Visual Value 3D

Efficient event propagation in scene graphs.

Anders Nilsson

Master Thesis in computer science, 20 credit points,
Department of Computer Science,
Faculty of Engineering, LTH
December, 2006

Abstract

Scene graph based applications for 3D graphics normally supports event listeners in order to interact with displayed data. Events that eventually change data held in the scene graph need to be processed and handled in a robust manner. If a scene graph contains a complex structure with multiple dependencies between attributes, the event processing mechanism through the scene graph becomes a problem difficult to solve in an optimal way. Problems that seem to occur are multiple and partly inconsistent updates of scene graph data which in the end will affect the rendering performance.

In this paper I will argue that a technique called Visual Value 3D solves the event propagation problem in a robust and optimal way. Visual Value 3D should mainly be used as a complement framework for scene graphs with demands on complex dependencies. Designing dependencies between scene graph objects and attributes is easily made with Visual Value 3D since the entire scene graph with including dependencies is expressed in xml.

Preface

This master thesis project was initiated in January 2005 by Winsider AB and the Department of Computer Science at Lund University in Sweden.

I would like to thank my both my supervisors, Lars Andersson at Winsider and Lennart Ohlsson at Lund Unviersity for their commitment and contributions to this project.

Table of contents

1 - Introduction

In most 3D graphics systems the entire 3D scene is described in a scene graph. A scene graph is a reusable set of data structured in nodes that contains necessary information for the rendering software when a scene is rendered. Typically a scene graph contains object's transforms, materials and geometrical data as well as information about lights and viewer. Structural and spatial relationships between objects are normally also included in the scene graph.

Scene graphs are in most cases a variety of a directed acyclic graph. They exist in game engines, computer-aided design (CAD) programs, scientific and commercial visualization applications, simulators, and modeling programs. To render a 3D scene is the process of converting the data held in the scene graph into colored pixels on screen. Constructing and maintaining the 3D data is handled with software whereas the rendering is normally done in the graphics hardware.

1.1 - Event handling

To process events in a desirable way has always been hard. Events are hot, meaning that there is no way of knowing when they occur, but when they do it means that something has to be done and depending on their kind and how well the system is designed, they can cause frustrating and sometimes disastrous effects.

When a scene graph has dependencies between nodes, their attributes, such as world matrices and colors, are in need to be calculated somehow for each object in the scene. Normally, 3D rendering software exploits the scene graph structure in order to calculate the correct attributes. But as a consequence complex scenes become sensitive for events that cause changes in the scene graph data. Extensive recalculations may take place and attributes might be calculated multiple times. In some situations attributes may even hold incorrect data while the scene graph isn't fully updated.

1.2 - Why was this thesis written?

Rendering a scene with a 3D rendering software is a well known subject and has been developed a lot during recent years. Especially graphics card has developed tremendously and many 3D graphics related areas have drawn benefits of this. Because of the rapid development on hardware the demands on software development have not been high enough to push development forward at equal speed.

Having a complex chain of dependencies in a 3D scene is not an uncommon situation, especially not when rendering a component composed by several subcomponents that is controlled by for example a user. Today's existing methods to express a scene graph filled with complex dependencies are not satisfying since the playground of defining dependencies between

nodes are often limited to the scene graph structure or too clumsy to define complicated dependencies. Moreover the updating of scene graph data when complex dependencies are present is not optimal since the data sometimes is updated multiple times and always updated even though it doesn't need to. Thus, finding a general way of handling events with optimal performance and making it certain that scene graphs hold correct data is the reason for why this paper was written.

1.3 - Visual Value 3D

Visual Value 3D is a framework used for holding scene graph data and protect it when events occur. Complex dependencies between nodes in the scene graph are expressed within Visual Value 3D and event propagation through dependencies are handled in a robust and optimal way. The technique uses Forward Propagation[1] to keep consistency in the scene graph when changes occur and scene graph data is calculated dynamically when access is made in order to optimize the amount of recalculations.

Visual Value 3D acts as a complementary layer to a scene graph, that exists between the scene graph and event sources. It receives and process events in order to insure that all scene graph nodes are correctly updated the next time rendering of the scene is performed.

The technique is developed inspired by an already existing technique called Visual Value. It is today used for configuration of composite industrial products. But more than that, Visual Value is an "under-the-hood" concept which encourages developers to keep a system's event handling clean, maintainable, easily exchangeable and generally dynamic. A concept like that is not just needed in configuration products or in scene graphs but in any system in need for larger scaled event handling.

Chapter 2 of this paper presents the main problem that was the reason for making this project. There is an overview of how the problem can be solved and how today's existing products solves it and following that; a short presentation of Visual Value 3D is made together with comparisons to today's existing products. The rest of the thesis covers the architecture and implementation of Visual Value 3D together with results and conclusions

[1] Explained in chapter 2.1.2

2 – Event Propagation in Scene Graphs

This paper is aimed towards the problem of maintaining a scene graph's structure when events occur that has either direct or secondary effects on the scene graph. Events should inflict as little impact as possible on the scene graph and when rendering it should be certain that values contained in the scene graph are consistent when they are accessed. In order to argue for why Visual Value 3D is needed, a presentation of the problem it solves is needed together with an explanation of why today's existing solutions not are good enough.

2.1 – Overview of the event propagation problem

A dynamic 3D scene with animations and/or user interaction is a normal seen feature today and excogitated methods for handling such scenes are needed. The reason for why a dynamic scene's data is structured into a graph is that logical and spatial relationships between objects can be expressed in the graph.

When designing a scene graph, an important issue is to decide how updating and calculating the data held in the scene graph should be made. There is no standard answer to this issue. It is a fact that programmers who implement scene graphs in their applications take the basic principles and adapt them to suit the needs of that particular application. This means that there is no hard and fast rule as to what a scene graph should or should not be. But from looking at the "common case", it can be seen that in general there are two ways of describing how calculating data held in the scene graph should be done and as they are presented below, the event propagation problem takes shape.

2.1.1 - Static scene graph nodes

The logic of calculating scene graph data is with static scene graph nodes placed in the rendering software traversing the scene graph. Starting out at the root of the graph, the context of nodes is after calculation passed down to their children by using a stack where node attributes are pushed and popped while traversing. This method requires that the scene graph is a tree since a node's attributes are updated relative its parent and with multiple parents there is no way of knowing which one of the parents the rendering software should choose. This method also requires that the entire scene graph is traversed each time a frame is rendered; otherwise changes on scene graph nodes will not be detected.

Updating the entire scene graph each frame is a relatively naive method for rendering. What we want is that after the initialization of the scene graph, nodes are only updated when changes occur and that the change is propagated to all dependant nodes. In such scene graph the rendering software only needs to traverse the scene graph's leaves and render the objects placed there. Changes will automatically eventually reach the leaves and update them. By presenting how that is solved leads us into the next technique of updating scene graphs.

2.1.2 - Dynamic scene graph nodes

A popular way of designing a scene graph is to make nodes “dynamic” and “lazy”, which means that nodes know how to calculate themselves (dynamic) and they do that only when they are invoked (lazy). A node’s parents are those that are accessed when the node’s attributes is calculated. Updating a scene graph with dynamic nodes can generally be made in two ways. As they are presented below, two important concepts are introduced.

Forward Propagation – pushing

Dynamic nodes need their parents to be updated before they are updated themselves, and the forward propagation algorithm solves that like this:

```
void Update()
{
   CalculateMe();
   foreach (Node node in childNodes)
  {
     node.Update();
   }
}
```

(“childNodes” is here an iterable structure over a node’s child nodes)

The initialization of the scene graph should be done by making an update call on the root of the scene graph which then first updates the root itself and then recursively its child nodes. Once that is done, an update call on a single node will propagate forward to all dependant child nodes until the leaves are reached. Update calls performed after the initialization, are typically triggered by events.

This is a popular technique in many scene graph based applications. By using this propagation model for events we find both advantages and disadvantages. The advantage is that a change is always detected in all affected nodes immediately when the event occurs. However there is one big disadvantage with this method. A scene graph containing complex dependencies where node paths are merged together will perform multiple updates on nodes with multiple parents when the event change propagates forward. This is shown in fig 2.1. The only way to avoid this problem is to use a tree structure as scene graph. It is then ensured that scene graph paths never merge together but on the other hand it hinders developers to use complex dependencies such as constraints on structural relationships. Multiple updates on nodes might not have a big impact on the rendering performance. Depending on the complexity of the scene and how extensive the calculations of nodes are the decrease in performance might not even be noticeable. In other cases, multiple updates could cause a very noticeable effect. In any case, the aim for every developer should be to reach optimal performance and a scene graph with merging node paths cannot have that when solving the event propagation mechanism with forward propagation.

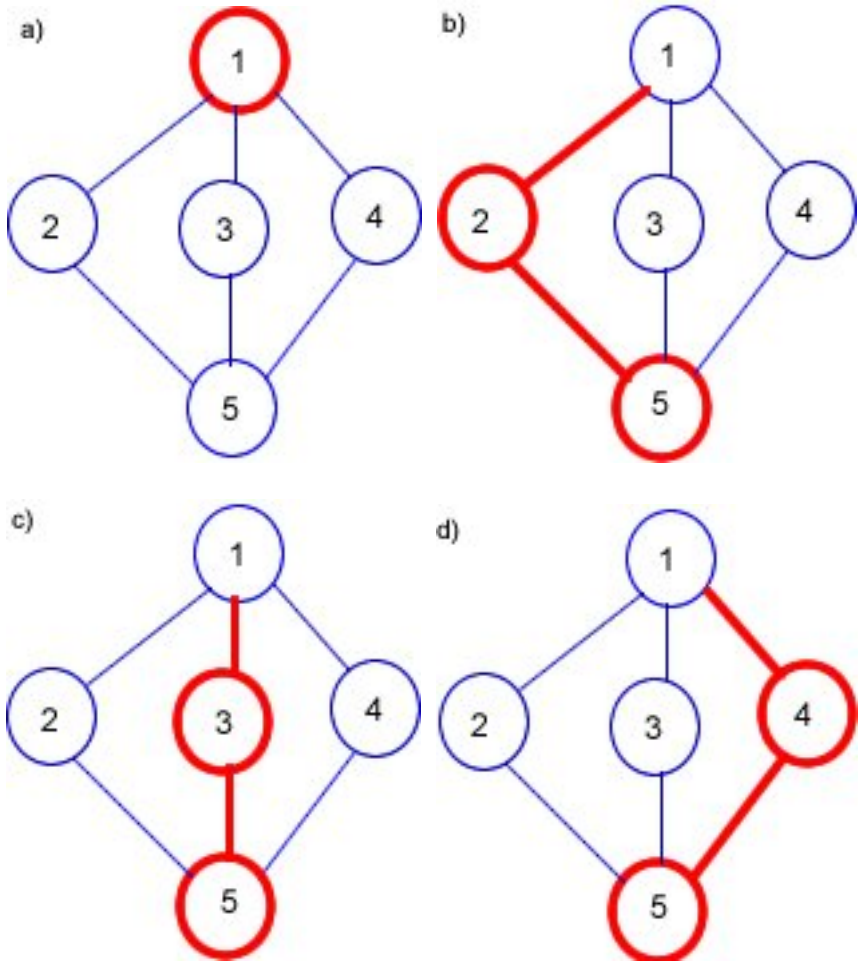

Fig 2.1 – Forward Propagation…
a) Update is first performed on the root (node 1)
b) Node 2 is updated with the recursive call from the root node and the recursion continues to node 5 which also becomes updated.
c) Still being in the recursive update call of the root node, node 3 is then updated which in turn means that node 5 is updated recursively, again.
d) The recursive update call for the root node continues to node 4 and further on to node 5 which becomes updated for the third time.
This example shows how the forward propagation mechanism causes nodes to be updated multiple times when scene graph paths are merged together.

Backward Extraction - Pulling

Another way to handle the updating logic in a scene graph is to pull the information from the nodes instead of pushing. In contrast to the forward propagation technique, the backward extraction technique starts the updating process at the leaves in the scene graph. Before a node is updated it makes sure that all its parents are updated.

```
void Update()
{
   if (alreadyUpdated) return;
   foreach (Node parent in parentNodes)
   {
      node.Update();
   }
   CalculateMe();
}
```

As can be seen, this is a recursive call from leaves to root. Note that this recursive method use a flag which indicates that the node already is updated which in turn will stop the recursion (if the flag is true). The advantage of using this method is that when a node once is updated it is certain that both the node itself and all parents and grandfathers hold a correct value and does not need to be updated again until any of them are invalidated. This solves the multiple updates problem that was introduced with the forward propagation technique. Backward Extraction is shown in fig 2.2.

While this might seem to be a good technique to optimize the calculation of scene graph data with complex dependencies a big disadvantage of this technique is that there is no good way of detecting events. The only way is to, before each frame, first invalidate all nodes and then perform the scene graph update again

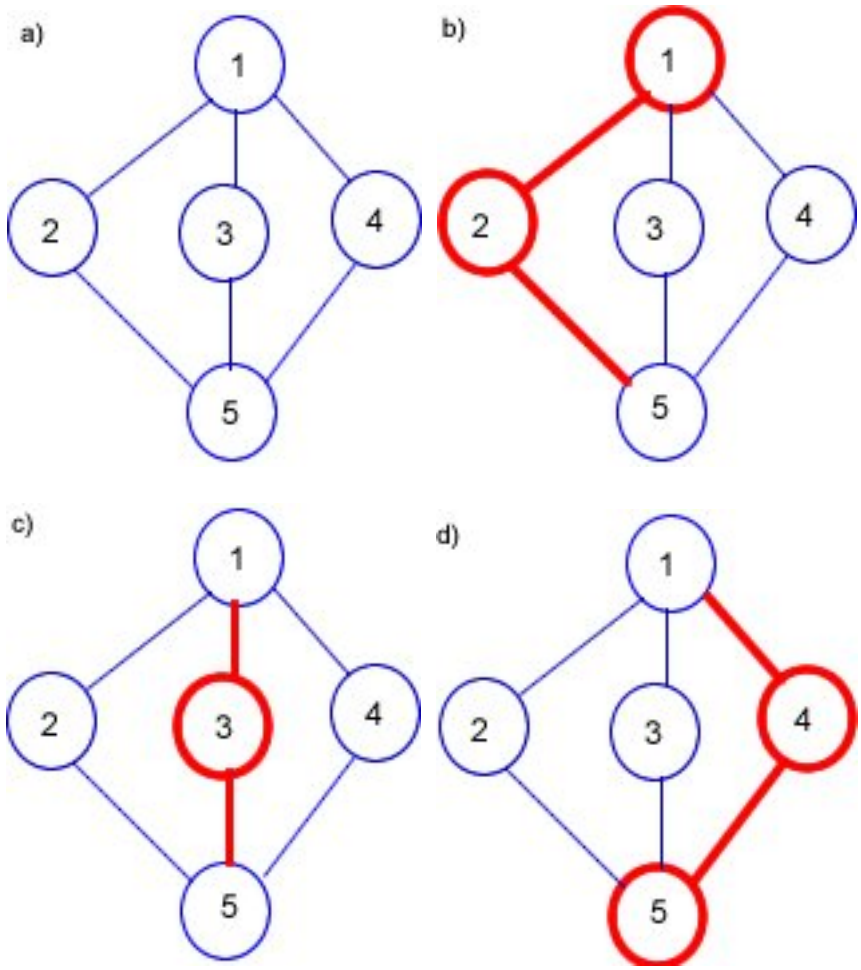

Fig 2.2 – Backward Extraction.
a) Starting at a leaf node (node 5) the update call recursively call parent node's update method.
b) At node 2 the recursive call first updates the root node and then itself.
c) The recursive call at node 5 continues to node 3 and since its parent already is updated it can update itself correctly.
d) Continuing with the recursive call at node 5, node 4 is updated and node 5 can now be updated completing the update call of the scene graph.
This example shows that by using backward extraction it is ensured that nodes only will be updated at once during an update session.

To summarize and come back to the headline of this section it can be stated that the event propagation in scene graphs containing complex dependencies is a problem that needs a solution. The forward propagation technique is good, because changes in the scene graph are immediately detected and pushed forward to all affected nodes, but the technique also causes multiple updates on nodes, unless the scene graph is a tree. On the other hand, if the backward extraction mechanism is used in a scene graph with complex dependencies we can insure that nodes won't be calculated multiple times but the only way of detecting that an event caused a change somewhere in the scene graph is to always invalidate all scene graph nodes and perform an update each time a frame is rendered. Depending on the complexity of the scene this solution will not hold for larger and complex scenes.

2.2 - The cylinder example

To point out and clarify the difficulties, a small and simple example is presented in fig 2.3 which will be a good example to refer to throughout the report. The scene contains two base cylinders independent of each other and one top cylinder that is dependant of both base cylinders. The big issue about this scene is that the top cylinder is constrained to lie on top of the two other cylinders. Events changing any of the two base cylinders should change the top cylinder's translation, rotation and scaling. Updating and maintaining the scene should be performed in an optimal way, meaning that recalculation of node values should only take place when it is actually needed and should only be performed at most once per frame rendered.

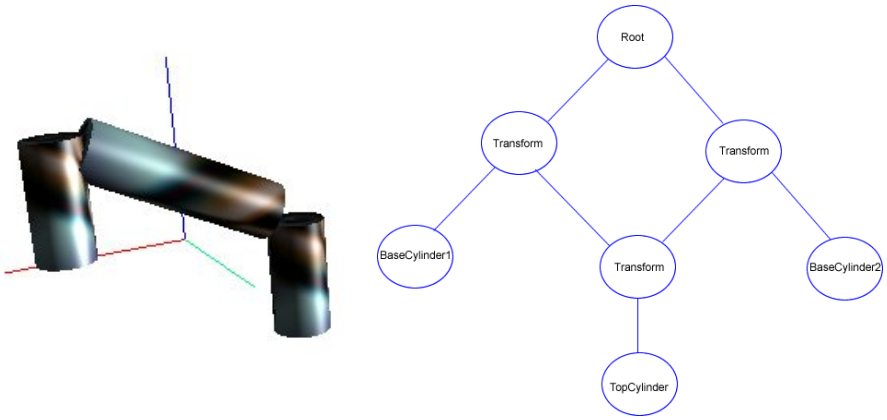

Fig 2.3 – A simple example of a 3D scene with a scene graph containing merging node paths, the top cylinder is dependant on both transform nodes used for the base cylinders. As we have seen this will cause difficulties when events occurs that updates scene graph data.

We will now take a look at how scene graphs are used in the applications out on the market today. A review of how the scene graph part of the application work will be made and the problems will be targeted. We'll also look into how these applications would create and maintain the scene graph in the cylinder example in order to later compare it with how Visual Value 3D would solve it.

2.3 - Existing solutions, related work

Many scene graph based applications supports updating through event driven mechanisms. OpenSceneGraph [4], NVSG [5], Java3D [9], Panda3D [10] are similar libraries used for developing advanced 3D graphics. The existing method that lands closest to Visual Value 3D is X3D [3] which is a successor of VRML [11].

2.3.1 – OpenSceneGraph, NVSG, Java3D and Panda3D

All of these scene graph based applications are cross platform graphics toolkits, used for developing high performance graphics applications such as flight simulators, games, virtual reality and scientific visualization. By using the scene graph as a concept and a base, they provide an object oriented framework on top of a Graphics API freeing the developer from implementing and optimizing low level graphics calls and provides many additional utilities for fast development of graphics applications. Event handlers are used as an interface between user and the scene graph. Nodes are updated dynamically and their update calls are exposed in the interface to the event handlers. So whenever an event occurs that changes a node, its update call is invoked and forward propagation is used to make the event affect all dependant parts of the scene graph. These scene graph based graphics applications all use a tree as scene graph. The following statement is taken from www.openscenegraph.org;

"It's a tree! Quite simply it is one of the best and most reusable data structures invented"

This is most certainly true, and as stated before, forward propagation is good when the scene graph is a tree since events that affect node attributes will affect that node and its children and multiple parents does not exist in a tree, hence multiple updates will not take place. But this hinders developers to create scenes with complex structural dependencies. E.g. the cylinder example presented in 2.2 cannot be created with a tree structure. A scene looking like the cylinder example can of course be created, but only with the restriction that the top cylinder only will be dependant on one of the base cylinders and the constraint that the top cylinder should lay on top of the two base cylinders will not in such case be maintainable. As long as the scene is static this won't be a problem, but whenever an event occur which imposes a change in the scene's structure, control will be lost over its updating process.

So to summarize it can be stated that a tree limits a scene's possibilities to have dependencies. A tree is a sufficient structure in most cases and is a well established and used concept in the game industry where most models are built on hierarchies. But even though OpenSceneGraph, NVSG, Java3D and Panda3D are all robust and fast 3D graphics framework they still won't satisfy developers in certain situations where complex dependencies is needed.

2.3.2 - X3D

X3D is the ISO-standard for real-time 3D computer graphics and the successor of VRML. It was developed to easily create and exchange 3D

scenes between networked computers. An X3D scene graph is a tree with typed nodes expressed in an xml file. The actual rendering is made with an X3D browser. Nodes in X3D are static, meaning that it is the browser that has to keep the logic of calculating node attributes and traverse the scene graph each frame and depending on how the browser is implemented, the performance of a rendered scene in X3D cannot be predicted, but the concept of X3D still remains the same in any case.

To be able to update the static nodes in an X3D scene graph it is possible in X3D to create event nodes which listen to e.g. keyboard strokes or mouse clicks and fire events on these occasions. To make an impact on the scene graph with event nodes so called routes are added between event nodes and the nodes that should be affected by the event. A route does nothing else but passing a specific value from the source node to the target node when the event is triggered. An example of a common way to use routes is to first create a clock node which generates time events each frame. A route is created between the clock node and an interpolator node which calculates a certain value e.g. a translation vector which then, with help from another route is passed further on to a transform node. In this way dependencies between nodes in the scene graph can be designed in a fairly flexible way which means that it is possible to create complex dependencies in a scene. Events are processed with forward propagation technique. Hence, values are updated as they are passed by the forward propagation mechanism. The disadvantages with this are:

- A route can only pass one value at a time between nodes. Hence, a node that needs 5 values to get the right transform matrix needs 5 routes. If this has to be done for each node in the graph the scene graph becomes extremely complex and overly hard to survey.
- Since X3D events are processed with forward propagation, a node's attributes become updated even though they might are unnecessary when rendering the next frame.
- It is completely possible to add routes that are animating the same value (fig 2.4)

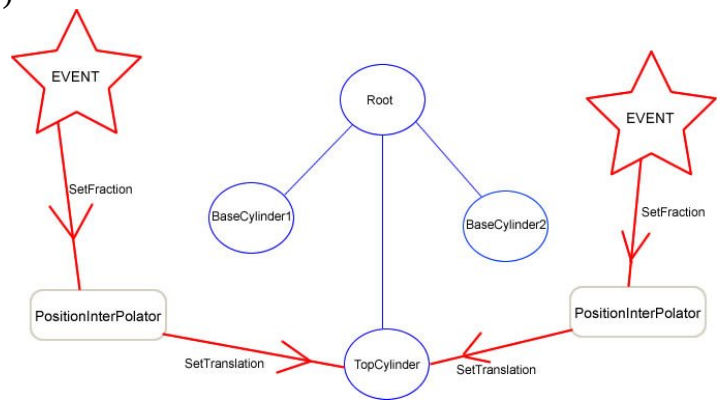

Fig 2.4 – When events reach the same node through different route paths, there is no control of which one of the events that will be used. The route which last performs its update will be the currently held value.

X3D is suitable for a streamed environment where the scene updating process has a clear forward scheme, from event nodes through interpolator nodes and in the end to some attribute at a scene graph node. With this streamed environment filled with routes passing values back and forth to static nodes, it is hard to create complex and dynamic scenes. Also, to design the scene graph by using routes to control every single dependency in the scene graph becomes complicated which in the end limit designers to only apply this method on small and simple scenes, where an overview of the system still is possible for the human brain.

There is however a way to make it easier to create complex scenes in X3D. So called script nodes can be created and with a route connected that sends events to the script node, a script written in JavaScript is executed every time an event is passed to the node. In this way a script node can make dynamic updates on nodes in the scene graph. This means that more complex relationships between nodes can be created. Even if this seems to be a good way to solve dependencies in a scene graph, we still come down to the fact that when an event occurs and starts its travel through the route system it will invoke each script function it finds on its way in the propagation chain. Hence, if more than one route is connected to a script node and events are fired from all routes each frame, the script will be executed multiple times and fire events forward holding incorrect values until the last time the script is executed. By using script nodes, the scene in the cylinder example can be created as shown in fig 2.5. An X3D implementation of the cylinder example is presented in Appendix B. The animation of the scene does not fully handle all dependencies it should, but it handles enough dependencies to show that the evaluation script for the top cylinder will be executed multiple times, and because of this, the scene will never be rendered with optimal performance. Adding more dependencies to the scene will just mean more complex code and worse performance since the script node will be executed more times.

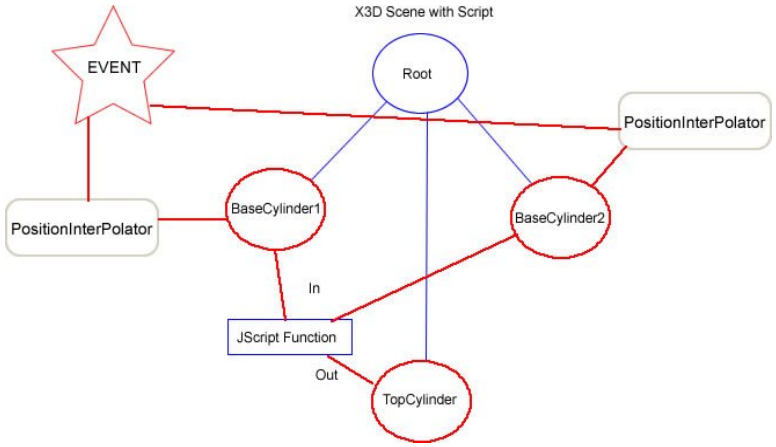

Fig 2.5 – Shows an X3D scene with a script function which calculates the transform for the TopCylinder node. A scene like this cannot be rendered with optimal performance since the script will be executed twice during the event processing and between the updates the top cylinder's transform will be incorrect.

2.4 - Conclusions

Most existing 3D scene graphs libraries are built upon a directed acyclic graph which in most cases has a tree shape and uses the structure logic for calculating attributes for scene graph nodes and forward propagation to process events. Event handling is made through either separate nodes in the scene graph or through event handlers with callback functions to scene graph nodes. When an event occurs they will be plugged in to a specific node in a scene graph and the change propagates forward affecting nodes connected either as children or with routes. If paths in the scene graph merge we have seen that it will cause multiple and inconsistent updates. In this right moment that this paper is being written techniques are being developed targeting the event propagation problem in scene graphs with merging dependencies, but until now there is no standard solution to this problem. So to summarize this section I will, once and for all, state how a solution to the event propagation problem should act.

"Attributes at scene graph node's should be able to be dependant on any number of existing nodes or attributes both in the scene graph itself and outside the scene graph. To render with optimal performance, merging scene graph paths should be handled correctly and updates on scene graph attributes should at maximum be performed once per update session and they should only take place when they really have to. Moreover dependencies should automatically be detected when designing the scene"

2.5 – Visual Value 3D, a solution to the event propagation problem

The concept of Visual Value 3D is that no matter how the scene graph looks like it should act as an abstraction layer used to process events and calculate node attributes making it possible for events to affect any part of the scene graph in any way. The scene graph and events do not know about each other. Events are fired into the Visual Value 3D framework and the scene graph extracts information from the same framework. Visual Value 3D makes sure that the scene graph always is fed with correct information (see fig 2.6).

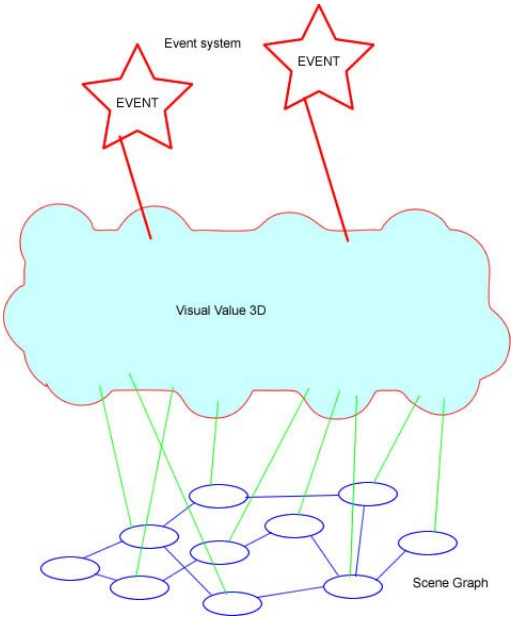

Fig 2.6 – Shows an overview over Visual Value 3D as a concept. It acts as a layer between the scene graph and the event system.

Visual Value 3D is built upon a similar technique that is used by spread sheet programs. The most used spread sheet program – Microsoft Excel - where cells can be updated by formulas using other cell values can have very complex dependencies between cells. Whenever a cell is changed, all cells that are dependant on the changed cell are invalidated, meaning that they will not be recalculated until the next time they should be displayed on screen. A similar technique is used by Visual Value 3D. By combining the two methods presented in chapter 2.1; Forward Propagation and Backward Extraction, we can extract the benefits and kill the drawbacks of both. What is left of that fusion is a powerful algorithm to update and evaluate a scene graph in an optimal way since change propagation and evaluation are performed in different steps but hand in hand in the same structures. By abstracting Visual Value 3D to its own layer everything that is profitable from the scene graph such as culling mechanisms will still be profitable but attribute updates will become more powerful. Visual Value 3D does not only provide a robust and reliable event processing mechanism but it also opens the door for “crazy” (or creative) dependencies in a scene graph. Changing color of an object depending on its rotation, making sure two objects always are a given distance away from each other (magnetism), changing resolution of objects depending on distance to the camera or just plain and simple animations are things easily made with Visual Value 3D. Visual Value 3D also works for a general purpose since all rules and dependencies are expressed outside the application in xml, more about that in chapter 3. The cylinder example shown in chapter 2.2 is simple to express in Visual Value 3D. The same scene expressed with X3D was much more complicated to design and it rendered with around 10% of the performance in Visual Value 3D.

2.6 - Comparing Visual Value 3D to related works

To sum up what was concluded in this chapter I can say that it is certain that Visual Value 3D has an area where it is a better solution than what exist today. When event paths are merged together in a chain of dependencies the other techniques I previously presented run into problems that are elegantly solved in Visual Value 3D. The cylinder example in chapter 2.2 is easily created and maintained with Visual Value 3D and renders around 10 times faster compared to the same scene expressed in X3D and rendered in FluxPlayer[2]. Also, defining dependencies between scene graph nodes are made a lot easier with Visual Value 3D compared to e.g. X3D where a hank of routes are needed stitched together with script functions. Now, for most purposes the traditional hierarchical structure used in OSG, Java3D, Panda3D and NVSG is sufficient. The game industry still relies on such structures, but in some situations e.g. solving a system where events control complex constraints Visual Value 3D could come in handy.

[2] http://www.mediamachines.com

3 –Visual Value 3D (VV3D) Architecture and Implementation

VV3D is written in C#.NET and is combined with a 3D rendering software developed with Managed DirectX

The main idea with VV3D is to let nodes in a scene graph have their attributes exclusively expressed as values plugged into the VV3D layer. Values in the VV3D framework know how to calculate themselves and they can be dependent on other values. The responsibility to calculate consistent values is now no longer on the rendering software or in the scene graph structure but to the values themselves leaving the renderer to what it is good at, rendering!

3.1 – The framework of Visual Value 3D

Since VV3D contains values who actively know when and how to calculate themselves, they were given the name “ActiveValue”. As described in the previous section they can be everything from colors to matrices which are solved by using generics in C#. The most important classes in the VV3D framework are:

3.1.1 - ActiveValue

This is an abstract class which acts as the layer that handles the forward propagation scheme.
An ActiveValue has a private member:

```
bool evaluated;
```

which indicates if an ActiveValue is evaluated meaning that it already holds its correct value. An ActiveValue also has a subscriber list containing other ActiveValues used to handle forward propagation. If ActiveValue A is dependent of ActiveValue B, A exists in B’s subscriber list. The most important method in the ActiveValue class is RippleChange:

```
protected void RippleChange()
{
  foreach (ActiveValue v in subscribers)
  {
     v.Invalidate();
  }
}
```

This is called to indicate that an ActiveValue has been changed. The invalidate call sets the evaluator flag to false and calls the RippleChange for that ActiveValue. This means that RippleChange is a recursive call and it is important that no circular dependencies exist since that would cause endless recurrence calls. By just setting the evaluated flag to false makes invalidation a cheap operation. Hence, invalidation does not affect performance if it’s done multiple times on an ActiveValue unless it’s not done zillions of times.

3.1.2 - ActiveValue<T>

The most important class of VV3D is the ActiveValue<T> class. It is a subclass of ActiveValue and it is implemented as a generic class where T has the constraint that it has to be of a value type. A value type in C# is e.g. an int, float, Color, Matrix or a type defined with the keyword struct[3]. The most important member for ActiveValue<T> is:

```
T value;
```

This is the actual value itself and access to this value is made through the property[4] val:

```
public T Val
{
  get
  {
    if (!evaluated && evaluator != null)
    {
       val = evaluator.Evaluate();
       evaluated = true;
    }
    return val;
  }
  set
  {
     val = value;
     evaluated = true;
     RippleChange();
  }
}
```

As can be seen, calculation of a value is not performed until somebody tries to access the value. This is what makes the evaluation lazy. An ActiveValue<T> needs an evaluator which implements the interface IEvaluator<T> to calculate its value. If the evaluator is null the ActiveValue is considered to be constant and will always return the same value.

3.1.3 - IEvaluator<T>

This is an interface used by an ActiveValue<T> to evaluate its value. If a class implements this value it provides two methods:

```
void SetSubscribers(Value sub);
```

In this method we make sure that all the other ActiveValues that this ActiveValue's evaluator needs to calculate the output value are added in the subscribers list of this ActiveValue. In practice this means that this ActiveValue is guaranteed to be invalidated whenever the other ActiveValues are changed.

[3] Information about the C# language can be found at http://www.c-sharpcorner.com/
[4] A property is an access layer in .NET

```
T Evaluate();
```

This method calculates and returns a value of type T.
Here follows an example of a simple evaluator.

```
public class EvalFloatAdd : IEvaluator<float>
{

ActiveValue<float> one, two;

public EvalFloatAdd(ActiveValue<float> one,
                    ActiveValue<float> two)
  {
     this.one = one;
     this.two = two;
  }

   #region IEvaluator<float> Members
  public void SetSubscribers(ActiveValue sub)
  {
     one.AddSubscriber(sub);
     two.AddSubscriber(sub);
  }

  public float Evaluate()
  {
     return one.Val + two.Val;
  }
  #endregion
}
```

The evaluator shown above is used to evaluate an ActiveValue<float> which is a sum of two other ActiveValue<float>. The evaluation of an ActiveValue can - just as well as the RippleChange mechanism - be recursive. Since this evaluator uses two other ActiveValues in its evaluation call the access of those values can as we have seen end up with an evaluation call for that value. The evaluators use the Backward Extraction technique which we have seen before makes this a recursive call. An evaluator like this is assigned to an ActiveValue through the method:

```
public void SetEvaluator(IEvaluator<T> ev)
{
   evaluator = ev;
   if (evaluator != null)
   {
      evaluator.SetSubscribers(this);
   }
   evaluated = false;
   RippleChange();
}
```

An application using ActiveValues should first create all ActiveValues that are going to be used and then assign evaluators to the ActiveValues. This should be done to make sure that an ActiveValue can be dependent on any other existing ActiveValue.

To handle events ActiveValues should be created only to store the event's value. When an event occur the only thing that has to be made is to set the ActiveValue it is connected with. It will then be considered to be evaluated and a RippleChange will take place invalidating all dependant values. This is an important pillar for the solution to the event propagation problem. The invalidating phase starts out when events occur and performs a set on an ActiveValue. The forward propagation technique is used to invalidate all ActiveValues is either directly or secondary a subscriber of the changed ActiveValue. The ActiveValues continue to be invalidated until anyone tries to access them. That's when the backward extraction technique takes over.

By using the combination of the two mentioned techniques above an application using an ActiveValue can be sure that it will always return a correct value when access is made. The application does not have to bother about checking for state changes or updating correctly. Everything is handled by the ActiveValue itself.

3.2 – Implementation

To show the strength of what VV3D can achieve an implementation of a renderer was made using the simplest scene graph possible; a root node only containing nodes one level down shown in fig 3.1.

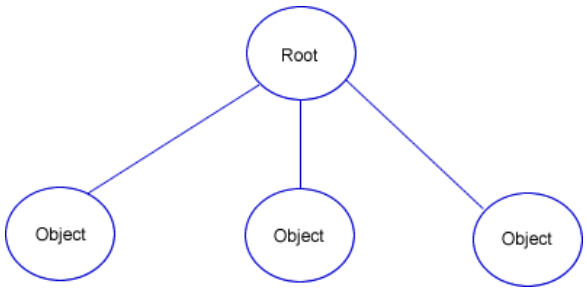

Fig 3.1 - A very simple scene graph containing 3 objects.

This means that no relation or logic is put into the scene graph but object's properties however are exclusively expressed with ActiveValues. Each renderable object in the scene graph contains an ActiveValue<Matrix> which is their transform in world space and an ActiveValue<Color> which describes their color. Together with the scene graph there is VV3D framework with containing ActiveValues connected with each other. When the renderer is rendering the scene it only needs to iterate over the scene graph's leaves and ask each object for the world matrix and other attributes which then is extracted from the VV3D framework. If no change occurred since last time the renderer asked there is no need to calculate the attributes again.

A render loop in Visual Value 3D can be described like this

1. Allow events from the user interface or the system interact with the VV3D framework. If an event cause a change in the structure it

automatically propagates to the rest of the subscribing values making them invalid.

2. Iterate over the scene graph leaves, ask the VV3D framework for transformations and other attributes and render the objects. Any update of ActiveValues will take place automatically during access.

There are two reasons for why this is good. First, a change will only affect the values that are subscribers of the change. Second, only the ActiveValues that need to be evaluated will be evaluated. When an ActiveValue that already is evaluated is asked for it will simply return its currently held value. If the chain of dependencies is not complex at all this method would not be very profitable but having very complex dependencies between scene graph nodes the combination of Forward Propagation and Backward Extraction becomes close to optimal for rendering where fast calculations of world matrices is very important.

Another thing worth to mention is that any kind of dependency can be defined. They may both be simple or complex. There is no limit on how many other ActiveValues a single ActiveValue depend on.

Fig 3.2 shows a typical situation when Visual Value 3D is built upon a scene graph.

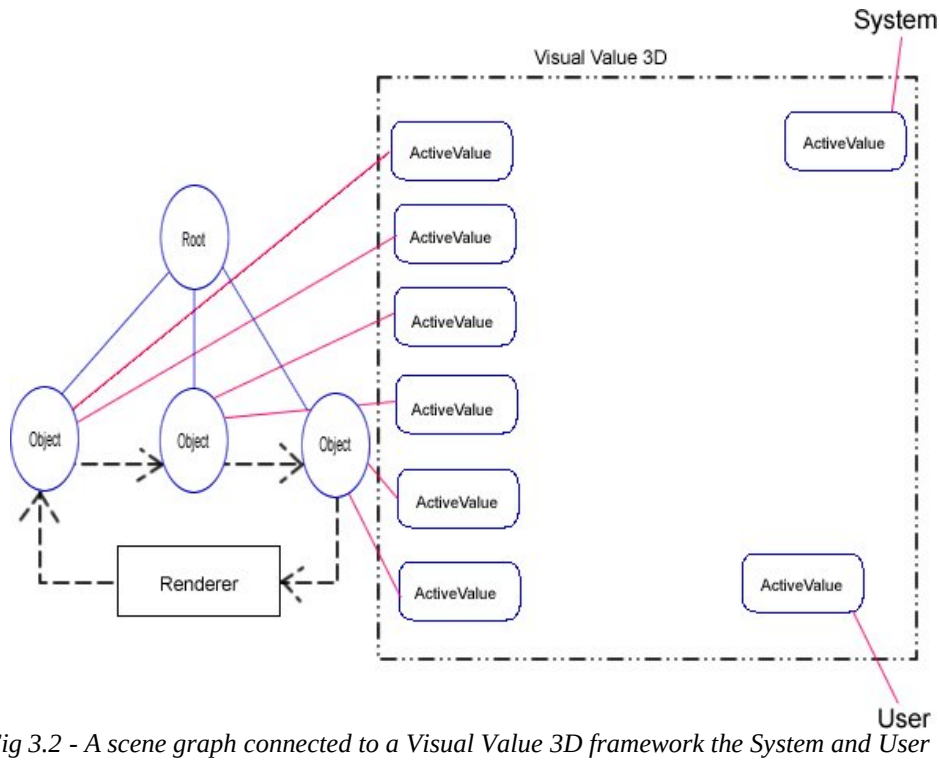

Fig 3.2 - A scene graph connected to a Visual Value 3D framework the System and User represents events that update ActiveValues in the VV3D framework.

As can be seen this scene is not very exciting. It contains 3 objects and their attributes that are connected to the VV3D framework does not have dependencies which makes them insensitive to events and therefore static. The scene could typically look like in fig 3.3.

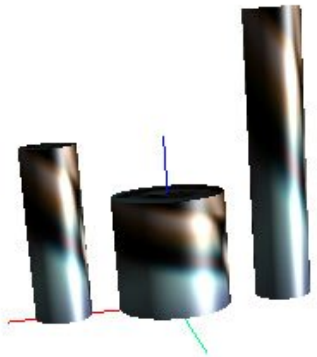

Fig 3.3 - A scene without dependencies as described in fig 3.2 could look like this. Three static cylinders are standing next to each other with three different animation functions.

By adding some more ActiveValues with evaluators a lot more interesting scenes can be achieved with the same scene graph. Fig 3.4 shows the Visual Value 3D framework in its element.

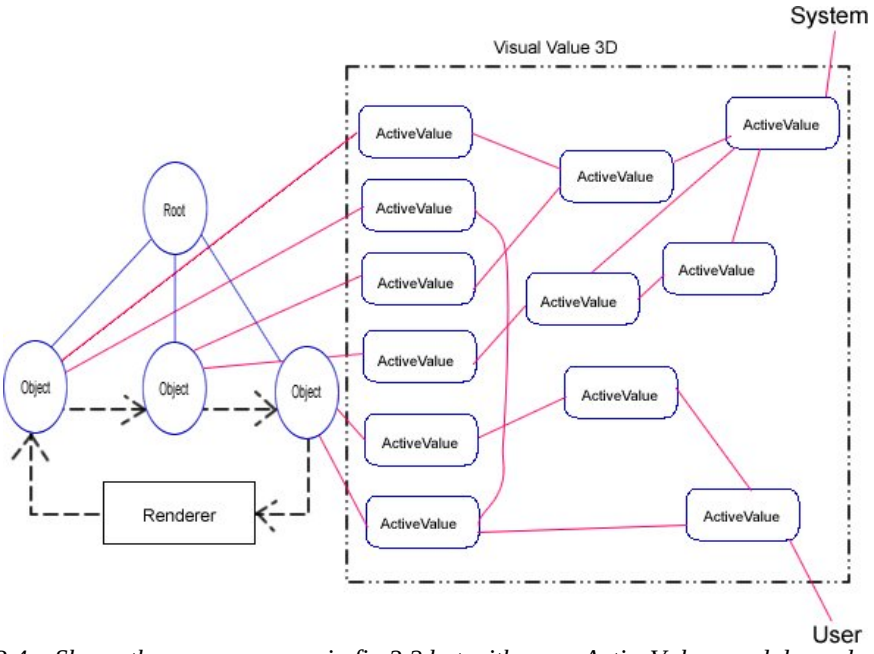

Fig 3.4 – Shows the same scene as in fig 3.2 but with more ActiveValues and dependencies added.

A Visual Value 3D framework with more complex dependencies which also reacts to events could look like in figure 3.5. These are screenshots taken from the same animated scene. The two cylinders standing up are animated to move up and down and the user is able to change scaling on both of them. The top cylinder's evaluator makes sure that it always lies on top of the other two and adjust it's rotation, translation and scaling in order to fulfill this demand. Note that there are still just three objects and the scene graph is just the same as before the dependencies were added.

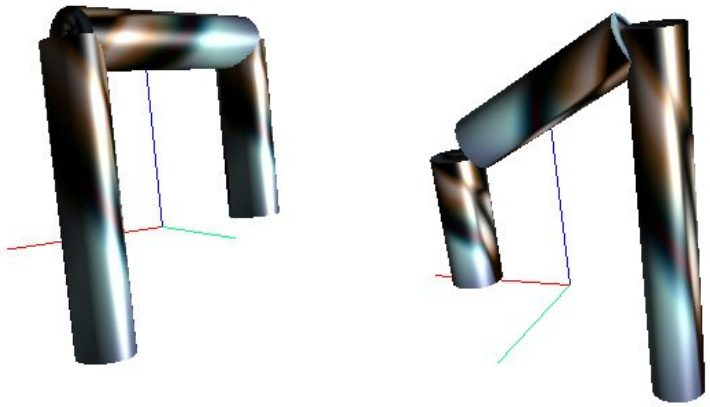

Fig 3.5 – Shows a scene with fairly complex dependencies. The evaluator for the top cylinder always makes sure that it lies on top of the two other cylinders. As can be seen the top cylinder reacts to changes of the two other cylinders. Changes can come both from the system and from a user interface.

Continuing this way, the scene graph which is really simple itself can with help from VV3D take shape with relationships that are very hard to express in a scene graph structure, such as the one shown in fig 3.6.

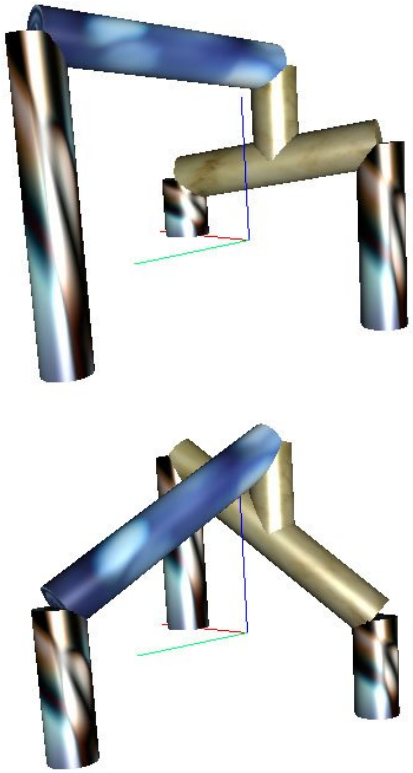

Fig 3.6 – Screenshots from an animated scene. The animation is entirely handled within the VV3D framework

Note that the evaluators used for calculating the transform for the cylinders that lie on top are exactly the same evaluator. The only difference is the parameters passed in when creating the evaluator.

ActiveValues can control all kinds of attributes in a scene graph. E.g. by making the mesh resolution for objects dependent of the distance between objects and the camera an easy level of detail engine can be created as seen in fig 3.7.

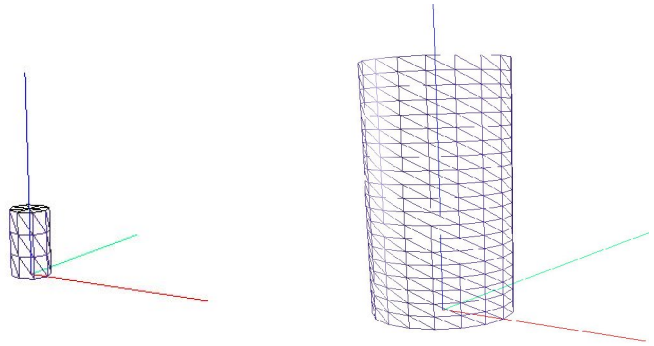

Fig 3.7 – Shows the same cylinder at different distance from the camera. As can be seen the cylinder gets a larger resolution when it is closer to the camera.

3.3 - Description of a scene

In order to make Visual Value 3D general and enable sharing of Visual Value 3D data between applications the entire scene is expressed in xml. By using xml serialization an xml file can be directly imported into a C# class. Benefits of this is that if a type can be expressed as strings e.g. int, double, float, dates etc it can directly be read into an ActiveValue of that type. ActiveValues that don't have the same kind of string representation will be created with the type's default value (e.g. a Vector will be created with the elements [0, 0, 0]) and set to their correct value later. Other solutions would require an xml parser used to read and translate the xml into the ActiveValues. That would require more and less efficient code. With serialization, ActiveValues are created on the fly and the only thing that needs a more advanced processing is the strings that represent the evaluators.

3.4 - The Cylinder Example in VV3D

The xml for the cylinder example shown in figure 2.3 is presented in Appendix A. The xml is straight forward and easy to understand. All attributes in the xml except some specific attributes (such as textures and mesh attributes) are serialized into ActiveValues. An ActiveValue with a defined evaluator is represented by an xml element with the two attributes "value" and "evaluator" and this is the only way to define evaluators to an ActiveValue. All ActiveValues without an evaluator are constant. The xml file are processed and serialized into a scene object holding all data and ActiveValues contained in the scene. During serialization the ActiveValues are created and put into a hash table making them easy to access with a unique key. The given key for an ActiveValue is related to the position the ActiveValue has in the XML file. After serialization the evaluators are just held in the scene object as plain strings, therefore a post processing step is performed after xml serialization to create and set the corresponding evaluators the ActiveValues should have.

In the serialization of the xml to the scene object some classes are frequently used. The most basic class is the ValueElement<T>. An xml element with the two attributes “value” and “evaluator” will be serialized into this object and during serialization a corresponding ActiveValue<T> is created. If the type T can be represented as a string, e.g. a float or an int, a default value can be given in the value attribute. In the evaluator attribute a string is given saying which evaluator to use for the corresponding ActiveValue. This string has to be known by the application.

Another class the scene object uses is the VectorElement class that is a subclass of ValueElement<Vector> which means that a corresponding ActiveValue<Vector> is created. In the xml an element which is serialized into a VectorElement must either have an evaluator attribute or three child nodes <X>, <Y> and <Z> where each of these three are serialized into a ValueElement<float>. If this is the case the evaluator VectorElement will automatically become a VectorFromFloats evaluator which turns the three given ActiveValue<float> to a Vector3.

The ColorElement is a subclass of ValueElement<Color>. Similar to the VectorElement this class creates an ActiveValue<Color> when serialization is made and the class’ corresponding xml element must either have an evaluator attribute or three child nodes <R>, <G> and <B> where each of these three are serialized into a ValueElement<int>. In this case the evaluator for the ColorElement will then automatically become a ColorFromInts which takes the three ActiveValue<int> and turns them into a color.

The TransformElement is a subclass of ValueElement<Matrix>. In the xml, an element which is serialized into a TransformElement must contain either an evaluator attribute or three elements <Translation>, <Rotation> and <Scaling> where each of these three are serialized into a VectorElement. If this is the case the evaluator for this class will be a MatrixFromTranRotScal which makes a matrix from the three ActiveValue<Vector> representing translation, rotation and scaling of the shape that holds the transform.

As can be seen when looking at the evaluators in the xml, All ActiveValues are given a unique name by combining their position in the xml with the name attributes and these unique names are used to store all created ActiveValues into a global hash table. Access to any ActiveValue can now be made by knowing the key. Passing ActiveValues as parameters to evaluators is expressed in the xml by using these unique keys.

3.5 - ActiveValue<Matrix>, examples

Fig 5.9 shows overview of how an ActiveValue<Matrix> can be created. An evaluator for an ActiveValue<Matrix> typically uses three ActiveValue<Vector> to represent the rotation, scaling and translation in world space. These vectors’s evaluators in turn use three

ActiveValue<float> and simply return a Vector object holding the three float values. Note that this is just an example of how an ActiveValue<Matrix> typically looks like. In any of the evaluator owned by either an ActiveValue<Vector> or an ActiveValue<float>, other dependencies can be defined.

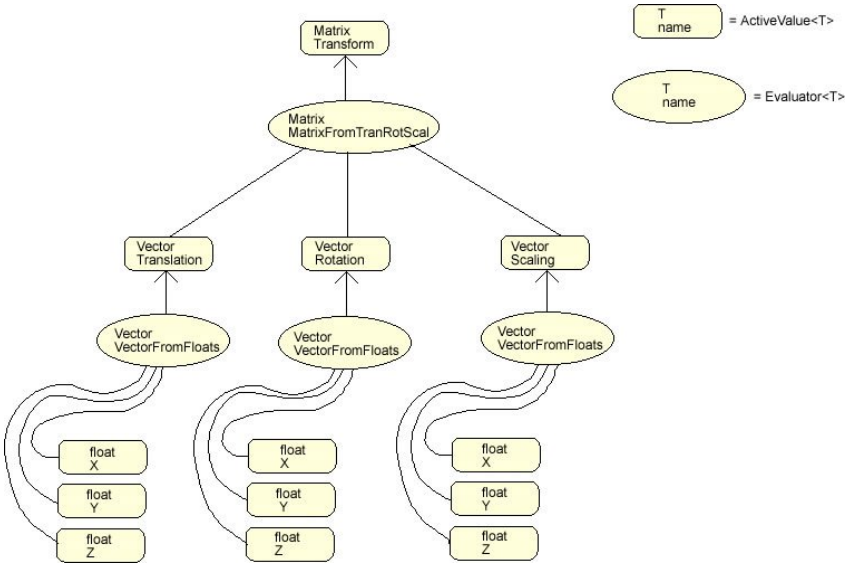

Fig 5.9 - An ActiveValue<Matrix> which describes an objects transform is typically built up by three ActiveValue<Vector3> representing translation, scaling and rotation, which in turn are built up by three ActiveValue<float> representing the x, y and z component in the vector.

An alternative way of defining how an ActiveValue<Matrix> can be constructed is shown in figure 5.9. In this case the evaluator uses one ActiveValue<Matrix> and returns the inverse of that matrix. In the sample scene the evaluator that calculates the transform for the cylinder that lies on top of the two others is similar to this one. It uses two other transform matrices in order to find its own transform.

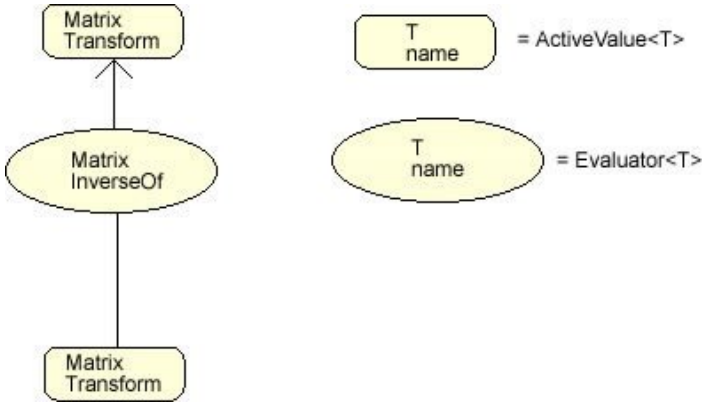

Fig 5.10 - This is another way to create an ActiveValue<Matrix>. The evaluator only uses one other ActiveValue<Matrix> and returns the inverse of that matrix.

4 – Results and Discussion

Visual Value 3D solves the event propagation problem presented in chapter 2 in an elegant way. The problems that OpenSceneGraph, NVSG and X3D had with the cylinder example in chapter 2.2 are with Visual Value 3D gone. Event handling is performed in a robust way and complex scene graphs are maintained with optimal performance. The scene in VV3D is entirely defined in an xml-file which means scenes can be built in a general way and data can be exchanged between VV3D applications.

Complex dependencies are what VV3D handles best. Therefore, in order to handle event propagation through the dependencies in the most optimal way, as many dependencies as possible should be moved from the scene graph to the VV3D framework.

As we saw when looking at related work in chapter 2, most scene graph based applications are built upon hierarchical structures. There are very few 3D applications that is built on a scene graph with merging nodes, mostly because there is no need for it, but also because it is a problem to handle events when the scene graph has merging nodes. With a VV3D framework connected to a scene graph a wider arena to play with is opened. Scene graphs attributes can be dependant on any other attribute in the scene graph, not just its parent's attributes, which will enhance creative scene graph effects.

X3D is close to VV3D in the aspect that it also describes scenes in xml and enables dynamic updates of scene graph nodes making them able to have complex dependencies. But by looking at Appendix A and B and comparing how dependencies are described in X3D and VV3D it is a lot less complicated in VV3D to design scenes with complex dependencies. A script function in X3D that calculates a value using 6 different values from other nodes in the scene graph needs to define 6 routes to be able to detect changes at the scene graph nodes the script needs. In VV3D this is automatically handled by the evaluators. When designing the scene all that is needed is to add an evaluator and define which parameters it should take and the evaluator will make sure that all the dependencies will be detected when changed. Also VV3D will just execute the evaluator at most once between two rendered frames. In X3D there is no control of this. A script node with 6 routes connected will, if all routes pass events to the script node during one frame, execute the script 6 times between two frames and the correct value will only be available after the 6^{th} time. The other 5 executions are completely unnecessary. Hence, VV3D will always perform faster and better than X3D.

Since VV3D does not yet handle culling mechanism, VV3D would be most useful together with a scene graph that handles culling but uses VV3D to calculate and maintain attributes for the scene graph nodes.

It can be clarified that the technique for Visual Value as it is used at Winsider is suitable for 3D rendering. This opens the door for Visual Value to in the future be integrated with 3D rendering in a natural way and since Visual Value today is used for configuration of industrial composite products we can expect that it is possible show a 3D representation of the configured products while it is configured by using the Visual Value technique.

I can also see that VV3D is a simple way to create creative animations since dependencies are unlimited. Movements are easy to synchronize by making different movements be dependent on the same values.

Visual Value 3D was not designed to be a game engine. The well known game engines are optimized to meet the demands on performance from the game industry. VV3D is developed to be a robust and consistent way to solve complex dependencies in a scene. But after implementing the use of backwards extraction I noticed a big difference in performance since all values in the dependency graph went from being evaluated each frame to just being evaluated when they actually had to. In some cases the performance increased with around 150%. The only thing that is done in overflow in Visual Value 3D is invalidating values. But since this is a very cheap operation it gives the least lack of performance possible. Comparing to what is gained while evaluating, which is that all unchanged values will not be evaluated, it usually is a big gain. Even if the worst case is that a change causes all values to be re-evaluated VV3D will with the backwards extraction make sure that nodes are just updated once no matter how complex dependencies are.

This project was initialized in the January of year 2005. Since that time new products have entered the market. One of them is FLEX [8] which is an AJAX based technology used to build rich multimedia applications on the internet with the Flash Player as base. By looking at the technology presented at their website they seem to have undertaken the same principles that VV3D is built upon. Events are propagated through dependencies making as little impact as possible, and when values are needed in order to display data on screen they are dynamically updated with as few function calls as possible. Combining this with the AJAX technique the communication between client and server becomes highly optimized.
The fact that more products undertake the same principles as VV3D proves that concept of Visual Value is a good design to solve the event propagation problem and that it probably will be well established amongst systems in need to solve complex dependencies the future.

4.1 - Future Work

Currently Visual Value 3D lacks the support of defining evaluators in a natural way in the XML file, the strings are now given as evaluators needs to be recognized by the system. A typical dependency like:

$$evaluator = Value2 + Value3$$

cannot be written that way in the XML. Instead the evaluator needs to be written like this:

$$evaluator = Add(Value1, Value2)$$

So making the application better and make it understand snippets of code and with that create evaluators as the xml file is parsed would lead to a big improvement.

Since the 3D renderer is built on DirectX there already are a big amount of possibilities to add a rich multimedia experience. VV3D already renders with vertex and pixel shaders but such shaders should also be possible to be defined in the xml file so it would be possible to add effects which make the rendered model look more realistic. Smoke, fog or water can be added together with sound.

We have seen that this configuration technique is usable both for industrial products for a manufacturing industry and for 3D rendering, two areas that don't have so much in common except for the use of a dependency graph. I see that all applications which use a graph will draw benefits of using this technique. It can be everything from planning traffic systems to Artificial Intelligence.

5 – References

Articles:

[1] - Stephan Diel, Jörg Keller “Constraints for 3D Graphics on the Internet”
[2] - Graham Smith, Tim Salzman, Wolfgang Stuerzlinger, “3D Scene Manipulation with Constraints”

Websites:

[3] - http://www.web3d.org/x3d/
[4] - http://www.openscenegraph.org/
[5] - http://developer.nvidia.com/object/nvsg_home.html
[6] - http://www.gamasutra.com/features/20030829/vanderbeek_pfv.htm
[7] - http://www.realityprime.com/scenegraph.php
[8] - http://www.adobe.com/products/flex/
[9] - http://java.sun.com/products/java-media/3D/
[10] - http://www.panda3d.org

Books

[11] - Ames, Nadeau, Moreland “VRML 2.0”

Appendix A – The cylinder example in Visual Value 3D

```
<?xml version="1.0" encoding="utf-8" ?>
<Scene xmlns="http://www.winsider.se/xmlns/scene3d">
  <Time />
  <!--Camera node, don't change this structure, only it's
internal values-->
  <Camera>
    <Eye>
      <X value="0"/>
      <Y value="2.0"/>
      <Z value ="-15.0"/>
    </Eye>
    <LookAt>
      <X value="0"/>
      <Y value="0"/>
      <Z value="0"/>
    </LookAt>
    <Proj fov="45" nearplane ="1.0" farplane="1000.0" />
  </Camera>
  <!--Lights node, here you can add DirectionalLight and
PointLight, but keep the structure-->
  <Lights>
    <DirectionalLight name="dirLight">
      <Direction>
        <X value="3"/>
        <Y value="3"/>
        <Z value="2"/>
      </Direction>
      <Color>
        <R value="250"/>
        <G value="250"/>
        <B value="250"/>
      </Color>
    </DirectionalLight>
  </Lights>
  <!--Model node .. here you can add as many shapes you
want, remember to apply a unique name for each shape-->
  <Model>
    <Shape type="cylinder" name="cyl1" material="default"
texture="blue_black_wave.jpg" height="2.0" radius="0.5"
stacks="3" slices="24">
      <Transform>
        <Translation>
          <X value="2"/>
          <Y value="0" evaluator="Cos(2,-2,Time)"/>
          <Z value="0"/>
        </Translation>
        <Rotation>
          <X value="0"/>
          <Y value="0"/>
          <Z value="0"/>
        </Rotation>
        <Scaling>
          <X value="1"/>
          <Y value="1" evaluator="Slider2(0,5,Cyl1 Y
Scaling)"/>
          <Z value="1"/>
        </Scaling>
```

```
      </Transform>
    </Shape>
    <Shape type="cylinder" name="cyl2" material="default"
texture="blue_black_wave.jpg" height="2.0" radius="0.5"
stacks="3" slices="24">
      <Transform>
        <Translation>
          <X value="-2"/>
          <Y value="0" evaluator="Sin(2,-2,Time)"/>
          <Z value="0"/>
        </Translation>
        <Rotation>
          <X value="0"/>
          <Y value="0"/>
          <Z value="0"/>
        </Rotation>
        <Scaling>
          <X value="1"/>
          <Y value="1" evaluator="Slider1(1,5,Cyl2 Y
Scaling)"/>
          <Z value="1"/>
        </Scaling>
      </Transform>
    </Shape>
    <Shape type="cylinder" name="cyl3" material="default"
texture="blue_black_wave.jpg" height="5.0" radius="0.5"
stacks="3" slices="24">
      <Transform
evaluator="OnTopOf(Scene/Model/cyl3/height,Scene/Model/cyl1/
Transform,Scene/Model/cyl1/height,Scene/Model/cyl2/Transform
,Scene/Model/cyl2/height)" />
    </Shape>
  </Model>
</Scene>
```

What do all the elements in the XML mean?

<Scene> - document element of the xml, holds all the other elements.
<Time> - if this Time node is given the application creates a global ActiveValue<double> which is updated each frame with the current application time.
<Camera>- holds data for the camera
<Eye>- tells the starting position of the camera. This element is serialized into a VectorElement.
<LookAt>- element similar to the Eye element showing the point the camera will be faced towards.
<Lights> - holds data about the scene's lights.
<PointLight> - describes a point light
<Position> -VectorElement describing a point light's position.
<DirectionalLight> - describes a directional light.
<Direction> -VectorElement describing a directional light's direction.
<Color> - describes a color. It is serialized into a ColorElement.
<Model> - holds data for the scene's shapes.
<Shape> -Represents a renderable object. This element has attributes describing the shape.
<Transform> - TransformElement describing a shape's transform.
<Translation> - VectorElement representing an object's translation in x-, y- and z-coordinates.
<Rotation> - VectorElement representing an object's rotation around x-, y- and z-axis.
<Scaling> - VectorElement representing an object's scaling in x-, y- and z-coordinates.

Appendix B – The cylinder example in X3D

```
<?xml version="1.0" encoding="UTF-8"?>
<X3D version='3.0' profile='Immersive'
xmlns:xsd='http://www.w3.org/2001/XMLSchema-instance'
xsd:noNamespaceSchemaLocation='
http://www.web3d.org/specifications/x3d-3.0.xsd '>
  <head>
    <meta name='title' content='cylinders.x3d'/>
  </head>
  <Scene>
    <Viewpoint description='View test' position='0 0 12'/>
    <Group>
      <Transform DEF='Cylinder1'>
        <Shape>
          <Appearance>
            <Material diffuseColor='0 1 0' shininess='1'
specularColor='1 1 1'/>
          </Appearance>
          <Cylinder radius='0.5' height='2'/>
        </Shape>
      </Transform>
      <Transform DEF='Cylinder2'>
        <Shape>
          <Appearance>
            <Material diffuseColor='0 0 1' shininess='1'
specularColor='1 1 1'/>
          </Appearance>
          <Cylinder radius='0.5' height='2'/>
        </Shape>
      </Transform>
      <Transform DEF='Cylinder3'>
        <Shape>
          <Appearance>
            <Material diffuseColor='1 0 0' shininess='1'
specularColor='1 1 1'/>
          </Appearance>
          <Cylinder radius='0.5' height='1'/>
        </Shape>
      </Transform>
      <PositionInterpolator DEF='PI1' key='0 0.5 1.0' keyValue='2 2 0,
2 0 0, 2 2 0'/>
      <PositionInterpolator DEF='PI2' key='0 0.5 1.0' keyValue='-2 -1
0, -2 2 0, -2 -1 0'/>
      <TimeSensor DEF='Clock' cycleInterval='4' loop='true' />
      <Script DEF='InterfaceScriptNode' directOutput='true'>
        <field name='cyl1Transl' type='SFVec3f'
accessType='inputOnly'/>
        <field name='cyl2Transl' type='SFVec3f'
accessType='inputOnly'/>
        <field name='cyl3Transform' type='SFNode'
accessType='inputOutput'>
          <Transform USE='Cylinder3'/>
        </field>
        <![CDATA[
        ecmascript:

        var cyl1translation;
        var cyl2translation;
```

```
        function initialize ()
        {
         Browser.print ('initialize ():');
          cyl1translation = new SFVec3f(0, 0, 0);
          cyl2translation = new SFVec3f(0, 0, 0);
        }

        function cyl1Transl (value, timestamp)
        {
          cyl1translation = value;
          setTransl ();
        }

        function cyl2Transl (value, timestamp)
        {
        cyl2translation = value;
          setTransl ();
        }

        function setTransl () {
          var deltaX = cyl1translation[0] - cyl2translation[0];
          var deltaY = cyl1translation[1] - cyl2translation[1];
          var a = (deltaY / deltaX);
          var angleZ = Math.atan(a) + 1.57;
          if (deltaX < 0) deltaX = -deltaX;
         cyl3Transform.scale = new SFVec3f(1, deltaX, 1);
          cyl3Transform.rotation = new SFRotation (0,0,1,angleZ);
          cyl3Transform.translation = new SFVec3f(0,1 +
(cyl1translation[1] + cyl2translation[1]) / 2, 0);
        }

        function shutdown()
        {
         if (DEBUG) Browser.print ('==============================');
          if (DEBUG) Browser.print ('script shutdown.');
         if (DEBUG) Browser.print ('==============================');
        }
      ]]>
      </Script>
      <ROUTE fromNode='Clock' fromField='fraction_changed'
toNode='PI1' toField='set_fraction'/>
      <ROUTE fromNode='Clock' fromField='fraction_changed'
toNode='PI2' toField='set_fraction'/>
      <ROUTE fromNode='PI1' fromField='value_changed'
toNode='Cylinder1' toField='set_translation'/>
      <ROUTE fromNode='PI2' fromField='value_changed'
toNode='Cylinder2' toField='set_translation'/>
      <ROUTE fromNode='PI1' fromField='value_changed'
toNode='InterfaceScriptNode' toField='cyl1Transl'/>
      <ROUTE fromNode='PI2' fromField='value_changed'
toNode='InterfaceScriptNode' toField='cyl2Transl'/>
    </Group>
  </Scene>
</X3D>
```