The VOID Shell

A Toolkit for The Development of Distributed Video Games and Virtual Worlds

Karl O'Connell*, Vinny Cahill, Andrew Condon,
Stephen McGerty, Gradimir Starovic and Brendan Tangney
Distributed Systems Group†,
Department of Computer Science,
Trinity College Dublin
Ireland

Abstract

This paper presents a brief overview of the VOID approach to the design and implementation of next-generation (distributed) video games and other interactive virtual world applications. The main features of the VOID Shell, a toolkit for object-oriented game and virtual world development, are described including its object model and associated class libraries as well as the tools provided for the game designer and programmer.

1 Introduction

The VOID (for **V**irtual **O**bjects **I**nteracting **D**ynamically) Shell [3, 2] is a toolkit for the development of 3D, (distributed) video games and other interactive virtual world (VW) applications which is being developed as part of the ESPRIT MOONLIGHT [1] project[1]. MOONLIGHT is a collaborative research and development project involving industrial and academic partners from several European countries which is addressing the development of arcade and PC video game systems. The project as a whole is addressing both game development and game execution environments including the development of tools for game design and implementation, hardware and software support for real-time 3D rendering, a real-time executive and the VOID shell. In addition to games, intended applications of MOONLIGHT include 3D videotext services such as teleshopping as well as distributed interactive simulations.

The VOID shell supports the use of object-oriented (OO) techniques for the design and development of games. Central to VOID is its object model which describes the way in which the objects representing entities in a game interact. The VOID object model, known as ECO, 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 [10]. Entities in VOID may be defined in two ways: either graphically using the VOID entity editor (a statechart-based graphical tool for describing an entity's behaviour) or programatically with the ECO language, an extension of C++ supporting events, constraints and objects. VOID also provides a library of generic game entity base classes with classes providing support for animation, collision detection and mobility. A graphics library supporting both GUL (a graphics library developed within MOONLIGHT which is optimised for real-time 3D rendering [5]) and GL is also included. VOID also provides a world editor, a graphical tool enabling the game developer to construct a game by positioning entities within the world. The VOID execution environment provides the runtime support for the execution of game entities including the ECO language runtime and is layered above an instantiation of the TIGGER object support operating system framework [4] providing a real-time executive [11] as well as support for object persistence as required.

The remainder of this paper is organised as follows: Section two describes the architecture of VOID in (a little) more detail and gives an overview of the game development process supported. The major features of VOID including the object model, class library, entity editor and support for distribution are described in section three while section four concludes the paper with a brief summary of current and future work.

2 Architectural Issues

The motivation behind the VOID architecture is simple: To allow game designers to produce executable games without burdening them with excessive low level detail while at the same time allowing them to see the overall progression of the game. To support this, significant use

*Karl.OConnell@dsg.cs.tcd.ie
†http://www.dsg.cs.tcd.ie/

[1] The MOONLIGHT project is partially supported by the Commission of the European Union under contract number 8676.

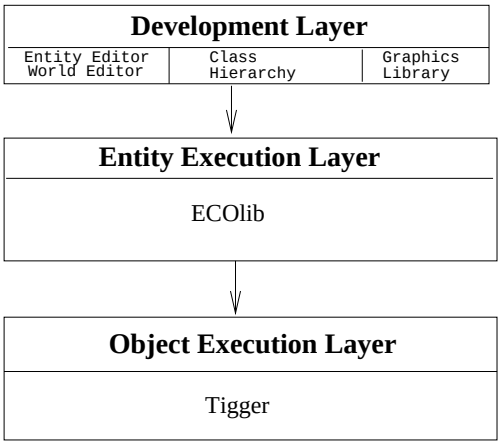

Figure 1: Overview of the VOID Architecture

of predefined code and game structures is required, saving the designer from carrying out the work themselves.

The object oriented paradigm provides an excellent basis for structuring a system (using frameworks) and reusing code (class methods), and as a result VOID is heavily object oriented.

Used together, tools such as the Entity Editor and the World Builder go to make up a designer environment that ties together intuitive visual programming, predefined game frameworks, and graphical tools in a way that allows game designers to create games in a more intuitive fashion.

As shown in Fig.1, VOID logically consists of three layers. The development layer supports the production of games and provides tools and libraries to be used by game designers and programmers as well as by artists, animators and musicians. Of course existing "off-the-shelf" tools can also be used with VOID. The entity execution layer provides the necessary runtime support for the execution of VOID applications and is layered above the so-called object execution layer.

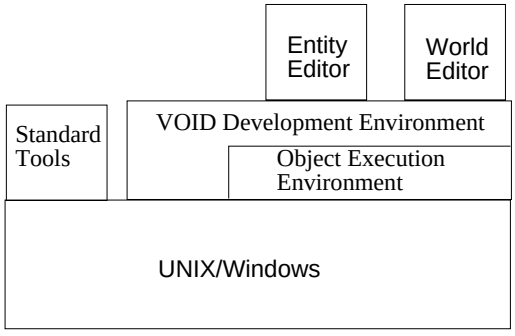

Figure 2: A VOID Development System

At runtime a VOID installation takes one of two forms. A development system (see Fig.2) runs on a standard workstation or PC and supports the VOID development layer including VOID tools such as the entity editor and world editor. The object execution environment may be used to connect to a target system during game debugging and tuning. A target system (see Fig.3) runs on a custom arcade platform, workstation or PC. While the same UNIX workstation might simultaneously host a development system and a target system, it is not envisaged that a single PC would simultaneously play both roles. The VOID execution environment is tailored to include only the necessary support for the entities making up the current application.

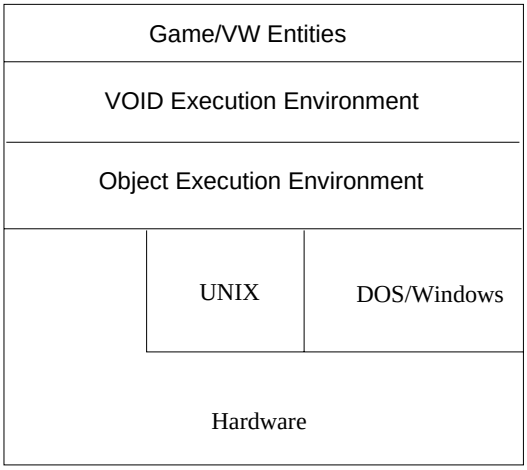

Figure 3: A VOID Target System

Developing a game using VOID (see Fig.4) involves the game designer (the person devising the game) using the entity editor to specify the behaviour of each class of entity that is to appear in the final game. Following this, the specification of the entity classes may be enhanced by linking artwork and animation specifications to an entities behavioural specification. Similarly any sound effects required are added at this stage.

The programmer then elaborates the statecharts of each Entity by adding appropriate ECO code fragments. With the help of the entity editor, a complete ECO source code representation of the entity can then be produced. Once the appropriate code has been compiled the designer and programmer use the world editor to place instances of these entity classes within the game world. Of course the entire process is unlikely to be carried out in a single iteration!

3 The ECO Model

In the ECO model, objects which are instances of classes communicate using events. An event represents a change to the state of the system. Each event has a name and zero or more parameters. The parameters of an event are typed. For the specific occurrence of an event the parameters are instantiated with values.

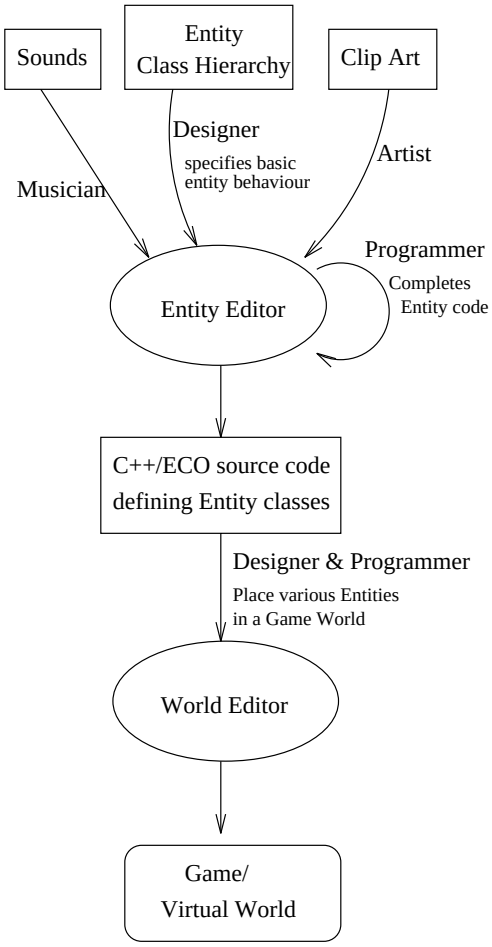

Figure 4: Overview of Game Production

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). Thus events may be considered as a form of one-to-many anonymous communication.

In addition to events, the object model supports constraints. Constraints are named conditions which control the propagation and handling of events. 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 finally *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. An object may decide, based on its local state when it is informed about an event of interest that the processing of the event should be postponed or even cancelled. If the local state is such that the processing should go ahead, it may be the case that multiple flows of control are allowed within the object.

3.1 Notify Constraints

A *notify* constraint is optionally provided by a destination object when it subscribes to an event. The only data which can be referred to by such a constraint are the parameters of the event and the identity of the source object. The destination object uses a notify constraint to express: *I want to be informed about those occurrences of this event which satisfy this condition.* For example, consider the common case of a `Collision` event whose parameters might include the identities of the objects that collided and to which many objects are likely to be subscribed. Many of them will be interested only in collisions between *themselves* and another object. Without notify constraints each collision event would be handled by every subscribed object, each of which would have to check whether the collision concerned it or not. Notify constraints restrict the propagation of events to only those objects which want to receive them. Since notify constraints do not depend on the local state of the destination object they can be evaluated in the context of the source object, or of some *event manager* object. In particular, in the distributed case notify constraints can be evaluated at the raising node avoiding unnecessary transmission of event notifications to remote nodes. An example of a notify constraint is given next:

```
Collision((object1=Monster AND object2=Wall) OR
(object1=Wall AND object2=Monster)).
```

3.2 The VOID class hierarchy

The VOID class hierarchy provides the designer with a class library containing much of the basic functionality shared by every game entity, a graphics library and a library of I/O classes.

When beginning a new entity definition the designer can explicitly combine classes from the hierarchy to synthesise a new class that has the properties necessary for the envisaged role. These properties come in three forms: concrete state (i.e. data); useful methods for maintaining this state; and encapsulation of the interface to standard classes (e.g., the Collision Manager, the

local Renderer etc). A "mixin" style of programming is supported whereby each entity includes only the necessary features via inheritance from a set of class hierarchies (see Fig. 5 which shows the bases classes of these hierarchies.)

Using existing code in this manner enables the developer to have usable classes with which to build a prototype system. The game-specific functionality of the new class is added subsequently by the designer (using statecharts) and the programmer (providing game logic in the form of event-handlers).

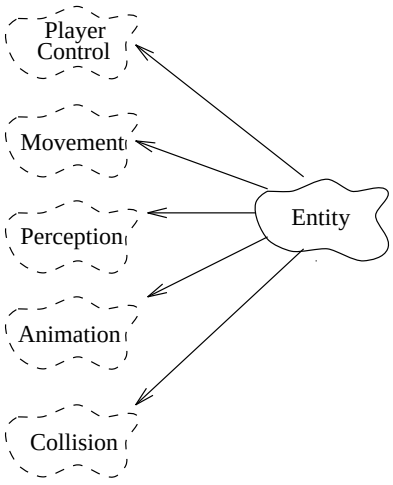

Figure 5: The VOID Class Hierarchy

The `Entity` class is the most basic class from which all other entities are derived. The main purpose of this class is to encapsulate the functionality common to all game entities.

The purpose of the `Movement` class is to provide entity classes which are derived from it with the appropriate state variables and the methods necessary to update them (see figure 5).

The `Animation` class is provided to give the designers the necessary functionality to control and display animated graphical representations of the entity, using facilities provided by the graphics library. In a 2D game the animations would consist of sequences of bitmaps. In a three-dimensional game the animations will consist of 3D graphical representations and associated matrices. In each case it is the responsibility of the sub-classes to ensure that the correct representation/matrix is sent to the renderer on each frame.

The `Collision Support` class is provided as a means of insulating the designer from the details of collision detection. It also obviates the need for the programmer to write very similar code for every entity. This class provides a bounding box and the tools to manipulate it and to choose different collision policies. It is the responsibility of this class to interface with the Collision Manager and to keep the Collision Manager informed of its position[2].

3.3 The Entity Editor

The Entity Editor is a tool used to specify how the entities in a game should behave. It allows game designers, with little or no programming knowledge, to create entity classes. The behaviour of an entity is defined by the actions that it performs in response to events, so the notation used by the Entity Editor must allow the designer to specify this simply and intuitively.

As the same event may not always trigger the same response from an entity (e.g., 'once bitten, twice shy'), a state-based notation is required for describing its behaviour. Rather than using ordinary State Transition Diagrams (STDs) as the basis for the notation, it was decided that Harel's statechart notation [6] would be a better choice, since:

- statecharts are a relatively intuitive notation, once a few basic concepts are understood;
- with appropriate state, event and action names, an entity statechart can be a very legible description of its behaviour;
- statecharts support the concept of modularity, allowing a hierarchical nesting of states. This supports the top-down design of entity behaviour;
- statecharts provide for orthogonality, allowing the state machine to be in two independent states at the one time;
- through modularity (or depth) and orthogonality, statecharts are significantly more efficient in the number of transitions and states required, when compared with STDs. This improves the clarity of the resulting statecharts.

The notation used by the Entity Editor is an extension of Harel's Statechart notation [8]. As well as specifying the states, the Entity Editor must allow the user to define actions (methods) and attributes (data members) of an entity class. In addition, the state transitions are be annotated as follows:

Event(params)[condition]/Action(params)

A transition may only be taken if the appropriate event has been delivered to the entity, and the condition evaluates to true. If the transition is taken, then the

[2] As you may guess, Collision isn't actually orthogonal to Movement: it is derived from it. To the users of the system, however, this is hidden by the tools and they don't need to concern themselves with it.

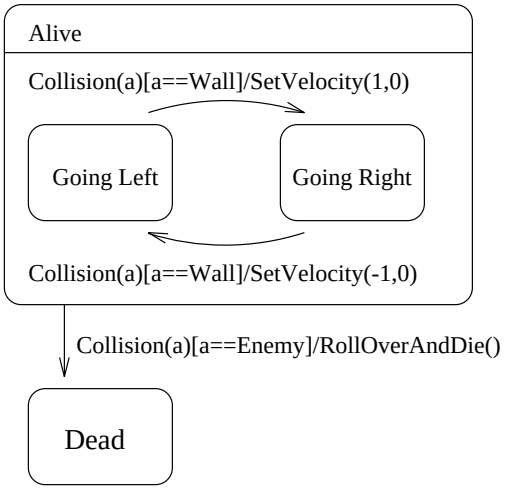

Figure 6: A Statechart example.

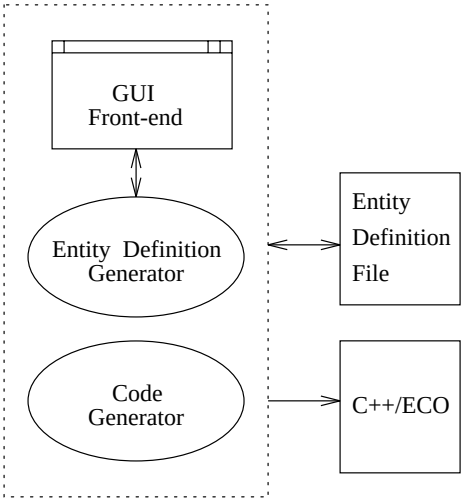

Figure 7: The Entity Editor.

state change occurs and the action is executed. It is important to note that work is only done when a state change occurs. Once in a new state, the entity does nothing but wait for the next event to be delivered.

Figure 6 depicts a simple entity statechart. Consider the entity in question to be a blind creature which spends all of its life walking back and forth between two walls. If it bumps into a wall, it just turns around and walks the other way. If it bumps into an enemy, it dies of fright.

The actions specified in a statechart can be inherited, or can be part of the entity class itself. Common actions, such as SetVelocity, could be provided by entity base classes. More specific actions, such as our creatures' RollOverAndDie, would be introduced for a specific entity, and would become part of that entity class definition.

Note that in this case the conditions may be implemented as notify constraints if the implicit parameter of the Collision event was available. The event indicates that the creature collided with something; the fact that the event refers to the creature is implicit. In general however, the condition may be a function of the data members of the entity, in which case it would not be implemented as a notify constraint.

As an application in its own right, the Entity Editor is somewhere between an integrated compiler and a drawing package. Its user interface borrows ideas from applications of both kinds. For drawing the statechart, typical functions such as resizing, moving, cutting and pasting of states must be available. As the Entity Editor is to be used for creating entity classes for video games, it must be easy for the user to build a set of classes for a single game. This is analogous to the way integrated compilers allow you to group many source modules in a single *project*.

The general structure is given in Fig.7. The Entity Definition Generator controls the GUI front end and is also capable of saving and loading Entity Definition Files (EDF). These files are purely a storage format for holding defined entity classes, and are not intended to be used for execution purposes.

The Code Generator is responsible for creating the code for a C++/ECO class that is an executable representation of the defined entity class. The code can then be compiled and linked within the VOID development environment.

Once all the entity classes have been defined, instances of them can be used to build the application universe. This is done through a world builder, which allows the initial internal state of each entity to be set, as well as enabling the game world to be built.

3.4 Distribution in VOID

VOID supports three scenarios for distributed execution: centralised, replicated and partitioned.

In the centralised case, only one copy of the game world exists and is managed by a single server. An advantage of this centralised architecture is that it is intrinsically simple with little work required for a new user to join the game. The synchronization is handled in one place and therefore is easy to realise. A disadvantage however, is that network traffic becomes high as output from the server must be broadcast to every participant. Also, as all activities are channeled through the central server, it may become a bottleneck as the number of participants grows. In addition there is a high dependency on the server.

With the replicated architecture, each participant has a replica of the game world. Every participant accepts input from other the participants' console, passes the input to its resident copy of the game, which then recomputes and generates the necessary output to the

user. A major advantage to this type of architecture is that a lower bandwidth is required, as only the input rather than output, needs to be distributed among users. In addition, all participants receive good interactive performance because they interact with local copies of the game or game. A disadvantage with replicated architectures, however, is that they are more complicated than centralised architectures and it is more difficult to maintain consistency across all copies of the shared game world. Thus synchronisation mechanisms are needed to ensure that every copy sees the same sequence of input events.

In the partitioned case the game is partitioned among the participant's consoles thereby supporting large scale applications which involve large numbers of state changes which may not be relevant to particular players at any given time. This may be particularly true with indoor scenes where certain entities within the world will not be visible [7]. MOONLIGHT is developing a mechanism for organising entities to overcome this problem. Entities may be grouped into sets called *zones* according to their location and/or functionality within the game world in a similar way to [9]. Entities in a particular zone are only subscribed to the events related to that zone. This limits the propagation of events to the nodes which have an entity present but which is not belonging to the zone. Zones may overlap and standard events may be defined for each zone to which all of its members are subscribed automatically as they join the zone (and unsubscribed as they leave).

4 Future Work

At the current time, the design and specification of the ECO model, the class library, and the entity and world editors is complete. A class library allowing C++ programs to use events and constraints is available while preprocessor and runtime support for the ECO language is being developed. First prototypes of the entity class library and graphics library are also available. In addition to language support for ECO current work is also concerned with the implementation of distributed event management and the detailed design of area management.

Acknowledgements

The authors would like to acknowledge the very many helpful contributions that our partners in the Moonlight consortium, from APD, Caption, Deltatec and NR, have made in the design of Void to date. Special thanks are due to Luis del Pino.

References

[1] Moonlight Management Board. Technical Annex to the Moonlight Contract, September 93.

[2] Vinny Cahill, Andrew Condon, Dermot Kelly, Stephen McGerty, Karl O'Connell, Gradimir Starovic and Brendan Tangney. VOID Shell Specification. Deliverable 1.5.1, TCD, 95.

[3] Vinny Cahill, Andrew Condon, Gradimir Starovic, and Brendan Tangney. VOID Shell and Execution Environment Definition. Deliverable 1.2.1 and 1.3.1, TCD, 94.

[4] Vinny Cahill, Christine Hogan, Alan Judge, Darragh O'Grady, Brendan Tangney, and Paul Taylor. Extensible systems - the Tigger approach. In *Proceedings of the SIGOPS European Workshop*, pages 151–153. ACM SIGOPS, September 1994. Also technical report TCD-CS-94-07, Dept. of Computer Science, Trinity College Dublin.

[5] Pascal Cottin. Moonlight Project Graphics Interface Definition. Technical report, Caption Groupe Telmat, 1994.

[6] David Harel. On visual formalisms. *Communications of the ACM*, 31(5):514–530, May 1988.

[7] R Kazman. Load Balancing, Latency Management and Separation of Concerns in a Distributed Virtual World. Technical report, Department of Computer Science, University of Waterloo, Ontario, Canada, 1993.

[8] Stephen McGerty, Vinny Cahill, Andrew Condon, Karl O'Connell, Gradimir Starovic, and Brendan Tangney. Moonlight Entities and Statecharts, 1995. Working paper TCD.

[9] Macedonia M.R., Zyda M, Pratt D.R., and T Barham P.T. Exploiting Reality with Multicast Groups: A Network Architecture for Large Scale Virtual Environments. Technical report, Computer Science Dept, Naval Postgraduate School, 1994. Also in proceedings of VRAIS'95.

[10] Gradimir Starovic, Vinny Cahill, and Brendan Tangney. ECO: Events + Constraints + Objects. Technical Report TCD-CS-95-02, Distributed Systems Group, Computer Science Dept.,Trinity College Dublin, 1995.

[11] Chris Zimmermann and Vinny Cahill. Roo: A framework for real-time threads. In *Proceedings of the Workshop on Parallel and Distributed Real-Time Systems*, 1995.