Experiences from UniView

– A Discussion on Real Time Standards –

Staffan Klashed ’ Carter Emmart ’ ’ Anders Ynnerman ’ ’ ’

’ Sciss AB
Bredgatan 12
602 21 Norrköping, Sweden

’ ’ Rose Center for Earth and Space
American Museum of Natural History
79th St. at Central Park West
New York, NY 10024

’ ’ ’ NVIS, Department of Science and Technology
Linköping University
602 74 Norrköping, Sweden

Abstract
UniView is a software framework for real time astronomical visualization. The development of UniView has been made in close collaboration with software developers, content creators and astronomers, and we have learned several valuable lessons regarding data formats and distribution of real time shows. This paper will discuss some of those lessons.

This paper will also discuss some other problems that we have encountered during the development of UniView. Among these problems are issues with numerical precision, and remote application control.

Presented at the IPS Full dome Standards Summit, this paper attempts to provide a generic perspective on the issues afoot. UniView is a joint effort between Sciss AB, Linköping University and the American Museum of Natural History.

Background and introduction
In the summer of 2002, a collaborative relation was initiated between the American Museum of Natural History (AMNH) and Linköpings University (LIU), Sweden. It initially had the shape of an internship where thesis students from LIU worked alongside with the Rose Centre production staff at AMNH. The initial idea of the internships was to create an interactive application that could bring functionalities of a solar system simulator, together with the functionalities of the Partiview [1] software, into a digital planetarium dome. After initial experiments on merging existing software, the problem with numerical precision became apparent and we decided to create a completely new platform for visualization of scale-independent objects. This was the embryo of what was to become UniView, and a first version was presented at the SGI VizSummit 03 in Paris.

During 2003 the effort was intensified. Emphasis was on creating solid structures, both for internal and external representation of data. At the same time, the embryo for the company Sciss AB was created, to ensure further development of the UniView software. A second and much improved version of the software was presented at the International Planetarium Society conference 2004 in Valencia, in cooperation with vendors SEOS and SGI.

Related work
The notion of flying freely in a virtual universe has been examined by several other projects. One significant effort was made with the NCSA Virtual Director and the desktop PC version Partiview [1]. Another was made by the Solar System Simulator Project [2]. To current date, we know of several more projects examining free flight in a virtual universe. Among them are the SGI Digital Universe, the RSA Cosmos “In Space” System [3], the Starry Night software [4], and the freely available Celestia [5].

During examination of different solutions, and our own work with parsing AMNH’s Digital Universe, we started to understand the depth of complexity of the standardization problem. There is a great need for standardized data formats. Realistically, we need to achieve such standards while not limiting the possibility for different solutions to compete.

The UniView project
The overall aim for the UniView project has been to create an interactive solution that can, in time, be a full planetarium solution in itself. We have from the very beginning of this project been influenced by the notion that real time software will sooner or later have the capability to replace traditional video material. Current development trends point in that direction and the quality of the recent real time benchmarks [6] compares well, in some aspects, to the quality of off-line rendered material. UniView has also been

influenced by the notion of the wider usability of digital planetariums than "just" astronomy. Therefore, UniView has been built as a generic visualization platform, with current focus on astronomy, rather than an astronomy-only visualization platform.

UniView is a pure C++ application, with few external dependencies. We rely heavily on scene graphs for rendering, as the hierarchical structure of such graphs is also the fundament of our solution to the numerical precision issue, the ScaleGraph. Originally, we used OpenGL Performer as the sole scene graph, but over time we have merged to a more abstracted use of scene graphs, which can be any of the well-known Performer or OpenSceneGraph, or some DirectX-based graph. This is because different planetariums use different solutions for image slicing, edge blending and geometry correction, and it will prove useful to have the ability to render to both OpenGL and DirectX contexts. Also, it may prove useful to maintain the capability to benefit from future improvements to any of the mentioned scene graphs.

We have invested a considerable amount of time into maintaining cross-platform compatibility of UniView. In retrospect, this was not necessarily a wise choice as development has been slowed down by many debugging and test sessions between Windows/Linux/IRIX. However, it gives UniView an advantage to other software as a user will run the same application, but different compiles, whether on a desktop in the office or a super computer in the planetarium dome. Over the last few months, we have gradually come to abandon the cross-platform compatibility and instead work with different versions for different OS. The core pieces of UniView (i.e. model and controller, see below) are relatively stable and have maintained cross-platform compatibility, but rendering and GUI are growing more and more specific for individual platforms. This has become an important advantage, as off-the-shelf rendering hardware, such as Sony Playstation [7], NViDIA [8] or ATI [9] can now deliver relatively high performance to a low price, and platform-specific libraries can provide standardized and efficient GUI components such as MFC [10].

A key component of UniView is remote control. UniView is equipped with entry points for remotely connecting and taking control over a UniView session. We have done this by opening our event kernel to events sent over TCP connections, and a remote guide can control UniView through such events. We have done only limited tests of this functionality, and have confirmed that our approach is good for "impulse" events such as starting pre-defined camera flights, enabling or disabling data groups. We have yet to examine in detail the use of continuous event streams, such as free flight camera manipulation, and it may prove that our event language needs to be optimized further to allow such functionality. It may also be that state synchronization between nodes provides a bottleneck in the communication speed, but we believe that to be less troublesome. By state synchronization we mean the functionality that ensures that attributes of data are set to the same values on all clients, and in the worst case, state synchronization can be done over a period of time instead of on each new frame.

In the effort of trying to combine small-scale objects, such as the International Space Station, with large-scale objects, such as galaxies, in the same visual context, we ran into the problem with numerical precision. The problem is not so much in the processor of the computer as it is in the rendering hardware. Here is a small example in C.

Example 1. The precision issue.

```
float a = 0.0;
float b = 1.0;

while (a != b)
{
        a += 1.0;
        b += 1.0;
}
```

This loop will not continue forever. After a while, the floating point variables contain high enough values that the addition of 1.0 will no longer make a difference. Actually, you can discover the relatively low precision of floating points just by trying to initialize a variable to for example the value 0.3. This is what makes it virtually impossible to represent ISS and the edge of the observable universe in the same visual context. In the CPU of a computer it is not such a big issue, just replacing the floats with doubles will significantly improve precision. However, graphics hardware commonly uses floating point precision which will lead to jittering pixels on the screen.

We solved this problem using an approach we call the ScaleGraph. The details of the method are complex and will not be covered here, however the general idea is to use dynamic coordinate systems and multiple passes of rendering. A first pass renders large-scale objects with a dynamic coordinate system in the unit of light years or similar, and another pass renders small-scale objects with a coordinate system in the unit of meters or similar.

Data representation
UniView uses the traditional model-view-controller, or "mvc", paradigm for data representation. In short, the mvc paradigm means that data is represented through a model, which is an abstraction of the data, a view that controls rendering of the model, and a controller that manages communication to and between the model and view. Typically, physical attributes, motion and shading attributes are parts of the model, whereas the view is the actual rendering (i.e. OpenGL/Direct3D).

Static and dynamic attributes
We wanted to equip UniView with the possibility to affect attributes of datasets in run-time. For example, it is often desirable to be able to change opacities, to fade in and out datasets. There are numerous such attributes that a user may want to affect, but there are also more static attributes where it is less likely that a user wants to make a change. For example, it is not likely that a user wants to change the properties of sun reflection in the oceans of Earth. This calls for a division of attributes into static and dynamic attributes.

Currently, we know of no datasets that have a clear separation of what are static and dynamic attributes. We believe the reason for this is because it is rare that a dataset is used with more than one software platform. To enable the use of various datasets on different software platforms, this issue should be investigated in further detail. Static and dynamic attributes will also help in a future effort to enable remote collaboration between different software platforms. If two software platforms cannot share control events with each other, a most likely scenario, synchronization between remote sites needs to be performed through comparing the dynamic attributes of the respective datasets.

Cross platform issues
When discussing data, there are no obvious issues with maintaining cross-platform compatibility, if we by platform mean operating systems and/or graphics drivers. Obviously, we all know of the always-appreciated issues with line endings. We also know about potential problems with little- and big-endian systems. From our experiences with UniView, these are well-defined issues and not really a threat to cross platform capabilities of datasets. However, if we instead refer to the software platform, or in effect the application, when discussing cross-platform compatibility, there are numerous issues with current datasets.

We have identified three main issues with datasets we have examined within the scope of the UniView project;

- Lack of scopes
- Lack of hierarchies
- Implicit relations

Lack of Scopes
Many datasets today are not using well-defined scopes. UniView makes use of the Digital Universe datasets developed by AMNH. These datasets have an implicit way of defining scopes, making use of an "active" group of data at the time.

Example 2. Implicit scopes

```
object g1 = MilkyWay
eval attribA true
eval attribB false

object g2 = HippoStars
eval attribA false
eval attribC 1.0

object g3 = Sun
eval attribB false
eval attribC 1.0 0.5 0.7 1.0
```

This may seem like a straight-forward definition. However, the objects are not arranged in well defined scopes. What would happen if we for example added another instruction to the dataset;

```
eval attribB true
```

Since g3 = Hipparcos was the last object we implicitly selected, the attribute modification will occur on that object. We suggest that content creators and dataset managers start making use of explicit scopes, which would lead to a structure as in Example 3.

Example 3. Explicit scopes

```
object g1 = MilkyWay
{
eval attribA true
eval attribB false
}

object g2 = HippoStars
{
eval attribA false
eval attribC 1.0
}

object g3 = Sun
{
eval attribB false
eval attribC 1.0 0.5 0.7 1.0
}
```

Now, if we again added that last instruction to the dataset;

```
eval attribB true
```

it would not be valid. So in order to use explicit scopes, we also need a way to address existing scopes. This should be done using scope names, or in effect the names of the objects.

```
eval HippoStars.attribB          true
```

We will later expand on this topic and suggest a structure of data that is suitable for scopes, and also hierarchies.

Lack of hierarchies

Astronomical datasets today often lack a built-in hierarchy. Most of the modern software platforms use some sort of scene graph, or built-in hierarchy, to improve rendering efficiency. A standardized specification for defining this hierarchy would greatly improve the usability of datasets and help maintaining good rendering performance.

Rearranging example 3 into a hierarchy would yield a data structure similar to example 4.

Example 4. Explicit scopes and hierarchy

```
object g1 = Milky Way {
    attribA     true
    attribB     false

    object g2 = Hippo Stars {
        attribA     false
        attribC     1.0

        object g3 = Sun {
            attribB true
            attribC 1.0 0.5 0.7 1.0
        }
    }
    attribB     true
}
```

Since the sun is one of the stars in the above data group, a subset of the Hipparcos catalogue, the Sun object should be defined as a child of the Hippo Stars. The brackets again define start of a scope and end of a scope, so the last attribute modification will, in this example, occur on the Milky Way object.

Implicit relations

Often, relations between an attribute and parameters of the attribute are implicit. For example, imagine the attribute colour. It could have three floating point parameters, four floating point parameters or a string parameter. It would be desirable to state what parameters are valid for different attributes of a data group, and also what parameters are static and dynamic, respectively. By doing so, any generic parser would automatically know how to interpret the data.

It would also be desirable to explicitly define what attributes are static and dynamic, respectively. Again, this would allow generic software to interpret both the static data and dynamic input, and the software would know what attributes need to be kept in updateable structures, with all the implications that may bring (double buffering, shared memory and similar).

Suggested structure

Defining data as in example 4 is good, since it provides both hierarchy and scope. However, it is far from perfect. Using data formatted like the above calls for software parsing to be customized. The format is also not standardized, or at least not used broadly enough to be considered a standard, and chances are that different software platforms will parse data differently and treat errors differently.

A commonly used approach in other software is to use an XML-based [11] structure of data, providing scope, hierarchy and easy means to explicitly identify relations between attributes, parameters of attributes and static/dynamic parameters. Example 5 is a suggested structure, with an XML approach, of how datasets should be designed.

Example 5. An XML-based structure

```
<header>
   <attributes>
      <dynamic> attribA bool </dynamic>
      <dynamic> attribA float </dynamic>
      <dynamic> attribB bool </dynamic>
      <dynamic> attribB bool </dynamic>
      <static> attribC float </static>
      <static> attribC float float float </static>
      <static> attribC float float float float </static>
   </attributes>
</header>

<body>
   <object>

      <name> Milky Way </name>
      <group> g1 </group>
      <attribA>true</attribA>
      <attribB>false</attribB>

      <object>

         <name> Hippo Stars </name>
         <group> g2 </group>
         <attribA>false</attribA>
         <attribC>1.0</attribC>

         <object>

            <name> Sun </name>
            <group> g3 </group>
            <attribB>true</attribB>
            <attribC>1.0 0.5 0.7 1.0</attribC>

         </object>
      </object>
   </object>
</body>
```

The use of XML-based data structure would simplify parsing. Not only do open source XML parsers exist in the public domain, but the standard is also well-defined when it comes to error handling and data validation.

Show distribution
An interactive planetarium experience is defined not only by its digital audiovisual content, but also by storytelling, audience participation and the possibilities to manipulate datasets. Thus for distribution of real time shows we must consider what are the items to distribute, since the quality of an interactive experience rests more with the skills of the guide than with the database or the software application. Through our experiences with UniView, and in considering these issues, we have identified a discrepancy between two different show formats;

- Remote lectures / collaborations
- Pre-scripted shows

Remote lectures
Real time remote lectures
The application areas of remote lectures would primarily be scientific. Through one single lecture, astronomers could reach out to hundreds or thousands of colleagues and/or pupils worldwide. There are several technical challenges involved when considering this feature.

First, the databases at the listening clients would need to be synchronized before a lecture starts. This could be done automatically, possibly by brute-force methods such as redistributing entire datasets from the host machine to all clients. The other, and more appealing, alternative would be to use the sophisticated version control techniques that already exist. Systems such as CVS and Subversion provide nice interfaces to version control, where data updates made by the host are automatically merged into the databases of the clients.

Second, the communication during run time needs to be optimized to minimize latency. For this, we suggest that clients would connect to the host and that no intercommunication between clients is accepted. We also suggest that state confirmation and attribute updates are made only on dynamic attributes, as previously discussed. It is crucial that attribute changes are in synchronization between the host and the clients, as these attribute changes will most probably constitute a major part of the content of the lecture. For example if a lecturer changes increases the scale of one of Saturn's moons, to give a detailed speech about imagery provided from Cassini, it is necessary that the scale of the moon is increased on all the clients as well as on the host. To optimize network communication, it is important to discuss in further detail what are the essential dynamic attributes of various datasets, and what attributes can be considered static.

Regarding camera motion, events need to be distributed every time the host lecturer interacts with the camera. However, when trying camera event distribution within UniView we have discovered that the accumulated error from round-offs and potentially lost network packages can be significant. Since only the events (i.e. a message saying "move the camera forward two units") are distributed, we miss the confirmation of actual camera position. To avoid this, UniView is currently distributing absolute camera positions. This approach is straight-forward. However, it introduces a problem when communicating between two different software platforms. Absolute camera positions are always relative to the coordinate system of the application. Different applications may use different coordinate systems, and in order to allow remote lecturing to work between different applications a standard needs to be set on how to define coordinate systems.

Pre-scripted shows
Using pre-scripted shows generally means that we take away the interactive parts of a real time show, to benefit from the simplicity of distribution. A pre-scripted show distribution would need to contain only a camera flight path, specifications on what events should occur on different positions along the flight path (i.e. fade in the radio sphere when the camera passes a certain key point), and narration.

We have also discussed a system for scripted interactivity, meaning that a show is scripted to the extent of guiding the audience to consecutive key points along a path of storytelling, allowing for interaction and hands-on experiences at these key points. In this way, we can script outlines of a story and distribute the scripts, yet maintain the navigation and manipulation freedom needed to create the presence of an interactive experience.

Conclusions

Through the efforts with UniView, we have learned several valuable lessons. Regarding standardization of data, this is a field where the planetarium community could grow more mature, and work towards data formats used in other disciplines such as gaming or medical visualization. The implementation of XML-based structures will be a good step towards that goal. Once we have a good standard for data, other issues will raise when we start to distribute and share data between different creators, users and software platforms. Validation will be a key issue, to make sure that data designed on system A looks equally good, or at least as intended, as on system B.

Distribution of real time shows is another challenge ahead. A first step is taken towards categorizing real time shows in different show formats, but many more definitions will need to be set before a standardized way of distributing real time shows between different platforms can be achieved. In remote lectures, we face only the problems of state synchronization and camera positioning. In pre-scripted shows, we also face the challenge of how to make an audience experience the true value of real time, interactivity.

We realize that the road ahead is long and possibly bumpy. Maybe the market powers will lead the way without any academic interference being necessary. Until we know for sure that is the case however, we welcome a dialogue on topics similar to those brought up in this paper.

References

[1] PartiView
http://www.haydenplanetarium.org/hp/vo/du/partiview.html

[2] Solar System Simulator Project
http://www.sssim.com/

[3] Rsa Cosmos - In space system
http://www.rsacosmos.com/anglais/produits/space_system.htm

[4] Starry Night
http://www.starrynight.com/promo/astronomypack.html

[5] Celestia
http://www.shatters.net/celestia/

[6] 3d Benchmark
http://www.futuremark.com/download/?3dmark05.shtml

[7] Sony Playstation
http://www.us.playstation.com/consoles.aspx?id=2

[8] NVIDIA
http://developer.nvidia.com/page/home

[9] ATI
http://www.ati.com/

[10] MFC
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vcmfc98/html/mfchm.asp

[11] XML Graphics
http://www.xml.com/graphics/