Techniques for Handling Scale and Distribution in Virtual Worlds

Karl O'Connell, Tom Dinneen, Steven Collins,
Brendan Tangney, Neville Harris and Vinny Cahill[*]
Distributed Systems Group[†] and Image Synthesis Group[‡]
Department of Computer Science
Trinity College Dublin
Ireland

Abstract

Lack of bandwidth and network latency are known to be major impediments to achieving realism in distributed virtual world (VW) applications with a large number of, potentially geographically dispersed, entities. This paper describes a combination of techniques that we are using to overcome these twin problems. The techniques described are intended to reduce both the volume and frequency of communication between the entities that make up the virtual world and include the use of anonymous event-based communication with notify constraints, scoping of event propagation with zones, and use of predictive approaches to replica management. Each of these techniques is described in turn.

1 Introduction

Lack of bandwidth and network latency are known to be major impediments to achieving realism in distributed virtual world (VW) applications with a large number of, potentially geographically dispersed, entities. This paper describes a combination of techniques that we are using to overcome these twin problems. The work is being done within the context of a wider project known as MOONLIGHT[1], within which we are designing and implementing a toolkit, known as VOID [9], for the development of distributed VW applications such as interactive simulations and advanced video games.

Central to VOID is a distributed object model, known as ECO, in which objects, or entities, communicate with each other using an anonymous event based paradigm. VOID also provides a class hierarchy that includes classes for graphics and the main types of entity, e.g. ones which can move, handle collisions, are animated etc., that populate the VW. Entitys classes are derived from this class hierarchy.

In order to scale well the amount of network traffic generated by entities must be reduced and the impact of network latency minimised. In the ECO object model events are only delivered to entities that have subscribed interest in them and additional techniques are utilised to further minimise the amount of traffic generated. In particular: *replication of entities* and *dead reckoning* are used to allow the behaviour of a remote entity to be simulated locally with a minimum amount of remote communication; *subscription to events can be parameterised* (using notify constraints) so that entities need only be notified of selected occurrences of a particular type of event; *entities can also be organised into zones* with the potential propagation of events restricted to entities within a zone.

The remainder of this paper elaborates upon the ECO model before going on to describe in turn the details of each of the techniques just mentioned. The current implementation of VOID is discussed and the paper concludes with a comparison of our approach with related work in the area.

2 The ECO Model

VOID supports the use of object-oriented (OO) techniques for the design and development of VWs. The VOID object model, known as ECO [11], combines three key concepts: *objects* representing entities, *events* providing the means for entities to interact and *constraints* which allow the specification of synchronisation, real-time, and notification requirements.

ECO objects, which are instance of classes, communicate using an event abstraction. An event represents a change to the state of the system and is a form of anonymous communication in that the object (or entity) raising the event does not know, or care, which other objects have subscribed an interest in the event. Each event has a name and zero

[*]email: Vinny.Cahill@dsg.cs.tcd.ie
[†]http://www.dsg.cs.tcd.ie/
[‡]http://vangogh.cs.tcd.ie/
[1]The Moonlight project is partially supported by the Commission of the European Union under contract number 8676.

```
class Entity
{
   protected:
      //...

   public:
      virtual void Subscribe (EventType,
                              EventHandler
                              NotifyConstraint,
                              PreConstraint,
                              PostConstraint);
      virtual void UnSubscribe (EventType,
                                EventHandler);
      virtual void Raise (EventType);
      //...
};
```

Figure 1: The Entity Class

or more parameters. The parameters of an event are typed. For the specific occurrence of an event the parameters are instantiated with values. These values, together with the event name, describe the state change that has occurred.

An object can inform other objects about a state change, and it can react if it is informed of some state change by other objects. The former is accomplished by **announcing** an event, and the latter by **binding** a method of the object to the required event. This binding can be static (at object creation time) or dynamic. The same method can be bound to several events, and the same event can have several methods (of the same or of different objects) bound to it. A binding can be established only if the signatures of the event and of the method match (if they have the same number of parameters, and the types of the corresponding parameters are the same).

ECO events are a richer abstraction than those of SimNet [5] and such systems. They are a language level concept and the ECO model supports *constraints* which are named conditions that control the propagation and handling of events. Figure 1 shows the basic event API.

The motivation for constraints is threefold. Firstly, *synchronisation constraints* provide a mechanism for associating synchronisation policies with a class. Secondly, *real-time constraints* provide a mechanism for associating various real-time requirements with a class and lastly *notify constraints* provide a means of restricting the propagation of events to objects which are specifically interested in particular occurrences of those events, thus enabling a more efficient implementation of event-handling.

Notify constraints are described in more detail below as are zones and dead reckoning which are the other main features of VOID which help the event abstraction to scale well.

3 Notify Constraints

The aim of notify constraints is to ensure that event handlers are only called when the entity to which they belong is definitely interested in the specific event in question.

For example in a distributed game a very common event would be `collision` whose parameters might include the identities of the entities that collided and to which many entities are likely to be subscribed. While the score keeping entity might be interested in all collisions, most other entities will only be interested in collisions involving *themselves*. The entity will only be notified when one of the parameters equals their own identification number. Without notify constraints each collision event would be handled by every subscribed entity, each of which would have to internally check whether the collision concerned it or not. The constraint system allows propagation of events only to those entities which want to receive them. Figure 2 illustrates how an entity would subscribe to collision events using notify constraints.

To support the distributed case, and to avoid the unnecessary transmission of event notifications, notify constraints may only refer to parameters of the event and never to the local state of an entity. As the constraint is independent of the actual state of the subscribing entity then it can be evaluated at the raising side and if found to be false then the event need never be propagated to that entity.

```
E1.Subscribe (CollisionEvent,
              EventHandler (Event),
              anEntity.ID == (CollisionEvent.EntityID1 || Collision.EntityID2),
              0,    // No preconstraint
              0);   // No postconstraint
```

Figure 2: Notify Constraints

```
class Entity
{
   protected:
      // ...

   public:
      // ...
      virtual void CreateZone (Zone newZone);
      virtual void JoinZone (Zone newZone);
      virtual void ChangeZone (Zone newZone,
                               Zone oldZone);
      virtual void LeaveZone (Zone zone);
      virtual void Raise (EventType,
                          Zone zone, ...);

      // The Zone parameter in the Raise function is optional and is
      // used only when targeting events at a particular zone.
};
```

Figure 3: Zone API

4 Scoping Events

Notify constraints are however only one way in which event propagations may be minimized. Events in ECO may also be **scoped**. Scoping ensures that objects do not receive notification of events, even with matching notify constraints, unless they are in the **same** scope as the object raising the event. In a typical distributed 'room based' game, for example, an entity may only be interested in events that are raised by other entities within same room.

In the ECO scoping model, objects are organised into zones [8], where a zone is simply a collection of related objects and events are only visible within the zone of the object raising the event. Objects are organised into zones at the discretion of the application programmer based on functionality, geographical location within the VW or physical location on the network. Communication is still anonymous as the object raising the event need not know which objects are members of the zone.

Partitioning VWs into virtual computer spaces within which people and VW entities interact, either with each other or with the various tools they find there, is a common technique employed by many CSCW systems (e.g Jupiter [3, 7]) and virtual environments (e.g. DIVE [2] and NPSNET IV [6]). Zones are a mechanism that facilitates the application developer in implementing whatever form of these spaces is required through extra features supported by the ECO language. Figure 3 outlines the C++ interface for zone management while in figure 4 a small program extract is given which illustrates how an ECO `Entity` object of type `VW_Entity` uses this interface.

Zone Membership

Objects may change zones **dynamically** and zones in ECO may also **overlap** allowing an object to be a member of two or more zones. This is useful for example in applications where an object may be a member of both a geographical and functional zone simultaneously. Consider the scenario in which objects *o1*, *o2*, *o3*, *o4* and *o5* in figure 5 are entities which are related geographically within the VW for example because they are all located in a research laboratory

```
class VW_Entity : public Entity
{
   //...
   public:
   //...
   VW_Entity (...);
   virtual void Navigate ();
   //...
};

VW_Entity (...) // constructor function
{
   JoinZone (zoneA, zoneB);
   // other initialisations
}

void VW_Entity::Navigate ()
{
   if (Bdry_collision == TURE)
      changeZone (oldzone, newzone);
   else
      changePosition (...);
}
```

Figure 4: An example program using zones

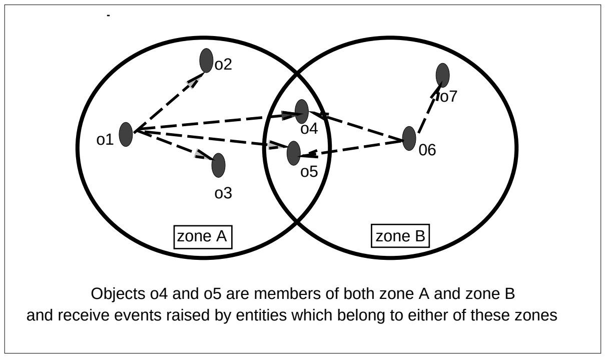
o2
o1
o4
o7
06
o5
o3
zone A
zone B
Objects o4 and o5 are members of both zone A and zone B
and receive events raised by entities which belong to either of these zones

Figure 5: Overlapping Zones

(zone A) within a university, while objects *o4*, *o5*, *o6* and *o7* have some functional relationship (represented by zone B), say through their role within the university (e.g. researchers in the computer science department). In this scenario objects *o4* and *o5* are members of both zone A and zone B and will receive event notifications raised by objects in both zones. Events raised by the objects *o4* and *o5* will be propagated to both zone A and zone B.

Nesting

Zones may be **nested**, allowing large zones containing many objects to be subdivided. In figure 6, zone A may be considered, for example, as a virtual computer science building with zone B representing the research lab. In this scenario events raised by an object in the research lab (e.g. *o1*) will be propagated to only those objects in its zone. Whereas an object raising an event in the outer enclosing zone (e.g. *o5*) will be propagated to all the objects in both zone A and zone B.

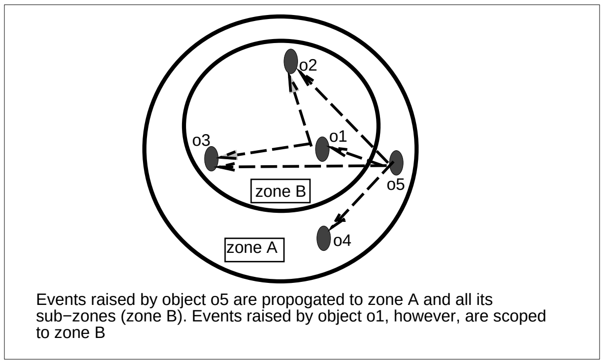
o2
o3
o1
o5
zone B
o4
zone A
Events raised by object o5 are propogated to zone A and all its sub−zones (zone B). Events raised by object o1, however, are scoped to zone B

Figure 6: Nested Zones

Targeting Zones

Finally the scope of an event may optionally be **targeted** at a particular zone of which the object raising the event is not a member. This may be useful in situations where an entity (e.g. *o5* in figure 7) is in a different building, say the admissions office in the university example (zone A) and may wish to send events to the computer science department (zone D) notifying all the entities within that building of some change in admission procedures.

5 The VOID Class Hierarchy and Dead Reckoning

This section briefly outlines the VOID class hierarchy and the dead reckoning protocol used to reduce the number of events that have to be sent accross the network.

5.1 The VOID Class Hierarchy

The VOID toolkit provides a class hierarchy for building VWs. The class hierarchy provides common functionality found in other VW toolkits, such as articulated fiqures, animation, collision detection and newtonian physics. This hierarchy provides the user with an infrastructure for a basic model of an autonomous entity. This entity can be tailored to suit particular requirements and as such the hierarchy is designed so that the application developer may extend the Entities functionality with facilities required by the application.

Figure 8 shows the general structure0 of the VOID class hierarchy. The main VOID classes are outlined below,

- Animation - provides functionality for mapping KeyFramed channels onto an Entity's node hierarchy.
- Collision Detection - provides support for receiving and handling Collision Events.

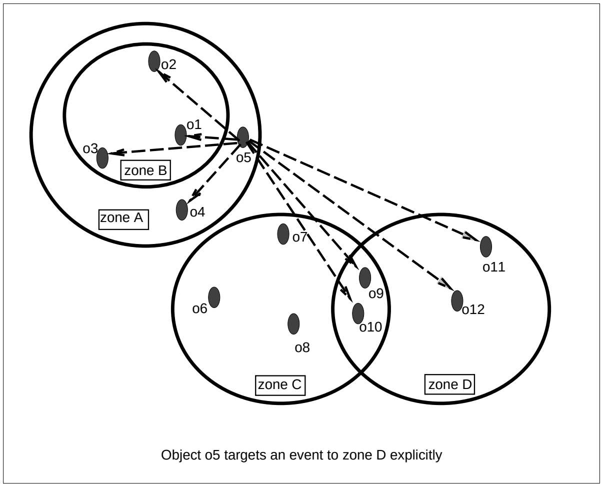

Figure 7: Scoping events explicitly

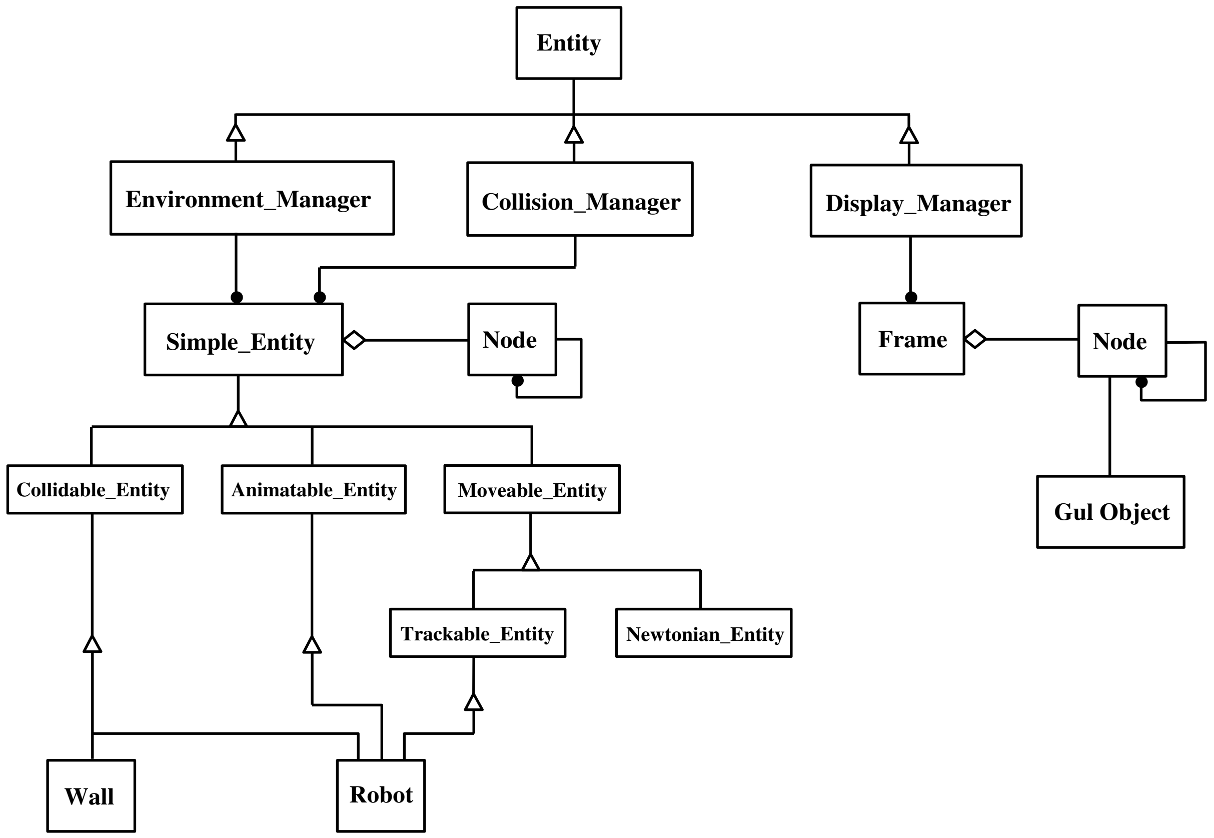

Figure 8: The VOID Class Hierarchy

- Moveable - this class provides a simulation core using integrator and derivative classes for updating an Entity's state to the next time step. A Moveable Entity can either represent a Particle (Position and Velocity) or a Rigid Body (Position, Momentum, Orientation and Angular Momentum).
- Newtonian - An Entities node hierarchy can be acted on by gravity and spring forces. With the addition of a newtonian class within a Physically-based simulator, to provide a means of enhancing the feeling imersion within the VW.
- Trackable - The Trackable Class provides the user with a predictor corrector model based on the track class which is described below. By simply deriving from this class instead of from the Moveable_Entity class all updates are tracked and events are only sent when the actual local behaviour differs from the tracked version by a threshold amount.

The above classes are derived from a common base class `Simple_Entity`, which provides the basic functionality of the object node hierarhcy.

The `Envionment_Manager` is responsible for managing a number of Entities. In figure 8 a `Wall` is derived from `Collidable_Entity`, while the `Robot` is derived from `Collidable`, `Animatable` and `Trackable_Entity`. The behaviour of an entity derived from an `Animatable_Entity` or a `Newtonian_Entity` is defined by an ASCII file read in at run time.

5.2 Dead Reckoning

The Display Manager is the component of VOID that is responsible for visualising the environment. It has to ensure that a given object appears in the same position at a given time to all users who can perceive it, regardless of the display frame rate of their machine and delays caused by the communication system. To ensure a realistic looking simulation and to cater for scalability the amount of communication between Entities and the display manager must be minimised. To achieve this a form of dead reckoning, similar to that outlined in [10], is used. Entities and the Display Manager both execute a function which will predict the future behaviour of the Entity. Updates are only sent from the Entity when the actual local behaviour differs from the predicted version by a threshold amount.

In VOID **any time dependent behaviour** can be predicted and corrected. The manner in which this is done is outlined in detail below for the case of movement but the same approach is also used to minimise the amount of traffic generated by the animation of entities (a crucial aspect for realistic games).

The Display Manager predicts future updates from Entities based on several previous updates which it uses to build up a track history of the Entity, it then uses this track history and curve fitting along with numerical extrapolation techniques to display the object at the local frame rate. The Entity maintains its true position along with the predicted path and it is only when the difference between them skews beyond a certain threshold that it raises an Update Event. All Display Managers subscribed to this event receive the new position update and re-fit their predicted path accordingly.

Further savings on network bandwidth may be made by sending only the positional and orientational updates instead of extra derivatives such as velocity and acceleration which can be calculated locally by the Display Manager. Only sending the position and orientation updates better approximates the behaviour of the object if updates are received too late as positions cannot change significantly until the velocity and acceleration have changed for some time.[2]

To ensure a realistic visualisation it is necessary to avoid 'jumping' when an update is sent to the Display Manger. An extra convergence step is introduced that smoothly converges with the tracked position at the Convergence Point. This step provides a seamless view of remote objects. This convergence path can then be extrapolated at the local frame rate.

6 Related Work

Like VOID, SimNet and DIS [5] also use predictive methods and dead reckoning to reduce communication between entities within the VW. However both of these systems have problems when scaled due to their homogeneous world databases, where all object state changes must be communicated among all users of the environment. The NPSNET IV [6] project has attempted to overcome this problem by introducing the concept of areas of interest (AOI), whereby *only* VW entities within a particular AOI would communicate and receive messages from each other. An IP Multicast address may then be assigned to the AOI. NPSNET IV, however, is based on the use of the DIS network protocol which is targeted primarily at the development of large scale military applications.

The DIVE [2] (Distributed Interactive Virtual Environment) system is a toolkit for building distributed interactive virtual reality applications in a heterogeneous network environment. DIVE allows a number of users and applications to share a virtual environment, where they can interact and communicate in real time. The run-time environment consists of a set of communicating processes, running on nodes distributed within a local area network (LAN) or wide area network (WAN). These processes can represent either human users or autonomous applications and have access to a number of databases which they update concurrently. Each database contains a number of abstracted descriptions of graphical objects that make up the VW. Multicast protocols are used for the communication within such a process group. A process may enter and leave groups dynamically, but at a given time will be a member of only one process group. A disadvantage with this approach however, is that it is difficult to scale because of the communication costs while maintaining reliability and consistent data.

MASSIVE [1, 4] (Model, Architecture, and System for Spatial Interaction in Virtual Environments) supports multiple VWs where each world may be inhabited by many concurrent users who can interact over ad hoc combinations of graphics, audio and text interfaces. In contrast to distributed VW systems based on a shared database approach, the processing model used in MASSIVE is of independent computational processes communicating over typed peer-to-peer connections (running over standard Internet transport protocols). Each world defines a disjoint and infinitely large virtual space which may be inhabited by many concurrent users. Portals allow users to jump from one world to another. Interaction between users is based on a spatial model, where users interact with each other provided they are within each other's *aura*. Auras are governed by spatial factors typically an object's relative position and orientation. Communication, however, is limited to point-to-point connections.

7 Implementation

The VOID graphics libraries and display manager, which together support the predictor corrector system, are implemented as C++ class libraries layered above both the GUL[3] and OPENGL rendering packages. Currently a C++ distributed version of ECO is implemented with zones and constraints. This implementation essentially maps zones onto IP multicast groups and will support event type definitions written in C++ with pre-processor support to be provided for stub generation in order to handle event marshalling. Planned future work will investigate optimisation techniques for local event dispatching perhaps based on dynamic code generation techniques as used in the event dispatching mechanism of the Spin micro-kernel. We also intend developing several VW applications to illustrate the scalability and flexibility of the toolkit.

[2]Position changes are thus delayed reactions to forces, they are less sensitive to sporadic changes.

[3]A graphics library developed within Moonlight.

8 Summary and Conclusions

This paper presented a brief overview of a number of techniques which we are currently developing in order to support scalable distributed VWs. Our approach is based around an object model which uses anonymous event based communication along with notify constraints, zones and dead reckoning to ensure effective event propogation.

Acknowledgements

The authors would like to acknowledge the very many helpful contributions of our partners in the Moonlight consortium.

References

[1] Steve Benford, John Bowers, Lennart Fahlen, Chris Greenhalg, John Mariani, and Tom Rodden. Networked Virtual realitty and Cooperative Work. *Presence*, 4(4):364–386, 1995.

[2] Carlson C. and Hagsand O. DIVE: A Platform For Multi-User Virtual Environments. *Computer And Graphics*, 17(6):663–669, 1993.

[3] Pavel Curtis, Michael Dixon, Ron Frederick, and David A Nichols. The Jupiter Audio/Video Architecture: Secure Multimedia in Network Places. In *ACM Multimedia Conference*, 1995.

[4] Chris Greenhalgh and Steve Benford. MASSIVE: a Distributed Virtual Reality System Incorporating Spatial Trading. In *15th International Conference on Distributed Computing Systems (DCS'95)*, Vancouver, Canada, May 30-June 2 1995. IEEE Computer Society Press.

[5] Locke J. An Intrduction to the Internet Networking Environment and SIMNET/DIS. Technical report, Computer Science Department, Naval Postgraduate School, August 1993.

[6] Macedonia, Michael R., Zyda, Michael J., Pratt, David R., Brutzman, Donald P. and Barham, Paul T. Exploiting Reality with Multicast Groups: A Network Architecture for Large Scale Virtual Environments. In *the Proceedings of the 1995 IEEE Virtual Reality Annual Symposium*, North Carolina., 1995.

[7] David A Nichols, Pavel Curtis, Michael Dixon, and John Lamping. High-Latency, Low-Bandwidth Windowing in the Jupiter Collaboration System. In *User Interface Systems and Technology (UIST'95)*, 1995.

[8] Karl O'Connell and Vinny Cahill. System Support for Scalable Distributed Virtual Worlds. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology*. ACM, July 1996.

[9] Karl O'Connell, Vinny Cahill, Andrew Condon, Stephen McGerty, Gradimir Starovic, and Brendan Tangney. The VOID shell: A toolkit for the development of distributed video games and virtual worlds. In *Proceedings of the Workshop on Simulation and Interaction in Virtual Environments*, 1995. Also technical report TCD-CS-95-27, Dept. of Computer Science, Trinity College Dublin.

[10] S.K. Singhal and D.R. Cheriton. Using a Position History-Based rotocol for Distributed Object Visualization. Technical Report STAN-CS-TR-94-1505, Stanford University, 1994.

[11] Gradimir Starovic, Vinny Cahill, and Brendan Tangney. An event based object model for distributed programming. In *OOIS (Object-Oriented Information Systems) '95*, pages 72–86, London, December 1995. Springer-Verlag. Also technical report TCD-CS-95-28, Dept. of Computer Science, Trinity College Dublin.