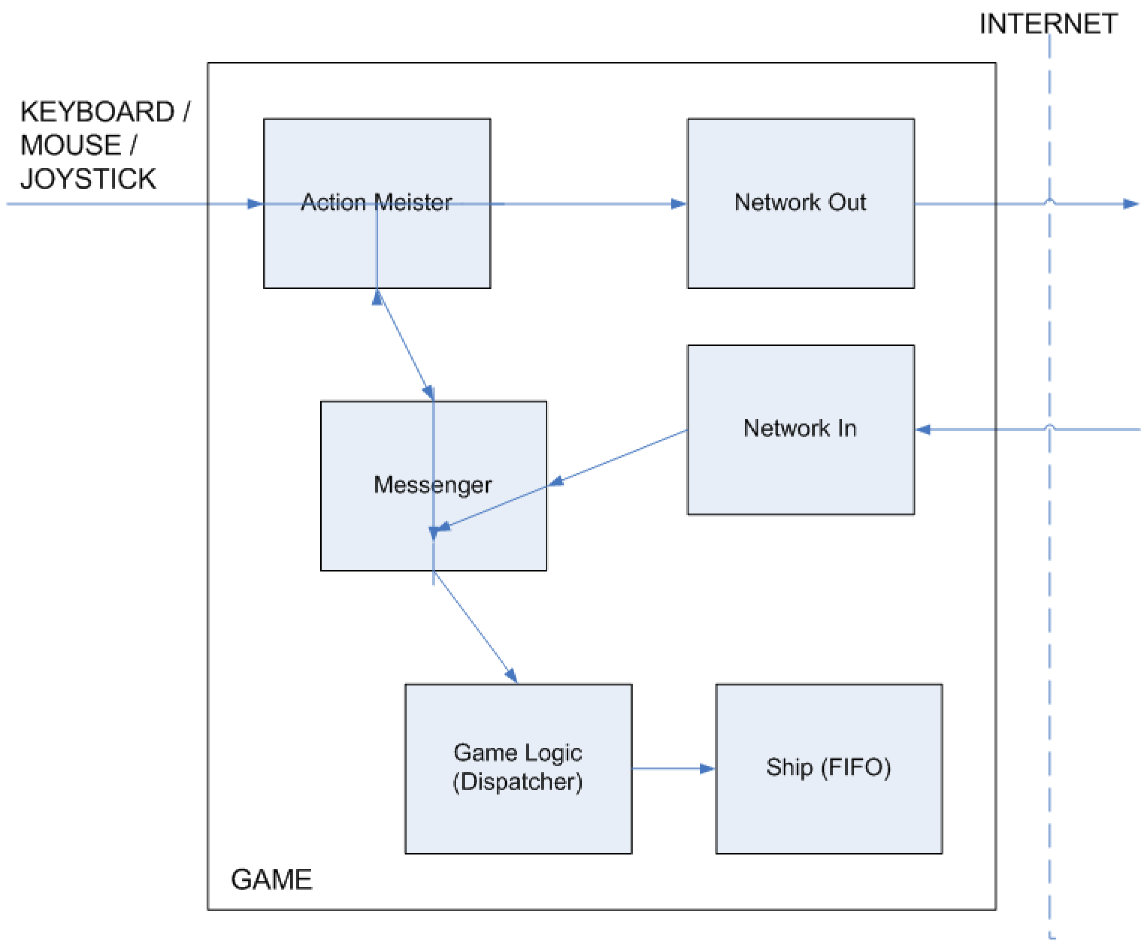


Figure 8: Overall design architecture, highlighting the various subsystems and the flow of messages between them.

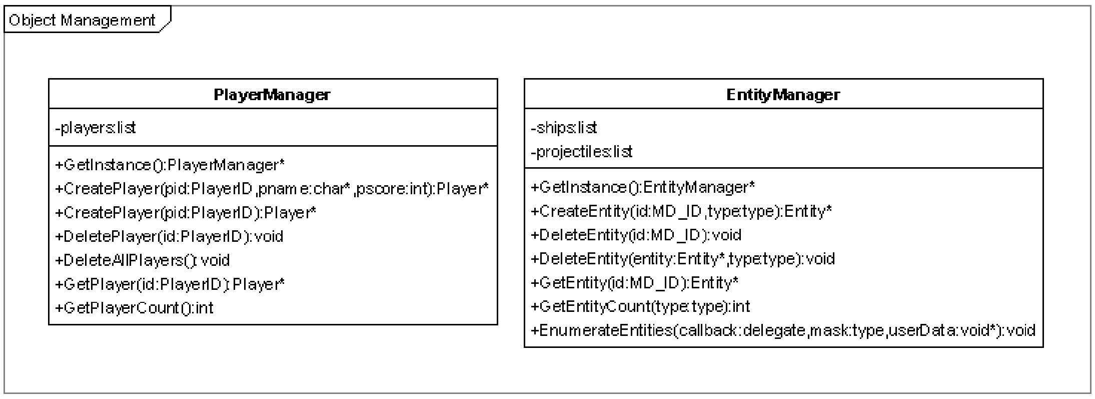

Object Management
PlayerManager
-players:list
+GetInstance():PlayerManager*
+CreatePlayer(pid:PlayerID,pname:char*,pscore:int):Player*
+CreatePlayer(pid:PlayerID):Player*
+DeletePlayer(id:PlayerID):void
+DeleteAllPlayers():void
+GetPlayer(id:PlayerID):Player*
+GetPlayerCount():int
EntityManager
-ships:list
-projectiles:list
+GetInstance():EntityManager*
+CreateEntity(id:MD_ID,type:type):Entity*
+DeleteEntity(id:MD_ID):void
+DeleteEntity(entity:Entity*,type:type):void
+GetEntity(id:MD_ID):Entity*
+GetEntityCount(type:type):int
+EnumerateEntities(callback:delegate,mask:type,userData:void*):void

Figure 9: A class diagram of the object management classes.

the manipulation of game objects from any point within the system.

3.2.2 Messaging System

Core to the game logic is the ability to pass arbitrary messages around the system. This is achieved by the use of a central messaging system which operates in a publish-subscribe manner. Message senders create the message block with the appropriate contents and then invoke a central method, passing it the message block. The central message dispatcher will then iterate its chain of registered clients and pass the message to each one. Client message handlers may prevent the message propagating to any subsequent clients.

Additionally, clients may register themselves with the message dispatcher with a given priority. When a message is dispatched, higher priority clients will recieve the message first. This allows overriding of message handling, should such functionality be needed.

Messages passed through the dispatcher all extend a common superclass, thus allowing the dispatcher to be completely generic. The dispatcher has no interest in the messages it is passing around anyway, so this works well. In order for the reciever to be able to determine what the type of each message it recieves is, there is a field within the message block which contains the type information (encoded as an enum). Receivers may then switch over the message type and handle it appropriately (by dispatching it to a handler function, for example).

As mentioned above, message recipients may also terminate the propagation of messages to any further clients registered with the dispatcher. This is achieved by having the receiving routine return a boolean value specifying whether to allow further propagation. Messages may also be modified by recipients such that any further receivers will handle the modified message. These two features are unused in Z Fighter Forever as there is no need for this functionality.

The implementation of this messaging system was extracted from a library of pre-existing code developed by various members of the group. This allowed rapid integration of the messaging system into the game such that the main effort could be directed at implementing the game-specific logic, as opposed to generic subsystems.

3.2.3 FIFO Queues

The design of the first-in, first-out (FIFO) queue functionality was not something that was undertaken as part of this project as the implementation was taken from a library of pre-existing code written by one of the group members.

It is, however, worth discussing this API here as the use of FIFOs for data transfer is integral to the way in which the software works.

FIFO data structure The FIFO is intended to be implemented as a queue of data blocks of varying size. Each block is intended to have a fixed-size header followed by a variable amount of payload. Each block is assigned a sequence number by the client. This may be any positive integer and the FIFO design places a single restriction on this; the sequence number of FIFO entry n must be $\geq$ to that of FIFO entry n - 1. Data contained within any given block is arbitrary and the FIFO design makes no assumptions about it. Blocks can be equated to network packets, which take the same kind of structure.

One important part of the FIFO design is the way in which ownership of data is handled. Once a block has been inserted into a FIFO, the FIFO itself assumes ownership of the data. Therefore, the client that inserts the data should discard any references it had to the data block prior to insertion into the FIFO. When a block is removed from a FIFO, the block's ownership is transferred to the client that removed it. The FIFO discards all information about the data once it has been removed. Therefore, it is the responsibility of the receiving client to free up the memory occupied by the data block.

FIFO API As the FIFO implementation is pre-existing, this API is unlikely to be modified during development (although extensions may be made, depending upon need).

The FIFO API provides for creating and destroying a FIFO:
struct fifo *fifo_create(void);
void fifo_destroy(struct fifo *f);
When manipulating the FIFO (e.g inserting or removing data from it), the

FIFO handle returned from `fifo_create` must be passed. This allows many FIFOs to be active simultaneously at runtime.

The FIFO manipulation routines are:

void fifo_insert(struct fifo **f*, struct fifo_entry **e*);

struct fifo_entry *fifo_remove(struct fifo **f*);

uint32_t fifo_num_to_process(struct fifo **f*, uint32_t *seqnum*);

The first two of these should be self-explanatory - allowing insertion and removal of FIFO entries from a given FIFO. `fifo_num_to_process` requires a little explanation. This is used to determine how many entries are located at the front of the FIFO given the condition that their sequence numbers are $\leq$ the value passed in as `seqnum`. This allows rapid and non-destructive handling of the case where there are no FIFO entries to process (there is no mechanism in the API at present for pushing removed entries back onto the FIFO).

The remaining API operates upon FIFO entries. Firstly, we have the entry creation and destruction API:

struct fifo_entry *fifo_alloc(uint32_t *seq*, uint32_t *len*, const char **data*);

void fifo_free(struct fifo_entry **e*);

`fifo_alloc` allocates a new FIFO entry large enough to contain the data passed in `data` (the length of which is passed in `len`). The entry's sequence number is set to `seqnum`. `fifo_free` frees the memory used by the FIFO entry provided.

Secondly, we have the accessor methods to allow the receiving client to extract data from a FIFO entry. This is needed as FIFO entries are opaque objects.

uint32_t fifo_get_seqnum(struct fifo_entry **e*);

uint32_t fifo_get_data(struct fifo_entry **e*, char **buf*, uint32_t *len*);

The first of these should be obvious; extracting the sequence number from a given FIFO entry. The second allows access to the data contained within the FIFO entry. If this function is called with `buf = NULL` and `len = 0`, then the value returned will be the buffer size required to hold all the data contained in the specified FIFO entry. The caller is then at liberty to call `fifo_get_data` a second time with the `buf` and `len` parameters suitably filled in in order to retrieve the actual data. This function's API is so designed to guard against the possibility of pointers to freed data persisting (which could be the case if `fifo_get_data` simply returned a pointer to the data block within the FIFO entry). The downside of this is that the process of extracting the data is slightly slower and a little more memory hungry, as two copies of the data will exist at once; one in the FIFO entry and the other in the client's buffer. The assumption is, however, that once a FIFO entry's data has been acquired, the FIFO entry will be destroyed.

3.2.4 Flight Model Encapsulation

The flight model is peculiar to the type of entity being controlled, it is well encapsulated and can thus be re-implemented in different ways if needed. The inertia behaviour is further encapsulated and abstracted from the flight model, removing the error proneness and extra work involved in implementing new flight models.

The purpose of the flight model is to process control messages and use them to manipulate the physical flight parameters of the entity. The physics system (detailed in section 4.5) then uses these to evolve the ship's motion naturally, according to physical laws.

4 Technology

4.1 True Portal Engine

4.1.1 Mandala

Z Fighter Forever uses the Mandala engine, which has also been developed as part of this project, based on a prototype created by Philip Boulain as an assignment for ELEC6024 Advanced Computer Graphics. Mandala is reusable, extensible, modular game engine, with full support for portal-based geometry, extreme customisation of the simulation and rendering processes, and dynamic control binding.

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.

4.1.2 Portals

Portals are, at a simple level, a visibility culling technique. The environment is carved into 'sectors', which are ideally concave, at least to a basic approximation. For example, for a house environment, each room may be a separate sector; for a dungeon, each chamber and corridor. Objects may be placed within these sectors, with their visibility tied to the sector containing them. Between the sectors are placed 'portals': invisible polygons which provide the subdivision of space into sectors. Optimal places for portals are usually in doorways, and from corners of major obstructions such as large 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 frustum. 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 runtime 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.

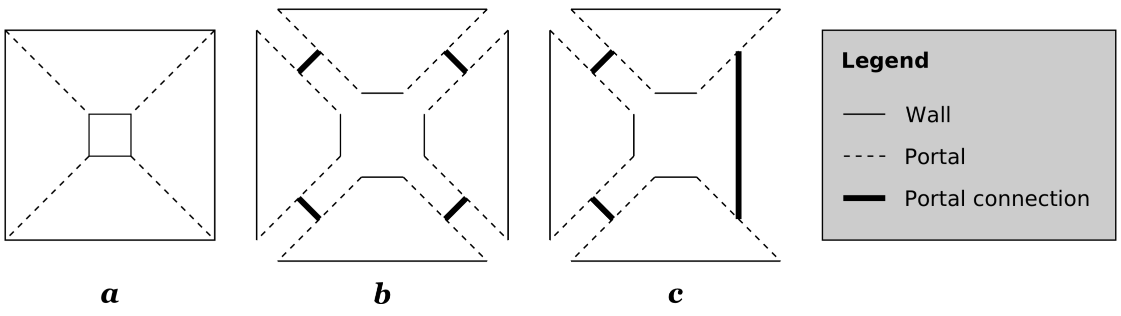


Figure 10: How to create a non-Euclidean triangular room

Figure 10 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.

10a 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. 10b shows an 'exploded view' of how this room can be constructed from separate, yet linked, sectors. In 10c, however, one of the sides of the room has been removed, and the portals either side linked to each other. 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 10c, 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.

Figure 11 further demonstrates this with another example: 'a' shows the actual orientation the sectors in their own, private frame of reference. In order to render this scene correctly for a camera in the left-hand sector, the right-hand sector must be rotated and translated into place, as in 'b'; conversely, the left-hand sector must be transformed if the camera is in the right-hand sector.

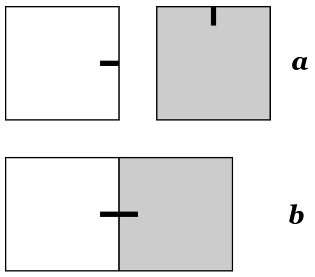


Figure 11: Transforming a sector to correctly align portals

Resultingly, very few engine using portals for visibility are actually what the author dubs *true* portal engines—those which are capable of handling completely disassociated, and thus 'impossible', space. UnrealEngine is a hybrid engine, which normally uses BSP[1], but also provides 'zone portals', which are used to segment ordinary geometry into sectors purely for visibility culling. It does, however, provide a crude approximation of non-contiguous space via 'Warp-Zones' [2]: *sectors* can be linked to each other, effectively linking a single pair of portals connected to them, 'erasing' the sectors as overlap space. This is somewhat of a 'toy' feature, however, largely unused in UnrealEngine games, and is thus poorly integrated with the rest of the engine. Notably, physics-controlled objects such as ragdolls and vehicles will ignore the portal, travelling into the theoretically-unseen overlap space.

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 available at the time of writing; a video trailer released at the gaming conference E3 showed that portal effects were in use, yet presumably these have also been 'kludged' on top of the licensed engine.

[1]Binary Split Partitioning

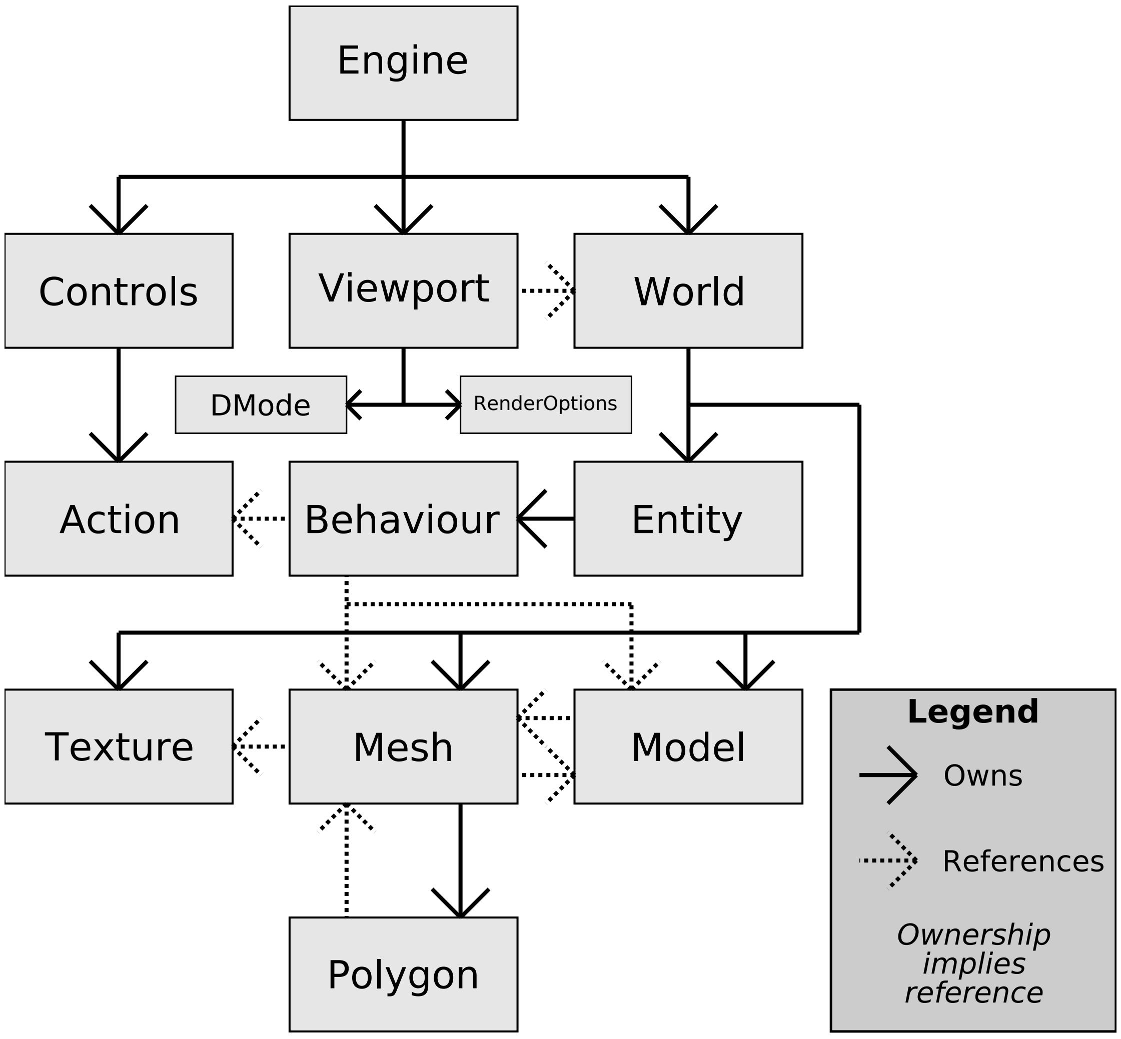


Figure 12: Main classes of Mandala, with memory information

4.1.3 Architecture

Mandala has a highly modular design, and is implemented in object-oriented C, compliant to the C99 standard. Figure 12 shows the memory ownership semantics between the main classes of the system; these are arranged such that shutdown of the engine singleton can cause a complete cleanup of all allocated resources.

In addition to those classes shown, Mandala includes a number of utility classes which implement data structures optimised for performance in their 3D engine role. For example, there is a bag implementation which maintains ordering on addition, supports O(1) addition and removal with memory chunking to reduce heap reallocations, and provides an macro-based iteration interface. This is used extensively as a general-purpose dynamic array, as often strict ordering is not required, yet rapid, efficient removal is. A hashtable implementation is also included for use in caching textures by filename, along with classes usually employed in 3D graphics: matrices, vectors, and quaternions, with a range of utility methods for calculating results such as the dot products of vectors, or for generating matrices from quaternions.

The API is split into public, package local (internal), and private headers, and fully documented. Mandala builds as a shared library (a Perl script was created to parse the public headers into an approximate abstract syntax tree, and from this generate a definitions file for building as a Windows DLL) and can be used to build games other than Z Fighter Forever.

4.1.4 Simulation

Mandala provides simulation abilities which encapsulate the game code in an extensible fashion. The coarsest unit is a 'world', which is a container providing the code to invoke these abilities upon a 'tick'—a configurable quantisation of time. The engine finds worlds through viewports (covered below), and potentially manually-forced ones (e.g. to allow simulation without rendering on a server), and triggers enough ticks each time it is 'stepped' (triggered to act within the main loop) to cover the time taken to render the last frame—that is, Mandala provides reliable, drift-free rate control of ticks.

Worlds own the memory for meshes, models, and textures, and provide a mechanism to load the latter two. Models are simply collections of meshes, and meshes are simply collections of polygons with a single texture: they also provide extensive caching of various data for speed. Textures are loaded via world so as to be shared across meshes; requesting to load an already loaded texture will simply return a reference to the existing texture object in memory.

Worlds contain entities, which are quite simply any kind of item with a presence—physical or visual. Entities form a parentage tree, and have positions and orientations relative to the parent entity. Those without a parent are 'top-level entities', and are functionally sectors. Entities have zero or more 'behaviours', which are the mechanism by which they influence the world; behaviours themselves consist of a set of 'facets', which are the various aspects of that behaviour. Behaviours can either be created from a set of stock types—such as "display as model", "collide as mesh", "light"—or created as custom behaviours from function pointers and a data pointer. Facet existence is cached at an entity level (which dispatches facets to the individual behaviours) to reduce function call overhead. The facets of a behaviour are:

Act A potentially parallelisable facet where entities can modify themselves, but not their peers. Movement takes effect immediately after this step.

Interact Entities are now permitted to interact: for example, for a projectile to home in on a target.

Display The entity should render itself: the renderer has already prepared OpenGL so that the behaviour only has to render the item at the origin, facing $-Z$, with $+Y$ as an upvector. This behaviour also returns to the renderer any portals which must be followed, by adding them to a provided bag structure—this will be covered in more detail below.

Pre-display Indicates to a 'relevant to environment' behaviour (see below) that they are now relevant: this is, for example, where lights should turn on.

Post-display As pre-display, but for when relevance is ending: lights, for example, should turn off.

Traversed Triggered when an entity crosses a portal. As Mandala itself has no understanding or dependence on the internal state of a behaviour, it cannot automatically transform any geometric data into the new co-ordinate system. This facet provides the behaviour with the necessary information to perform any such transforms itself.

Collision Detect a collision against an incoming line segment with a given radius. This is covered in more detail below. *Handling* collisions is the role of the 'act' facet. The first facet to detect a given collision terminated detection, effectively making multiple behaviours act like a short-circuiting logical OR test.

Destroy Indicates that the behaviour must now deallocate its resources, as the entity it belongs to is also being destroyed.

Behaviours may also have two flags, which are unified at the entity level: 'relevant to environment', and 'relevant in radius'. The former of these states that an entity should be considered relevant to all entities within its radius, regardless of visibility. This is useful for lighting: even if the light itself is not directly visible, it should still affect any visible entities within its range.

'Relevant in radius' is more applicable to sound, and is not currently fully designed or implemented. It indicates that an entity is relevant if it is within its radius of the camera entity, even if it is not visible: for example, a sound which is occurring behind the player.

It is possibly to set the position, parent, orientation and such of an entity. However, to perform normal movement, the method MoveBy (or WarpBy, which ignores all non-portal collisions) should be used, as they store a movement vector which the world later uses to correctly move and collide the entity. Rotation is currently safe to modify directly, and there are a number of utility methods for applying natural, relative rotations to the entity; along with accessors to get its local axes.

Collision detection is only supported for the top two levels of the entity tree—sectors, and one level of non-sector entities—for reasons of computational and implementation complexity. When ticking, the world iterates over the entities and moves and collides them. To do this, it generates a line segment, which is simply a start vector and an offset, compensates for the relative movement of the other entity, then dispatches to the entity's collision detection behaviours. The entity's radius is also passed, effectively simplifying it into a sphere collision over time for the potentially impacted entity to detect against. In theory, a behaviour could use the line segment to guarantee that the movement never intersects the impactee; in practice, however, this is often an exceedingly computationally complex problem, isomorphic to detecting four-dimensional hypercylinder intersection. As a result, the only stock collision behaviour currently in Mandala which uses this is mesh collision (and, thus, model collision), as polygons can reduce it down to plane-crossing test. This process is then repeated for any siblings of the moving entity, with line segments which are transformed into their co-ordinate systems.

When invoking the collision detection behaviour, the world provides a collision structure to populate if there is an impact. If so, the world will fill in useful fields, such as the other entity involved in the collision, then store the structure in the moving entity's collision bag. The entity's 'act' behaviours can use these collisions, after which they will be automatically discarded; its movement for this tick is also cancelled, to prevent it penetrating into geometry and becoming stuck. The structure may indicate that a 'detailed' collision is desired if a configurable flag on the entity is set to request such: in this case, the collision behaviour should provide information such as the surface normal at the point of impact, and the precise offset of the impact.

A field is also set should the collision be with a portal polygon. In this case, the entity traverses the portal into the target sector, and has is 'traversed' behaviour invoked so that it may compensate for the new co-ordinate system. No collision is recorded: from the entity's point of view, portals are just like any other section of empty space.

4.1.5 Renderer

Mandala supports run-time display mode modification and arbitrary screen subdivisions, through the DMode and Viewport classes. DMode is simply a representation of a screen mode and is used to tell the Engine singleton which mode to switch to, complete with the reinitialisation of OpenGL required on platforms incapable of maintaining the context across window changes[2].

Viewports represent proportional areas of the available display which are dedicated to rendering worlds; they also specify which entity acts as the camera, which may be a 'dummy' entity whose only role is to be attached at some offset from another. Viewports also encode information about the view frustum, including the near and far clipping planes, and a 'render options' structure which allows for miscellaneous metadata about the rendering process to be expressed to assist in debugging. Figure 13 demonstrates this.

As covered above, the actual rendering of entities is abstracted out. When Mandala steps and requires viewports to render, they set up the projection matrix if needed (a caching mechanism bypasses this stage for single viewports once initialised once), disable all lighting and deallocate all lights[3], and clears the depth (and colour, if requested) buffers for their area. Assuming that a camera exists, rendering proceeds by recursing up the entity chain from the camera, applying reverse GL transforms until the top-level entity containing the camera is reached (i.e. the current sector). This effectively 'positions' the camera by applying the inverse of that position to the level.

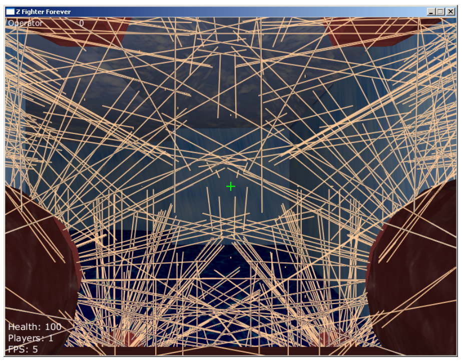
Ideally, the renderer would now calculate which entities are relevant to their peers by the 'relevant to environment' criteria; due to time constraints, the current version of Mandala assumes relevance based on visibility to the camera. For the correct algorithm, see the lighting design document in the appendix.

The meat of the renderer is the dual-recursive subtree-rendering stage, starting from the identified current sector. Figure 14 demonstrates what the subtree renderer is attempting to achieve: the entity tree under the current sector must be rendered, and any portals within it must be followed. This then triggers rendering of the target sector, which may potentially have been rendered before, but in a different relative position. Calls to the subtree renderer also push and pop the OpenGL matrix stack.

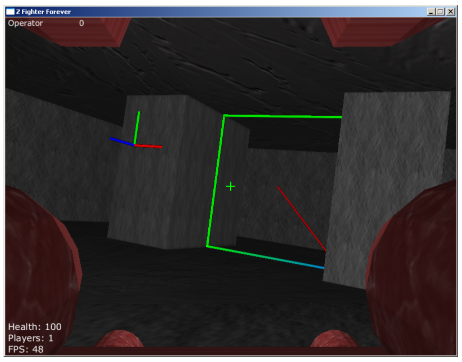
To protect against infinite portal recursion (e.g. an infinite corridor, where one end links to the other), each entity has a render count; the subtree renderer

[2]e.g., Microsoft Windows

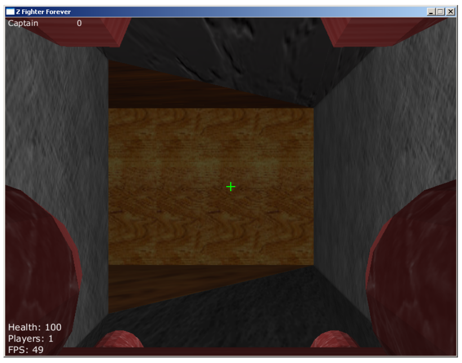
[3]Rather than using GL lights directly, entities must ask the viewport to allocate them one—this allows intelligent re-use of lights up to the GL implementation's limitation, at which point the viewport falls back to reallocating the zeroth light. Lights which are deactivated again must release their allocated light.



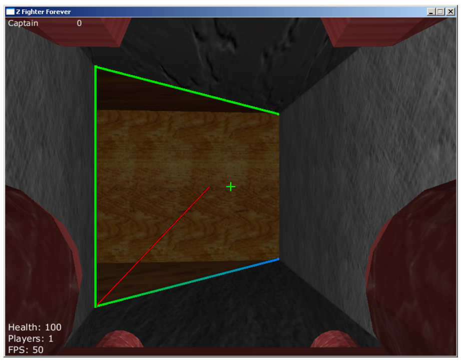
(a) Polygon normals



(b) Sector axes: red, green and blue map to X, Y, and Z respectively



(c) A portal, rendered normally



(d) Debugging portal view: red line indicates centre and anchor; cyan vertex is zero

Figure 13: Support for debugging visualisations

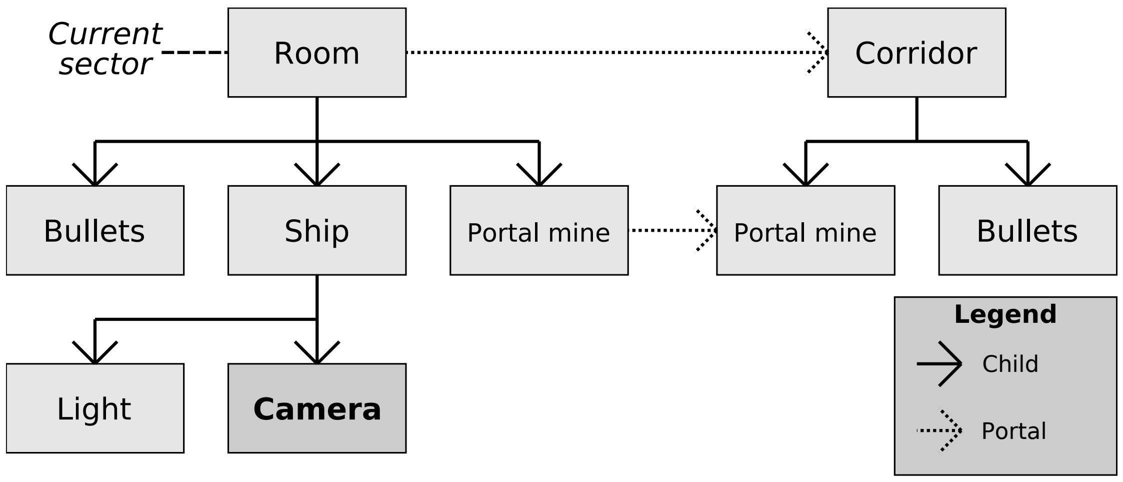


Figure 14: The entity tree with overlaid portal graph

first increments and tests this against a configurable limit. Next, the forward entity transform is applied, to move the co-ordinate system into that of the entity from its past, parental state. In the case of start point (the camera's current sector), the previous co-ordinate system is simply the OpenGL identity matrix. All entities which are relevant to this one (e.g. lights) are now told to pre-display, then the entity itself, which also retrieves the set of portal polygons required to complete rendering. In the future, these two stages should be separated so as to render portals first, then the entity, allowing correctly for geometry through portals which is further away than overlapping geometry in the current entity.

If the entity returned portals to render, relevant entities are post-displayed again, to disable lighting which may not be applicable to the portal's target sector. Then, for each portal, the following sequence of actions must be taken:

1. A visibility test is performed by rendering an untextured version of the polygon, with writes to the depth buffer disabled, in `GL_SELECT` mode; if this indicates that the portal is being clipped away (e.g. is not within the view frustum), rendering for this portal is aborted.

2. A stencil is created in the stencil buffer matching the visible section of the portal, by again rendering the polygon's shape with depth and colour buffer writes disabled.[4]

3. *Planned improvement.* Entities within the current sector which are likely

[4]This strictly fails to clip portals by any previous portals in the recursion chain required to reach them. A more correct algorithm would either restrict the stencil buffer in this manner or, better, form a clipping stack using separate stencil planes.

overlapping the portal (radius comparison) must be rendered, else they may be clipped away later when the portal is depth-sealed.

4. The matrix stack is pushed, the portal's co-ordinate is transformed, ready to render the other sector, and a recursive call is made to the subtree renderer. It is provided with a bag structure for passing back entities, which will be covered later.

5. The matrix stack is popped to return to the current sector, and the portal is sealed by again rendering the portal's shape, this time only writing to the depth buffer. This prevents geometry in the current sector from appearing in the other sector in the case of overlapping space.

6. Stencil clipping is disabled.

7. *Planned improvement.* Entities which have been back-passed by the bag are rendered.

Relevant entities are then re-enabled, and rendering continues on to children: making recursive calls. If recursion to the current entity was caused by following a portal, those children which are found to be within their radius of the portal's plane are added to the pass-back bag. This is required to render any fragments of entities which pass through the portal from the target sector into the current one, and expand beyond the bounds of the stencil clipping.

Immediately prior to and, separately, after the main recursive phase of the rendering, a single entity may be displayed if set. This provides a mechanism to trigger a set of behaviours either side of the rendering: the post-rendering entity is used to display the HUD after all geometry, using a glOrtho projection matrix with the depth test disabled. The pre-rendering behaviours could be used to, for example, configure fogging, or configure pixel shaders.

4.1.6 Entity Tree and Portal Graph Transforms

Mandala represents linear quantities such as position and velocity using 3-vectors; angular quantities such as orientation and rotational speed are stored as quaternions. While vectors are sufficient for representing directions in space, they are inappropriate to representing three-dimensional orientations as they fail to specify a "roll' component. The alternative is to use an orthonormal matrix to represent the rotational transform between global (i.e. sector) and local (i.e. entitiy) co-ordinates. Matrices allow easy composition of rotations through multiplication, but are computationally expensive.

Quaternions are a four-dimensional variety of complex numbers which have a unique applicability to representing rotations in three-dimensional Euclidean spaces in ways largely analogous to the matrices they supercede. A quaternion comprises four components, w, x, y, and z, where w is the real part and x, y, and z the imaginary part.

Quaternions representing rotations are invariant under linear scaling, and therefore are less easily degraded through rounding errors than matrices. Multiplication of quaternions is non-commutative in the same manner as matrix multiplication, and represents the same composite rotation.

Unlike matrices, quaternions relate relatively directly to the equivalent axis rotations (another method for representing arbitrary resultant 3-dimensional rotations). Quaternions are easily generated from single-axis rotations. This allows the use of generic basis quaternions in control systems and flight models to apply local rotations to objects.

Local co-ordinate definitions The transformations required to map objects between the co-ordinate systems of Mandala sectors are performed in terms of subtransformations to and from *face* co-ordinates: those peculiar to a given polygon, and invariant between geometrically similar polygon definitions. Mandala's portals define their local Z axis as the face normal, and use an agreed definition of the local X axis which places a labelled point (the "anchor") on the positive X axis. The local origin, O, of a face is defined as the geometrical mean node position, and the local Y axis, coplanar with the X axis, is defined as the cross-product between the normal and local-X vectors. A right-handed system, appropriate to OpenGL, results.

Portal definitions In order to create a portal, two polygons are effectively placed back to back by the engine; therefore, a prerequisite for flush interfacing is that the given portals are congruent through three-dimensional rotation. In terms of the shapes of the portal polygons, this means that when viewed from the same side, they should appear congruent through reflection. A further requirement for correct interfacing is that the portal's polygons, once placed back to back with matched face origins, be rotated to geometric congruence. While computed-geometry approaches to this problem exist, not all polygon shapes are sufficiently asymmetrical to produce unique solutions; therefore, it was deemed inappropriate to attempt a runtime approach.

Instead, it was decided to define face-relative roll in terms of the anchors: by labelling correspondent anchors on each polygon, a complete spatial mapping may be defined in terms of the matrices $F(s)$ (the to-face-coordinates matrix for the source polygon s) and $F(d)^{\mathrm{T}}$ (the inverse [from-face-coordinates] matrix for the destination polygon d, the transpose of matrix $F(d)$. One correction remains if objects are to translate correctly: in the presence of a common X axis and antiparallel Z axes (normal to the respective polygons), the polygons' Y axes will necessarily lie antiparallel. This effect will appear to produce a 180°rotation in objects mapped through any given portal. The solution is to introduce a compensatory artificial rotation of 180°between $F(s)$ and $F(d)^{\mathrm{T}}$, implemented as a negation of the face-relative Y and Z co-ordinates (represented by the diagonal matrix C).

Inter-sector transformations Therefore, the function, $M(s,d)$, to map a vector from the sector containing polygon s to that containing polygon d, where s connects to d, is defined thus:

$$M(s,d) = F(d)^{\mathrm{T}} \cdot C \cdot F(s)$$

A vector V may therefore be transformed between the two co-ordinate systems through multiplication by this matrix:

$$M(s,d) \cdot V$$

A point P is transformed by pre-translating into face-origin-relative co-ordinates and post-translating to sector-origin-relative co-ordinates, modifying the above to:

$$M(s,d) \cdot (V - O(s)) + O(d)$$

The problem remains as to how an entity's orientation, represented by a quaternion Q, is transformed through a portal. The solution is to derive a separate quaternion G for each portal face, representing the to-face rotation encoded by the matrix T; the derivation is performed via an obscure matrix-to-quaternion conversion algorithm based on matrix-trace analysis[5].

The following multiplication, then, transforms an orientation quaternion through a portal:

$$G(s) \cdot Q$$

Entity-relative transformations Since Mandala implements sectors as modified entities, making constructs such as linked pairs of portal-mines theoretically possible, the possibility of nested entities appears, creating the need for conversion of co-ordinates between parent and child entities. This problem is materially identical to that of defining to- and from-face co-ordinate transforms: the only difference is in the definition of an entity's origin and axes.

[5]`http://www.ecs.soton.ac.uk/~prb102/junk/notmine-matrixandquaternion.htm`

Each entity defines a local co-ordinate system, for purposes of mesh rendering and collision, that places the entity's local origin at its position within the parent entity and the negative Z axis in the facing direction; local "up" corresponds to Y and local "right" to X. The generation of a set of parent-relative axis vectors from an entity's orientation quaternion is trivial, and corresponds directly to the definitions given for face-relative axes. The conversion operations are otherwise exactly as given above.

4.1.7 Control Abstraction

When stepping, as well as rendering and running ticks, Mandala pumps the SDL event queue. Some of these events are mundane, such as correctly handling the closure of the window. However, many are input devices.

Mandala provides a complete control abstraction layer which converts a range of SDL input events into a consistent representation: an 'action' which has a value from -1.0 to 1.0. Multiple input devices may be bound to a particular action, and the game code does not need to know about the details. For example, a keyboard key, which has a value of 1.0 when down, and 0.0 when up, may be bound to the 'move forward' action; as may a joystick axis, which returns values in the range -1.0 to 1.0. A per-binding sensitivity/scaling system allows a different key to also be bound to this action, but with a value of -1.0 when pressed. The result is that game code can simply interpret this single 'move forward' action's value as an amount to move forward (or backward, if negative) by, and it will just work, whether the user uses the keyboard or joystick.

A state machine is maintained on actions which provides a way to detect that an action has been 'pressed' (taken non-zero) without concerns of missing the exact tick during which it happens. This provides a reliable and responsive mechanism to detect commands such as 'open in-game menu', even if the machine is under heavy load.

SDL input events can also be diverted, which disables Mandala's control system in favour of sending them to a custom callback, including mouse capture. In Z Fighter Forever, this is used to provide KUI with the events needed to interact with the interface; when leaving the menu, KUI disables the callback, and Mandala resumes command of the controls.

4.1.8 Testing

Mandala was tested using a test harness which contained trivial, manually defined geometry, a loaded model, and a set of spherical entities used to test various aspects. Additionally, it can perform numerical, black-box testing of some of

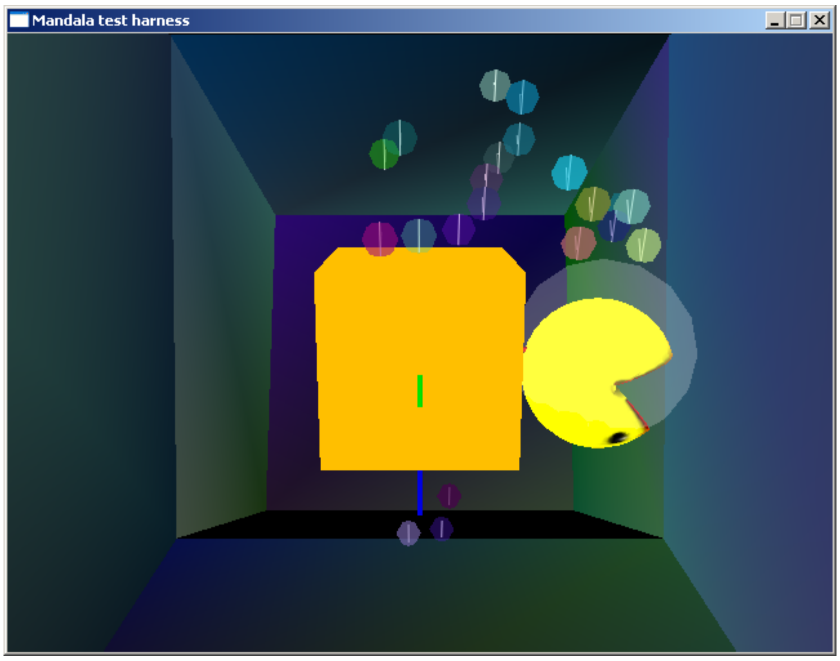


Figure 15: Mandala test harness

Mandala's functionality: this was mostly used to verify the transforms without the additional complications of simultaneously attempting to debug the renderer.

Figure 15 shows the test harness in a late configuration, with the spheres acting as 'missiles', firing towards a second sector via a portal, and colliding with a cube and rotating Pac-Man.

4.2 Level Loader

Rather than manually defining geometry, as for the Mandala test harness—a tedious and error-prone process—Z Fighter Forever has a runtime level loader. Levels are defined in a simple, directive-based format, not unlike a simple procedural program. In fact, the level loader does actually contain a complete lexer (tokenisation on whitespace, with special rules for statement terminators) and parser/interpreter.

In general, attributes are set up for geometry, and are 'sticky': if it is specified that the texture is 'stonewall', then all future meshes use that texture. Points can be added to polygons, and polygons may be defined as portals, which are connected by load-time string identifiers. The level loader builds a set of STL-based data structures to convert these mappings into Mandala-level portal connections after loading has completed, to avoid the necessity for forward declaration, and facilitate circular references. There is also a ‘syntactic sugar’ approach, used in the]|[space level, which builds ‘rooms’: 2.5D structures generated from a sequence of wall points, and a floor and ceiling height.

Finally, the level loader is also responsible for selecting spawn points, as they are encoded in the level. The server can simply request a spawn location (sector and position), and the level loader will select one randomly.

For the full level format definition, see the appendices.

4.3 GUI

4.3.1 Alternatives

The development of a full GUI widget set was borne out of the unavailability of a good, modular, OpenGL widget toolkit for the game interface. Many toolkits were looked at and tested but tended to have one or more of the following undesirable attributes:

- Not designed for games or flexible enough to be integrated into one.
- Depended upon many 3rd party libraries.
- Poorly documented.
- Bloated and too generic.
- Some or all of the toolkit was not cross-platform.

Two promising GUI toolkits were tested more thoroughly: Crazy Eddie’s GUI, and SXML GUI. The former was highly regarded, well documented, and versatile but was impossible to integrate cleanly. The latter was barely documented, but its clean API and XML-described screen/page format compensated for this problem. Unfortunately, a critical bug was found in the page-changing procedure and when the author of the toolkit failed to respond to the problem, the use of it was abandoned.

4.3.2 KUI

Instead, a clean, object-orientated, widget toolkit of our own was developed. The toolkit was named “KUI” with the only external dependencies being FreeType (for font rendering) and FTGL (for FreeType rendering within OpenGL). The widgets inherit from the base class KUIWidget, as can be seen in figure 16. Almost all of the other widgets use a combination of the KUIIcon and KUILabel widgets to draw images and text respectively. KUIIcon loads and renders images as OpenGL

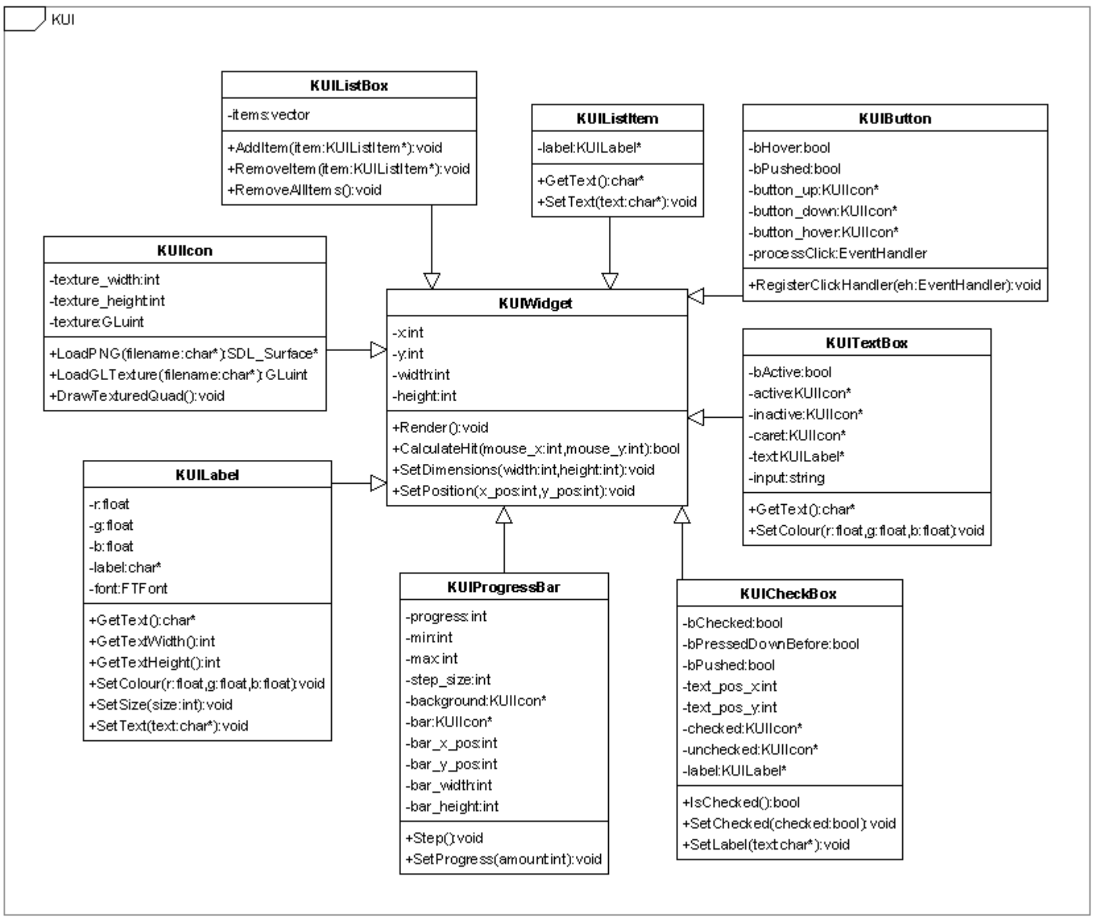


Figure 16: A simplified class diagram showing the variety of KUI widgets available and how they all extend the KUIWidget base class.

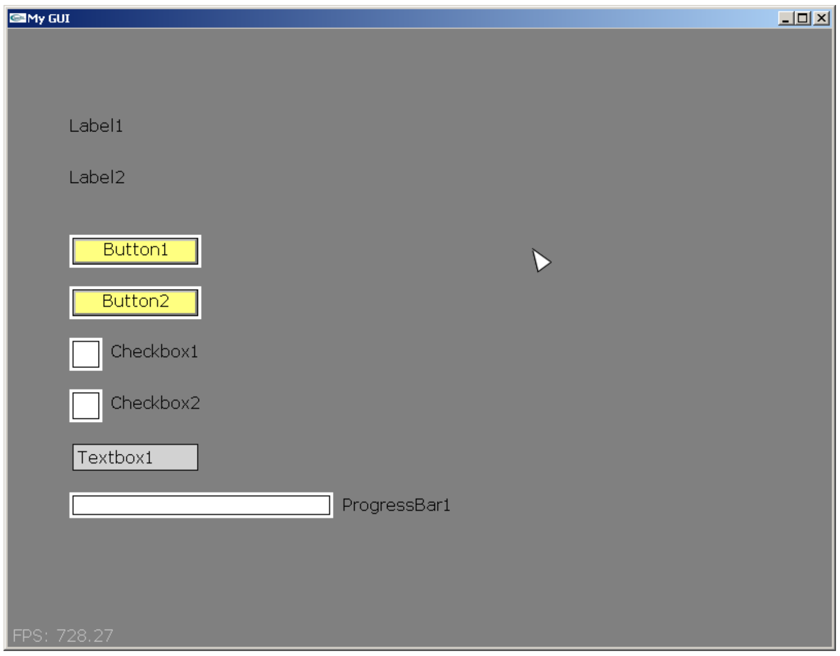


Figure 17: The test harness within which all of the KUI widgets were developed.

textured polygons, making the entire GUI very fast - for example, the test harness (see figure 17) ran at ˜730 FPS.

KUI has the concept of 'pages' within which all of the widgets for that screen are contained; this allows the entire UI to switch in an instant by choosing to render a different page. The page is responsible for the rendering of all of its widgets (the RenderAll() method of the page calls the Render() method on every widget), as well as keyboard and mouse event propagation (with CalculateKeys() and CalculateHits() methods appropriately). Finally, a page-wrapper class can be created to offer page-specific functionality (e.g. the HUD contains methods like SetHealthValue() SetPlayerCount(), etc.).

4.3.3 In-Game

The game keeps track of the many pages it has and switches between them, giving or taking authority from Mandala over the keyboard/joystick/mouse messages as appropriate. Figures 18, 19, 20, 21 show the variety of pages within the game. Notice the alpha-blended background in figure 21, and the custom mouse cursor and hover state of buttons in figures 18 and 21. With regards to the FPS count, the game is limited to a maximum of ˜50 (as seen in figures 18 and 21) while the low FPS count on figures 19 and 20 is due to the screenshot-taking machine running a server and 3 clients at the time.

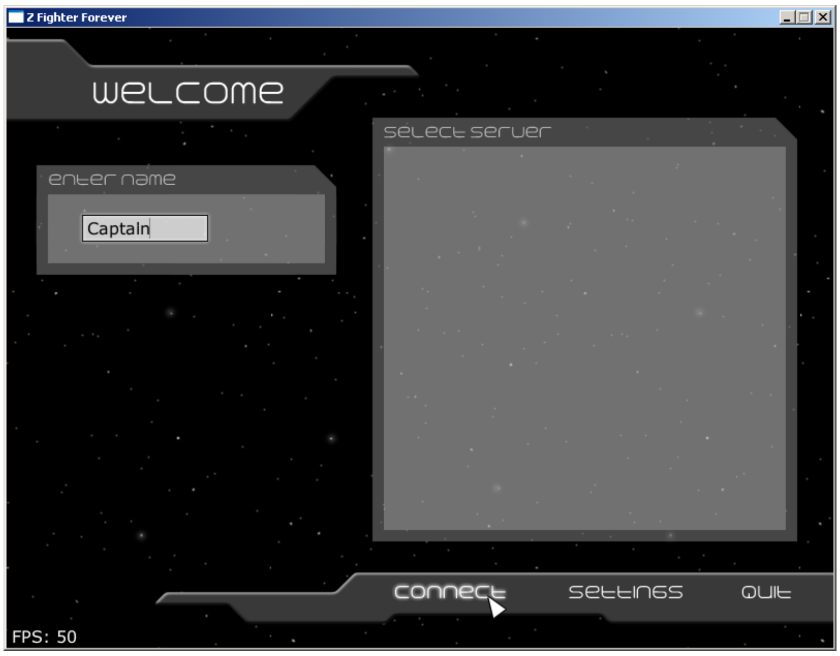


Figure 18: A screenshot of the main menu screen.

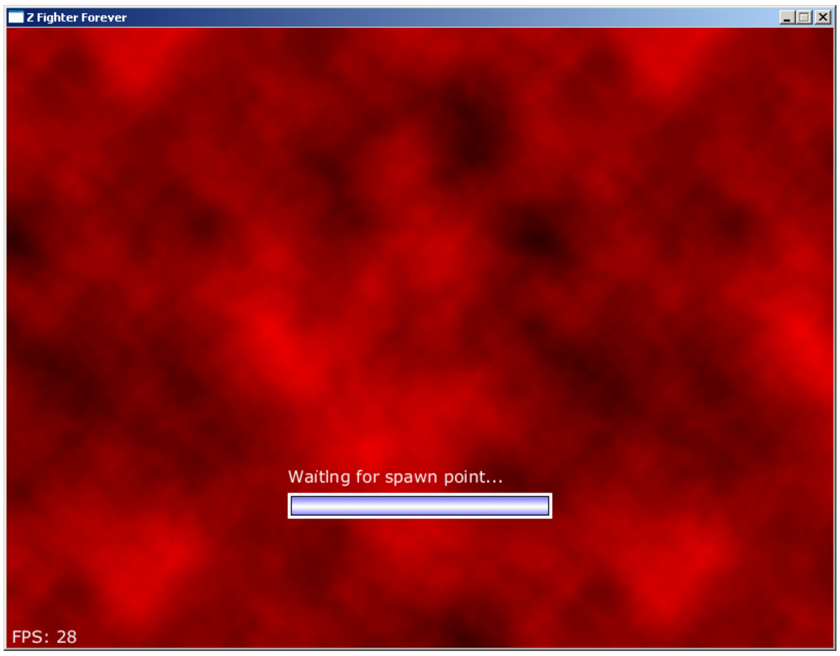


Figure 19: A screenshot of the connecting screen.

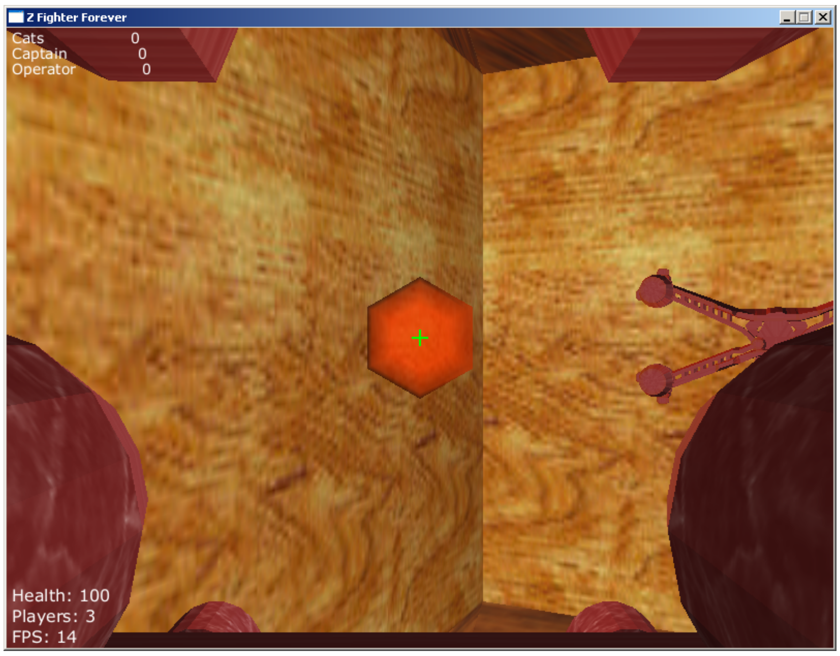


Figure 20: A screenshot of the game screen.

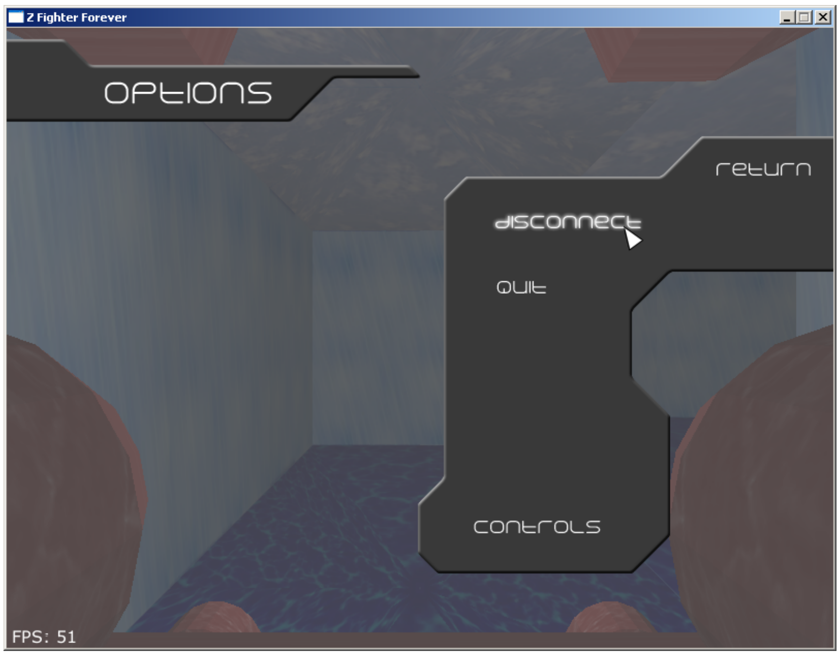


Figure 21: A screenshot of the in-game menu screen.

4.4 Networking

4.4.1 Basics

For a twitch game such as Z Fighter Forever, TCP is unsuitable for the multiplayer networking for many reasons: packet header overhead, long timeouts, etc. UDP, however, does not offer the necessary features that TCP would otherwise provide: reliable packet delivery, stateful connection/disconnection, etc. The solution is to implement these features as an abstraction layer above any UDP-related function calls. RakNet is a cross-platform library, self-described as a "multiplayer game network engine", that provides a lot of the networking basics needed. It uses UDP but provides the best of both worlds, including:

- Connection / disconnection.
- Various packet priorities.
- Various packet reliabilities.
- Synchronised time information.
- Unique player id generation.

RakNet provides functionality for peer-to-peer and client-server architectures. For Z Fighter Forever, we chose the client-server architecture as used by similar action games. The core idea of this architecture is for enough messages, in frequency and content, to be sent between clients and server such that an authoritative game state is present on the server, with as accurate a representation as possible (of this game state) present on the clients. A full listing of these messages, complete with their reliability, priority, direction, and content can be found in appendix D.

Globally-addressable entity IDs are generated server-side. These act much in the same was as the synchronised timestamp generation that RakNet handles, together they both allow every client and server to reference the same entity or point in time in the same way.

These timestamps are typically used when sending important messages such as control inputs or state updates. Control input messages contain a client's changes to yaw/pitch/roll, acceleration, deceleration, braking and firing. These are fed into the flight model for simulation, updating a ship's internal state indirectly, and could be considered relative information. State update messages contain the physics and firing state of every ship entity in the arena, these messages could be considered absolute information, and they are used to update every entity's internal state directly. Projectile entities do not need explicit replication over the network, as these can be deduced from a ship's firing state combined with its position and orientation at the moment of fire.

The following sections describe the original network protocol, and the revised network protocol due to time-constraints. Damage, death, and spawn information replication are constant between these different implementations, the processes are also rather straight forward and can be deduced from appendix D. Ship movement, the flow of control input and state update messages, is the main difference between the 'theory' and 'practice' sections.

As a note, both public and private chat messages are supported in the network protocol, and largely implemented in the game logic of both client and server, but the user interface presently provides no method of textual input so this feature is currently unusable.

In the next two sections, the term 'simulation' is used to refer to the immediate input of information into the physics model. The term 'resimulation' is the process of rewinding the game state, applying the input information at the correct time, and simulating from the past to the present with this new information. The latter will occur frequently given the fact that, in effect, every message received will always be out-of-date. Further details of this process have already been described in section 1.3.1.

4.4.2 Theory

From many networking articles read [3, 4, 5, 6, 7], the following ideal networking protocol was devised. The procedure is as follows:

- Control input messages are sent client-to-server when a player's control input state has changed. Thus, in the case when the player is doing nothing, there is no network traffic; and without position information is hacker unfriendly. These messages are also replicated locally and immediately applied to the player's game state.
- The server receives the control input messages and uses them to re-simulate its game state. The server has authority, and with resimulation should have an accurate representation of every client at any point in time.
- The server then, at timed intervals, broadcasts state updates to every client connected client.
- These state update messages are then received by the client, who uses them to re-simulate the state of every entity present in the message.

This procedure can be improved further by the following:

- The information in a control input message contains may be represented as an initial bitfield within the packet, allowing multiple state changes (e.g. roll, yaw, and fire) at once to be encoded within one packet.
- Data types may be casted shorter to save a packet's size in bytes.
- The dead-reckoning algorithm can be used to only send updates to the clients when a ship has moved away (from the position the clients think they are) by a pre-defined range.
- Smoothing/blending algorithms (e.g. linear, exponential, or cubic splines) can be used to improve, visually, the correction any state update may make to the client's state.

In practice, only a subset of this procedure and these features could be achieved. Time constraints, created by, among other things, the critical-path nature of game subsystem development, meant that there was not enough time to implement everything we wanted. Server-side resimulation proved to be more complex than expected and a lot more difficult than client-side resimulation. Combined with the time constraints, server-side resimulation was abandoned in favour of purely client-side resimulation.

4.4.3 Practice

The abandonment of server-side resimulation had a big impact on the other areas of the networking procedure. This was due to the fact that, with locally replicated control input messages, the client and server would now be applying these control state changes at different times. Presented with a choice between being heavily out of sync but with smooth gameplay movement, or a synchronised client-server state with potentially jerky controls, the latter was chosen in the hope that, together with reducing the scope to LAN-only, the networking model would still be playable.

The procedure is now as follows:

- With the removal of server-side simulation, client and server would apply a control input message at different times - resulting in wildly out-of-sync game states. The control input messages therefore should not be replicated locally.
- Instead, when the server receives the control input message, it re-writes the packet's timestamp, applies it to the simulation and sends the message out to every client, including the client it came from.

- The client waits for the remote echo of its control input message, with its timestamp re-written to server time, then applies it locally with resimulation. This server approval is the reason for the aforementioned jerky controls, and LAN-only limitation.
- The server then, as before, at timed intervals, broadcasts state updates to every client connected. The state update messages are received by the client and processed as before.

Far from ideal, but a playable simulation to some degree. With more time, not only would the ideal procedure mentioned above be implemented but with consideration to several lessons that were learnt during development:

- Unreliable messages are preferable to reliable ones when possible (no resending needed).
- Sequenced information is better than unsequenced information when possible (old messages received can simply be discarded).
- Fewer larger packets are better than more smaller ones.
- Handling of timestamps, with regards to resimulation, is difficult and CPU-spike inducing.

An implementation of the ideal networking procedure, with consideration to the above lessons learnt, should produce a vastly superior networking model.

4.5 Physics

4.5.1 Flight model and physical simulation

Physical simulation Z Fighter Forever extends Mandala's support for entity positioning and orientation by adding variables to the client-side Entity class to store linear and angular velocity respectively. Further, a behaviour is automatically registered that provides for automatic procession of the entity's position and orientation over time.

Physically accurate frictionless elastic rebound behaviour is made possible by Mandala's provision of collision normals in the colliding entity's co-ordinate system. Converting this to sector co-ordinates gives the direction of momentum transfer for the collision. The following equation then gives the closing velocity, S, of the collider on the system's centre of mass:

$$S = C \cdot \frac{M_2}{M_1 + M_2}$$

where C is the relative speed of the collidee (the entity imparting momentum), M_1 is the mass of the collider, and M_2 is the mass of the collidee. It is then possible to derive a rebound momentum delta, R:

$$R = S(1 + E)$$

where the unity term represents the momentum delta required for purely inelastic collision, and E is the coefficient of elasticity.

Having determined the momentum delta, it is subtracted from the collider's velocity in the direction of the collision:

$$V_{n+1} = V_n - R \cdot N$$

where N is the collision normal.

Controls and flight modelling Z Fighter Forever's client-side Ship class encapsulates a flight model, responsible for tracking control state and updating position and/or velocity to reflect that state.

Member variables store the current control state, and update this information whenever control messages are received. Meanwhile, this control state updates the parameters of the physical model, providing thrust modelled on an annihilating Bussard ramjet design, with a top speed limited by the maximum expulsion velocity of the interstellar medium from the engine exhaust, as asymptotically increasing perpendicular intake speeds limit the power output of the engine; this operation is materially similar to that of a turbofan engine.

5 Development Environment

5.1 Setup

The development environment used to develop Z Fighter Forever has evolved from that designed and produced for the Real-Time Editor GDP project. This brought together a number of different resources under one, unified, system. This has the advantage of abstracting away differences between platforms; providing a reusable utilities source tree, which contains useful functionality written in C and C++. The build system was explicitly designed to allow compilation of the same source code under both Linux and Microsoft Windows operating systems. A Mac OS variant of the system is planned.

In order to aid the goal of multi-platform development, resource libraries which are known to be portable were chosen. This removed the need for cluttering the game source tree with platform-specific code. Under Linux, the GCC compiler was used (in various incarnations, ranging from version 3.3.x through 4.1.x). Under Windows, Microsoft's Visual C++ compiler was used. This variety in compilers used allowed us to ensure that the code written is as portable as we desired.

One deficiency of the current system is that it has separate build-control files; the Linux build uses a Makefile whereas the Windows build uses a MSVC project file. At some point in future, it would be useful to combine these into a meta-build file which generates the relevant build-control files on the fly.

6 Future Work

6.1 Release Preparation

Although Z Fighter Forever is playable in its current incarnation, there are a number of issues which need addressing before a proper release could be made. Some of these issues are due to incomplete (on unimplemented) engine features. This section considers only game-side features, engine deficiencies are documented elsewhere.

The current implementation is particularly network-traffic heavy and is likely to result in an unplayable game on anything less than a LAN. No testing of this has been undertaken, but such degradation in performance would not be surprising, given the amount of data transferred. There is a large amount of optimisation which could occur in this area.

Due to time reasons, the current resimulation mechanism is suboptimal, in that it requires control inputs to be sent to the server and only processed when the server echos the message back. The system is designed such that control inputs may be processed locally (and, indeed, this is implemented; controlled by a compile-time switch). Therefore, it would be desirable to fully implement the planned resimulation mechanism, allowing control inputs to be handled locally and synchronisation with the server to occur at appropriate points.

The resimulation mechanism does not currently attempt any form of smoothing between the client's concept of the current game state and that dictated by the server. This can result in jerky movement in some conditions, as the client is periodically "snapped" to the server's concept of state. Fixing this would be a priority for enhancing the gameplay experience.

The game server currently performs no resimulation to cater for late messages; it simply treats them as lagged and rewrites the timestamp to the time received. This can therefore produce some inconsistencies. It is thought, however, that for the present, this behaviour is reasonable as server-side simulation was discovered to be prohibitively complex to implement in the time available.

Addressing this is a prime candidate for future work.

Z Fighter Forever currently only has a single level. For a release version, multiple levels would be required and some form of level negotiation protocol added to the network connection mechanism. Adding new levels is relatively cheap to do; all levels are stored in files and may be loaded as appropriate; therefore creating a new level is simply a case of designing it and encoding this in the appropriate file format. Adding level negotiation to the network protocol should not be overly complex a task. Various other areas of the game may need modifying, given that it currently assumes a single level that never changes.

The game is currently intolerant of attempts to reconnect after disconnecting from the server. This is likely due to some state from the previous connection not being cleared correctly. Therefore a fix for this should not be particularly difficult.

There is currently no server detection mechanism although the UI has facility to list all known servers. Additionally, there is no dynamic reconfiguration of various game settings (e.g. window size, control input bindings etc.). These issues would require addressing before any release.

7 Concluding Remarks

Development was greatly hindered by the complexity of debugging a system when the most effective visualisation aid for a particular problem was often another component of the system which may also have been faulty. However, we now have not only the solid foundation of a multiplayer game, but a modular and reusable game engine with robust support for true portals.

References

[1] Various, "Portal Engines." Usenet thread, 1999. Viewed online at `http://groups.google.com/group/comp.graphics.algorithms/browse_thread/t%hread/d8e6be3323c9f5f8/?fwc=1`.

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

[3] P. Lincroft, "The Internet Sucks: Or, What I Learned Coding X-Wing vs. TIE Fighter." Viewed online at `http://www.gamasutra.com/features/19990903/lincroft_01.htm`.

[4] T. Sweeney, “Unreal Networking Architecture.” Viewed online at `http://unreal.epicgames.com/Network.htm`.

[5] Valve, “Source Multiplayer Networking.” Viewed online at `http://www.valve-erc.com/srcsdk/general/multiplayer_networking.html`.

[6] Various, “Continuum UDP Game Protocol.” Viewed online at `http://wiki.minegoboom.com/index.php/UDP_Game_Protocol`.

[7] D. Andrews, “A Primer For RakNet Using Irrlicht.” Viewed online at `http://www.daveandrews.org/articles/irrlicht_raknet/`.

A Who Wrote What

Report Jim, John, Karl, Phil.

Mandala Phil (overall), Jim (entity/portal transform matrices; collision mathematics; quaternions) and John (model and mesh loader; general assistance).

Flight model and physics Jim.

Game infrastructure John, Karl.

KUI and GUI Karl.

Control handling Jim, Phil.

Level loader and level Phil.

Networking John and Karl (overall), Jim (general assistance and testing).

Ship models Karl.

B Design Documents

B.1 Level Design

See figure 22.

B.2 Lighting Design

See figure 23.

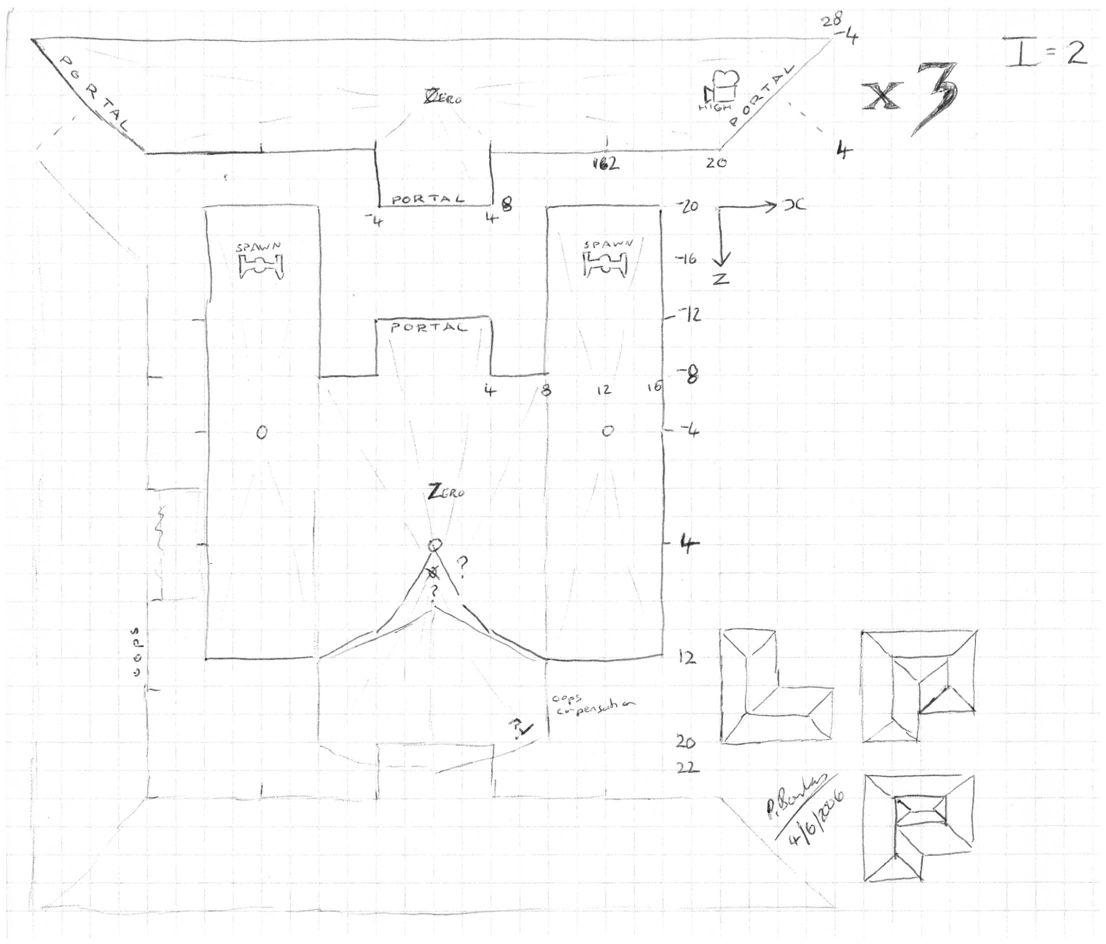


Figure 22: Design of the ‘]|[space’ level

C Level Format

A level is a list of directives, separated by semicolons. Directives may take arguments, which follow the directive name, separated by whitespace (*not* commas, and no parentheses). Comments are just ‘#’ directives terminated, as usual, with a semicolon. Directives are case-sensitive.

For directives which start creation of some kind of data structure, that data structure will be completed either when the next of its type (or containing type) is started, or at the end of the file. At the start of the level, there is no current sector, mesh, or polygon, and trying to add points or such is an error.

Some directives make use of IDs. The ID ‘-’ is magical, and is basically a null. (A portal polygon with a ‘-’ target is just a polygon with an ID.) IDs are only useful within the scope of the level loader. IDs are any string, but remember that

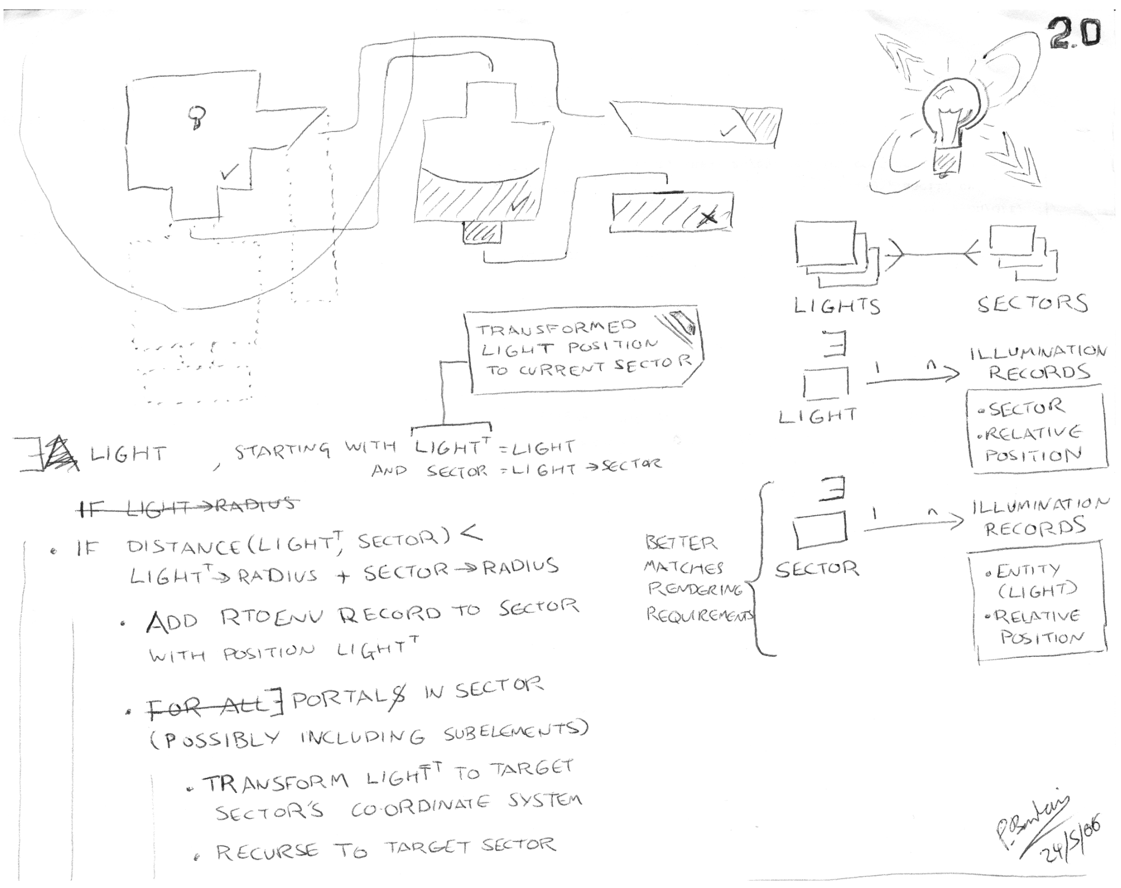

2.0
TRANSFORMED LIGHT POSITION TO CURRENT SECTOR
LIGHTS
SECTORS
∃
LIGHT
1
n
ILLUMINATION RECORDS
• SECTOR
• RELATIVE POSITION
∃
SECTOR
1
n
ILLUMINATION RECORDS
• ENTITY (LIGHT)
• RELATIVE POSITION
BETTER MATCHES RENDERING REQUIREMENTS
∃ LIGHT, STARTING WITH LIGHT^T = LIGHT AND SECTOR = LIGHT → SECTOR
~~IF LIGHT → RADIUS~~
• IF DISTANCE(LIGHT^T, SECTOR) < LIGHT^T → RADIUS + SECTOR → RADIUS
• ADD RTOENV RECORD TO SECTOR WITH POSITION LIGHT^T
• ~~FOR ALL~~ ∃ PORTALS IN SECTOR (POSSIBLY INCLUDING SUBELEMENTS)
• TRANSFORM LIGHT^T TO TARGET SECTOR'S CO-ORDINATE SYSTEM
• RECURSE TO TARGET SECTOR
24/5/06

Figure 23: 'Relevant to environment' determination algorithm design

the level parser splits on whitespace, with no escaping or quoting.

C.1 Directives

C.1.1 Metadata

name *string* Set the name of this level.

author *string* Set the author of this level.

C.1.2 Geometry

sector Start creation of a new sector.

mesh Start creation of a new mesh.

texture ***name*** Set the texture name of the next mesh; '-' for none. This is relative to 'resources/textures/'.

poly Start creation of a new, regular polygon.

portal ***id target*** Start creation of a portal polygon, with the ID of the first value (not necessarily ever referenced), and a target of the other ID (doesn't have to exist yet, or at all if one-way). The target sector is automatically determined.

point Create a point (of a polygon).

anchor ***index*** Set the anchor point for future portal polygons. In the case of rwalls, it refers to the wall segment (as opposed to the floor/ceiling). **Note:** this takes effect at the point at which the polygon is **finished**, not when it is created.

r, g, b, a, s, t, x, y, z Define the properties (colour; texture co-ordinate; position) of future points. These all take one, numeric argument.

room ***low high*** Sets Y heights for a room. Rooms are a shortcut to create multiple polygons which form some concave, 2.5D structure. The two arguments specify the floor and ceiling heights respectively. These surfaces are created as multiple polygons in a radial configuration from some centre point in order to guarantee polygonal convexity. *Low* may exceed *high*, but you probably actually want 'rinvert'. Rooms must have a containing mesh, because they are just sugar for polygons. If you want rooms to be 'closed', you must repeat the first point. Heights default to -1 and 1.

roomgencf ***low high*** Indicate whether ceiling and floor should actually be generated for the next room segment. Arguments are integers: zero for false, non-zero for true. Defaults to true.

roomc ***x z*** Set the centre point for a room, from which floors and ceilings radiate.

rinvert Indicate that a room is inverted, and should actually act like a convex object. This can be used to create simple geometry. You can trigger this multiple times to flip between the two, to potentially interesting effect.

rwall ***x z*** Specify a point along the room's wall. Sequential points are connected. In order to create polygons with the correct winding, build walls in an anticlockwise direction from the start point, on the XZ plane. The texture co-ordinates for wall segments are (1.0, 0.5) to (0, 0) from bottom old (right) to top new (left); for the floor, (0.25, 1.0) at the centre to (0.5, 0.5) and (0.0, 0.5) for the edges; for the ceiling, the same pattern, but with X values of 0.75, 1.0, and 0.5.

rwallp *x z id target* As 'rwall', but the wall part is a portal. IDs are shared with manual polygons. Note also that the 'anchor' is also only applicable to walls: floors and ceilings always use point zero, which is the centre of the room for the floor, and nothing useful for the ceiling.

rwalli *x z* Internal wall: it doesn't actually have the wall part; just the floor and ceiling.

C.1.3 Items

orientz Set the orientation of all future items to -Z, the default entity orientation.

orientby *x y z a* Rotate the orientation to use for all future items by $a\pi$ radians around the vector [x, y, z] (i.e. 1.0 = 180°). You can do this multiple times.

spawn *x y z* Add a spawnpoint at the given co-ordinates within the current sector

camera *x y z* Add a camerapoint: just like a spawnpoint, but the camera starts here.

D Network Packet Types

Type	Priority	Reliability	Timestamped	Parameters

ID_REMOVE_ENTITY	HIGH_PRIORITY	RELIABLE	Yes	MD_ID entityId;
				unsigned long timestamp;

ID_SPAWN_REQUEST	MEDIUM_PRIORITY	RELIABLE	No	

ID_SPAWN_GRANTED	MEDIUM_PRIORITY	RELIABLE	No	MD_ID ship,
				sector;
				double pos_x, pos_y, pos_z; double ori_w, ori_x, ori_y, ori_z;

ID_CONTROL_INPUT	HIGH_PRIORITY	RELIABLE	Yes	MD_ID entity;
				unsigned long timestamp;
				ActionType actiontype;
				double state;

ID_STATE_UPDATE	HIGH_PRIORITY	UNRELIABLE_SEQUENCED	Yes	uint32_t count;
				ShipState *states;

ID_DAMAGE_RECEIVED	MEDIUM_PRIORITY	RELIABLE	No	uint32_t damage;

ID_PLAYER_DIED	MEDIUM_PRIORITY	RELIABLE	No	PlayerID killed;
				PlayerID killer;

ID_CHAT_MESSAGE	LOW_PRIORITY	RELIABLE	No	PlayerID source;
				PlayerID target;
				char* text;

Table 1: Packet Types

E Game Usage

E.1 Compilation

Compiling Z Fighter Forever is relatively simple, providing that the required libraries are installed on the system. Z Fighter Forever depends upon the following libraries:

- RakNet (`http://www.rakkarsoft.com/`)
- FTGL (`http://homepages.paradise.net.nz/henryj/code/#FTGL`)
- Freetype 2 (`http://www.freetype.org`)
- SDL (`http://www.libsdl.org`)
- SDL_image (`http://www.libsdl.org/projects/SDL_image/`)
- libmd3 (libmodelfile from `http://sf.net/projects/worldforge/`)

At the time of writing, RakNet requires a small patch to the code before it will work correctly with Z Fighter Forever. The details of this patch may be found at `http://www.rakkarsoft.com/raknet/forum/index.php?topic=605.0`. We are the actual submitters of the patch, having found a bug in RakNet during development.

Once the prerequisites are installed, compilation should simply be a case of running the `buildlatest` script in the *game* directory under Linux (you may have to modify the include paths, however). Alternatively, use the MSVC project file under Windows.

E.2 Starting Up

To start Z Fighter Forever, it is necessary to run two programs. Firstly, `iesserver`, which is the game server. This has a single command line parameter, which is optional; this is the port number on which to listen (default is 2101). Secondly, `iesgame`, which is the game client. This takes up to two command line parameters, thus: `iesgame <hostname> [<port>]` where `port` is optional.

Under Linux, it may be the case that library paths require setting up. There exist scripts to do this; `runclient` and `runserver`. These take the same parameters as the binaries themselves (all command line parameters are passed on).

E.3 Control Bindings

The control bindings for Z Fighter Forever are currently hard-coded to the following:

- Forwards (Up arrow)
- Backwards (Down arrow)
- Yaw Left (Left arrow)
- Yaw right (Right arrow)
- Pitch up (S)
- Pitch down (W)
- Roll left (A)
- Roll right (D)
- Fire (Space)
- Brake (Right control)

Additionally, support for joysticks is implemented. The bindings were designed for 4-button joysticks (e.g. a Thrustmaster) and emulate the game 'Star Wars: Rogue Squadron'.

- Forwards (Button 3)
- Backwards (Button 4)
- Yaw axis (X-axis)
- Pitch axis (Y-axis)
- Roll (Button 2 with X-axis)
- Fire (Button 1)
- Lateral thrust (Joyhat)