Some Useful Abstractions for Re-Usable Virtual Environment Platforms

Anthony Steed*

Department of Computer Science, University College London

ABSTRACT

Within the virtual environments community there is a large cost of maintaining software demonstrations and applications whilst hardware and low-level software changes. In our own laboratory, over the last 15 years, students and staff have spent considerable time and effort writing demonstrations and applications on at least 40 significant VE software systems, ranging from relatively low-level APIs that impose little constraint on software-engineering up to relatively high-level platforms that have quite fixed structure. In this short paper we discuss some observations about some abstractions that have served us well both in the creation of virtual environments (VEs) and in the development of platforms.

Keywords: virtual environments, interactive systems, event mechanisms, inter-operability.

1 INTRODUCTION

There is and has been a plethora of software solutions for the creation of virtual environments (VEs). Partly this is because the field has changed quite a lot, and software engineering practices and languages, have changed; but mainly it’s because VE software is a large domain, covering scientific visualization through to social spaces. The range of concerns including quite disparate areas of computer science, from parallel languages through constraint satisfaction to efficient peer-to-peer networking protocols. Each VE software package thus necessarily covers a relatively small area of the domain.

Just with our own lab, a partial list of the software solutions that we have invested more than a person-month of time in using would include: dVSe/dVISE, AVS, OpenSG, OSG, Performer, Torque, CrystalSpace, OGRE, Alice, CAVElib, Panda3D, Chromium, Direct3D, Inventor, COIN, Java3D, JoGL, MASSIVE1/2, SecondLife, Quake1/2/3, PowerVR, Avango, DIVERSE, VRJuggler1/2, Unreal2, LightSpace, VTK, OpenVRML, Oblivion, GoogleEarth, Renderware, MetaVR, XVR, GKS, VRPN, Superscape, Core, WorldToolkit, XP, CoVISE, Flow, EyeCVE, Uni-verse, StudierStube. Each of these software systems, which range from high-level libraries such as scene-graphs (e.g. OSG) through to full platforms such as Avango. This list doesn’t include a much larger list of supporting libraries and formats such as OpenGL, XGL, Cal3D, RakNet, Collada, PhysX, etc.

There are several reasons why we have used so many different systems, ranging from technical through to social:

- Collaboration with the authors of the toolkit
- Good fit to student or staff knowledge
- Good fit to the application domain
- Known to support a very specific feature we wanted

- An application we wanted to build was similar to an existing demonstration
- System was highly popular at the time, and students came wanting to use a specific system.

More telling, we stopped using systems for a variety of reasons:

- Knowledge was lost (usually by the owner graduating)
- System was retired by the authors
- System was no longer up to scratch (e.g. visually)
- System was perceived as being hard to program or lacking in capability, suggesting a move to a similar, but potentially more powerful system
- Lack of a critical facility (e.g. no cluster support)
- Hardware or operating system support was no longer available

There is no doubt that a cycle of system software changes will continue and in some ways this is healthy: software does get more reliable, easier to program and has richer functionality. However, we, as a lab, end up re-writing a lot of code on different toolkits.

In this short paper we give some observations based on the experience of VE software engineering in our laboratory. These indicate some abstractions that have served us well both in maintaining demonstrations whilst software and hardware changes around them, and also that have facilitated easy re-implementation in other packages as necessary. Thus this paper does not attempt to suggest a particular framework, or make a proposal for the software engineering of future platforms. Such discussions are very valuable, and we can learn a lot about from the motivations of the development of such frameworks (e.g. see [6] [7][8] [11]). Rather, this paper makes some modest suggestions about the pragmatics of writing VE code so that it can be re-used in the future. Most of our examples will be based on the DIVE system [2][3][4]. This is because system is the one our group has had longest experience with, and has been able to influence the later development of. However, the remarkable thing is of all the software we have, DIVE demonstrations from 10 years ago are still running, without change to the demonstrations themselves, whilst the hardware and operating systems in the lab have changed. This is because there is a clear separation between what the demonstration scene contains, and what the platform itself provides.

2 THE DIVE SYSTEM

The DIVE system went through several iterations, and many papers were written about its engineering and evaluation. It was designed from the outset to support networked environments. At its core was a set of libraries, written in C in a modular form, with different modules responsible for maintaining a scene, interfacing to devices and so on. DIVE defined its own data-structures, called entities, which described a scene in the manner of a scene-graph, though there were entities that described non-visual data. The simplest DIVE process would create a scene directly by instancing nodes, but there were modules to load external scene description formats, including VRML97 and its own “vr” format. A scene loaded by one process could then be shared between processes over the network, the second and subsequent processes to connect
to a scene, would load it from one of the already connected processes. Certain processes could be renderers, and over the years, renderers were written in XGL, RenderWare, OpenGL, Performer and OpenSG: the entity data structures were quite neutral about geometry and texture formats. A process could disconnect from a scene and connect to another one whilst it was running. All resources could be downloaded from the web or from local disk.

An important aspect was that there were standardized application that could connect to a scene and provide a well-known 2D & 3D interface to the scene. The most common of these in use was *vishnu*, which provided a 2D desktop view of the scene, and allowed various menu and mouse interfaces. Different processes would provide different interfaces: *spelunk* was a variation of *vishnu* which supported CAVE-like displays through CAVElib [12] and later VRJuggler; there have been a variety of specialized devices for mobile and AR systems. Both vishnu and spelunk were scriptable in the TCL language, and interfaces to CAVElib or VRJuggler were added in the core C libraries.

DIVE had integration with TCL at two levels: scenes could contain several small TCL scripts to executed based on events (see Section 3.1), and many processes contained an embedded TCL interpreter to control their 2D user-interface. For a much more detailed description of the later iterations of DIVE, see [2][3]. It is worth mentioning that a process did not have to have a TCL interpreter in order to connect to the system. In fact, it was planned, though never implemented, to support multiple (optional) scripting languages in the run-time.

The other facilities that vishnu and spelunk would provide would be lists of other users, tools to setup audio and video links, a scene tree browser and scene editing tools.

3 ABSTRACTIONS FOR THE INDIVIDUAL SYSTEM

3.1 Object-Referential Publish-Subscribe Event System

Event systems under-pin 2D windowing systems, but their use in 3D application systems is not so common. Many 2D user-interface toolkits “hide” the event system, by having callbacks implemented as (possibly delegated) member functions within classes. In 3D applications, such an approach is not so attractive, as although, with a 2D widget, it is possible to easily vary the appearance, with a 3D application, any 3D object can be a “button”, and thus the representation is primary, not the behavior. A publish-subscribe event system provides, in our opinion, a neat way of separating the two issues: in a scene-graph, there are logical objects that act as containers for scene fragments. These can be named, and identified, and scripts can register interest in certain events associated with those nodes. These can be standard graph manipulation, but also application-level events such as select. Scripts can also register interest in an event on any object.

Systems like Inventor and VRML, describe an event system based on data-flows, defined in the model. There is quite considerable interest in data-flow and similar abstractions for describing data processing for 3D (e.g. [1][10]) but despite our own interest in the design of such languages, in practice we have found that systems with event filters are easier for programmers to understand and they have a more natural interpretation in a networking situation, where there are more potential sources of specific events such as interaction events.

In the DIVE system, the event mechanism was quite pervasive, in that most changes to data structures shared between different code modules, was done through the event system, such as collision and simulation, interaction, device reading and triggering of rendering functions (audio & video) and external interfaces. This allowed for fluid interactions between script at an application level, the underlying run-time and any run-time plugins, since they could all send and consume events, and there was no need for recourse to API for different parts of the system to share data. This was most typically done for light-weight interactions between modules and scripts (e.g. notification of data loaded, specific application event occurring such as a user joining), but it was robust and complete enough that the networking system could be built on top of it, to share the whole scene and for the Performer renderer to be built on it without recourse to walking an in-memory scene representation [12].

3.2 Decoupling Interface from Application

The key problem that VE application programmers face is that of platform abstraction: there are lots of hardware facilities out there that use a variety of different software solutions. In our experience, this is the principle cause in problems in maintaining demonstrations and sharing them with colleagues; not only do our own facilities change, or someone wants to run a networked version of a demonstration, requiring porting to a new machine, but our colleagues, ostensibly similar facilities are configured in a subtly different way. We have used three abstractions that support device transparency: User, Body and Vehicle.

3.2.1 Vehicle Abstraction

Today, there are several libraries that support a variety of devices: VRPN, CAVElib VRJuggler and DIVERSE are four that we have used for various demonstrations. None is completely comprehensive, and one reason that we have used all four is that we have legacy software with an interface to a subset. However, these all tackle device abstraction at an API level: you can name devices, and retrieve them by abstract identifier. It is still up to the programmer to do the first layer of interface: deciding what buttons do what. With applications written to use these APIs is quite usual to see a construct at the application level such as:

```
if there is a joystick then
  rotate camera if joystick displaced
else if there is a hand tracker
  rotate camera if middle button pressed
else
  rotate camera on arrow keys
```

This code will commonly be found in dozens of applications written for that API, and one of the most common problems that we have had with maintenance is the upgrading of applications when a new mechanism is required (e.g. adding a new display-type with multi-touch). The ability to use these abstract tracker names is useful, but we would argue that the basics of moving, selecting and manipulating is actually a configuration of the system, not the application.

The DIVE system had a notion of a vehicle which was responsible for moving the user around, and implementing object selection and manipulation. The concept was that applications could create at least one vehicle when they were started. The vishnu process had many options for enabling and configuring vehicles (e.g. one vehicle used on-scene virtual widgets, another a physical joystick, another a “QuakeKeys” metaphor). The spelunk vehicle would check for the presence of joysticks and implement flying in the direction of gaze or point, or aim, etc. It was important for re-usability that these configurations were not in the scene. However, if you really wanted a new interaction technique for a particular scene you could write a new vehicle with TCL scripting, or a plugin – you would just disable any existing vehicle (or leave it running if it didn’t interfere).

The vehicle thus provides a default set of interactions. All vehicles need to generate events that publicise the decision made, they must issue move or velocity changes events on the user...
object (see below), they issue select and manipulate events. For manipulation the vehicle issues a “grab” event, and then moves the object. The scene description would include flags or hints about what objects can be manipulated, but they can also veto the grab by sending a release event. This convention has worked well, in that the scenes are concerned with reactions to interaction, not the mechanism to enact interaction. The vehicle concept in DIVE is similar to the similarly named concept in VRML, but interactions are generated by devices, not by sensors in the world – this fits better with the publish-subscribe mechanism.

3.2.2 User Abstraction
Vehicles move users about the world, but the second key abstraction is that the user is a specific entity type, which will usually be associated with a visual representation as an avatar, but may not. The user abstraction is useful because the entity is visible in the scene inventory, is known to be uniquely associated with a physical user and a specific system-bound process. This means that it can be queried, and sent event, and these events can be handed to process that created the user for any processing. An identifier for the user is also supplied with any event that is interaction-based; this means that multi-user interactions are quite easy to implement, because the origin of an incoming interaction event is well-known, and we can easily distinguish multiple grabs or selects of the same or associated entities.

3.2.3 Body Abstraction
A related abstraction is concerned with a representation of the body of the user. All processes can create one or users in the scene, but each gains a default body if one is not provided. This default body contains, amongst other things the position of a head, hands, foot, eyes and ears. Rendering is controlled by the position of the eyes and ears, the foot is used by the vehicle if surface following is enabled. Each body part can have a visual representation, and most processes would create a full avatar, including the default objects as part of a full graph. The important part of the body abstraction is that all events generated by the input devices associated to the process that created a user are sent from the objects of body parts. That is, a selection event might be sent from the “right hand” (for a CAVE user) or the “eyes” (for a desktop user, where the vehicle implements selection by mouse clicks through the camera view). This association of event with a body part allows some simple generalization of interaction techniques (e.g. an object must be selected by the eyes and a hand). Scripts and plugins, thus can ask for “any selection”, or a specific selection without needing to know the particular source.

This is similar to spirit to the technique in many device APIs of abstract naming of devices. The subtle difference is that you don’t need to know the source of an event, and the names and positions of particular body parts are guaranteed to exist, even if there are no trackers. The body parts will be in a known relationship, and are moved ensemble by the vehicle.

3.3 Scene Descriptions, Graphs and Meta-Data
Although most VR systems, including scene-graphs, provide external formats for describing the scene, they often rely on a format that is directly exportable from a modeling package. Obviously support for export from modelers is necessary, but we would advocate having a scene description language that is easily human readable and which can be easily produced from scripts or web services. The key reason for this is that meta-data can easily be added in to this scene file. DIVE had a “.vr” file format, which could store complete geometry descriptions, but this facility was rarely used once 3DS and VRML import was available. The “.vr” did allow for node naming, meta-data additions, through arbitrary named nodes. Several other VR systems have a similar facility explicitly, or it is the programmers common practice to have one file that inlines several others.

An important observation is that the scene-graph is rarely a unidirectional graph. Whilst it is common in scene-graphs to allow nodes to be shared, because they represent the same state or geometry, we would often find ourselves wanting to re-arrange the scene-graph to represent relationship, such as ownership, locations such as “within” or “joining”. Again in DIVE, there was a facility for creating named associations between objects, which could be looked up, and altered up in the scripts and code, but which also could be described in the external file format.

Finally, we have hinted that meta-data is an important part of any system. Meta-data is the hard to preserve through the production process, but a good facility to name and share key/value pairs within the scene representation not only makes coding easier, but makes network programming straight-forward. Too many times, we have discovered that code that works on a single machine fails in multi-mode because of the side-effect of a global variable only existing inside a script interpreter, not being part of the scene itself. One level of meta-data that we have found extremely useful are associations of hierarchical relationships: “isA” and “hasA”, without the resort to creation of “classes” of object. VRML for example allows you to define your own classes, with interfaces, but actually the simplest thing we wanted to do, for which this mechanism is way over the top, is to label all objects “isA DOOR”, and then, in our plugins or scripts, create an iterator over these types. Finally, the very most common function we ever used in DIVE was a function that would find the closest object to the user by a given type. This code construct is so common in use, it is one of the first things we re-implement in other systems’ scripting language.

4 ABSTRACTIONS FOR THE NETWORKED SYSTEM
The previous section focused on programming constructs that assist with the creation and maintenance of scenes. At the time it was written DIVE was one of very few systems that supported networking out of the box, and now many other systems have been built. However, some aspects of DIVE are still fairly novel.

4.1 Event Scoping
The system was based around the partial sharing of the scenes database. Events describing scene changes were propagated between hosts. This was done as an extension of the internal event mechanism. The simple extension to the main event system was that certain events would be distributed to the network, whilst others would remain on the machine. Thus key presses and render events were local, changes to geometry or meta-data were shared. Events were ordered, so that even cascades would propagate in the same order as generated, but there were notable issues with this due to the way network events were re-transmitted on network failure [5]. One distinct advantage of event scoping was that making a record and replay mechanism for networked events was quite straightforward and flexible, in that a networked, or local view of proceedings could be regenerated [13].

4.2 High-Level Events
To make the networked event system scale to moderate numbers, a refinement to the scoping was introduced called lightweight groups [2][3]. The problem with event cascades was that the person who triggered a behavior would be responsible for generating and then distributing all the state change events. This does work, but means that simple, deterministic behaviors must be propagated across the network. A lightweight group “hides” an event cascade, to only those processes that have expressed an interest. Interest would be controlled through the scripting
language or plugin code. This was designed originally as a partitioning mechanism, but its implementation was done by event suppression. Shortly after implementation, it was realized that we could have the “suppressed” parts of the graph communicate high-level events, by relaying them via scene node outside the lightweight group. Thus with suitably described events, we could easily control large scene changes which was deterministic on only a few variables. This mechanism might sound complicated to achieve the effect of having an abstract interface to a behavior, but this important concept is that a new client joining the session doesn’t need to know about any new classes, and can get the state by copying the contents of a lightweight group, and then continuing to update based on the high-level events. Thus the programmer doesn’t need to deal with boot-strapping the scene in all possible configurations.

4.3 Subjective Views

The final general abstraction in the networking system was that of subjective views[9][3]. A subjective view was a part of the scene state whose values would depend on the user’s name and role. Originally it was intended for the description of visual differences, thus different people would see different geometry or surface models. This was done simply by putting all the potential representations would be in the scene, and these would be selected between at run-time. However, to implement this, within the scene, there were a series of associations which could be matched. Later this came to be used to load different functionality to different users, thus combining the subjective view concept, with the lightweight group mechanism. This could allow, for example, two users to receive different bits of user interface when interacting with an object: in extreme a desktop user could receive switch-like structure where the condition is made on some known other processes are using specific views. Thus in implementation create all the views of the object, and doesn’t need to know which widget that would manipulate with a tracked device.

We have found in other systems that having something similar to subjective views is very useful. This is because one process can create all the views of the object, and doesn’t need to know which other processes are using specific views. Thus in implementation it can actually be very straightforward: it can boil down to a switch-like structure where the condition is made on some known property of the user or system. If there is event routing occurring on the different branches, some care needs taking: within DIVE, as mentioned, this can be done with light-weight groups.

5 OTHER COMMENTS AND REQUIREMENTS

All of the abstractions we have described were implemented at some stage of the life-time of the DIVE software. A few of these were abstractions that were implemented or designed by the author and colleagues, integrating techniques from other software that we had found useful. It is worth noting however, that the DIVE platform is now no longer being actively maintained, and although the applications continue to run because of the various abstractions away from scene-graph and interaction devices, no new applications are being built. This is for a variety of reasons: the visual appearance is now dated, the scripting language is out of fashion and there are some parts of the run-time that have non-obvious behavior. One of the critical things mistakes was not ensuring that the order of specific events was deterministic. This was particularly a problem when incrementally loading files over a network: some code was written with arbitrary delays to “wait” until it was likely that all assets were loaded before searching for objects to perform animations. Also the code is primarily C, where there is an obvious role for an object-oriented representation in some areas.

Porting existing demos to alternate systems has been done piece-meal, usually taking the opportunity to improve the demonstration, but integrating the original assets. However, at the moment this is done to a variety of different systems, mostly PC-based, whilst we search for a platform to commit resources to. As yet, we have not found a system that is easy to use, supports all of our hardware, has networking built and supports a “platform” style of programming with world description kept separate from interface. We believe getting commonly-used facilities exposed to scripting and plugins in a simple to use way, should motivate the design of future systems, over the desire to make elegant, extensible software frameworks.

REFERENCES

Note that to save space, links to all the associated web pages for the VE software packages can be found at:
http://www.cs.ucl.ac.uk/staff/A.Steed/ve-list/