Real + Virtual = Clever
Thoughts on Programming Smart Environments

Mads Haahr[1], Vinny Cahill[1], and Eric Jul[2]

[1] Department of Computer Science, Trinity College, Dublin 2, Ireland
{Mads.Haahr,Vinny.Cahill}@cs.tcd.ie
[2] Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 København Ø, Denmark
eric@diku.dk

Abstract. Event-based communications has been used successfully in many application domains, one of which is virtual environments. Events are a useful concept in this context because they embody the notion of something happening in an environment, be it real or virtual. We claim that the notion of events is not only a suitable communications paradigm to model purely virtual environments but that it can also be used to *interface* an area in a real environment with a corresponding area in a virtual environment by relaying events in both directions. This idea, in turn, could turn out useful as a programming model for smart environments. As part of our recent work in distributed event systems for virtual world support, we have implemented an event model called ECO and used it to build a virtual model of a real world environment. In this position paper, we describe how such a virtual environment can be interfaced to its real world counterpart. We argue that this technology is promising as a way of programming smart environments because it maps naturally onto the application domain and simplifies the programming of such environments. To support this view, we present some examples of functionality often found in smart environments and explain how they can be programmed with the technology presented in this paper.

1 Introduction

Event-based communications has been used successfully in many application domains, one of which is virtual world support [5]. Events are a useful concept in this context because they embody the notion of something happening in a world, be it real or virtual. We claim that the notion of events is not only a suitable communications paradigm to model purely virtual worlds but that it can also be used to *interface* one or more areas in the real world with a corresponding area in a virtual world by relaying events in both directions. This idea, in turn, could turn out useful as a programming model for smart environments.

This position paper describes how event-mapping can be done and presents an application which can map a single type of event between a real and a virtual environment. We argue that this technology is promising as a way of programming smart environments because it seems to map naturally onto the application domain and make the programming of such environments simpler. To support this view, we present some examples of functionality often found in smart environments and explain how they can be programmed with the technology presented in this paper.

2 The ECO Model

The ECO Model is an event model developed for distributed virtual world support. It was used in the Moonlight [4] project (a distributed 3D video games project) and, as event models go, it is relatively simple. It has only three central concepts and its application programmer interface (API) contains only three operations. The intent of the model is that it is applied to a given host language and extends that language's syntax and facilities so as to support the ECO concepts. Though the ECO model has traditionally been used in a virtual world context [8, 6], the model itself is generic, and can easily be used in other domains where event-based communication applies. This section describes the ECO concepts and operations.

2.1 Events, Constraints and Objects

The acronym ECO stands for *events, constraints*, and *objects*—the three central concepts in the event model:

Objects in the ECO model are much like objects in a standard object-oriented language. However, instead of invoking other objects for communication ECO objects communicate with other ECO objects via events and constraints as explained below. ECO objects are often implemented as programming language objects but not all programming language objects are necessarily ECO objects. In order to distinguish the two, ECO objects are often referred to as *entities*. Entities have identifiers that are unique within an ECO world and they may contain threads of control.

Events are the only means of communication in the model. Entities do not invoke each other's methods directly but instead raise events which may, or may not, lead to other entities' methods being invoked. Any entity can raise an event. Events are typed and have parameters, and they are propagated asynchronously and anonymously to the receiving entities in no particular order. The type of events is usually specified using the type system of the underlying language.

Constraints make it possible for entities to impose restrictions upon which events they actually receive. The ECO model specifies several types of constraints: *pre, post, synchronisation*, and *notify* constraints. Notify constraints (known as *filters* in some event models) can be used by an entity to specify what events it is interested in receiving *notification* about.

The three concepts are illustrated in figure 1 which shows two ECO entities communicating. Entity A raises an event which may, or may not, reach entity B

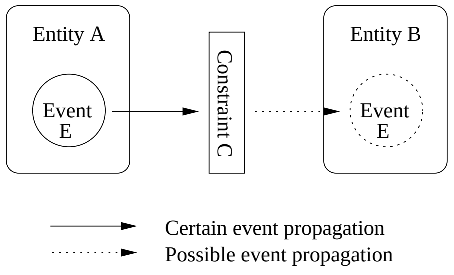

Fig. 1. The Three ECO Concepts in Relation

because of the constraint C. The constraint is imposed by entity B. The raising of an event can be thought of as an announcement to the rest of the ECO world that the event has occurred. A notify constraint can be thought of as a filter that decides whether or not a given entity is to receive the event, and receiving an event can be thought of as invoking an appropriate method (called an *event handler*) of an entity in response to the event. When an entity uses a notify constraint to enable it to receive certain events, we say that it *subscribes* to those events. An entity can subscribe multiple times to the same events using different constraints and handlers. It is also possible to subscribe without using a constraint, in which case no filtering is performed.

2.2 The ECO Model's API

The ECO model's API contains three operations which are used by entities to communicate:

`Subscribe(event-type(notify-constraint), event-handler)` is used by entities to register interest in events. An entity that subscribes to a certain type of event will receive an invocation of one of its methods when a matching event is raised. The event is passed as a parameter to the handler. When an entity performs a subscription, it can also choose to specify a constraint. An event must be of the right type and must match the constraint (if any) to be delivered to a particular subscriber.

`Raise(event)` is used by an entity to announce the occurrence of an event. The event is delivered to all entities subscribing to events of that type, subject to filtering against their respective constraints.

`Unsubscribe(event-type(notify-constraint), event-handler)` is used by an entity to cancel an existing subscription.

3 The Real and the Virtual Environment

As mentioned in section 1, our environment consists of a real and a virtual part. This section presents the two halves of this environment and the next discusses how events can be mapped between them.

3.1 The Real Environment

The real environment is a series of research laboratories found at the University of Cambridge. The laboratories are equipped with specialised hardware called an active badge system. The badge system consists of a number of infrared sensors (called stations) placed in the rooms and hallways of the university buildings. The stations pick up signals emitted by battery-driven badges worn by personnel in the labs. When a station detects the presence of a badge, it raises a so-called sighting event to announce that this particular badge has been seen in that particular location. Stations are grouped into networks, each being a part of a particular laboratory. Each badge carries a unique badge identifier which is picked up by the sensors.

Because we do not have access to the actual badge system, the events raised in the virtual environment are replays of events that happened at an earlier point in time. The data we have obtained from the system consists of 35,811 sighting events collected over period of almost 21 hours by 118 stations distributed over 12 networks. It is worth noting that the virtual environment is run as a simulation (i.e., using previously collected data rather than real-time data) only because we do not have access to the data-generating hardware. Given the required hardware, the virtual environment could easily be maintained in real-time. For each sighting, the following information is available:

Location Identifier identifies a physical location in the environment. The location identifier consists of a *network* (a symbolic name) specifying an area of the building and a *station* (an integer) specifying a location within that area. Assuming there is only one station in each physical location, the location identifier can be mapped uniquely to a location in the virtual world.

Badge Identifier identifies the sighted badge by a sequence of six eight-bit hexadecimal numbers separated by dashes. This identifier can be mapped to the real name of the badge owner.

Time Stamp identifies the moment when the sighting was made in seconds and microseconds, since 00:00:00 UTC, January 1, 1970, as returned by the `time(3)` Unix system call.

3.2 The Virtual Environment

The virtual environment is a model of the real environment and forms a shadow world where events happening in the real world are echoed. For each station in the real laboratories there is a corresponding location in the virtual environment and for each badge identifier there is an entity in the virtual environment representing that person. The entities are moved between locations in the virtual environment as sighting events are raised by the stations.

The station entities in the virtual environment represent physical pieces of equipment and echo real events by raising virtual (ECO) events. In addition to the station entities, the virtual environment also contains entities with no physical counterparts. These entities can interact with each other and the rest of the virtual environment by using the three ECO operations to subscribe to events (such as those raised by the station entities) and raise events of their own. Below is a description of a series of such entities.

Location entities implement rooms in the virtual environment. For each station there is a corresponding location entity keeping track of which entities are currently in that location. This location entity subscribes to all sighting events from the corresponding station in order to detect arrivals of new entities. Entities are assumed to remain in the location where they were last sighted until they are sighted elsewhere. For this reason, a given location entity also subscribes to all sighting events featuring badge identifiers which are currently in its own location. A location entity can be queried (using an ordinary non-ECO invocation) as to who is currently in that location.

Ghost entities implement people in the virtual environment who have no physical counterparts in the real environment and therefore do not cause the sensors to raise sighting events. A ghost entity is controlled remotely by a user using a text-based command line interface. Ghost entities interact with location entities in order to move around and to present their remote users with views of their current locations in the form of a textual descriptions.

CCTV (closed circuit television) entities implement security cameras in the virtual environment. A CCTV entity subscribes to all sighting events occurring within a particular network and in this way monitors a small area of the entire virtual environment.

4 Interfacing the Real and the Virtual Environment

This section discusses a possible mapping of events between the two environments by looking at how one type of event, the *sighting* event, can be mapped in both directions. In general, it is important to distinguish between the representation and the presentation of a virtual environment. The former is the way the environment is structured and stored in the computers' memory and is independent of any input/output devices individual users may have. The latter is the way the environment is presented to individual users and is therefore dependent on such hardware. In particular, it is possible to have the same virtual environment presented differently to different users.

There are many ways of presenting virtual environments to users, ranging from simple text-based interfaces as those used in MOOs (Multi-User Environments, Object-Oriented) [1] over first person perspective 3D graphics commonly used in popular games such as Quake, to full-blown virtual reality equipment such as stereoscopic VR goggles or even CAVE theaters [3].

For the purposes of this paper, we use the simplest possible presentation: a text-based interface akin to that used in MOOs. It is important to stress, however, that the ideas discussed here are general and will be usable with other presentation techniques. It should also be noted that researchers from the Computer Laboratory at the University of Cambridge are doing work which is somewhat related to that presented here. Their approach is to use the active badge system to maintain a VRML2 representation of the labs. This enables users to interact with the environment using a graphical rather than a text-based interface [2].

It should also be noted that this is work in progress and most of our effort so far has been to implement an application that can map real to virtual sighting events. Therefore, the discussion of real-to-virtual event mapping is more thorough than that of virtual-to-real mapping.

4.1 Example

The following is an example of a purely virtual user interacting with the environment. At the user's location is another virtual user, John, and a real person, Jane. Jane herself is physically present at the corresponding location in the real environment whereas the two virtual users are only present at the virtual location. User commands issued by the virtual user from whom the transcript is taken are preceded by a prompt (>) character.

```
You are on the 3rd floor of the Research Laboratory.
A sign on the wall reads, ‘Area 8.’  A sensor in the ceiling is marked, ‘11.’
John (virtual) is here.
>look at john
John is transparent and has no badge identifier.
Jane (real) arrives.
>look at jane
Jane’s badge identifier is 0-0-0-0-10-14.
```

There are several things to note about this example. First, John and Jane are marked as virtual and real mainly for reasons of clarity. Distinguishing between users in this way is not a strict necessity. Second, John has no badge (he is purely virtual) and therefore no badge identifier. Third, this scenario is limited because it features only one type of event: the sighting event. Though this event enables users to see each other and move around (by being sighted in different places) it offers few possibilities of interaction.

4.2 Sighting Events

In the example, the location entity keeps track of any real and virtual users present at the location. When Jane arrives physically at the location, the signal emitted by her infrared badge is picked up by the sensor in the ceiling and a sighting event is raised. This event is detected by the location entity because it subscribes to all sighting events raised by that sensor. Hence, Jane's arrival in the physical location of the real environment triggers her virtual arrival in the corresponding location of the virtual environment. This is effectively a mapping from a real to a virtual sighting event.

For sighting events, the purpose of a virtual-to-real mapping is to let Jane in the real location obtain a sighting of a virtual person in the virtual location. (If the person is real rather than virtual, we assume Jane will be able to see him with her own eyes.) This is a presentation issue and can be adressed in many different ways. In case the location is Jane's office and she has a terminal on her desk, a simple terminal window could be used to inform her about the presence of virtual users in her virtual office. If the location is a hallway or meeting room, a screen (possibly flat and wall-mounted to take up as little space as possible) could be placed in the location and messages about the presence of virtual entities could be either printed (MOO-style) or drawn (as images) on the screen. More ambitious approaches could use more advanced hardware to give a more lifelike presentation of events in the virtual world. For example, a holographic projection of each virtual entity would be ideal but is hardly feasible with current technology. Comparable results, however, might be achieved by equipping Jane with a wearable rendering computer and a pair of transparent VR goggles where an image of the virtual environment is superimposed that of the real environment. Regardless of whether a low- or a high-tech approach is adopted, the presentation would let Jane observe events happening in the virtual world. Analogously, Jane's badge would let virtual people see her. Together, these two mappings constitute a full mapping of sighting events in both directions between the environments.

5 Smart Environments

The previous sections have described a real and a virtual environment and explained how they could be closely interfaced by mapping events in both directions. So, the next question is, in what way is this technology useful for smart environments research? A smart environment can be seen as an 'ordinary' real environment with certain extra functionality that enables it to *monitor* its own state and to *modify* that state. When a smart building detects the presence of a disabled person in front a door, the detection is a result of the building monitoring its own state. (We here assume people in the building to be part of the environment.) When the building opens the door to let the person through, it is modifying that state.

Sections 3 and 4 argued that such state can be naturally represented as a virtual environment mirroring the the real environment and that *events* can be used to keep the two environments synchronised. In fact, event mapping is just another word for monitoring and modifying the real environment's state. When events are mapped from real to virtual, the real environment's state is being monitored. When they are mapped from virtual to real, the state is being modified.

The section 4.1 example used an extremely simple environment, featuring only surveillance-type functionality and using only a few types of entities and one type of events. *Surveillance* is only one area in which this technology can be applied. In the following, we discuss this and two other areas known from smart environments, namely *planning-* and *action-type* applications. The example from section 4.1 is extended to illustrate the points, and actual C++ code with ECO statements is used to explain how the events are used. It is worth noting that this section is work in progress and because the ideas are largely untested, the discussion is somewhat speculative. The code given in this section assumes the existence of a sighting event class along the lines of that given below.

```
class SightingEvent : public Event {
  string badge_id;  // The badge seen.
  string network;   // Location identifier:
  int station;      // (network, station).
};
```

5.1 Monitoring and Surveillance

Many ‘dumb’ buildings are equipped with security cameras to monitor areas which are deemed sensitive for one reason or another. Such cameras generally have the disadvantage that they have to be monitored by people rather than software because detailed digital image analysis is not only extremely difficult but also very time-consuming. Consequently, the information that can be gathered from such cameras is in practice rather limited. Areas are often monitored only for security purposes; other information, such as the movement patterns of (legitimate) users, is not recorded.

Smart environments can incorporate user tracking, for example via active badges as those described in section 3.1. This makes it easy to analyse the movement patterns of the users of the environment. The CCTV camera from section 3.2 is an example of a simple analysis tool, monitoring a small area of the complete environment. The following C++ code with ECO statements shows how a simple CCTV camera could be implemented; it subscribes to all sightings in a particular area and logs them to standard output. Recall that the *network* and *station* constitute a location identifier and that a *network* consists of a series of (related) locations. Asterisks ($*$) denote wildcards, i.e., fields matching any value.

```
class CCTV {
  string network;
  int station;

  CCTV(string nw) : network(nw) {
    // Detect all users in this area (‘network’) of the building.
    Subscribe(SightingEvent(*, network, *), &EventHandler);
  }
  ~CCTV() {
    // Cancel subscription.
    Unsubscribe(SightingEvent(*, network, *), &EventHandler);
  }
  void EventHandler(SightingEvent se) {
    // Log the sighting to stdout.
    cout << "Badge " << se.badge_id << " seen.\n";
  }
};
```

Entities like this CCTV camera are useful because the complete event flow through any sizable smart environment will be large and difficult to comprehend. Entities such as this can be used to provide meaningful views of this event flow by dynamically extracting events according to certain patterns, and in this way make it easier for humans to monitor the system at runtime. The CCTV camera extracts sightings on the basis of their *location* but other entities could for example monitor certain *users*, or use a combination of the two.

Using smart environments with user tracking doubtlessly has its advantages but also raises serious ethical issues about privacy. It is easy to picture technology like that described here being used to implement surveillance of Orwellian [7] proportions on the grounds that it is for the users' own good.

5.2 Planning

Planning routes through buildings is a useful functionality for visitors and robots and one that is often discussed in the context of smart environments. To make a plan from one location to another, a building must be aware of its structure, e.g., must know the layout of floors and the locations of lifts, stairs and doors. It must also take into account the abilitites of the user (be it a visitor or a robot), e.g., whether he/she/it can use lifts and stairs and holds the right keys and access codes to doors on the way. Even after a suitable plan has been laid out, a smart building may still help the user follow it for example by displaying encouraging messages on screens on the way.

Making the plan is an algorithmic problem and beyond the scope of this paper. However, once a plan has been made, the environment can assist the user in carrying it out. In the context of the section 4.1 example, this could be implemented with virtual *signpost* entities placed in locations along the route. Consider the following version of the example:

```
You are on the 3rd floor of the Research Laboratory.
A sign on the wall reads, 'Area 8.'  A sensor in the ceiling is marked, '11.'
Jane (real) arrives.
A green arrow bearing the name 'Jane' in large, friendly letters suddenly
appears on the wall.  It is pointing north.
Jane (real) goes north.
The green arrow vanishes.
```

This example features a signpost visible in the virtual world. Obviously, signposts would also have to be visible in the real world, in order to be useful for real users. For human users, they could be printed on displays in the various locations. For robots, they could be transmitted via short-range wireless links such as infrared. Note that directions only appear when the user enters the location and that they disappear when the user has left. This prevents an environment with many visitors from becoming cluttered with messages. Also, in the case of human users, chances are that the appearance of the message will attract the user's attention at the right moment.

The following C++ code with ECO statements shows how signposts could be implemented. A series of signposts could be created by a planning application and placed in the environment along the projected route. The exact placement of signposts is of course open to discussion; one could for example claim that a user does not need signposts when following the projected path, only when straying from it. However, for the purposes of this example, we will assume they are placed in the locations on the path the user is expected to follow.

```
class SignPost {
  string badge_id, network;
  int station;

  SignPost(string bid, string nw, int st)
    : badge_id(bid), network(nw), station(st) {
    // Start looking for our user in our location.
    Subscribe(SightingEvent(badge_id, network, station), &Activate);
  }
  // Activation Handler.
  void Activate(SightingEvent& se) {
    // User is here; become visible (code not shown) and
    // stop looking for the user in our location.
    Unsubscribe(SightingEvent(badge_id, network, station), &Activate);
    // Start looking for the user anywhere.
    Subscribe(SightingEvent(badge_id, *, *), &Deactivate);
  }
  // Deactivation Handler.
  void Deactivate(SightingEvent& se) {
    if (se.station != station || se.network != network) {
      // User is sighted somewhere else; stop looking.
      Unsubscribe(SightingEvent(badge_id, *, *), &Deactivate);
      // Destroy ourselves (code not shown).
    }
  }
};
```

During the lifetime of a signpost entity, each of the above methods will be invoked once in the order of *constructor*, *activation handler* and *deactivation handler*. The two handlers will be invoked when the appropriate sighting events (matching the subscriptions) occur. The signpost example shows how a simple guidance system can be implemented rather easily given a virtual representation of the real environment and mapping of sighting events in both directions.

5.3 Action

Another function of smart environments is to perform certain *actions* automatically and on user request. Common examples are calling lifts and opening/closing doors in the environment. This type of functionality is useful for users with limited physical abilities, such as disabled people and robots, but also for users who need to control certain aspects of the environment remotely.

Consider a smart environment with automatic doors that can be opened by anybody with the right key. When opened, a door stays open for 30 seconds whereafter it closes. In the virtual environment, doors are represented as ECO entities and keys as strings. A door can be attempted opened by raising a `PleaseOpenDoorEvent` in the virtual environment. This event contains the key (placed there by the entity raising the event) and is received by the door entity. If the key matches, the door opens. This is signalled by the door with a `DoorOpenEvent`. When the door closes, it raises a `DoorCloseEvent`. Each door has a unique identifier in the form of a string. C++ code for the three event types is given below.

```
// Raised by somebody who wants to open a door.
class PleaseOpenDoorEvent : public Event {
  string door, key;
};
```

```
// Raised by the door when it opens.
class DoorOpenEvent : public Event {
  string door;
};
// Raised by the door when it closes.
class DoorCloseEvent : public Event {
  string door;
};
```

Note that ECO events are global; there is only one event space where all events are raised. Hence, anybody can raise a `PleaseOpenDoorEvent` regardless of their location; in particular, it is not necessary to be anywhere near the door. In practice, the `PleaseOpenDoorEvent` could be raised either by actual users (when standing in front of the door), by security personnel in remote locations or by other parts of the building, such as the signpost entities described in section 5.2. The code for the door entity is given below.

```
class Door {
  string my_id, my_key;
  bool open;

  Door(string id, string key) : my_id(id), my_key(key) {
    // Subscribe to all request to open us.
    Subscribe(PleaseOpenDoorEvent(my_id, my_key), &Handler);
  }
  ~Door() {
    Unsubscribe(PleaseOpenDoorEvent(my_id, my_key), &Handler);
  }
  void Handler(PleaseOpenDoorEvent& ev) {
    // Rudimentary security.
    if (ev.key == my_key) {
      // Physically open door (code not shown) and raise
      // event to tell the environment it is now open.
      Raise(DoorOpenEvent(my_id));
      // Wait a while to let people pass through.
      sleep(30);
      // Physically close door (code not shown) and raise
      // event to tell the environment it is now closed.
      Raise(DoorCloseEvent(my_id));
    }
  }
};
```

This example shows how event-based programming combined with a virtual representation of the environment gives a high degree of flexibility in controlling and programming the building. For example, a remote control button for the door is easily implemented as an entity that raises the appropriate `PleaseOpenDoorEvent`. It can be placed anywhere in the building and even moved at runtime.

6 Conclusions and Future Work

This paper has presented ideas for closely interfacing real and virtual environments by mapping events between the environments in both directions. We have also described an ongoing implementation of these ideas which features only a single type of event, the sighting event, and only the simplest possible presentation layer, a text-based interface. Despite these limitations, we claim that the principles generalise and will remain valid in more complex scenarios with many event types and multiple (and more advanced) presentation layers. We have also briefly described the ECO event model which the implementation uses for event-based communication.

We have argued that this technology can be applied in the context of smart environments by giving examples of some typical applications from the domain and explained how they could be implemented with the technology. The examples suggested that at least some smart environment applications could be kept quite simple even for large environments. Our conclusion is that closely interfaced real/virtual environments is a promising programming model for smart environments and is worth investigating further.

Acknowledgements

The authors are very grateful to John Bates of AT&T Laboratories, Cambridge for supplying the data from the active badge system.

References

1. R. A. Bartle. Interactive Multi-Player Computer Games. Technical report, MUSE Ltd, Colchester, Essex, UK, 1990.
2. John Bates, Jean Bacon, Ken Moody, and Mark Spiteri. Using Events for the Scalable Federation of Heterogeneous Components. In *Proceedings of the Eighth ACM SIGOPS European Workshop*, September 1998.
3. C. Cruz-Neira, D. J. Sandin, T. A. DeFanti, R. V. Kenyon, and J. C. Hart. The CAVE: Audio Visual Experience Automatic Virtual Environment. *Communications of the ACM*, 35(6):65–72, June 1992.
4. TCD Team Moonlight. VOID Shell Specification. Project Deliverable Moonlight Del-1.5.1, Distributed Systems Group, Department of Computer Science, Trinity College, Dublin 2, Ireland, March 1995. Also technical report TCD-CS-95-??, Dept. of Computer Science, Trinity College Dublin.
5. Karl O'Connell. *System Support for Multi-User Distributed Virtual Worlds*. PhD thesis, Trinity College, Department of Computer Science, Dublin 2, Ireland, October 1997.
6. Karl O'Connell, Tom Dinneen, Steven Collins, Brendan Tangney, Neville Harris, and Vinny Cahill. Techniques for Handling Scale and Distribution in Virtual Worlds. In *Proceedings of the Seventh ACM SIGOPS European Workshop*, pages 17–24. Association for Computing Machinery, September 1996.
7. George Orwell. *1984*. New American Library, 1990. ISBN 0-451-52493-4.
8. Gradimir Starovic, Vinny Cahill, and Brendan Tangney. An Event Based Object Model for Distributed Programming. In John Murphy and Brian Stone, editors, *Proceedings of the 1995 International Conference on Object Oriented Information Systems*, pages 72–86, London, December 1995. Dublin City University, Ireland, Springer-Verlag.