DEE: an architecture for distributed virtual environment gaming

1998 Distrib. Syst. Engng. 5 107

(http://iopscience.iop.org/0967-1846/5/3/004)

Distrib. Syst. Engng **5** (1998) 107–117. Printed in the UK PII: S0967-1846(98)95406-7

DEE: an architecture for distributed virtual environment gaming

Simon Powers, Mike Hinds and Jason Morphett

PP 7.2, MLB 3/7, BT Laboratories, Martlesham Heath, Ipswich IP5 3RE, UK

Received 6 March 1998

Abstract. This paper presents a distributed virtual environment architecture targeted specifically at network games. It outlines the requirements for supporting a game genre known as graphical multi-user dungeons or dimensions, and shows how these requirements can be met by a novel approach to the distribution of the game environment model. An implementation of the architecture is covered, together with the initial conclusions drawn from the implementation process.

1. Introduction

1.1. Background

Gaming has proved to be one of the more successful applications of distributed virtual environments (DVEs). Titles such as Doom and Quake have not only dominated the game sales charts† and gaming press, but have even penetrated the cultural mainstream. Books, television features and even movie licences have all been offshoots of their phenomenal success. With the home PC now offering network connectivity and sophisticated 3D graphics as standard, there is every sign that this type of DVE application will continue to grow in popularity.

In terms of DVE architecture design, games are a fascinating area to explore. Because they are a comparatively recent phenomenon, standards have not yet had time to develop. Each title uses its own proprietary architecture, message formats, databases, etc. Hence, any new design has a clean sheet to draw upon, without the constraints of supporting predefined interfaces.

This design freedom is particularly useful because of the demanding requirements games place on a DVE architecture. Players have a predefined set of expectations about the experience a game should provide. This includes factors such as a rapid speed of response, detailed and fluid graphics, complex interaction with both the environment and its inhabitants, large environments to explore and structured content that provides an entertaining experience. Hence, multiplayer gaming provides both a useful and tough yardstick to measure any DVE architecture against.

The current developers of commercial gaming DVEs have a background predominantly concerned with producing single-player titles. This means they tend to focus heavily on the graphics engine (their traditional skillbase) and neglect development of the network and distributed processing architecture. By taking experience and skills from the academic and industrial DVE design arena into the gaming community, the opportunity exists to combine the best of both areas, and produce a showcase DVE.

† id software reports that over 15 million shareware copies of Doom have been downloaded since its launch.

1.2. Multi-user dungeons or dimensions

Mass market penetration of multiplayer gaming was achieved with visually appealing, graphically rich games such as Doom, Quake, Duke Nuke'em, etc. These are generically called Doom-style games after their most successful variant. However, whilst the resultant interest in multiplayer games is welcome, shared virtual gaming environments, in the form of multi-user dungeons or dimensions (MUDs) [1], have been around since the 1970s.

MUDs are one of the earliest types of DVEs, and their method of modelling a virtual environment reflects this. A single central server maintains a database containing text descriptions of all possible locations, together with the objects found in each location. Players use a single text channel to communicate with the MUD server, and issue simple commands to control their playing personae (e.g. go west, take sword, say hello, etc). The MUD server is responsible for updating its database in response to the player's commands, and describing any relevant changes to the players.

Whilst using a very simple architecture (see figure 1), MUDs have a number of interesting features not found in the more modern Doom-style titles.

- Persistence. The environment continues to exist irrespective of the presence of players. Changes made by players can persist long after they have logged off.
- Many simultaneous connections. With sufficient server and network capacity hundreds of players can explore the same environment apparently simultaneously.
- Size. A MUD environment, whilst subdivided into distinct locations, appears to be a single large world. By contrast, most Doom-style titles have a collection of distinct small levels with defined start- and endpoints.

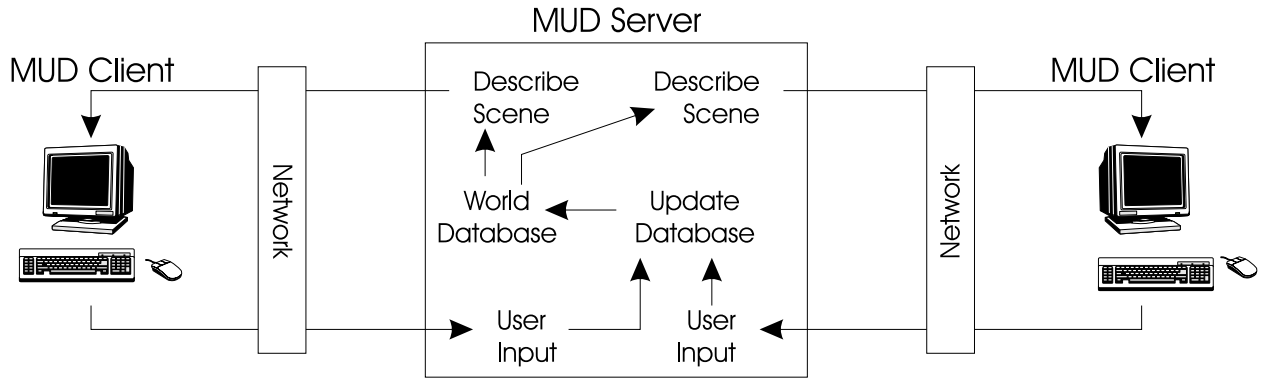

Figure 1. MUD architecture.

• Emphasis on interplayer communication. Whilst most graphical games focus on the simple task of annihilating the opposition, MUDs tend to set more complex goals, and hence encourage both cooperation and competition between players.

• User authored content. Experienced MUD players are often allowed to add or modify locations within a MUD, producing a more organic, evolutionary environment.

The combination of these features makes MUDs more than simple competitive gaming spaces. They can become complex on-line societies, where factors such as politics, economics, crime and even romance can flourish.

1.3. Distributed entertainment environment

We believe there is a potentially lucrative new gaming genre to explore, found in the crossover between MUDs and Doom-style games. This game style is termed a graphical MUD, and attempts to combine the appeal of interactive graphical environments with the complex social dynamics found in a MUD.

Distributed Entertainment Environment (DEE) was established to explore this new game style. The primary goal of DEE is to design and implement a DVE architecture that can support a graphical MUD. Additionally it is hoped DEE will provide useful ideas that can be applied to other, nongaming DVEs. Derived from the key features of both the MUD and Doom-style game genres, we set the following requirements for DEE.

(1) Environment modelled as a 3D space which players can visualize and interact with in real-time (i.e. >10 frames per second (fps)).

(2) Tight consistency maintained between the player's visualization and the environment's actual state. Consistency is one of the classic problems of DVEs [2], and different approaches result in varying degrees of consistency between clients. Loosely consistent systems allow clients to temporarily diverge in their views of the environment and resynchronize them either after a set time or when the divergence becomes too great. Whilst this normally increases the responsiveness of the client interface, it can mean two clients temporarily show different interpretations of the same scene. In the case of a game, where players quickly form playing strategies on the basis of what they see, this is not acceptable.

(3) Scaleable to include >100 players simultaneously. Indefinitely scaleable with respect to the environment volume.

(4) Balancing facilitated during run-time. Achieving an environment balance is a problem unique to persistent multiplayer games. In a single-player game the game designer can control what a player encounters at each stage of the game. In this way the game can be structured to provide a progressive challenge. However, once a persistent environment has been inhabited and modified by players, the designer has very little control on the individual experience of each player. Hence, the designer's task becomes one of balancing the behaviour and replication rate of each entity† within the environment in order to provide a self-sustaining environment all players can enjoy.

(5) High integrity. To produce an environment players can believe in, and become immersed in, it is vital that integrity cannot be breached by players attempting to cheat the system [3]. Any breaches in an on-line game's integrity (e.g. players able to make themselves indestructible) are always quickly exploited by players eager for an advantage, ruining the game for others.

(6) Persistency.

(7) Real-time dynamic interactions between entities (e.g. collision, bounce, rebound, etc). DEE should not simply be a visual front-end for a standard MUD.

(8) Extensible during run-time, so not only can new content can be introduced, but also the size of the environment grown to include new areas.

(9) Robust and predictable. The environment should behave as its designer intended for all clients. Clients with poor performance and/or poor network links should not impact on the predictability of the environment for other clients.

(10) Maximum component re-use between game titles.

† An entity is anything that can be placed within the virtual environment (VE). This includes everything from avatars, through movable objects, to fixed scenery.

2. Architecture

2.1. Network

Before designing a DVE architecture it is necessary to determine the network infrastructure that will support it. Factors such as expected latency, bandwidth, facilities offered and the topology will all greatly influence the final architecture.

A graphical MUD, in order to establish any sizeable playing community, must be accessible over a wide area network (WAN). The dominant WAN currently available is the Internet, based around the Internet protocol (IP) standard. However, the suitability of the Internet for supporting highly interactive services is very much open to question. Both its latency and bandwidth can be highly variable, making it difficult to maintain a consistent shared environment with the sort of rapid interaction and complex behaviour that a game player expects.

DEE is therefore targeted at a WAN designed specifically to support on-line gaming. Called Wireplay™, it uses the well-defined characteristics of the public switched telephone network (PSTN) to guarantee the latency and bandwidth between any two nodes. Using PCs with standard 28.8 kbit s^{-1} modems Wireplay can deliver consistent latency of around 120 ms between any two clients anywhere in the UK. The Wireplay architecture has the additional advantage of a known central aggregation point, which allows servers to be located at a fixed packet propagation time from all clients. By contrast the topology of the Internet is far more variable, making it difficult to specify a single ideal location for a server.

2.2. Environment model

We propose that a graphical MUD environment may be treated and implemented as three distinct models. These are the conceptual model, the dynamic model and the visual model.

The conceptual model describes the high-level game related concepts, and how entities relate beyond their spatial interaction. Typical examples of this might be as follows.

- A door can be opened with a key.
- Guns hold bullets, and pulling the trigger produces a bullet with a high velocity.
- Avatars† with a particular item of treasure get bonus points.

It is through the conceptual model that the designer can specify game goals and plotlines, and controls how the environment entities can be used to achieve these goals.

The dynamic model describes the interaction of the environment at the spatial level. Included in this model are factors such as collision management, trajectory/rebound calculations and movement. Where the conceptual model of a door describes that it can be opened by avatars, the dynamic model describes the volume it occupies and how this volume changes as the door swings open.

The visual model describes the environment with respect to the information and behaviour required to allow a player to access it. This includes the geometric data required for rendering and the actions that the player can perform on each entity. Thus, where the conceptual model knows that a door can be opened, and the dynamic model knows how it opens, the visual model knows what it looks like as it opens.

† An avatar is the graphical representation of a player's on-line personae.

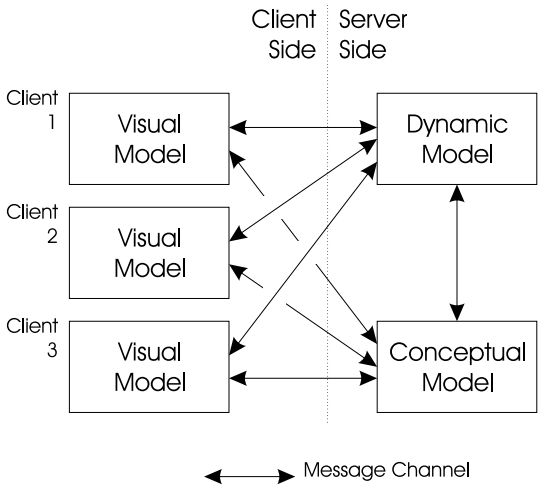

Figure 2. Coupling between models.

2.2.1. Coupling models. The three types of models are coupled together by message channels. This coupling, for a system with three clients connected, is shown in figure 2. For any single instance of a game one conceptual model and one dynamic model will exist. By contrast a visual model will be created for every client.

Typical examples of the information flow between these models are shown in figure 3.

Information flow A. Conceptual model requests a change to the dynamic model. This might occur due to a timer event or because a predetermined set of conditions have been met. The dynamic model evaluates the request for potential conflicts within its description of the environment, and assuming it is valid, performs the requested modification. The dynamic model then communicates its changing state to all relevant visual models.

An example of this scenario might be a door to a special game area that only opens for a short time period each day. The conceptual model knows when the door should open and close, and sends the necessary requests to the dynamic model. The dynamic model handles the opening, ensuring the door responds correctly to any collisions, and informs the visual models, allowing players to perceive the changed state. Note that a single request from the conceptual model might map into numerous updates between the dynamic and visual models. For example, in the case of an opening door, the dynamic model might generate both a 'start hinge rotation' message and a 'stop hinge rotation' message.

Information flow B. Client 2 places a request, via its visual model, with the conceptual model. The conceptual model evaluates the request and maps it into a modification to the

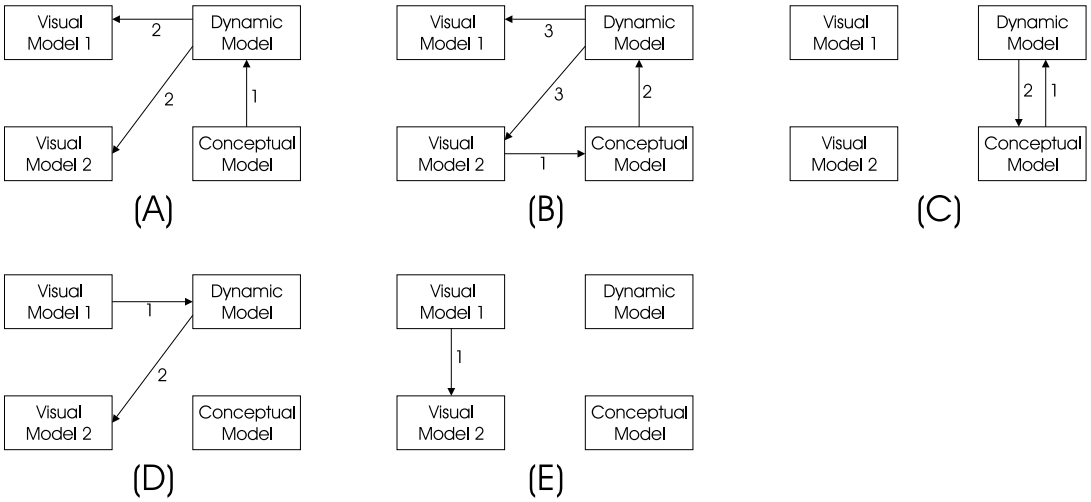

Figure 3. Example of information flow between models.

dynamic model. From this point the sequence of events is the same as A. For example, a player wishing to open a door would place the request with the conceptual model. The conceptual model would decide if this was an allowable action (e.g. does the player have the key?) and then send the necessary 'open door' requests to the dynamic model.

Information flow C. The conceptual model registers an interest in a specific set of dynamic conditions. When these conditions are attained the dynamic model informs the conceptual model. An example of this scenario might be a treasure room that players are awarded points for reaching. The conceptual model indicates to the dynamic model that it wishes to know about any avatars entering the specified area. Feedback from the dynamic model, as it detects avatars entering the room, could then be used to allocate points to the relevant players.

Information flow D. Client 1, via its visual model, places a request to modify the dynamic model. The dynamic model may either trust the client and accept the modification, or perform some validation on the request prior to acceptance. An example of request validation might be ensuring that strict collision rules are observed when moving an entity. Having modified its internal state the dynamic model informs all relevant visual models. This scenario would be enacted when a player wishes to move their avatar.

Information flow E. Visual model 1 requests a temporary modification to visual model 2. Such an information flow could be utilized for exchanging temporary modifications to the avatar's state (e.g. making avatars graphically emote at their player's request).

Note that an exchange of information between models can have a number of different contexts. In the simplest case information is simply a statement of fact. For example, in information flow C the message from the dynamic to the conceptual model states that a particular state condition has arisen. The dynamic model neither requires nor expects any response to this message. Alternatively the information exchange may be in the request/response form, where the originator can make no assumptions about the result of its request. For example, in information flow B, when the conceptual model requests an entity move to a specific position. Since the new position may be invalid due to object collision, the conceptual model must stall further operations on that entity until the dynamic model has informed it of the request's result.

In a few cases a request/response/rollback mechanism must be utilized. For example, in information flow D, where a player is making changes to the dynamic environment via their visual model, enforcing a request/validate/respond mechanism would heavily impact the responsiveness of the client–user interface. Hence, the visual model would allow changes to be performed locally, but be prepared to rollback to a previous state if the dynamic model rejects any particular modification. Note that rollback mechanisms in game scenarios should normally be avoided, as they break the consistency requirement specified in section 1.3. It is therefore envisaged that clients would carry out as much local validation to player input as possible (using their visual model), before assuming the request is valid and sending it to the dynamic model.

2.2.2. Advantages of environment model split. Separating the conceptual/dynamic modelling from the client side visual modelling offers a number of advantages with respect to the requirements laid out in section 1.3.

- The client can concentrate its processing power on producing high-quality rapidly updating visuals.
- The conceptual and dynamic models provide a point of sequencing for environment modification requests, allowing consistency to be maintained.
- Implementations can be both robust and predictable, as clients reflect the status of the servers rather than provide environment modelling of their own. Hence, an unreliable client will merely offer a poor visualization affecting just its user, rather than impact on other clients.

• Integrity is achievable, as end users can only access the environment state description through a predefined interface (the visual model to the conceptual/dynamic model coupling).

• Persistence is provided by holding a complete description of the environment on a set of central servers.

Separating the environment into separate conceptual and dynamic models also offers a number of important advantages. These primarily derive from the fact that both models have very different sets of problems to solve, and separation allows each implementation to be optimized differently.

A single dynamic model is re-usable across multiple game titles, as features such as collision handling and projectile trajectory calculation tend to be common to all. Dynamic modelling is also traditionally highly processing intensive, requiring a tuned and optimized compiled code to support real-time interactions. It therefore makes sense to invest time and energy in a single optimized dynamic model implementation, and not attempt to support extensive modification between game titles.

By contrast the conceptual model will require extensive modification between game titles to support the new rules and ideas that make each game unique. It is also possible that the conceptual model will need extensive modification during the life of a game to support balancing by the designer (requirement 4 in section 1.3). A good implementation of the conceptual model will therefore aim for clarity over performance, and offer facilities such as dynamic linking of new behaviour. This approach is feasible as the processing required to support the conceptual model should be far lower than that for the dynamic model. For example, a player moving around collecting treasure will be constantly making dynamic modifications to the environment by virtue of their avatar motion. However, the conceptual model only handles the occasional requests to gather the treasure.

The dynamic/conceptual split also potentially aids scalability. Modelling the behaviour of a VE on a single processing unit gives a nonscalable system, where the number of entities that can be placed within the environment is limited by the power of the single processing unit. A standard solution to this problem is to treat each entity in the environment as having a limited range of interaction over a predefined time period. This allows entities to be grouped on the basis of their spatial proximity, and each grouping allocated to a separate processing unit.

Whilst this allows the model of the environment to be distributed, its basic premise is not tenable for a graphical MUD. The game rules and puzzles within a MUD may cause a large number of interactions between entities with no clear spatial relationship. For example, a typical game puzzle might involve setting a collection of widely spaced switches/levers/buttons into a certain configuration. Achieving the correct settings would result in a modification to another part of the environment, allowing players access to a new area. Hence, nonspatially proximate entities would be affecting each other. An alternative example exists for a game requiring support of an economy. As items in the world are destroyed, sold to nonplayer characters or hoarded by players, it is necessary to introduce new items and control their selling price to ensure an effective game balance. This again introduces coupling between entities with no spatial relationship.

By utilizing separate dynamic/conceptual models the manner in which each model is subdivided can be varied. For the dynamic model, with the potentially high-load associated with dynamic simulation, the entity aggregation groups can be defined by spatial proximity. This will result in comparatively small but numerous entity groups, allowing the dynamic model to be scaled across many processing units. In contrast the conceptual model, which is not expected to generate a high processing load, may be supported from a single processing unit, with the entire model held in a single address space. Alternatively, if the game consisted of a number of separate 'levels' or 'worlds', each of these could be used to determine the division of the conceptual model.

Note that this approach to splitting the environment is only feasible because of the nature of the network available. Without low latency between the client and servers the coupling proposed is not feasible, and more of the environment behaviour would have to move out to the clients.

2.3. Entity representation

To implement the three coupled models of the VE we propose an approach termed entity representation.

As stated previously, an entity is anything that can be placed within the VE. This includes everything from avatars, through movable objects, to fixed scenery. Entities are usually defined by a derivation from a generic entity type, and this derived type will determine both their behaviour and the state they possess.

For example, a switch might be implemented as a switch entity. Its behaviour would include the ability to render itself, to toggle on and off (together with appropriate animations) and to trigger behaviour on another entity when switched on. Its state would specify the geometry data required to render it, its location, its setting (on or off) and the entity instance to inform when being toggled on. Multiple instances of the switch entity type would be created within the environment to represent different switches.

We propose that rather than define entities through a single type, and instance it at a single location, each entity is defined by a triplet of representatives. Each of these representatives is associated with a particular environment model (i.e. conceptual, dynamic and visual). The combination of three representative types defines an entity type, and an entity instance is produced by instancing its three different representatives.

This approach, for a door entity in a system with two clients, is shown in figure 4. The door entity is defined by the triplet ⟨door dynamic rep, door conceptual rep, door visual rep⟩, and to simulate it within the environment each model instances a representative of the appropriate type.

The function of each representative is to characterize its entity, through provision of appropriate state and behaviour,

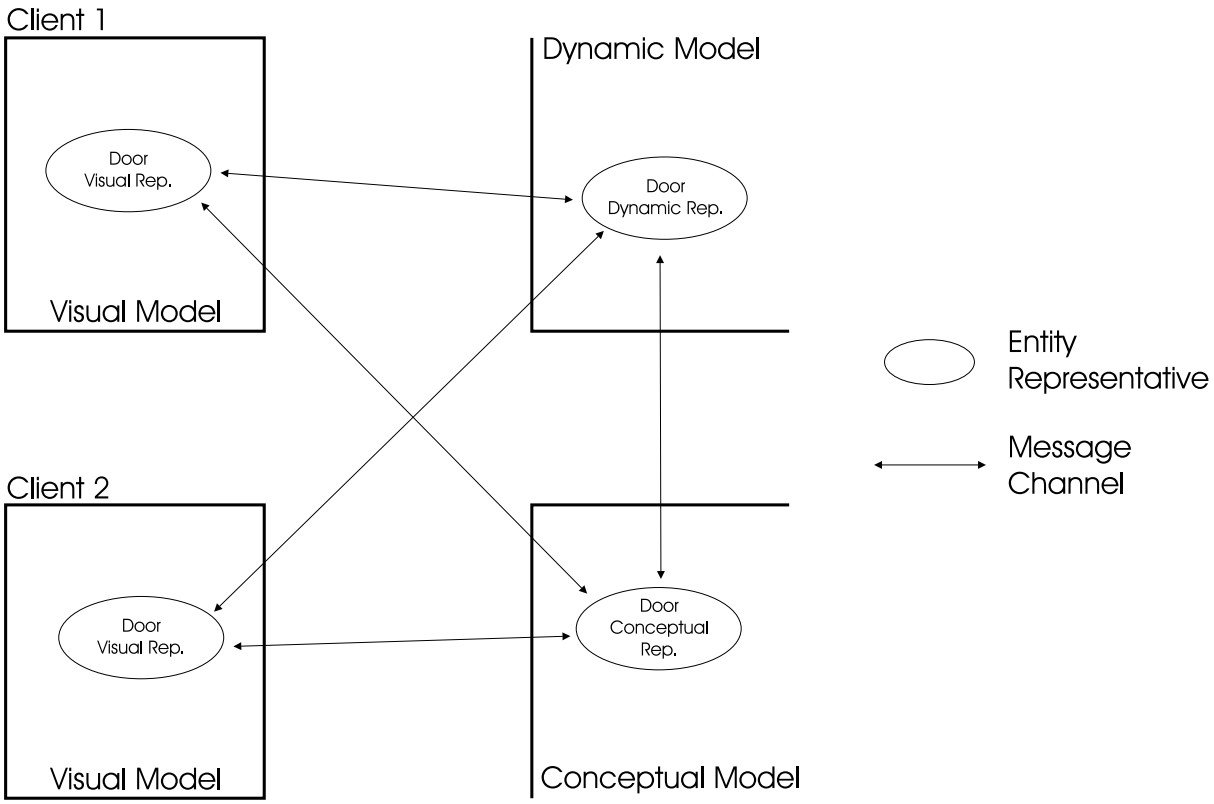

Figure 4. Door entity representatives.

in a manner appropriate to the instancing model. This functionality, for each different representative type, can be broadly categorized as follows.

- Visual representative. Holds the description of the entity appearance and the actions a player can perform on the entity. Able to provide visual description to client rendering engine, map player requests to the appropriate dynamic/conceptual representative and provide animation effects.
- Dynamic representative. Holds the description of the entity collision volume together with factors such as velocity, spin, position, orientation, etc. Able to model movement and collision of entity.
- Conceptual representative. Holds the behaviour and state specified by the game designer to make the entity perform correctly with respect to the rules of the game.

2.3.1. Entity representative coupling The information exchange between models is performed relative to each entity instance. Each representative possesses a reliable and ordered communication channel to all other representatives of the same entity. All requests or modifications to entity state are communicated via this message channel.

For example, a player wishing to open a door would use the client interface to place an 'open' request with the door's visual representative. This request would be passed to the door's conceptual representative for assessment. If the conceptual representative decides the player can perform this action, the conceptual representative informs the dynamic representative of the request. The dynamic representative then models the opening action, providing any necessary updates to the door's visual representations.

In addition to the reliable communication channels between representatives, an unreliable channel should also be available. This is useful for transmitting sequences of rapid state changes which are transitory in nature.

For example, a dynamic model wishes to inform visual models about a moving entity. Since each position update will be invalidated by the next, unreliable messaging can be used for each update. If position updates are lost consistency is only temporarily affected. Note that the final position message in the sequence must be sent reliably to ensure all visual models maintain long-term consistency.

This approach helps make effective use of limited bandwidth links.

All interaction between entities is handled within the context of a single model, with the representative of one entity unable to message to the representative of another in a different model. Any communication between entities occurs through method invocations within the various models. For example, in the case of a switch which opens a door, the switch's conceptual representative would instruct the door's conceptual representative to open. It would not message directly to the door's dynamic representative (see figure 5). This approach makes a necessary separation between the interface used to make requests to an entity and the coupling needed between representatives to accurately model the entity as a whole.

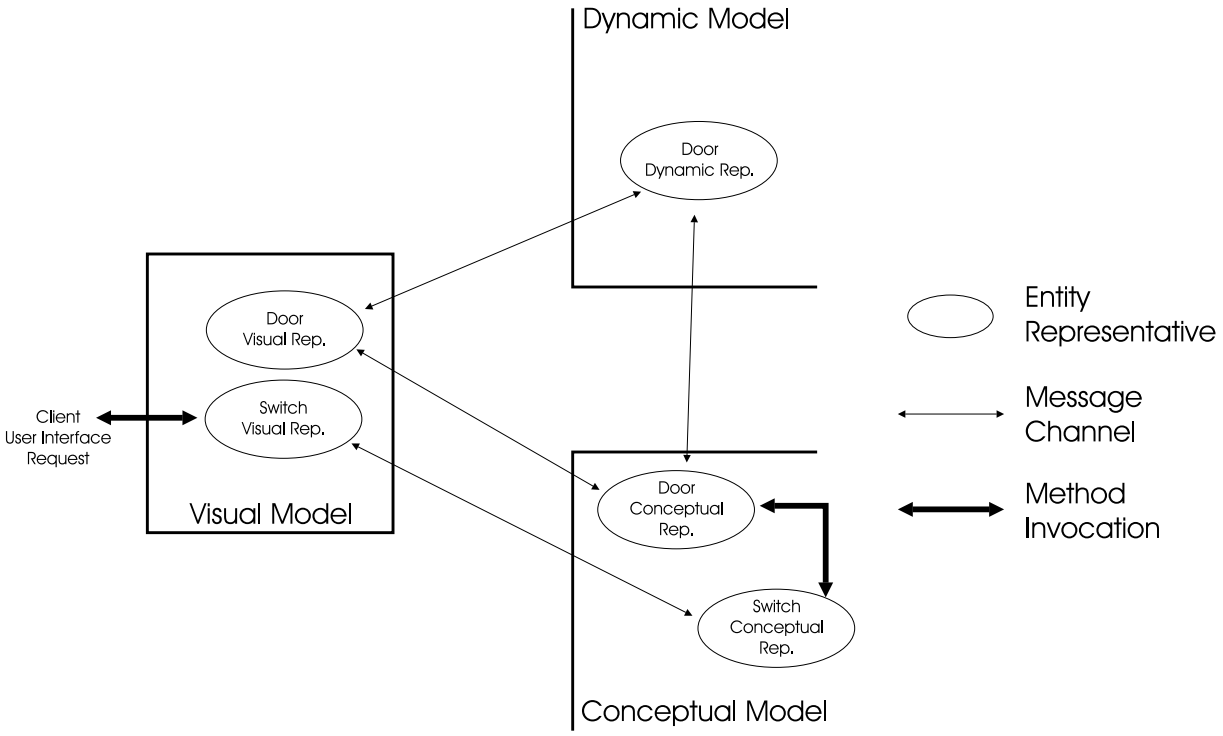

Figure 5. Opening a door from a switch.

2.3.2. Representative hierarchies. The type of entity is defined by the triple of its representative types. However, it is not expected that a unique representative type will be defined in each model for each required entity type. Instead, representative types will be re-reusable between different entities.

In terms of object orientated (OO) design, these representative types are drawn from a class hierarchy unique to each model. An example of a possible set of class hierarchies to support two different styles of door (sliding and swinging) and two different styles of gun (revolver and machine gun) is shown in figure 6.

The representative types from figure 6 would be combined as follows.

- Revolver entity = ⟨revolver conceptual rep, mobile dynamic rep, gun visual rep⟩.
- Machine gun entity = ⟨machine gun conceptual rep, mobile dynamic rep, gun visual rep⟩.
- Sliding door entity = ⟨door conceptual rep, sliding door dynamic rep, animator visual rep⟩.
- Swing door entity = ⟨door conceptual rep, swing door dynamic rep, animator visual rep⟩.

All hierarchies are rooted by a single representative type, termed a generic representative. This provides the common features required by all representatives in each model (e.g. providing rendering information to the client's scene graph† for the visual representative). The generic representative also provides the functionality for messaging between different instances of an entity's representatives.

Conceptually, a revolver and a machine gun are different in their behaviour both when fired and reloaded. Hence, in the conceptual model, whilst they share a common base class (gun), they are defined as different types. However, in visual and dynamic terms they are remarkably similar. Assuming factors such as recoil are ignored, both can be treated in the dynamic model as mobile entities occupying a specified volume of space. Visually, they share features such as 'muzzle flash' when fired, and hence can be modelled with a single visual representative type. Note, that whilst both gun types share a common visual representative type, this does not mean different instances of the guns will look the same. By modifying the geometry data references different gun instances maintain, they can be varied in appearance whilst sharing common behaviour.

In contrast, both types of door can share the same conceptual representative as they both perform the same function on a conceptual level—act as movable barriers that avatars can open and close. However, in the dynamic model the two doors are not similar, with very different behaviours. Hence, two different dynamic representatives are defined, each capable of handling the necessary movement and collision handling for their particular type of door. Within the visual model it might not even be necessary to have a special type of representative for a door. A more generic representative, capable of generating a smooth animation between defined geometry keyframes, would probably suffice.

Note that whilst the hierarchy shown in figure 6

† A scene graph is a tree-like data structure holding the description of a 3D scene. It contains polygon datasets, colour information, texture maps, etc. To produce a 2D image of a scene its scene graph is traversed, feeding its contents into a rendering pipeline.

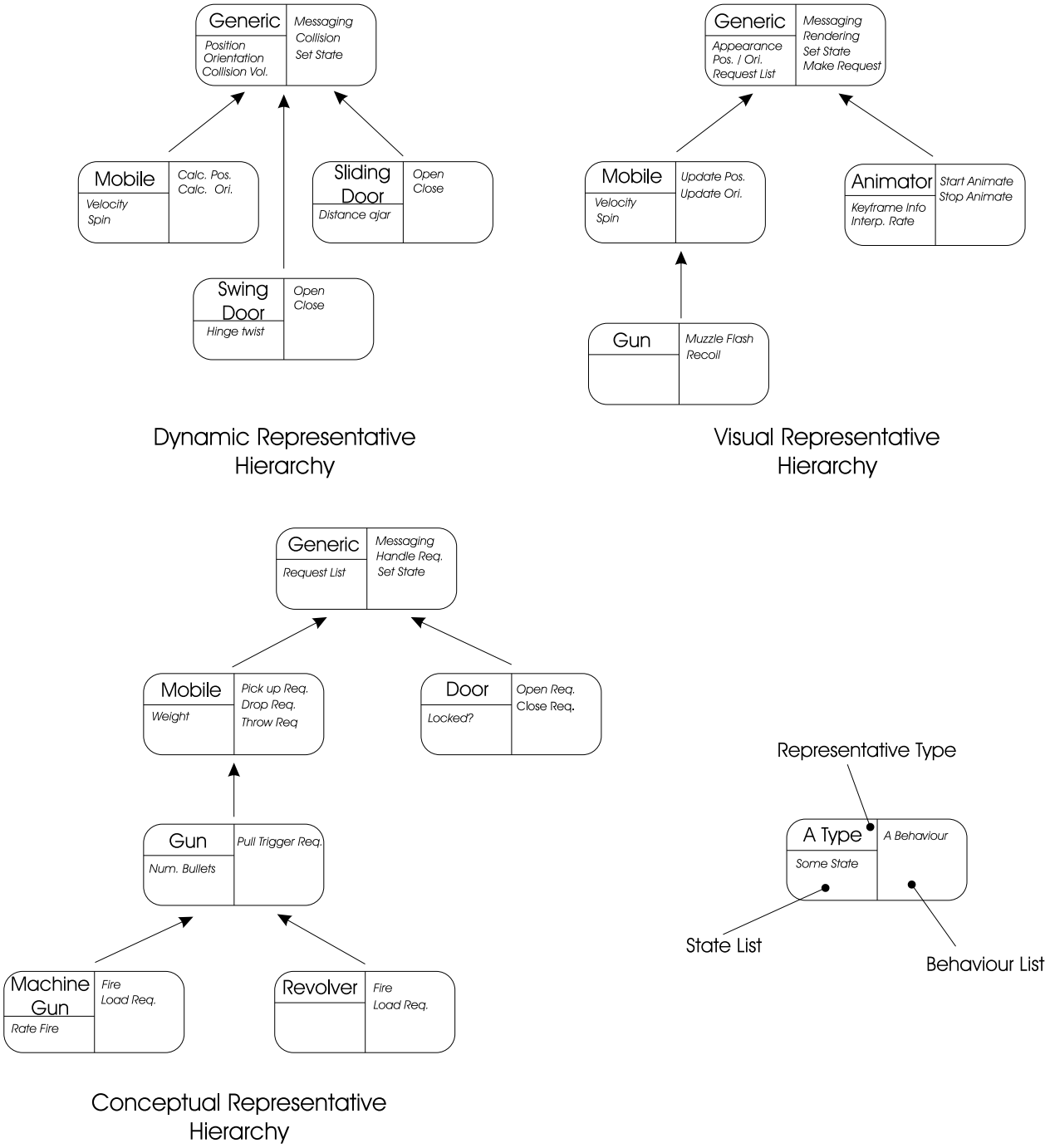

Figure 6. Example of class hierarchies.

was produced to support the door and gun entity types, the flexibility of the approach allows other entities to be supported from the representative types shown. For example, a rock could be instanced in the environment by using the representative triplet as follows: ⟨mobile conceptual rep, mobile dynamic rep, mobile visual rep⟩.

3. Implementation

The first version of DEE was implemented as follows.

- Conceptual model—Java, running on a Sun UltraSparc.
- Dynamic model—C++, running across a number of Silicon Graphics Workstations.
- Visual model—C++ with OpenGL, running on a Windows 95 PC.

Java was chosen for the conceptual model as many of its features (e.g. garbage collection, run-time linking, dynamic class loading, etc) are highly suited to the sort of balancing and tuning we expect during the life of a game title. For

the dynamic model, which should not require modification during the life of a game title, performance was a greater issue and hence C++ was selected.

The machines used to handle the dynamic and conceptual models were connected together via IP over an ATM network. The PCs running the visual model were connected though the Wireplay system using 28.8 kbit s^{-1} modems.

3.1. Scalability

Separating the environment into conceptual and dynamic sections aids scalability. For the conceptual model, which should handle far less player requests than the dynamic model, scalability was not considered an issue. Therefore, all conceptual representatives are instanced within a single process running on an UltraSparc. This simplifies the task of the game designer, as all the objects representing the environment are available within one process.

By contrast, the dynamic model must handle two issues associated with a DVEs scalability.

- Distributing the load associated with calculating the results of changes to the dynamic model (e.g. movement and collision).
- Constraining the amount of traffic that must be sent to a client to allow it to visualize the environment.

To handle these problems DEE divides the dynamic environment into a number of zones, each zone being a predefined volume of space. This is one of the standard approaches to achieving scalability, used in systems such as NPSNET [4] and Spline [5]. Every dynamic representative is associated with a single zone by virtue of its position. A zone therefore has state (the sum of the state of its dynamic representatives), processing requirements (the updates to the dynamic state) and a network traffic volume (the messaging between the dynamic representatives and the other representative types). By distributing the management of each zone between separate processes (potentially on different processor hosts), we achieve distribution of the processing load. Additionally, by treating each client to have visibility of only a limited number of zones, centred around the player's viewpoint, we can allow the environment size to increase without increasing the network load on each client.

Whilst straightforward in principle, the environment subdivision proved highly complex in practice, and for reasons of brevity the details are not covered here [6]. Figure 7 shows the distribution of processes between hosts for the complete system. Note that a new process is introduced in figure 7, termed the world manager. This is necessary to provide intelligent routing of messages from the conceptual model to the distributed dynamic model.

3.2. Representative messaging

To couple the various representatives together we used a fairly simple wire protocol. Each message header specifies a particular representative within a model and a method to invoke on that representative†. The message data is then passed to the specified representative as a variable length byte list. Any unpacking and marshalling of this data is performed within the invoked method, using a set of utility in lines.

Although simple, this approach offered the advantage of great flexibility. Whilst more formal remote procedure call technologies were considered (e.g. CORBA), it was felt that the problems of sequencing requests from representatives moving between processes and handling unreliable transport technologies was better served through a message based approach.

4. Conclusions

It is currently too early to provide quantitative measurement of the DEE system's performance. Whilst an implementation has been produced, it does not yet support sufficient features for a typical game. Hence, without realistic content it was felt that any measurements taken would not be representative of a final configuration.

However, a number of key points have arisen from the implementation to date.

- In terms of content authoring and game balancing, the Java implementation of the conceptual model works very well. It offers a clear view of the game rules and allows them to be easily modified either prior to or during run-time. The grouping of all conceptual representatives into a single process makes setting up complex puzzles involving multiple entities comparatively simple. New types of entities can be quickly added to the environment by extending a conceptual representative type and re-using existing dynamic and visual representative types.
- The time taken for a request to propagate from a client's visual model, through the conceptual model and the dynamic model, back to the visual model (e.g. an 'open door' request) must be minimized. The main area for concern in this loop is the propagation through the Java based conceptual model. If the conceptual model is performing numerous complex calculations it can significantly increase this time, which the user will perceive as an unresponsive visual interface. Threading the conceptual model implementation, optimizing its message handling, providing a sufficiently powerful machine to run it on and avoiding processing intensive game rules are therefore all recommended.
- Some commonly found game features do not easily fit the approach of splitting the conceptual and dynamic. For example, intelligent creatures who roam the environment making decisions on the basis of what is happening around them. Whilst they have highly dynamic behaviour, their decision making process is something a game designer will want to customize and balance for each game. However, attempting to place their 'brains' into the conceptual model would require their conceptual representative to be kept fully informed of their dynamic surroundings. This breaks the dynamic/conceptual split and would place too high

† Because of the distribution of the dynamic model (see section 3.1), the message header also contains sequencing information to allow message streams between representatives to be ordered correctly.

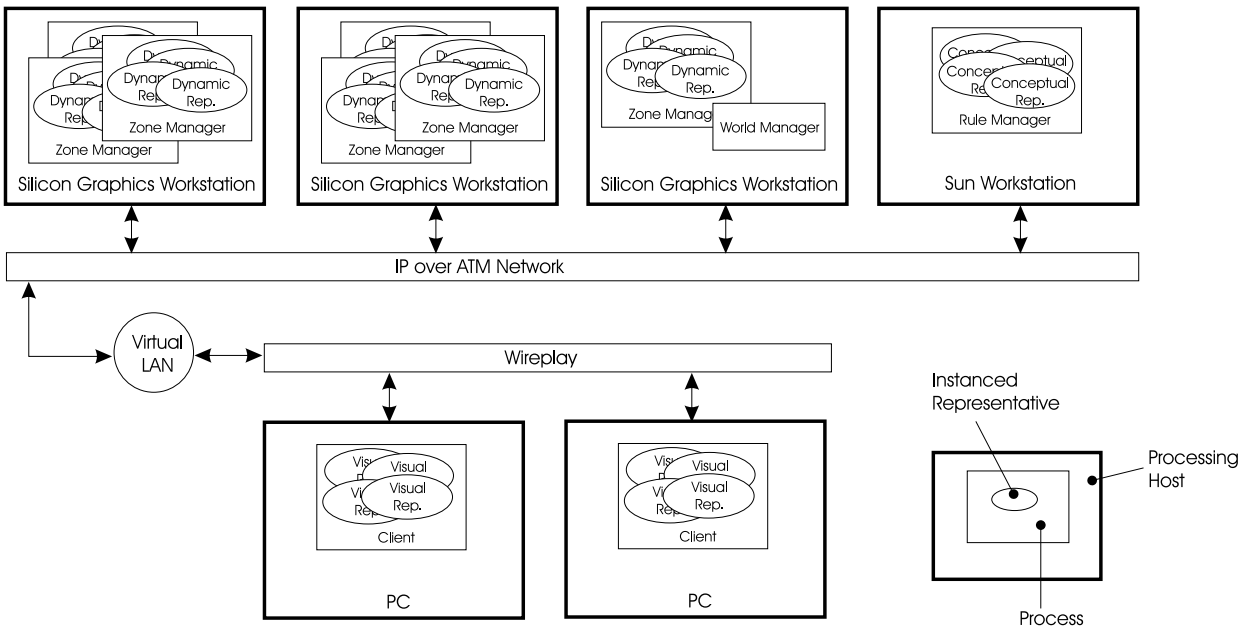

Figure 7. Distribution of system amongst processing hosts.

a processing and network load on the conceptual model. Whilst we believe this problem can be resolved by treating such intelligent creatures in the same way as players, placing their behaviour in a server side client style process, it is an area requiring further research.

• The separation of the dynamic model simplifies plugging in a variety of standard libraries to support the dynamic modelling. This means that not only can different libraries be easily assessed against each other, but the final selection can be modified depending on the processing power available for the dynamic model and the number of players expected at any one time.

• Distributing the dynamic model across multiple processing hosts is vital. For anything other than trivial dynamic behaviour (i.e. anything more than simply distributing position updates) a single processing host quickly becomes a bottleneck. Handling this distribution is definitely nontrivial, and a rich area of further research.

• Validating all client changes to the dynamic model, in order to guarantee environment integrity, creates too great a load on the dynamic model. Instead a variable level of trust should be assigned to each client, and the dynamic model validate only the input of untrusted clients (e.g. checking that they are not moving their avatars through walls). The trust level could be set either on the basis of feedback from players (e.g. complaints about a particular avatar) or on the basis of random sampling of change requests from all clients.

• Not all entities need a triplet of representatives. For immobile scenery that a user cannot perform any actions on (e.g. walls), a conceptual representative is not necessary. Alternatively, dynamic sensors (which allow the conceptual model to sample some aspect of the dynamic state) require no visual representative. Hence, some entities can be defined with a representative pair.

• To maximize performance it is important to minimize the number of interactions between the various models. For example, firing a gun could be treated as an example of projectile creation, where the conceptual model creates each bullet and specifies a velocity to the dynamic model. However, with potentially hundreds of players running around repeatedly firing at each other, this would place a heavy load on the conceptual model and its link to the dynamic model. Instead, if clients distribute the fire request via the dynamic model all relevant clients can perform the animation and assess if they have been hit. The conceptual model then assesses the result of a bullet strike when it has been informed of one from a client. Note that this alternative implementation raises the issue of integrity, with clients being trusted to perform collision detection. Game designers must therefore ensure the performance gains acquired are necessary before adopting such approaches.

4.1. Further work

Future work on the DEE design will concentrate on three main areas.

Producing quantitative results to show how equipped it is to meet the requirements laid down in section 1.3. This involves implementing enough entity types to allow production of an environment that is representative of a typical game. It will also require research on producing dummy clients that can generate realistic input into the system.

Specifying a framework for each environment model to facilitate authoring different game titles. The current design allows authoring of a game titles at a number of levels.

• Design of a world around existing entity types.

• Extending conceptual model class hierarchy to implement new entities.

• Extending dynamic model class hierarchy to support different types of dynamic behaviour.

• Implementing a client for a specific game title to provide a custom graphical user interface (GUI) and rendering pipeline.

The different model frameworks should aim to support all of these authoring approaches by providing and formalizing the base functionality all game titles will demand.

Determining the scalability of the dynamic model. As covered in section 3.1, scaling the dynamic model across multiple processing hosts is a nontrivial task. We therefore aim to focus on testing the value of the current implementation's approach, and determining if it has uses in other DVE architectures or applications.

Acknowledgments

Thanks to Tim Regan, Graham Walker and Chris Seal for their help and support.

References

[1] Curtis P 1992 Mudding: social phenomena in text-based virtual realities *Proc. Directions and Implications of Advanced Compuing (DIAC'92) Symp. (May 2-3, Berkeley, CA)*

[2] Lea R, Honda Y, Matsuda K and Rekimoto J 1995 *Technical Issues in the Design of a Scalable Shared Virtual World (Sony Research Forum SRF'95, Tokyo)*

[3] Morningstarr C and Farmer F 1990 *The Lessons of Lucasfilm's Habitat. Cyberspace: First Steps* (Cambridge, MA: MIT Press)

[4] Macedonia M R, Zyda M J, Michael J P, David R, Barham P T and Zeswitz S 1994 NPSNET: a network software architecture for large scale virtual environments *Presence* **3**

[5] Barrus J W, Waters R C and Anderson D B 1996 Locales and beacons: efficient and precise support for large multi-user virtual environments *IEEE Virtual Reality Int. Symp. (VRAIS)*

[6] Powers S J, Hinds M and Morphett J 1997 Distributed entertainment environment *BT Tech. J.* **15**