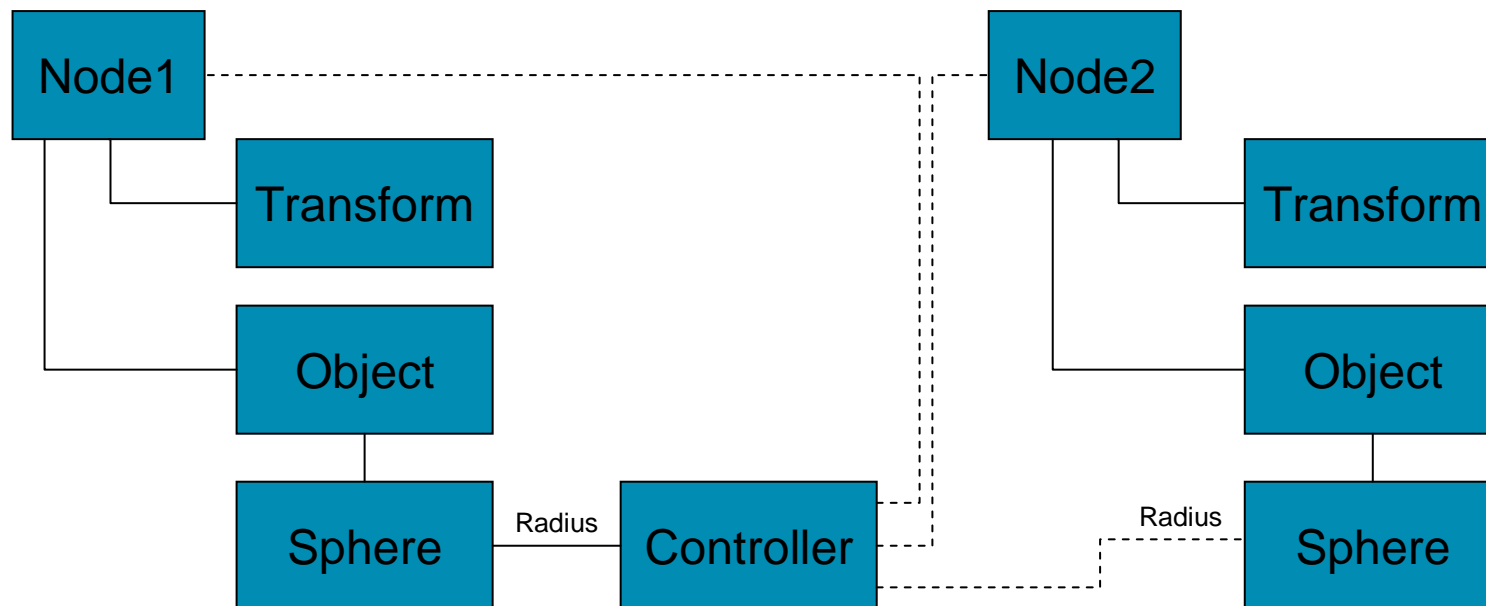


**IWeakReferenceMaker  
RefTargMonitorRefMaker  
NodeTransformMonitor**

Larry Minton  
Software Engineer  
Autodesk Media & Entertainment Division

# What was the need?



Objective: Radius of Node1's sphere depends on distance between Node1 and Node2, and Node2's sphere's radius

Problem: Have a circular reference:

Node1->Object->Sphere->Controller->Node1

# Weak vs. Strong References

A weak reference is a way of pointing to a ReferenceTarget without holding a direct reference to it.

Only want a few specific messages from the weak reference.

Don't want to prevent deletion of the weak reference.

The access methods for “strong” references are via class ReferenceMaker's NumRefs, GetReference, and SetReference.

The access methods for “weak” references are via class IWeakReferenceMaker's NumWeakRefs, GetWeakReference, and SetWeakReference.

# IWeakReferenceMaker

IWeakReferenceMaker is a virtual interface class:

## Public Member Functions

virtual int NumWeakRefs ()=0

*The number of weak references.*

virtual RefTargetHandle GetWeakReference (int i)=0

*Retrieve the indexed weak reference.*

virtual void SetWeakReference (int i, RefTargetHandle rtarg)=0

*Set the indexed weak reference.*

virtual int RemapWeakRefOnLoad (int iref)

*Specifies remapping of weak references on load.*

There is default implementation of RemapWeakRefOnLoad, all other methods are pure virtual.

# Scene Files and IWeakReferenceMaker

When a scene is saved, each object stores a list of its references. This list tells MAX which other objects are to be saved on a save selected, or loaded on a merge. This list now includes both strong and weak references.

The only part of 3ds Max core that knows about weak references is the scene load/save code.

When saving an IWeakReferenceMaker, ReferenceTargets it points to will also be saved.

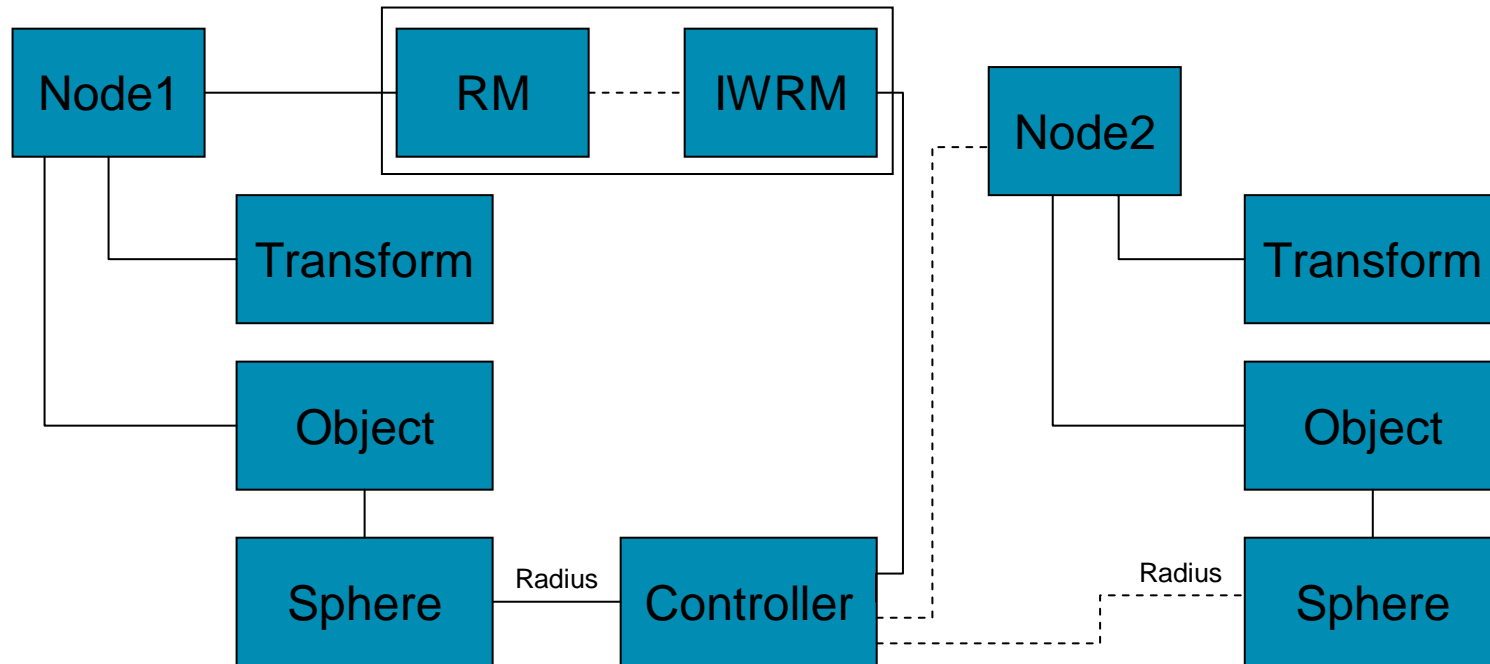
When loading an IWeakReferenceMaker, ReferenceTargets it points to will also be loaded.

# IWeakReferenceMaker interface implementation - I

The class that implements the IWeakReferenceMaker interface acts as the “loop breaker” when enumerating references downward.

The class that implements the IWeakReferenceMaker interface creates instances of a ReferenceMaker-derived class to hold references to the weak references. These class instances along with the IWeakReferenceMaker act at the “loop breaker” when enumerating dependents upward.

# IWeakReferenceMaker interface implementation - II



# RefTargMonitorRefMaker - I

An example of one of these ReferenceMaker-derived classes is RefTargMonitorRefMaker.

RefTargMonitorRefMaker:

- Holds a single reference
- Returns FALSE from method IsRealDependency
- Passes NotifyRefChanged messages to the owner
- Passes EnumDependents calls to the owner

# IRefTargMonitor

The RefTargMonitorRefMaker's owner must derive from class IRefTargMonitor. Messages from the RefTargMonitorRefMaker instance are passed to the owner through this interface.

## Public Member Functions

virtual RefResult [ProcessRefTargMonitorMsg](#) (Interval changeInt, RefTargetHandle hTarget, PartID &partID, RefMessage message, bool fromMonitoredTarget)=0

*Messages from the RefTargMonitorRefMaker instance are passed to its owner through this method.*

virtual int [ProcessEnumDependents](#) (DependentEnumProc \*dep)=0

*Calls to the RefTargMonitorRefMaker instance's EnumDependents are passed to its owner through this method.*

The RefTargMonitorRefMaker owner is responsible for ensuring that an infinite recursion does not occur. Typically the owner would set a flag while propagating a message or enumerating dependents, and not propagate a new message or enumerate dependents if that flag is set.

# RefTargMonitorRefMaker - II

RefTargMonitorRefMaker is a concrete class that derives from ReferenceMaker:

Public Member Functions

CoreExport [RefTargMonitorRefMaker](#) (IRefTargMonitor &myOwner, RefTargetHandle theTarget=NULL)

*Constructor for class instances.*

CoreExport void [SetRef](#) (RefTargetHandle theTarget)

*Set the object being watched.*

CoreExport RefTargetHandle [GetRef](#) ()

*Get the object being watched.*

IOResult [Save](#) (ISave \*isave)

*Save a pointer to the object being watched.*

IOResult [Load](#) (ILoad \*iload)

*Load a pointer to the object being watched.*

int [Proc](#) (RemapDir &remap)

*The PostPatchProc used when cloning.*

RefResult [NotifyRefChanged](#) (Interval changeInt, RefTargetHandle hTarget, PartID &partID, RefMessage message)

*The implementation of this method passes those messages to the owner via the owner's ProcessRefTargMonitorMsg callback*

BOOL [IsRealDependency](#) (ReferenceTarget \*rtarg)

*Specifies that this reference to the watched object should not prevent the watched object from being deleted.*

int [EnumDependents](#) (DependentEnumProc \*dep)

*The implementation of this method passes the DependentEnumProc to the owner via the owner's ProcessEnumDependents callback.*

# RefTargMonitorRefMaker - III

If the RefTargMonitorRefMaker instance's owner derives from IWeakReferenceMaker, don't need to worry about Save and Load methods, as 3ds Max will call GetWeakReference on scene save and SetWeakReference on scene load.

If the RefTargMonitorRefMaker instance's owner does not derive from IWeakReferenceMaker, the owner is responsible for loading and saving the weak reference. See the expanded documentation on the Save and Load methods for more information.

Highly recommend that RefTargMonitorRefMaker instance's owner derive from IWeakReferenceMaker.

Owner is responsible for cloning of the RefTargMonitorRefMaker.

# NodeTransformMonitor - I

The NodeTransformMonitor class derives from ReferenceTarget, INodeTransformMonitor, IWeakReferenceMaker and IRefTargMonitor.

INodeTransformMonitor is a virtual interface class:

Public Member Functions

virtual INode \* GetNode ()=0

*Retrieves the node being watched.*

virtual void SetNode (INode \*theNode)=0

*Sets the node being watched.*

virtual bool GetForwardTransformChangeMsgs ()=0

*Retrieves whether to monitor for REFMSG\_CHANGE/PART\_TM messages.*

virtual void SetForwardTransformChangeMsgs (bool state)=0

*Sets whether to monitor for REFMSG\_CHANGE/PART\_TM messages*

# NodeTransformMonitor - II

Typical creation code is:

```
ReferenceTarget* rntm =  
    (ReferenceTarget*)CreateInstance(REF_TARGET_CLASS_ID,  
    NODETRANSFORMMONITOR_CLASS_ID);  
INodeTransformMonitor *ntm =  
    (INodeTransformMonitor*)rntm->GetInterface(IID_NODETRANSFORMMONITOR);  
ntm->SetNode(node);  
MakeRefByID(FOREVER, i, rntm);
```

In the SetNode method, a RefTargMonitorRefMaker will be created with the NodeTransformMonitor instance as the owner and the node as the ReferenceTarget to watch.

# NodeTransformMonitor - III

The NodeTransformMonitor's ProcessRefTargMonitorMsg method watches for PART\_TM/ REFMSG\_CHANGE, REFMSG\_TARGET\_DELETED, and REFMSG\_FLAG\_NODES\_WITH\_SEL\_DEPENDENTS messages. These messages are passed to the NodeTransformMonitor's instances dependents. All other messages are ignored.

```
// only pass along a message if we are not already passing the same
// message. This breaks the message loop recursion
if ( !suspendTMMMessagePassing && fromNode &&
    message == REFMSG_CHANGE && partID==PART_TM &&
    forwardPartTM)
{
    suspendTMMMessagePassing = true;
    NotifyDependents(changelnt, partID, message);
    suspendTMMMessagePassing = false;
}
```

# NodeTransformMonitor - IV

```
if (!suspendDelMessagePassing && fromNode && message == REFMSG_TARGET_DELETED)
{
    // We can't send the REFMSG_TARGET_DELETED message, because it will look like this
    // NodeTransformMonitor is being deleted rather than node. That does very bad things to any
    // RestoreObjs that hold this NodeTransformMonitor as a reference. Send
    // REFMSG_NODETRANSFORMMONITOR_TARGET_DELETED instead, and pass
    // the node being deleted as hTarg.
    // We are by definition being called from theNodeWatcher. If an attempt is made to delete this
    // NodeTransformMonitor via DeleteThis while processing this message, delay the deletion
    // until after the NotifyDependents call returns (since we are accessing member variables)
    // and return REF_AUTO_DELETE so that theNodeWatcher is deleted.
    suspendDelMessagePassing = true;
    NotifyDependents(changeInt, PART_ALL,
        REFMSG_NODETRANSFORMMONITOR_TARGET_DELETED,
        NOTIFY_ALL, FALSE, hTarget);
    suspendDelMessagePassing = false;
    if (deleteMe)
    {
        delete this;
        return REF_AUTO_DELETE;
    }
}
```

# NodeTransformMonitor - V

```
NodeTransformMonitor::~NodeTransformMonitor()
```

```
{
    theHold.Suspend();
    // if deleteMe is true, we are in delete resulting from the watched node being deleted. We are being
    // called from ProcessRefTargMonitorMsg which is handling the message
    // REFMSG_NODETRANSFORMMONITOR_TARGET_DELETED. This message is being sent by
    // theNodeWatcher, and we don't want to delete it here. Instead, we will return
    // REF_AUTO_DELETE from ProcessRefTargMonitorMsg to delete theNodeWatcher
    if (theNodeWatcher && !deleteMe)
        theNodeWatcher->DeleteThis();
    theNodeWatcher = NULL;
    DeleteAllRefs();
    theHold.Resume();
}
```

```
void NodeTransformMonitor::DeleteThis()
```

```
{
    // if suspendDelMessagePassing is true, we are in the middle of handling a
    // REFMSG_NODETRANSFORMMONITOR_TARGET_DELETED message in
    // ProcessRefTargMonitorMsg (the watched node was deleted). Delay deletion until we are back in
    // ProcessRefTargMonitorMsg.
    if (suspendDelMessagePassing)
    {
        deleteMe = true;
        return;
    }
    delete this;
}
```

# NodeTransformMonitor - VI

A RefTargMonitorRefMaker owner is responsible for cloning the RefTargMonitorRefMaker:

```
RefTargetHandle NodeTransformMonitor::Clone(RemapDir& remap)
{
    NodeTransformMonitor* newobj = new NodeTransformMonitor();
    newobj->forwardPartTM = forwardPartTM;

    RefTargetHandle watchedNode = theNodeWatcher->GetRef();
    RefTargetHandle targetNode = remap.FindMapping( watchedNode );

    // if targetNode is NULL, the watched node hasn't been cloned. It might be cloned
    // later, or it might not. Register a post patch proc to check after all cloning is
    // finished. If watched node isn't cloned, newobj should watch this watched node.
    bool doBackPatch = false;
    if (targetNode == NULL)
    {
        targetNode = watchedNode;
        doBackPatch = true;
    }
    newobj->theNodeWatcher = new RefTargMonitorRefMaker(*newobj, targetNode);
    if (doBackPatch)
        remap.AddPostPatchProc(newobj->theNodeWatcher, false);

    BaseClone(this, newobj, remap);
    return newobj;
}
```

# Current Use in 3ds Max

NodeTransformMonitor is used by:

- The expression controller for Vector variables assigned a node
- The scripted controller for Node values

NodeTransformMonitor instances are creatable by MAXScript:

```
s=sphere()
```

```
ntm = NodeTransformMonitor node:s
```

```
ReferenceTarget:NodeTransformMonitor
```

```
showinterfaces ntm
```

```
Interface: INodeTransformMonitor
```

```
Properties:
```

```
.node : node : Read|Write
```

```
.forwardTransformChangeMsgs : bool : Read|Write
```

```
Methods:
```

```
Actions:
```

```
ntm.node
```

```
$Sphere:Sphere01 @ [0.000000,0.000000,0.000000]
```

# Can use in Scripted Plugins

```
plugin Helper HelperTest
name:"HelperTest"
classID:#(0x47db14fe, 0x4e9b5f90)
extends:dummy
(
    parameters pblock
    (
        myNode_ntm type:#maxobject
    )
    on attachedToNode theNode do
        myNode_ntm = NodeTransformMonitor \
            node:theNode \
            forwardTransformChangeMsgs:false
)

h1 = HelperTest()
h1.myNode_ntm.node
```

# Example use in scripted controller - I

Solving the need:

```
s1 = sphere pos:[100,100,0]
s2 = sphere pos:[-100,100,0]
s1.radius.controller = sc = float_Script()
sc.addNode "node1" s1
sc.addNode "node2" s2
sc.addTarget "node2Radius" s2.baseobject[#radius]
sc.setExpression "(distance node1 node2)-node2Radius"
displayControlDialog sc ""
```

# Example use in scripted controller - II

-- Create a sphere with torus that appears to slice it

```
fn CreateSlicedSphere =
```

```
(
```

```
  sn = sphere radius:50
```

```
  tn = Torus radius1:75 radius2:2
```

```
  md = slicemodifier slice_type:2
```

```
  addmodifier sn md
```

```
  c = transform_script()
```

```
  md.slice_plane.controller = c
```

```
  c.AddNode "sn" sn
```

```
  c.AddNode "tn" tn
```

```
  c.SetExpression "(tn.objecttransform * (inverse sn.objecttransform))"
```

```
)
```

```
CreateSlicedSphere()
```

End of Presentation  
Slide