Ogre 2.0 Porting Manual

(DRAFT)

Last revision: 2015-01-20

Table of Contents

1 Legal

This manual is aimed at making the transition to Ogre 2.0 as less painful and traumatic as possible.

2 Changes: Objects, Scene & Nodes

2.1 Names are now optional

Names no longer need to be unique and are optional (ie. two SceneNodes can have the same name). To identify uniqueness, classes derive from IdObject, and use IdObject::getId()

Note that, for example, Entities are completely different from SceneNodes (they don't even share a common base class), so it is possible for an Entity and a SceneNode to have the same Id.

You won't find two Entities (or rather, two MovableObjects) with the same Id. Otherwise it's a bug.

This change quite affect the creation calls. For example it's quite common to see this snippet:

```
sceneManager->createEntity( "myEntityName", "meshName.mesh" );
```

However the new definition of create Entity is as follows:

```
Entity* createEntity( const String& meshName, const String& groupName =
ResourceGroupManager::AUTODETECT_RESOURCE_GROUP_NAME,
                        SceneMemoryMgrTypes sceneType = SCENE_DYNAMIC );
```

In other words, your old snippet will try to look for the mesh “myEntityName” in the group “meshName.mesh”; which will probably fail. To port to Ogre 2.0; one would just need to write:

```
Entity *myEnt = sceneManager->createEntity( "meshName.mesh" );
myEnt->setName( "myEntityName" ); //This call is optional
```

2.2 How to debug MovableObjects' (and Nodes) data

All relevant data that needs to be updated every frame is stored in SoA form (Structure of Arrays) as opposed to AoS (Arrays of Structures)

This means that data in memory instead of being placed like the following:

```
class Node
{
        Vector3         pos;
        Quaternion      rotation;
        Vector3         scale;
};
```

It is layed out as the following

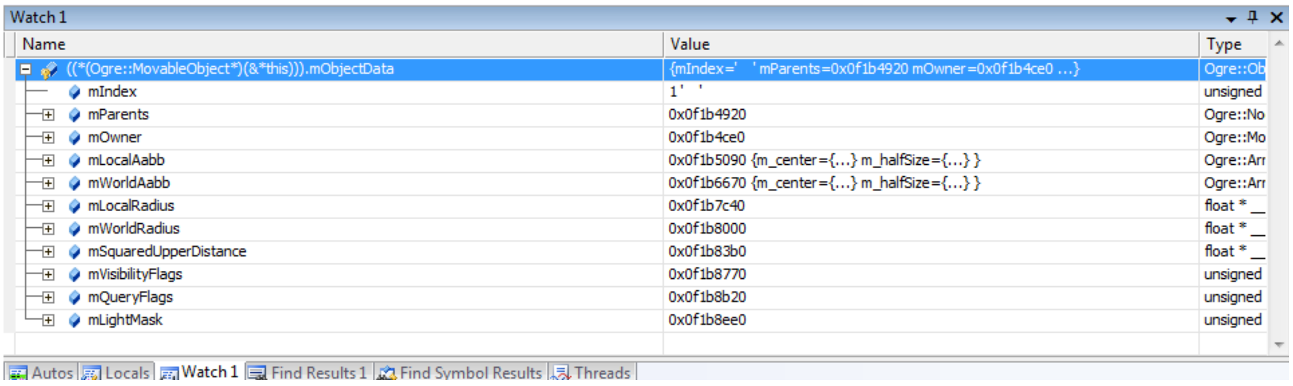
```
class Node
{
        Vector3         *pos;
        Quaternion      *rotation;
        Vector3         *scale;
};
```

However our setup is actually quite different from other engines approaches, as 4 Vector3s are packed like the following in memory:

Vectors [0; 4)	Vectors [4; 8)
XXXX YYYY ZZZZ	XXXX YYYY ZZZZ

Debugging SIMD builds can be quite counterintuitive at first, in which case defining "OGRE_USE_SIMD 0" and recompiling will force Ogre to use the C version of ArrayMath, and hence only pack 1 Vector3 per ArrayVector3.

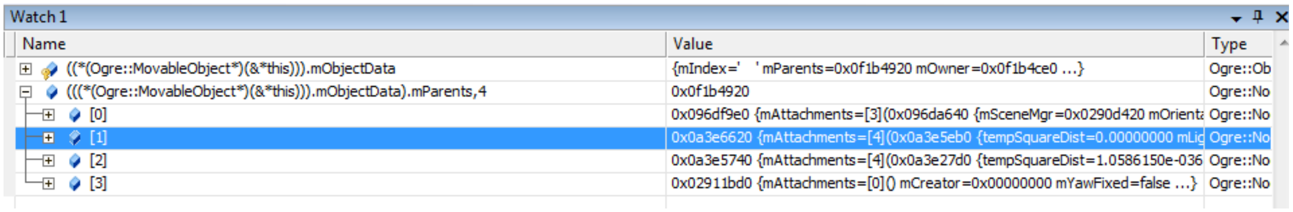
Nevertheless, debugging SSE builds directly isn't that difficult. MovableObjects store their SoA data in *MovableObject::mObjectData*. The following screenshot shows it's contents:



Name	Value	Type
((*(Ogre::MovableObject*)(&*this))).mObjectData	{mIndex=' ' mParents=0x0f1b4920 mOwner=0x0f1b4ce0 ...}	Ogre::Ob
mIndex	1' '	unsigned
mParents	0x0f1b4920	Ogre::No
mOwner	0x0f1b4ce0	Ogre::Mo
mLocalAabb	0x0f1b5090 {m_center={...} m_halfSize={...} }	Ogre::Arr
mWorldAabb	0x0f1b6670 {m_center={...} m_halfSize={...} }	Ogre::Arr
mLocalRadius	0x0f1b7c40	float * __
mWorldRadius	0x0f1b8000	float * __
mSquaredUpperDistance	0x0f1b83b0	float * __
mVisibilityFlags	0x0f1b8770	unsigned
mQueryFlags	0x0f1b8b20	unsigned
mLightMask	0x0f1b8ee0	unsigned

The most important element in this picture is **mIndex**. Because this was taken from a SSE2 (single precision) build, it's value can range between 0 and 3 (inclusive). The macro "ARRAY_PACKED_REALS" is defined as "4" for this build to let application know how many floats are being packed together.

In this case, ObjectData::mParents contains the parent nodes of all four MovableObject. In this case our parent is in mObjectData.mParents[1]; because mIndex = 1



Name	Value	Type
((*(Ogre::MovableObject*)(&*this))).mObjectData	{mIndex=' ' mParents=0x0f1b4920 mOwner=0x0f1b4ce0 ...}	Ogre::Ob
(((*(Ogre::MovableObject*)(&*this))).mObjectData).mParents,4	0x0f1b4920	Ogre::No
[0]	0x096df9e0 {mAttachments=[3](0x096da640 {mSceneMgr=0x0290d420 mOrient	Ogre::No
[1]	0x0a3e6620 {mAttachments=[4](0x0a3e5eb0 {tempSquareDist=0.00000000 mLig	Ogre::No
[2]	0x0a3e5740 {mAttachments=[4](0x0a3e27d0 {tempSquareDist=1.0586150e-036	Ogre::No
[3]	0x02911bd0 {mAttachments=[0]() mCreator=0x00000000 mYawFixed=false ...}	Ogre::No

In the picture above, we can now inspect the parent node of our object. Note that in the watch window adding a comma followed by a number forces MSVC debugger to interpret the variable as an array.

Otherwise it may only show you the first element alone. Example: “mObjectData.mParents**,4**”

The following is part of the declaration of Transform (which is used by Nodes):

```
struct Transform
{
        /// Which of the packed values is ours. Value in range [0; 4) for SSE2
        unsigned char           mIndex;
        Node                    **mParents;
        /* ... */
        bool      * RESTRICT_ALIAS mInheritOrientation;
};
```

When ARRAY_PACKED_REALS = 4 (i.e. SSE builds), though not strictly correct, we could say mParents is declared as:

```
struct Transform
{
        /* ... */
        Node                    *mParents[4];
        /* ... */
};
```

Hence to know which mParents belongs to us, we have to look at mParents[mIndex].

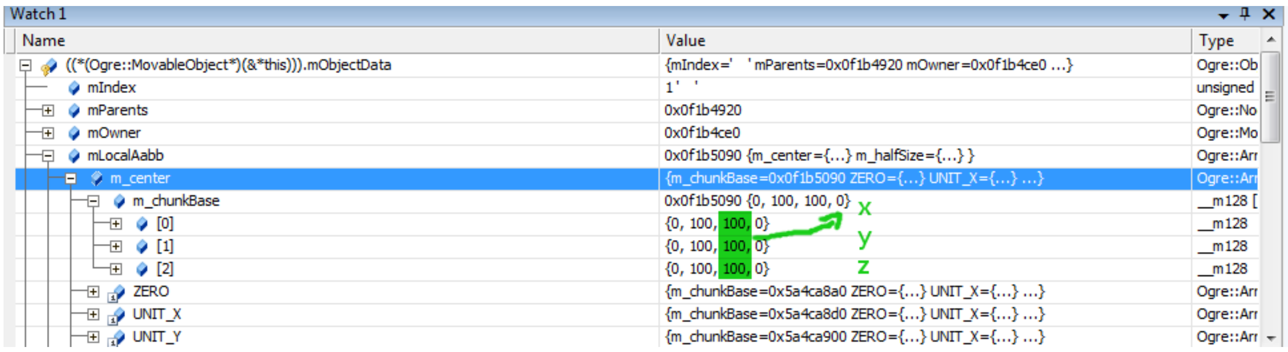
Same happens with mInheritOrientation. When 9 consecutive Nodes are created, if we take a look at mParents pointers, we would notice that first 4 point to the same memory location:

1. Transform::mParents = 0x00700000
2. Transform::mParents = 0x00700000
3. Transform::mParents = 0x00700000
4. Transform::mParents = 0x00700000
5. Transform::mParents = 0x00700010
6. Transform::mParents = 0x00700010
7. Transform::mParents = 0x00700010
8. Transform::mParents = 0x00700010
9. Transform::mParents = 0x00700020

The Transform of the first 4 Nodes will have exactly the same pointers; **the only difference is the content of mIndex**. When we go to the 5th, the pointers increment by 4 elements. mIndex is used to determine where our data really is. This layout may be a bit hard to grasp at first, but it's quite easy once you get used to it. Notice we satisfy a very important property: all of our pointers are aligned to 16 bytes.

2.2.1 Interpreting ArrayVector3

ArrayVector3, ArrayQuaternion & ArrayMatrix4 require a bit more of work when watching them through the debugger:


Watch 1

Name	Value	Type
((*(Ogre::MovableObject*)(&*this))).mObjectData	{mIndex=' ' mParents=0x0f1b4920 mOwner=0x0f1b4ce0 ...}	Ogre::Ob
mIndex	1' '	unsigned
mParents	0x0f1b4920	Ogre::No
mOwner	0x0f1b4ce0	Ogre::Mo
mLocalAabb	0x0f1b5090 {m_center={...} m_halfSize={...} }	Ogre::Arr
m_center	{m_chunkBase=0x0f1b5090 ZERO={...} UNIT_X={...} ...}	Ogre::Arr
m_chunkBase	0x0f1b5090 {0, 100, 100, 0}	__m128 [
[0]	{0, 100, 100, 0}	__m128
[1]	{0, 100, 100, 0}	__m128
[2]	{0, 100, 100, 0}	__m128
ZERO	{m_chunkBase=0x5a4ca8a0 ZERO={...} UNIT_X={...} ...}	Ogre::Arr
UNIT_X	{m_chunkBase=0x5a4ca8d0 ZERO={...} UNIT_X={...} ...}	Ogre::Arr
UNIT_Y	{m_chunkBase=0x5a4ca900 ZERO={...} UNIT_X={...} ...}	Ogre::Arr

Here the debugger is telling us that the center of the Aabb in local space is at (100; 100; 100). We're reading the 3rd column because mIndex is 1; if mIndex were 3; we would have to read the 1st column.

m_chunkBase[0] contains four "XX**X**X" which are read right to left.

m_chunkBase[1] contains four "YY**Y**Y", ours is the second one (starting from the right)

m_chunkBase[3] you should've guessed by now contains "ZZ**Z**Z"

Note that if, for example, the 4th element (in this case it reads (0, 0, 0)) is an empty slot (i.e. there are only 3 entities in the entire scene); it could contain complete garbage; even NaNs. This is ok. We fill the memory with valid values after releasing memory slots to prevent NaNs & INFs; as some architectures are slowed down when such floating point special is present; but it is possible that some of them slip through (or it is also possible a neighbour Entity actually contains an infinite extent, for example).

Is m_chunkBase a transform matrix? NO. In SSE2 SIMD builds, ArrayVector3 packs 4 vectors together (because ARRAY_PACKED_REALS = 4). If 4 Nodes are created named A, B, C, D; the picture above is saying:

- m_chunkBase[0] = { D.x, C.x, B.x, A.x }
- m_chunkBase[1] = { D.y, C.y, B.y, A.y }
- m_chunkBase[2] = { D.z, C.z, B.z, A.z }

So, to know the contents of B, you need to look at the 3rd column.

2.2.2 Dummy pointers instead of NULL

Seeing a null pointer in ObjectData::mParents[4] is most likely a bug unless it's temporary. During SoA update; those memory slots that were not initialized (or whose MovableObjects haven't been attached to a SceneNode yet) are set to a dummy pointer owned by the MemoryManager instead of setting to null.

This prevents us from checking that the pointers are null every time we need access to them in SoA loops (which are usually hotspots); with the associated branch misspredictions that may be associated. **This is a pattern in Data Oriented Design.**

Note however, that MovableObject::mParentNode **is** null when detached (since it isn't a SoA variable) while MovableObject::mObjectData::mParents[mIndex] points to the dummy node. When attached, both variables will point to the same pointer.

Same happens with other pointers like ObjectData::mOwner[] and Transform::mParents[]

2.3 Attachment and Visibility

In Ogre 1.x an object “was in the scene” when it was attached to a scene node whose ultimate parent was root. Hence a detached entity could never be displayed, and when attached, calling setVisible(false) would hide it.

In Ogre 2.x, objects are always “in the scene”. Nodes just hold position information, can be animated, and can inherit transforms from their parent. When an Entity is no longer associate with a node, it hides itself (setVisible(false) is implicitly called) to avoid being rendered without a position. Multiple entities can share the same position, hence the same Node.

This means that when attaching/detaching to/from a SceneNode, the previous value of MovableObject::getVisible is lost. Furthermore, calling setVisible(true) while detached is illegal and will result in a crash (there is a debug assertion for this).

2.4 Attaching/Detaching is more expensive than hiding

Due to how slow was Ogre 1.x in traversing SceneNodes (aka the Scene Graph), some users recommended to detach its objects or remove the SceneNode from its parent instead of calling setVisible(false); despite the official documentation stating otherwise.

In Ogre 2.x this is no longer true, and we do a significant effort to keep updates and iterations as fast as possible. This may have in turn increased the overhead of adding/removing nodes & objects. Therefore hiding objects using setVisible is much more likely to be orders of magnitude faster than destroying them (unless they have to be hidden for a very long time)

2.5 All MovableObjects require a SceneNode (Lights & Cameras)

Unless hidden (see Attachment and Visibility), all MovableObejcts like Entities, InstancedEntities, Lights and even Cameras require being attached to a SceneNode; since Nodes are the beholders of the transform (position and rotation).

There are no node-less Lights anymore. Their transform data is in the Nodes as it works perfectly with the optimized, streamlined functions (particularly update derived bounding box for computing visibility) and have no longer their own position and direction; which is now hold by the Node.

Another reason for this decision is that combining the transform data from the node with the Light's was inefficient, while the overhead of using the additional Node is virtually eliminated in Ogre 2.0; and furthermore it works better for lights that require more than direction, but a full quaternion (e.g. textured spot lights and area lights).

Functions like *Light::getDirection* and *Light::setDirection* will redirect to the scene node, and fail when not attached. Lights when created aren't attached to a SceneNode, so you will have to attach to one first before trying to use it.

Cameras do however have their own position and rotation for simplicity and avoid breaking older coder so much (unlike Lights). Performance wasn't a concern since it's normal to have less than a dozen cameras in a scene, compared to possibly thousands of lights. By default cameras are attached to the root scene node on creation.

Therefore if your application was attaching Cameras to SceneNodes of your own, you will have to detach it first calling *Camera::detachFromParent;* otherwise the engine will raise the well-known exception that the object has already been attached to a node.

2.6 Obtaining derived transforms

In the past, obtaining the derived position was a matter of calling SceneNode::_getDerivedPosition. Ogre would keep a boolean flag to know if the derived transform was dirty or not. Same happened with orientation and scale.

Ogre 2.0 removed the flag has for the sake of performance[1] (except for debug builds which use a flag for triggering assertions).

The following functions will use the last cached derived transform without updating:

- Node::_getDerivedPosition
- Node::_getDerivedOrientation
- Node::_getDerivedScale

What this means that the following snippet won't work as expected, and will trigger an assert in debug mode:

```
sceneNode->setPosition( 1, 1, 1 );
Ogre::Vector3 derivedPos = sceneNode->_getDerivedPosition();
```

All derived transforms are efficiently updated in SceneManager::updateAllTransforms which happens inside updateSceneGraph. You should start using the derived transforms after the scene graph has been updated. Users can have fine-grained control on when the scene graph is updated by manually implementing (or modifying) Root::renderOneFrame

Nonetheless, if the number of nodes that need to be up to date is low, users can call the “Updated” variants of these functions:

- Node::_getDerivedPositionUpdated
- Node::_getDerivedOrientationUpdated
- Node::_getDerivedScaleUpdated

1 This is a performance optimization. For a reasoning behind this, read the Ogre 2.0 design slides.

These functions will force an update of the parents' derived transforms, and its own. It is slower and not recommended for usage of a massive number of nodes. If such thing is required, consider refactoring your engine design to require the derived transforms after SceneManager::updateAllTransforms.

The following snippet demonstrates how to use updated variants:

```
sceneNode->setPosition( 1, 1, 1 );
sceneNode->setOrientation( Quaternion( Radian( 4.0f ), Vector3::UNIT_X ) );
sceneNode->setScale( 6, 1, 2 );

Vector3 derivedPos        = sceneNode->_getDerivedPositionUpdated();
//There's no need to call the Updated variants anymore. The node is up to date now.
Quaternion derivedRot     = sceneNode->_getDerivedQuaternion();
Vector3 derivedScale      = sceneNode->_getDerivedScale();
```

MovableObject's world Aabb & radius follows the same pattern and subject to the same issues.

2.7 SCENE_STATIC and SCENE_DYNAMIC

Both MovableObjects[2] and Nodes have a setting upon creation to specify whether they're dynamic or static. Static objects are meant to never move, rotate or scale; or at least they do it infrequently.

By default all objects are dynamic. Static objects can save a lot of performance on CPU side (and sometimes GPU side, for example with some instancing techniques) by telling the engine they won't be changing often.

2.7.1 What means a Node to be SCENE_STATIC:

- Nodes created with SCENE_STATIC won't update their derived position/rotation/scale every frame. This means that modifying (eg) a static node position won't actually take effect until SceneManager::notifyStaticDirty(mySceneNode) is called or some other similar call that foces an update.

If the static scene node is child of a dynamic parent node, modifying the dynamic node will not cause the static one to notice the change until explicitly notifying the SceneManager that the child node should be updated.

If a static scene node is child of another static scene node, explicitly notifying the SceneManager of the parent's change automatically causes the child to be updated as well.

Having a dynamic node to be child of a static node is perfectly plausible and encouraged, for example a moving pendulum hanging from a static clock. Having a static node being child of a dynamic node doesn't make much sense, and is probably a bug (unless the parent is the root node).

2.7.2 What means a Entities (and InstancedEntities) to be SCENE_STATIC:

Static entities are scheduled for culling and rendering like dynamic ones, but won't update their world AABB

2 i.e. Entities, InstancedEntities

bounds (even if their scene node they're attached to changes) Static entities will update their aabb if user calls SceneManager::notifyStaticDirty(myEntity) or the static node they're attached to was also flagged as dirty. Note that updating the node's position doesn't flag the node as dirty (it's not implicit) and hence the entity won't be updated either.

Static entities can only be attached to static nodes, and dynamic entities can only be attached to dynamic nodes.

2.7.3 General

On most cases, changing a single static entity or node (or creating more) can cause a lot of other static objects to be scheduled to update, so don't do it often, **and do it all in the same frame**. An example is doing it at startup (i.e. during loading time)

Entities & Nodes can switch between dynamic & static at runtime by calling setStatic. However InstancedEntities can't.

You need to destroy the InstancedEntity and create a new one if you wish to switch (which, by the way, isn't expensive because batches preallocate the instances) InstancedEntities with different SceneMemoryMgrTypes will never share the same batch.

Attention #1!

Calling SceneNode::setStatic will also force a call to MovableObject::setStatic to all of its attached objects. If there are objects you wish not to switch flags, detach them first, and then reattach.

If there are InstancedEntities attached to that node, you have to detach them first as they can't directly switch between types. Otherwise the engine will raise an exception.

Attention #2!

Calling setStatic(true) when it was previously false will automatically call notifyStaticDirty for you.

Q: Do the changes mean that you can set a "static"-flag on any entity and it automatically gets treated as static geometry and the batch count goes down when there are many static entities sharing the same material?

A: No and yes. On normal entities, "static" allows Ogre to avoid updating the SceneNode transformation every frame (because it doesn't change) and the AABB bounds from the Entity (because it doesn't change either). This yields massive performance bump. But there is no batch count going down.
When using Instancing however, we're already batching everything together that has the same material, so it is indeed like Static Geometry, except that we cull per instance basis (which puts a bit more strain on CPU, but allows for very fine grained frustum culling for the GPU, giving it less work), and 2.0's culling code is several times faster than 1.9's.

When using normal entities, batch count won't go down when using the "static" flag. However it will greatly improve performance compared to 1.9, because we're skipping the scene node transform & AABB update phases, and that takes a lot of CPU time.

2.8 Ogre asserts mCachedAabbOutOfDate or mCachedTransformOutOfDate while in debug mode

If you get assertions that "*mCachedAabbOutOfDate*" or "*mCachedTransformOutOfDate*" are true, they mean mean that the derived world AABB wasn't updated but was attempted to be used, or that the derived transform[3] was out of date and an attempt to use it was made; respectively.

They can trigger for various reasons:

1. Nodes/Objects are of type SCENE_STATIC and their data changed (i.e. you called setPosition) but no call to SceneManager::notifyStaticDirty was made.
    - **Solution:** Call SceneManager::notifyStaticDirty, or use a function that implicitly calls that function.
2. You've manually modified their data (i.e. you called setPosition) after SceneManager::updateAllTransforms or SceneManager::updateAllBounds happened; most likely through a listener.
    - **Solutions:**
        1. Move your calculations so that they happen before updateAllTransforms.
        2. If it's just a few nodes, call Node::_getFullTransformUpdated or MovableObject::getWorldAabbUpdate after modifying the object to force a full update on that node/entity only. You only need to call these functions once, and its transform will be updated and cached; if you keep calling the "*Updated" variations, you're just going to eat CPU cycles recalculating the whole transform every time.
        3. If it they're many nodes/entities, manually call updateAllTransforms/updateAllBounds again.
3. It's a bug in Ogre. Because the refactor was very large, some components still try to modify nodes and movables after the call to updateAllTransforms/updateAllBounds.
    - **Solution:** Report the bug to JIRA. When we get to refactor that faulty component, the node will be touched before the call to updateAllTransforms, but if the component isn't yet scheduled for refactor we might just fix it by calling getDerivedPositionUpdated

3 Derived position, orientation, scale or the 4x4 matrix transform.

3 Technical Overview

3.1 Overview

ArrayMemoryManager is the base abstract system for handling SoA memory. Derived classes (like NodeArrayMemoryManager or ObjecDataArrayMemoryManager) provide a list of with the bytes needed for each SoA pointer that will be used. I wrote a few slides to [4]help with understanding these concepts.

There is one NodeArrayMemoryManager per Node parent level. Root node is parent level 0; the children of Root is parent level 1. And Root's children's children are parent level 2.

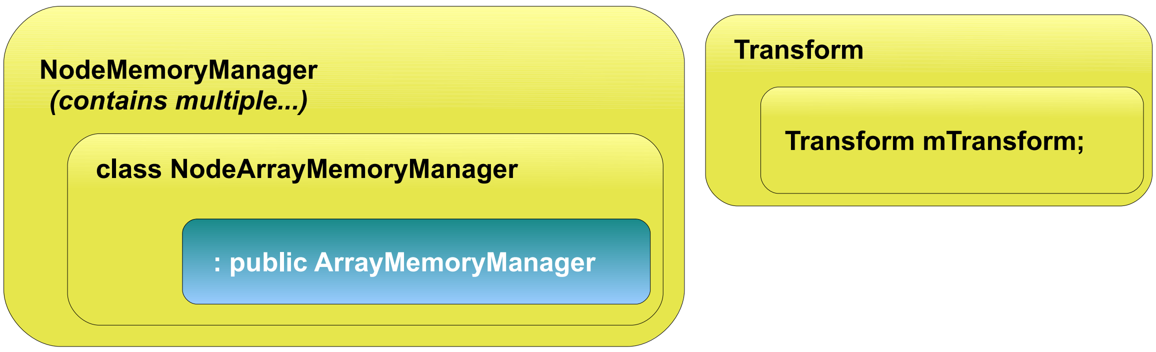
There is also one ObjectDataArrayMemoryManager per Render Queue.

NodeMemoryManager is the main public interface, which handles all the NodeArrayMemoryManagers.

Likewise, ObjectDataMemoryManager is the main public interface, which handles all the ObjectDataArrayMemoryManager.

NodeMemoryManager

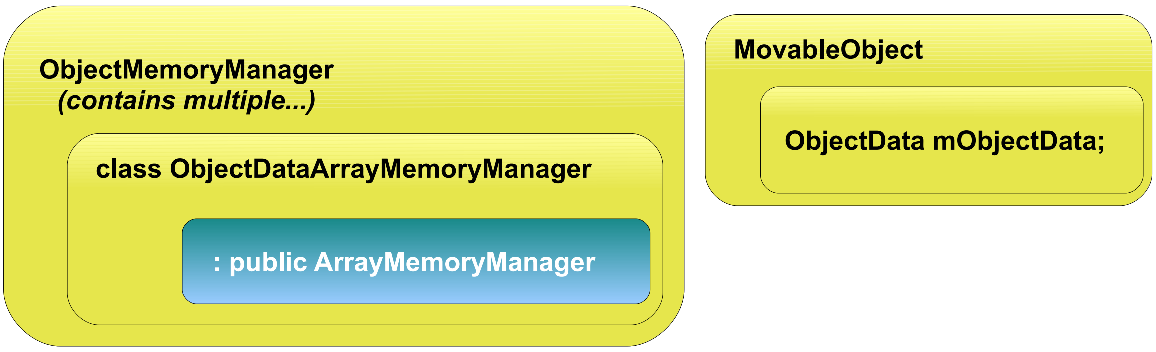
- Multiple NodeArrayMemoryManager. One per parent level.
    - Each one return a Transform (contain 4x4 Matrices, position, rotations, scale, the derived pos, rot, scales; etc) which is all SoA pointers.



ObjectDataMemoryManager

- Multiple ObjectDataArrayMemoryManager. One per render queue.
    - Each one return an ObjectData (contain Aabbs in local space, in world space, radius, visibility masks; etc) which is all SoA pointers.

4 Mirror 1; Mirror 2



3.1.1 SIMD Coherence

Using SSE2 single precision, OGRE usually processes 4 nodes/entities at a time. However, if there is only 3 Nodes; we need to allocate memory for 4 of them (ArrayMemoryManager already handles this) and initialize the values to sane defaults even if they aren't used (eg. Set quaternion & matrices to identity) to prevent NaNs during computations. Certain architectures slowdown if one of the elements in an xmm register contains a NaN.

Furthermore, null pointers are not valid, but dummy pointers are used instead.

SIMD Coherence is very important for both stability and performance, and is 99% of the time responsability of the Memory Managers

3.2 Memory Managers usage patterns

ArrayMemoryManagers work with the concept of slots. When a Node asks for a Transform, it is asking for a slot. When a MovableObject asks for an ObjectData, it is asking for a slot.

In an SSE2 build, 4 slots make a block. How many slots are needed to make a block depends on the value of the macro ARRAY_PACKED_REALS.

Slots are most efficient when requested and released in LIFO order.

When LIFO order is not respected, a release (i.e. destroying a Node) puts the released slot into a list. When a new slot is requested, the one from the list is used and removed from it. When this list grows too large, a cleanup will be performed. The cleanup threshold can be tweaked.

3.2.1 Cleanups

Cleanups happen when the number of slots released in non-LIFO order grows too large. A cleanup will move memory so that it is contiguous again.

Why are cleanups needed? Simply put, **performance**. Imagine the following example (assuming ARRAY_PACKED_REALS = 4): The user created 20 nodes, named A through T:

ABCD EFGH IJKL MNOP QRST

Then the user decides to delete nodes B, C, D, E, F, G, H, I, J, L, N, O, P, Q, R, S, T; the resulting memory layout will be the following:

A*** **** **K* M*** ****

where the asterisk '*' means an empty slot. When parsing the SoA arrays (i.e. while updating the scene nodes, updating the MovableObject's world Aabbs, frustum culling) everything is accessed sequentially.

The code will loop **4 times**, to process A, then nothing, then K, them M. If ARRAY_PACKED_REALS would be 1; the code would loop 13 times.

This is obviously inefficient if it stays for a long time. During real world application this issue won't impact performance if only done for 4 nodes, but if this kind of “fragmentation” is happening to thousands of nodes, performance drop would be noticeable.

The cleanup will move all nodes to make them contiguous again:

AKM* **** **** **** ****

Hence the code will only loop once for SSE builds (or 3 times if ARRAY_PACKED_REALS = 1)

3.3 Memory preallocation

ArrayMemoryManagers preallocate a fixed amount of slots (always rounded up to multiples of ARRAY_PACKED_REALS) specified at initialization time.

When this limit is reached, the following may happen:

- The buffer is resized. Any pointer to the slots will be invalidated (i.e. holding an external reference to Transform::mPosition). The Nodes &MovableObjects internal pointers are automatically updated.
- The hard limit is reached (hard limit is also set at initialization time, the default is no limit). When this happens the memory manager will throw.

3.4 Configuring memory managers

Note: at the time of writing, memory managers have not a direct way of setting up the amount of preallocations or how often they perform cleanups.

3.5 Where is RenderTarget::update? Why do I get errors in Viewport?

Advanced users are probably used to low level manipulation of RenderTargets. As such they're used to setting up their custom Viewports and calling RenderTarget::update.

That is too low level. **Instead, users are now encouraged to setup Compositor nodes and multiple workspaces to perform rendering to multiple RTs, even if it's for your own custom stuff** . The new Compositor is much more flexible than the old one (which has been removed). See section about

Compositors for more information.

Viewports are no longer associated with cameras as they're now stateless (they used to cache the camera currently in use) and a lot of settings they used to hold (like background colour, clear settings, etc) have been moved to nodes. See *CompositorPassClearDef* and *CompositorPassClear*.

RenderTarget::update has disappeared because the render scene update has been split in two stages, cull and render.

If you still insist in going low level, see the code on *CompositorPassScene::execute* to understand how to prepare a RenderTarget and render it manually. But again, we insist you should try the Compositor.

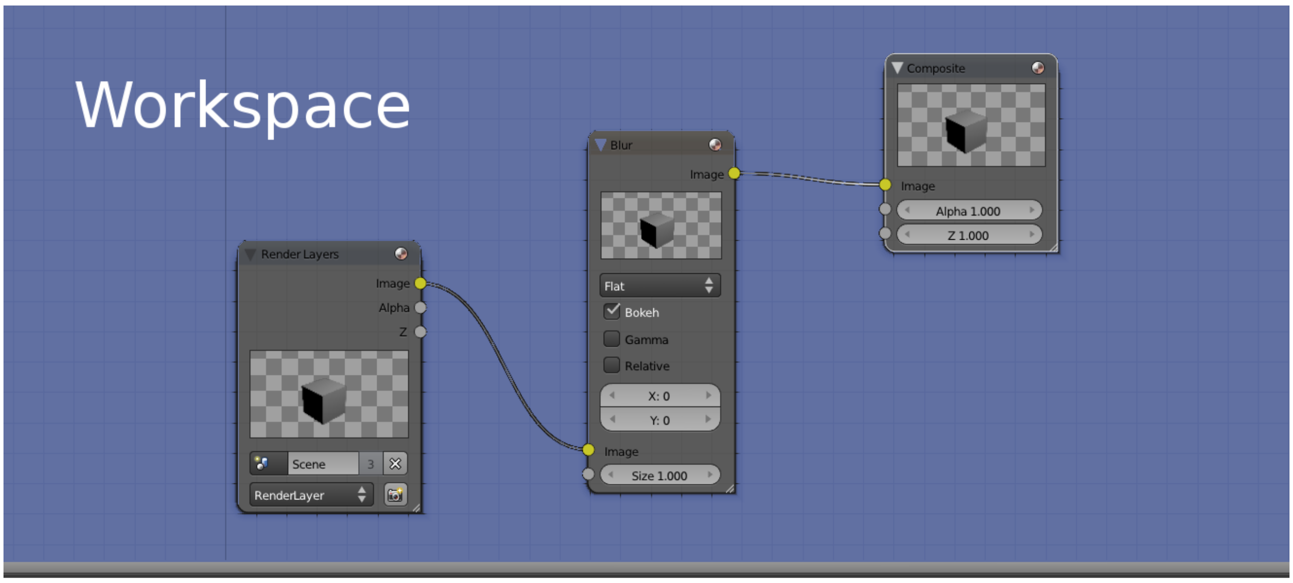
4 Compositor

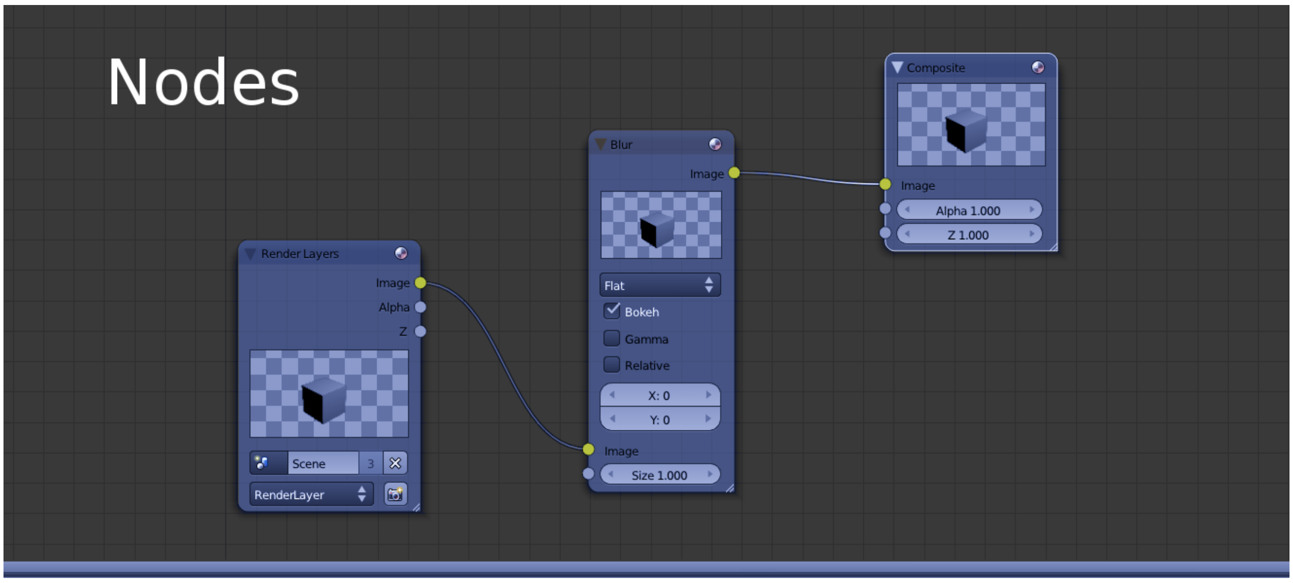
The Compositor is a Core and key component in Ogre 2.0. In 1.x, it was just used for fancy post-processing effects. In 2.0, it's the way to tell Ogre how you want to render the scene. Without setting it up, Ogre won't render to screen.

With the Compositor, the user stops having to deal with setting Viewports, RenderTargets, updating these RenderTargets every frame, etc.

Instead the user has now to setup Nodes and a Workspace. The workspace is the top level system where the user indicates which Nodes he wants to use, how they will be connected, which global render textures will be declared (which can be seen by all nodes from the same workspace), where to render the final output (i.e. RenderWindow, an offscreen RenderTexture) and which SceneManager to use. The user can have multiple workspaces active at the same time.

The new Compositor system was heavily inspired by Blender's Compositor system. Picture from Blender:





Compositor script syntax hasn't changed much, which should make porting them quite easy. Internally though, the new system was written from scratch (while reusing & reviewing some of the existing code); as the previous Compositor was stack-based, while the new one is node-based.

So, if you used to manipulate the Compositor directly from C++; porting efforts could be considerably bigger.

4.1 Nodes

A compositor node most likely resembles what used to be “a compositor” in 1.x

The following node clears the RT and draws everything that is in the render queue #50

```
compositor_node MyNode
{
	in 0 Input_as_MyLocaName // Take input texture #0 and use the local name "Input_as_MyLocaName" for reference

	target Input_as_MyLocaName
	{
		//Clear to violet
		pass clear
		{
			colour_value 1 0 1 1
		}
		pass render_scene
		{
			visibility_mask	0xffffffff //Viewport's visibility mask

			rq_first	50		//Inclusive
			rq_last		51		//Not inclusive
		}
	}

	out 0 Input_as_MyLocaName
}
```

Where is “Input_as_MyLocaName” defined? What is its resolution? Its bit depth? The RT comes from the input channel, so the answer is that it depends on how the Workspace will connect this node. The Workspace may pass a local RTT declared in a previous node or it could pass RenderWindow.

4.1.1 Input & output channels and RTTs

A node's RT may come from three different sources:

1. It was locally declared.
2. It comes from an input channel.
3. It is a global texture declared in the Workspace. Global textures must use the ***global_*** prefix

4.1.1.1 Locally declared textures

The following script declares a local texture of resolution 800x600, clears it to violet, and puts it in the output channel #0 (so other compositor nodes can use it as input):

```
compositor_node MyNode
{
	texture rt0 800 600 PF_R8G8B8

	target Input_as_MyLocaName
	{
		//Clear to violet
		pass clear
		{
			colour_value 1 0 1 1
		}
	}

	out 0 rt0
}
```

You may have noticed the syntax for declaring the RTT is **almost exactly the same** as it was in Ogre 1.x

Refer to Ogre's 1.9 documentation for more information about it.

There are a couple of small changes:

- **New parameter "no_gamma"**: In 1.x HW gamma would be on by default if the global gamma settings were on; with no way to turn it off. But it could be forced always on using the keyword 'gamma'. In Ogre 2.x, leaving the option blank uses the system's settings, writing the keyword 'gamma' forces it on, and using the keyword '*no_gamma*' turns it off.
- The parameter "scope" is no longer available.
- The parameter "pooled" is no longer available.

4.1.1.2 It comes from an input channel.

Input channels are numbered. An input channel must be given a name so that they can be referenced locally at node scope by all target passes. There can't be any gaps (i.e. use channels 0 & 2 but not 1)

Output channels are also numbered and can be assigned an RTT. This number will be later used by the

Workspace to perform the connections.

The workspace will be responsible for connecting node A's output channels with node B's input channels. In **other words, channels are a way to send, receive and share RTTs between nodes.**

The only restriction is that global textures can't be used neither as input or output (global textures are referenced directly). There can be more input channels than output channels, viceversa, and there may be no input nor output channels (i.e. when working with global textures alone).

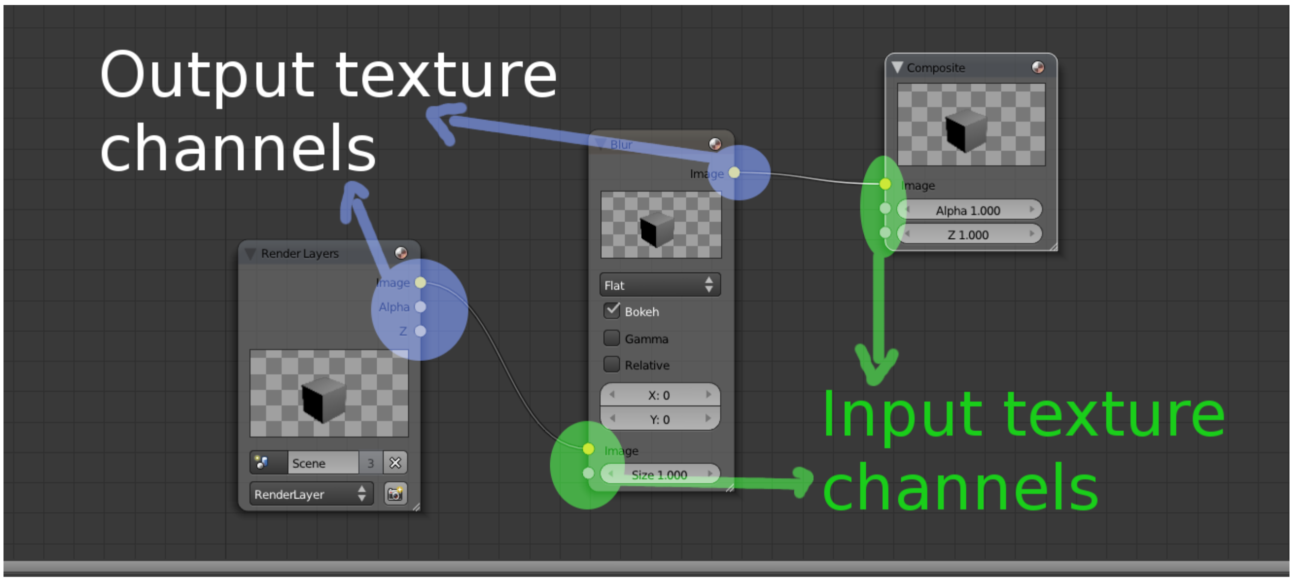
The following (rather useless) snippet takes channel #0 and sends it to output channel #1, takes input channel #1 and sends it through output channel #0 (in other words it flips the channels), and also sends an uninitialized texture created locally to channel #2:

```
compositor_node MyNode
{
        in 0 myFirstInput
        in 1 mySecondInput

        texture rt0 target_width_scaled 0.25 target_height_scaled 0.25 PF_R8G8B8

        out 0 mySecondInput
        out 1 myFirstInput
        out 2 rt0
}
```

Drawing inspiration from Blender's compositor system, the little dots on the left would be the input channels, while the dots on right would be the output texture channels:



4.1.1.3 *It is a global texture*

Global textures are declared at workspace scope and have the same syntax as local textures (which is the same Ogre 1.x's syntax). There are a few restrictions:

1. Global texture names must contain the ***global_*** prefix. Inversely, any local texture or input channel trying to use a name that starts with “global_” is illegal.
2. They can't be used in input or output channels.
3. Global textures can be seen by any node belonging to the same workspace, but they can't see a global texture that belongs to a different workspace.
4. If a Node uses a global texture that the Workspace didn't declare, execution of the workspace will fail.

Global textures are as powerful/dangerous as global variables are in C++. The main reason of their existence is that many nodes may use temporary rtts to perform their work, and it's *very* wasteful to declare these intermediate rtts on every node when they can be shared and reused.

Sharing and reusage can also be achieved through input & output channels, however for temporary rtts (or rtts that are accessed very frequently, i.e. a deferred shader's G Buffer) it would lead to connection hell; and hence global textures are a much better fit.

- in <channel_id> <local_texture_name>

  channel_id is a number in range [0; inf) but must be consecutive and continuous (no gaps, i.e. define channel 0, 2, but not 1). loca_texture_name cannot start with “global_”. A node definition may have no input.

- out <channel_id> <local_texture_name>

  channel_id is a number in range [0; inf) but must be consecutive and continuous (no gaps, i.e. define channel 0, 2, but not 1). loca_texture_name cannot start with “global_”. A node definition may have no output.

- custom_id <string>

  Custom string that will be hashed to identify this Node definition. Useful for classifying nodes into categories.

4.1.1.4 Main RenderTarget

When creating the Workspace instance, the C++ code will ask for the RT which should be the ultimate target (i.e. the RenderWindow). This RT is very important as keywords like “target_width_scaled” and settings like fsaa & hw gamma will be based on the attributes from this main RT. This feature will be seen in more detail in the Workspace section.

Attention #1!

By default you cannot use the main RenderTarget as a texture (because it's usually the RenderWindow and D3D and OpenGL don't allow it), and doing it may result in a crash.

It is possible to manually call node->connectFinalRT and supply a texture pointer (i.e. if the final RenderTarget is a RenderTexture) that can be bound. An automated way of doing this is not yet implemented.

4.1.2 Target

Targets include multiple passes. Their main purpose is define to which RenderTarget the passes will render to.

What's worth noting is that targets accept an optional parameter '*slice*'.

The slice parameter is optional and refers to which slice or face from a 3D texture (or cubemap or 2D array) to use from the given texture. Valid values can be numeric or the hint '+X'. '-X', '+Y', '-Y', '+Z', and '-Z'.

Note: When the target is a 3D/Cubemap/array texture, if the slice goes out of bounds, an exception will be raised. If the target is a 2D or 1D texture, this value is silently ignored. Default: slice = 0

4.1.3 Passes

Passes are the same as they were in Ogre 1.x. At the time of writing there are 6 types of passes:

- clear (PASS_CLEAR)
- quad (PASS_QUAD)
- resolve (PASS_RESOLVE)
- render_scene (PASS_SCENE)
- stencil (PASS_STENCIL)
- custom (PASS_CUSTOM)

More passes are planned including the ability for users to extend with custom passes which used to be present in 1.x.

Planned passes are:

- Paraboloid mapping passes (useful for efficient env. mapping and point light shadow maps)
- N pass (optimizes Ogre data for two or more passes i.e. a Z pre-pass with minimum overhead in the engine since the cull data is the same)

All passes support the following script parameters:

- pass <type> [customId]

  '*type*' must be one of the supported types: clear, quad, resolve, render_scene, stencil, custom.

The *customId* parameter is optional and is used by custom passes to give the registered custom pass provider the means to identify multiple types, in case there are more than one type of custom passes.

- num_initial <number>

  Number of times this will be executed. Default is -1, which means always execute. When the execution count hits that value, it won't executed again until a d3d device reset, resize or workspace recreation (the execution count is reset and executed N times again)

  This parameter replaces the “only_initial” parameter in Ogre 1.x.

- identifier <number>

  An arbitrary user-defined numeric ID used for identifying individual passes in the C++ code.

4.1.3.1 clear

The syntax for clear passes is the same as 1.x; except that by default now Stencil is also cleared. This follows performance reasons, as GPU architectures where the Z buffer is tied with the stencil buffer, clearing only the Z buffer hinders the driver from discarding the buffer entirely or using fast Z clears.

Additionally, all passes can define the viewport area they will work on, meaning clearing specific regions is now possible.

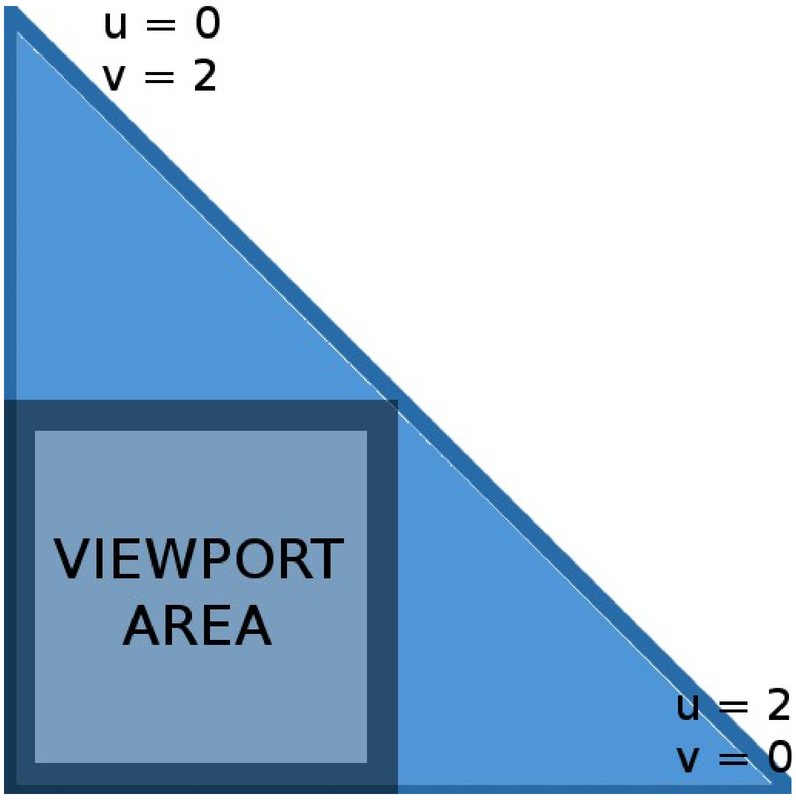
4.1.3.2 quad

Quad passes have the same syntax as 1.x; plus the following keywords have been added:

- use_quad [yes|no]

  Default is *no*. When *no*; the compositor will draw a fullscreen *triangle*. Due to how modern GPUs work, using two rectangles wastes GPU processing power in the diagonal borders because pixels are processed *at least* in 2x2 blocks; and the results from the pixels out of the triangle have to be discarded. A single triangle is more efficient as all blocks are fill the viewport area, and when the rectangle goes out of the viewport, the gpu efficiently clips it.

  When the viewport is not 0 0 1 1; this value is forced to *yes*. The following picture illustrates a fullscreen triangle:



Interpolation will cause that the effective UV coordinates will be in the [0; 1] range while inside the viewport area.

Using camera_far_corners_world_space will also force to use a quad instead of a tri (but camera_far_corners_view_space works with tris)

For an explanation of why this is a performance optimization, refer to "Optimizing the basic rasterizer" by Fabien Giesen.

The following setting was available in Ogre 1.x; but was not documented:

- quad_normals [camera_far_corners_view_space|camera_far_corners_world_space]

  Sends through the "NORMALS" semantic the camera's frustum corners in either world space or view space. This is particularly useful for efficiently reconstructing position using only the depth and the corners.

  Interesting read: Reconstructing Position From Depth Part I, Part II, Part III

Starting Ogre 2.0, UV coordinates are always sent to the vertex shader in pass quads.

Attention!

In Ogre 2.x; you need to apply the **world-view-proj matrix** so that the the pass being drawn compensates for texel-to-pixel aligning reads in Direct3D9. Failing to do so will not only cause the aforementioned alignment issue, but also will cause glitches when the viewport is not 0 0 1 1

In Ogre 1.x, only the proj matrix was necessary to fix texture flipping issues when rendering to FBOs in OpenGL.

4.1.3.3 resolve

Note: at the time of writing, resolve passes have not been fully implemented

TBD

When the Render System doesn't support explicit resolves (or textures were created with no msaa setting), textures are treated as implicitly resolved and all resolve passes are ignored.

See *MSAA: Explicit vs Implicit resolves* section for more information.

4.1.3.4 render_scene

The syntax is also similar to 1.x; but there were a couple modifications:

- rq_first <id>

  Replaces first_render_queue. The default is 0. Must be a value between 0 and 255. The value is inclusive

- rq_last <id>

  Replaces last_render_queue. The default is “max” which is a special parameter that implies the last active render queue ID. If numeric, value must be between 0 and 255. The value is **not** inclusive.

- viewport <left> <top> <width> <height>

  Specifies the viewport. Also supported by all other passes (i.e. clear & quads), The default is “0 0 1 1” which covers the entire screen. Values should be between 0 and 1.

  The Compositor will automatically share Viewport pointers between different passes to the same RenderTarget (even for different nodes) as long as they share the exact same parameters.

- shadows <off|shadow_node_name> <reuse|recalculate|first>

  Off by default. Specifies the shadow node to use for rendering with shadow maps. See section about *Shadow Nodes* for more information. When a shadow node's name is provided, the second parameter defaults to “*first*”.

- camera <camera_name>

  When not specified, the default camera is used for rendering the pass (this default camera is specified when instantiating the workspace from C++).

When a name is given, the Compositor will look for this camera and use it. Very useful for reflection passes (mirrors, water) where the user wants to be in control of the camera, while the Compositor is associated with it. The Camera must be created by the user before the workspace is instantiated and remain valid until the workspace is destroyed.

- lod_camera <camera_name>

  The camera point of view from which the LOD calculations will be based from (i.e. useful for shadow mapping, which needs the LOD to match that of the user camera). When an empty string is provided, Ogre will assume the lod camera is the same as the current camera, except for shadow nodes in which it will assume it's the lod_camera from the normal pass the shadow node is attached to. Default: Empty string.

- lod_update_list [yes|no]

  When No (or false), the LOD list won't be updated, and will use the LOD lists of calculated by a previous pass. This saves valuable CPU time. Useful for multiple passes using the same lod_camera (without a pass in the middle with a different lod_camera that would override the cached LOD lists). If your application is extremely CPU bound, and hence you don't need LOD, turning this setting to false in all passes will effectively turn lodding off (and alleviate the CPU). Default: Yes; except for passes belonging to shadow nodes, which is forced to false unless lod_camera is a non-empty string.

- lod_bias <bias>

  Applies a bias multiplier to the lod. Valid values are in range [0; Inf). A higher lod bias causes LOD to pop up sooner. Default: 1.0

- camera_cubemap_reorient [yes|no]

  When Yes, the camera will be reoriented for rendering cubemaps, depending on which slice of the render target we're rendering to (3D, Cubemaps and 2D-array textures only). Its original orientation is restored after the pass finishes. The rotations are relative to its original orientation, which can produce counter-intuitive results if the Camera wasn't set to identity (unless that's the desired effect). See Passes section on how to indicate which slice should we render to. Default: No.

  Note: if the target is not a cubemap, Ogre will still try to rotate the camera, often to unintended angles.

4.1.3.5 stencil

Stencil passes haven't changed at all since Ogre 1.x; remember to restore the stencil passes before leaving the node otherwise next nodes that will be executed may use unexpected stencil settings.

4.1.4 Textures

4.1.4.1 MSAA: Explicit vs Implicit resolves

Not long ago, MSAA support was automatic, and worked flawlessly with forward renderers and no postprocessing. Direct3D 9 and OpenGL were not able to access the individual MSAA subsamples from shaders at all.

Fast forward to the present, MSAA resolving should be performed after HDR to avoid halos around edges, and deferred shading can't resolve the G-Buffer otherwise aliasing only gets worse.

Direct3D10 and GL 3.2 introduced the ability of access the MSAA subsamples from a shader, also giving the ability to write custom resolves.

For those unaware what "resolving MSAA" means; a very brief explanation is that when rendering using 2xMSAA, we're actually rendering to a RT that is twice the resolution. "Resolving" is the act of scaling down the resolution into the real RT (i.e. think of Photoshop or Gimp's downscale filter modes). See the Resources section at the end for links to detailed explanations of how MSAA works.

To cleanly deal with this new feature without breaking compatibility with D3D9 & older GL render systems while at the same time being able to effortlessly switch MSAA on and off; the notion of "Explicit" and "Implicit" resolves were added.

Implicit resolves

By default all RTTs are implicitly resolved. The behavior of implicitly resolved textures mimics Ogre 1.x (except for implementation and design issues in Ogre 1.x that could cause an RTT to resolve multiple times per frame unnecessarily)

The RTT has an internal flag for being "dirty". The texture gets dirty when rendering to it; and it stops being dirty when it is resolved.

When you attempt to bind a dirty RTTs as a texture, you're forcing Ogre to resolve it. This means that you should try to delay using the RTT as a texture as much as possible until you've done with rendering to it.

Otherwise the RTT may resolve more than once if you render to it: render (gets dirty), use it as a texture, render to it again (gets dirty again), and use again as a texture (all in the same frame). In some cases this is unavoidable, but often it isn't.

Explicit resolves

Explicit resolves are used when you want to either implement a custom resolve other than the API's default; or you want access to the MSAA subsamples directly through shaders.

Like implicit resolves, RTTs have a dirty flag. However:

1. Attempting to bind a dirty RTT as a texture will cause Ogre to send the MSAA buffer; granting the shader the ability to access subsamples.

2. Attempting to bind a non-dirty dirty RTT as a texture will cause Ogre to send the resolved buffer. The shader won't be able to access subsamples.

In summary, shaders can access subsamples while a texture is dirty. To clean the dirty flag and perform a resolve on the RTT, use the PASS_RESOLVE pass. This is why they're called “explicit” resolves; because you have to *explicitly* tell Ogre to resolve an msaa target and unset the dirty flag[5].

On RenderSystems where explicit resolving is not supported, all textures will be treated as implicitly resolved and PASS_RESOLVE passes will be ignored; which should work straightforward and without issues except for a few corner cases.

Use the RSC_EXPLICIT_FSAA_RESOLVE Render system capability flag to check if the API supports explicit resolves.

Resources

- A Quick Overview of MSAA
- Experimenting with Reconstruction Filters for MSAA Resolve

4.2 Shadow Nodes

The only way to have shadows in Ogre is through shadow nodes.

Stencil shadows and “textured shadows” have been removed from Ogre 2.0; only depth shadow maps are supported.

A shadow node is a special type of Node (in fact, the class inherits from CompositorNode) that is executed inside a regular node (normally, a render_scene pass) instead of being connected to other nodes.

It is possible however, to connect the output from a Shadow Node to a regular Node for further postprocessing (i.e. reflective shadow maps for real time Global Illumination), but Shadow Nodes cannot have input. *This particular feature (output to regular nodes) is still a work in progress at the time of writing since ensuring the regular node is executed after the shadow node has been executed can be a bit tricky.*

4.2.1 Setting up shadow nodes

Shadow nodes work very similar to regular nodes. Perhaps their most noticeable difference is how are RTTs defined. The following keywords are supposed at shadow node scope:

- technique <uniform|planeoptimal|focused|lispsm|pssm>

  Specifies which shadow technique to use for the subsequent shadow map declarations. The default is uniform.

Note: at the time of writing, lispsm technique is broken and has terrible artifacts. This bug seems to affect

5 Note: You're allowed to keep an explicitly resolved textured dirty forever (i.e. never resolve, in case your main purpose is to always access fsaa subsamples)

both Ogre 1.9 & 2.0. Use focused technique instead.
planeoptimal has also not been implemented yet.

- use_aggressive_focus_region <true|false>

  Used by Focused, LispSM & PSSM techniques. Default is true. If you're having shadow mapping glitches, try setting this value to false. Note however, quality degradation for disabling it can be noticeable.

- optimal_adjust_factor <factor>

  Used by LispSM & PSSM techniques. Default is 5. Lower values improve the quality of shadow maps closer to camera, while higher values reduce the "projection skew" which is the key feature of lispsm; thus causing quality to revert and look more like focused.

  Tip: When this value is negative, lispsm reverts to a focused shadow map technique. This is very useful for users who want to use PSSM techniques with focused setups instead of LispSM.

- light_direction_threshold <angle_in_degrees>

  Used by LispSM & PSSM techniques. Default is 25. Originally, LispSM produces artifacts when the camera's direction is parallel to the light's direction. When the angle between the camera direction and the light's is lower than the threshold Ogre begins to gradually increase the optimal_adjust_factor so that it reverts back to focused setup, thus getting rid of these artifacts.

  Notes: When optimal_adjust_factor is negative (i.e. pssm + focused), this setting is ignored.

- num_splits <num_splits>

  Only used by PSSM techniques. Specifies the number of splits per light. Can vary per shadow map. The number of splits must be greater than 2. Default is 3.

- pssm_lambda <lambda>

  Only used by PSSM techniques. Value usually between 0 & 1. The default is 0.95. PSSM's lambda is a weight value for a linear interpolation between exponential and linear separation between each split. A higher lambda will use exponential distribution, thus closer shadows will improve quality. A lower lambda will use a linear distribution, pushing the splits further, improving the quality of shadows in the distance.

- shadow_atlas <name> light <light index> [split <split index>]

  See shadow atlas section.

shadow_map <Name> <Width> <Height> <Pixel Format> [<MRT Pixel Format2>] [<MRT Pixel FormatN>] [gamma] [fsaa <fsaa_level>] [depth_pool <poolId>] light <lightIndex> [split <index>]

Shadow maps declaration order is important. The first shadow map declared becomes shadow map #0; the second shadow map declared becomes #1; and so on. Most of the settings are the same as for regular textures. So only the new settings or the ones that behave differently will be described:

- <Name>

  Like regular textures, a locally unique name must be assigned (and cannot start with *global_* prefix). To avoid confusions, it is highly recommended that you name them by the number of shadow map. i.e. name the first shadow map “0”, the second one “1”

- <fsaa>

  The default value is always 0 regardless of system settings. Setting this to higher values will activate fsaa. This behavior differs from regular texture declarations.

- <gamma>

  When not present, shadow maps will never use HW gamma conversion, regardless of system settings. When this setting is present, the texture will perform automatic HW gamma conversion for you (when supported by the hardware). This behavior differs from regular texture declarations.

- light <index>

  Indicates which light index will be associated with this shadow map. i.e. the Shadow map #0 may contain the Nth closest shadow mapping light to the entity, not necessarily the first one.

- split <split index>

  Default: 0; only necessary when using PSSM techniques. Indicates which split this shadow map refers to.

shadow_map <shadowMapName0> <shadowMapName1> {}

Declaring a shadow map is not enough. You need to tell Ogre what do you want to render to it. And for that you need render_scene passes.

Shadow nodes can be written with the regular “target { pass render_scene {} }” syntax. However when you have 6 shadow maps with the same exact pass settings, it's cumbersome to write the pass six times. Instead the “*shadow_map*” keyword repeats the passes for you.

4.2.2 Example

The following is a basic script that will set a single shadow map with a focused setup:

```
compositor_node_shadow myShadowNode
{
	technique focused
	shadow_map 0 2048 2048 PF_FLOAT16_R light 0
	//Render shadow map "0"
	shadow_map 0
	{
		pass clear { colour_value 1 1 1 1 }
		pass render_scene
		{
			rq_first 0
			rq_last max
		}
	}
}
```

The typical setup is to have one directional light for the sun, and then multiple point or spot lights. This means directional light should use a PSSM setting for best quality, while point & spot lights shadow maps could use focused or uniform.

The following script creates 3 shadow maps for 3 PSSM splits, and 3 additional ones for the remaining lights:

```
compositor_node_shadow myShadowNode
{
	technique pssm
	//Render 1st closest light, splits 0 1 & 2
	shadow_map myStringName 2048 2048 PF_FLOAT16_R light 0 split 0
	shadow_map 1 1024 1024 PF_FLOAT16_R light 0 split 1
	shadow_map 2 1024 1024 PF_FLOAT16_R light 0 split 2

	//Change to focused from now on
	technique focused
	shadow_map 3 1024 1024 PF_FLOAT16_R light 1
	shadow_map 4 1024 1024 PF_FLOAT16_R light 2
	shadow_map 5 512 512 PF_FLOAT16_R light 3

	//Render shadow maps "myStringName", "1", "2", "3", "4" and "5"
	shadow_map myStringName 1 2 3 4 5
	{
		pass clear { colour_value 1 1 1 1 }
		pass render_scene
		{
			rq_first 0
			rq_last max
		}
	}
}
```

Showing off the full flexibility of shadow nodes: The following example is a bit large but shows off that shadow map rendering can be done in any order, with different settings, custom render queues for just one of the maps, etc. The user has a lot of power on defining how the shadow maps will be rendered:

```
compositor_node_shadow myShadowNode
{
	technique pssm
	//Render 1st closest light, splits 0 1 & 2
	shadow_map myStringName 2048 2048 PF_FLOAT16_R light 0 split 0
	shadow_map 1 1024 1024 PF_FLOAT16_R light 0 split 1
	shadow_map 2 1024 1024 PF_FLOAT16_R light 0 split 2

	//Change to focused from now on
	//(we can also change optimal_adjust_factor, etc)
	technique focused
	 //Render 2nd closest light in the 4th shadow map
	//(could be directional, point, spot)
	shadow_map 3 1024 1024 PF_FLOAT16_R light 1
	//Render 3rd closest light in the 5th shadow map
	shadow_map 4 1024 1024 PF_FLOAT16_R light 2

	//Don't use aggressive focus region on shadow map 5 (and from anything
	//declared from now on, we can also turn it back on later) just for
	//showcasing how this works
	use_aggressive_focus_region false
	shadow_map 5 target_width_scaled 0.25 1024 PF_FLOAT16_R light 3

	//Render shadow maps "myStringName", "1", "2", and "5"
	shadow_map myStringName 1 2 5
	{
		pass clear { colour_value 1 1 1 1 }
		pass render_scene
		{
			visibility_mask 0xffffffff
			rq_first 0
			rq_last max //Special value implying maximum
		}
	}
	//Render shadow map "4" with a different setting
	shadow_map 4
	{
		pass clear { colour_value 1 1 1 1 }
		pass render_scene
		{
			shadow_map_idx    4 //Note this is needed!
			visibility_mask 0xfaffffff
			rq_first    40
			rq_last     51
		}
	}
	//Regular way of declaring passes also works.
	target 3
	{
		pass clear { colour_value 1 1 1 1 }
		pass render_scene
		{
			shadow_map_idx    3 //Note this is needed!
			visibility_mask 0xfaffffff
			rq_first    42
			rq_last     51
		}
	}

	//Send this RTT to the output channel. Other nodes can now use it.
	out 0 myStringName
}
```

Setting up a shadow node is very flexible and very powerful. When environment-mapped passes are implemented, it should be possible to implement point lights that can fully render their 360° surrounding.

4.2.3 Shadow map atlas

Warning: The following feature is experimental and may crash, not work as expected, and may be subject to change.

The new Compositor supports shadow map atlas. Instead of writing one shadow map per RTT, it is now possible to render multiple shadow maps into the same RTT, just different non-overlapping viewport regions.

The first thing to do is to declare a regular shadow map. Then the subsequent shadow maps need to be declared using the "*shadow_atlas*" keyword:

```
compositor_node_shadow myShadowNode
{
	technique pssm
	//Render 1st closest light, splits 0 1 & 2, and 2nd light into myAtlas
	shadow_map myAtlas 2048 2048 PF_FLOAT16_R light 0 split 0 viewport 0 0 
0.5 0.5
	shadow_atlas myAtlas light 0 split 1 viewport 0 0.5 0.5 0.5
	shadow_atlas myAtlas light 0 split 2 viewport 0.5 0 0.5 0.5
	technique focused
	shadow_atlas myAtlas light 1 viewport 0.5 0.5 0.5 0.5
	/* ... */
}
```

Now we just need to declare the render_scene passes. We'll avoid using the convenient shadow_map so that we can perform one full clear instead of 4.

```
	/* ... */
	target myAtlas
	{
		pass clear { colour_value 1 1 1 1 }
		pass render_scene { viewport 0.0 0.0 0.5 0.5 }
		pass render_scene { viewport 0.5 0.0 0.5 0.5 }
		pass render_scene { viewport 0.0 0.5 0.5 0.5 }
		pass render_scene { viewport 0.5 0.5 0.5 0.5 }
	}
	/* ... */
```

The Compositor Node should automatically detect which shadow map you're referring to based on the viewport coordinates. It should throw an error if the viewport settings don't match any of the declared shadow map atlases.

> **Tip:** You may want to explore Ogre's powerful scripting capabilities (i.e. abstract passes) instead of copy pasting each render_scene pass.

On receiver objects, the texture projection matrix should already be scaled to point in the viewport range.

4.2.4 Reuse, recalculate and first

Each PASS_SCENE from regular nodes have three settings:

1. SHADOW_NODE_REUSE
2. SHADOW_NODE_RECALCULATE
3. SHADOW_NODE_FIRST_ONLY

This affect when shadow nodes are executed and how they cache their results. The default value is “SHADOW_NODE_FIRST_ONLY”; in which means Ogre should manage this automatically; however there are times when SHADOW_NODE_REUSE could be useful.

It's easier to explain what they do with examples.

Suppose the user has two render_scene passes, both have the same shadow node associated:

1. One for opaque geometry.
2. Another for transparent geometry,

If using SHADOW_NODE_FIRST_ONLY, when the first pass is executed (opaque geometry), Ogre will first execute the shadow nodes, updating the shadow maps; then render the opaque geometry.

When the second pass is executed (transparent geometry), the shadow node won't be executed as the shadow maps are supposed to be up to date; hence the transparent geometry will reuse the results.

Another example: Suppose the user has three passes:

1. One for opaque geometry.
2. Another for reflections, seen from a different camera.
3. The last pass for transparent geometry, rendered using the same camera as opaque geometry.

If using SHADOW_NODE_FIRST_ONLY; the shadow node will be executed before the opaque geometry pass.

Then the reflections' pass comes. It uses a different camera, which means there could be a different set of lights that will be used for shadow casting (since some techniques set shadow cameras relative to the rendering camera for optimum quality, pssm splits become obsolete, some lights are closer to this camera than they were to the player's camera, etc). ***Ogre has no choice but to recalculate and execute the shadow node again,*** updating the shadow maps.

When the third pass kicks in, the camera has changed again; thus we need to execute the shadow node... again!

Ogre will log a warning when it detects a suboptimal compositor setup such as this one. To be more specific, Ogre detects that the 3rd pass uses the same results as the 1st pass, but the shadow node is being forced to recalculate in the third one, instead of reusing.

There are several ways to solve this problem:

1. **Render reflections first:** This is perhaps the most obvious one. If there are no data dependencies;

first perform the reflection pass, and then the opaque & transparent passes; so the shadow node is executed twice instead of three times.

2. **Use two shadow nodes:** When the first option isn't viable (i.e. there's a data dependency) using two shadow nodes will guarantee the results don't get overwritten. This option needs more VRAM though.
3. **Use SHADOW_NODE_REUSE in the reflection render_scene pass:** This will force Ogre not to execute the shadow node. This assumes you know what you're doing or else you may experience glitches (i.e. pssm splits aren't fully usable from a camera with a different position). This is useful though, if you wish to maintain consistency in the light list being used (since recalculation may cause a different set of lights to be used for shadow maps, since it depends on proximity to the active camera). Another reason to force reusage could be performance: The shadow node is only being executed once.

The setting “SHADOW_NODE_RECALCULATE” forces Ogre to always recalculate. Ogre will not issue a warning if it detects your node setup is suboptimal because of passes using SHADOW_NODE_RECALCULATE.

Forcing recalculation only makes sense when the application makes relevant changes to the camera between passes that Ogre cannot detect (i.e. change the position or the orientation through listeners)

4.2.5 Shadow mapping setup types

Ogre supports 5 depth shadow mapping techniques. Although they're as old as Ogre 1.4 or older, they've never been mentioned in the manual, and the doxygen documentation is quite cryptic, assuming the reader is quite familiar with the original papers. Here each is technique explained.

4.2.5.1 Uniform shadow mapping

The oldest form of shadow mapping, and the most simple one. It's very basic and thus probably glitch-free. However it's quality is very bad, even on high resolutions.

The user needs to call SceneManager::setShadowDirectionalLightExtrusionDistance & SceneManager::getShadowFarDistance let Ogre know how far directional lights should be from camera (since theoretically they're infinitely distant). If the value is too low, some casters won't be included and thus won't cast a shadow. Too high and the quality will quickly degrade.

Most likely only useful for testing that shaders are working correctly, and shadows not showing up correctly is not an Ogre bug or the scene (i.e. casters with infinite aabbs can cause trouble for Focused techniques).

4.2.5.2 Focused

An improved form over uniform shadow mapping. The technique uses the AABB enclosing all casters, an AABB enclosing all receivers visible by the current camera and the camera's frustum to build a hull (which is the intersection of all three, also known as the “intersection body B”). With this hull's information, Focused shadow mapping is able to deduce the optimal extrusion distance (no need to set it like in uniform shadow mapping), and create a much tighter near and far plane, resulting in much superior quality.

SceneManager::getShadowFarDistance is still used, and it can cause major quality improvements, because the camera's frustum used to build the hull is first clipped at the shadow far distance (instead of using the camera's far plane)

Most of the time, this is one of the best choices for general shadow mapping.

4.2.5.3 LispSM

LispSM inherits from Focused shadow camera setup, and hence all that applies to Focused shadow maps applies to LispSM as well.

LispSM goes one step further by modifying the projection matrix, applying some “skewing” to the rendered image. This results in better quality for shadows closer to the camera, but can introduce some artifacts. optimal_adjust_factor can control how strong the skewing is, thus giving a blend between artifacts and quality.

Note: at the time of writing, lispsm technique is broken and has terrible artifacts. This bug seems to affect both Ogre 1.9 & 2.0

4.2.5.4 PSSM / CSM

PSSM stands for Parallel Split Shadow Mapping aka. Cascaded Shadow Maps.

Shadow maps are divided into “cascades” or “splits”; in order to improve quality. So instead of getting one RTT per light, the user gets multiple RTTs per light. Usually the depth in camera space is determining factor to know which cascade/split to use.

Ogre's PSSM inherits from LispSM and hence has separate lispsm settings for each split. Because of the bug that causes LispSM to be filled with strong artifacts, it is advised that users set optimal_adjust_factor for all splits to a negative value, thus causing PSSM to use Focused setups.

There's a lot of resources on internet regarding PSSM / CSM:

- A Sampling of Shadow Techniques
- Cascaded Shadow Maps
- Sample Distribution Shadow Maps
- Parallel-Split Shadow Maps on Programmable GPUs

The original technique was introduced by Fan Zhang, Hanqiu Sun, Leilei Xu & Lee Kit Lun

4.2.5.5 Plane Optimal

TBD

4.2.6 Writing shaders

Writing the necessary shaders to get depth shadow mapping work can be difficult due to the amount of factors that weight in and the flexibility that Ogre offers.

It is often better to use a shader generator or a material system that is less flexible but allows easier setting up of shadow maps, like RTSS or Shiny

That being said, in order to make a shadow mapping shader work, the following checklist will come in handy:

Written receiver's Vertex & Pixel shader to use depth shadow mapping	
Material uses right pair of receiver vertex shader & caster vertex shader	
Caster's vertex shader's math matches the receiver vertex shader's (i.e. skinning, instancing)	
VTF Instancing: The texture_unit is set to use a VTF texture for the caster.	

TBD

4.3 Workspaces

Nodes are useless without setting up a workspace.

A workspace defines what nodes are going to be used and how they're going to be connected. They also need to declare global textures. **Declaration order is very important**.

Nodes are automatically in use when their connection is specified.

```
connect <Node Name 1> [<output ch #>] [<output ch #>]  <Node Name 2> [<input ch #>] [<input ch #>]
```

Connects the Node “Node Name 1” output channels to “Node Name 2” input channels. This implicitly means “Node Name 1” & “Node Name 2” will be used and executed by the workspace (even if they're isolated and never reach the screen)

- <Node Name 1>

  The name of the Node that will be executed before “*Node Name 2”*

- [<output ch #>] [<output ch #>] … [<output ch #>]

  Channel numbers from “*Node Name 1”*s output channels that will be connected to “*Node Name 2*”.

- <Node Name 2>

  The name of the Node that will be executed after “*Node Name 1”*

- [<input ch #>] [<input ch #>] … [<input ch #>]

  Channel numbers from “*Node Name 2”*s inputs channels that will be connected from “*Node Name 1”* bindings.

Examples:

```
//Connect nodeA to nodeB
//A's output channel 0 ==> B's input channel 1
//A's output channel 1 ==> B's input channel 2
//A's output channel 2 ==> B's input channel 0
connect nodeA 0 1 2 nodeB 1 2 0

//Connect nodeA to nodeB
//A's output channel 0 ==> B's input channel 0
//A's output channel 2 ==> B's input channel 1
connect nodeA 0 2 nodeB  0 1

//Connect nodeC to nodeB
//C's output channel 3 ==> B's input channel 1
connect nodeC 3 nodeB 1
```

Not all output channels must be used. Take in mind that if an output is not used at all, it will still take CPU & GPU processing time. MRT (Multiple Render Target) textures are designed to travel through a single channel.

Attention #1!

All nodes must have their input channels connected. If a node has a disconnected input channel, the workspace will fail to initialize and throw a warning.

Attention #2!

Nodes that have no input channels will be the first to be executed, regardless of declaration order (but nodes without input channels declared first should run before nodes declared later with no input channels). Take this in mind if you plan to use global textures as a means of passing information (usually a very bad idea).

connect_output <Node Name> <input channel #>

Connects the final render target (i.e. the RenderWindow) to the specified input channel from the node. Implicitly the node will be used and executed. Only one connect_output is allowed. The final render window is passed in C++ code when initializing the Workspace.

It is possible for a Workspace to not use this variable (though rather pointless)

- <Node Name 1>

  The name of the Node that will receive the final RTT

- <input channel #>

  The number of the input channel from *Node Name 1*.

Example:

```
//Pass the render window to nodeA through channel #0
```

```
connect_output nodeA 0
```

Tip: Only one connect_output is allowed. However, if you need more, you can create a splitter node to redirect the RT to different nodes:

```
// This node is an example of a splitter node, useful for distributing the
// RenderWindow (connect_output) to more than one node
compositor_node RttSplitter
{
	in 0 MyRenderWindow
	out 0 MyRenderWindow
}

workspace MyWorkspace
{
	connect_output RttSplitter 0

	connect RttSplitter nodeA
	connect RttSplitter 0 nodeB 0
	connect RttSplitter 0 nodeC 1
}
```

alias <Node Name> <Aliased Name>

Normally, a Node is always reused. So, if node "*A*" connects to *B* and *C; and D* connects to *A*; it's always the same node A the one we're talking about. The definition is only instantiated once.

However there may be cases where you want to have multiple instances of the same node definition (i.e. because you want unique local textures, or because you want to repeat a process on a different set of nodes), and hence that's what node aliasing does. Once an alias is declared, the node will be instantiated with a different name (its aliased name), and will be possible to make connections with it.

- <Node Name>

  The name of the original instance

- <Aliased Name>

  The alias name to give to this separate instance. The alias must be unique across the workspace, and must also be unique across the names of original node definitions.

Example:

```
workspace MyWorkspace
{
	alias nodeA UniqueNode1		//Instantiate nodeA, calling it UniqueNode1

	connect nodeA		0 UniqueNode1 0
	connect nodeA		0 nodeB 0
	connect UniqueNode1	0 nodeB 1
}
```

4.3.1 Data dependencies between nodes and circular dependencies

The Compostor will solve data dependencies and reorder node execution as necessary. It will also detect some circular dependencies (i.e. node A connecting to A; A connecting to B and B connecting to A) report the error and refuse to initialize the workspace, but it may not detect more complex cases (i.e. node A connecting to B, B to C, C to D, D to B) and attempting execution could result in crashes or graphical glitches.

If you happen to encounter a circular dependency that is not reported by Ogre, we would be intereste in knowing more about it. You can submit your bug report to JIRA

4.4 Setting up code

The Compositor Manager must be created after the RenderSystem has been initialized.

```
if( mRoot->restoreConfig() || mRoot->showConfigDialog() )
{
        // If returned true, user clicked OK so initialise
        // Here we choose to let the system create a default rendering window by passing 'true'
        mWindow = mRoot->initialise(true);

        // Create the compositor after the first RenderWindow has been created (D3D9 complains)
        mRoot->initialiseCompositor();
}
```

4.4.1 Initializing the workspace

To create the workspace, just call the following function with the name of the workspace:

```
CompositorManager2 *compositorManager = mRoot->getCompositorManager2();
compositorManager->addWorkspace( mSceneMgr, mWindow, mCamera, "MyOwnWorkspace", true );
```

You can have more than one Workspace instance of the same Workspace definition. This is mostly useful if you're trying to render to two or more different RTs (i.e. two Render Windows, a RenderWindow and an offscreen RTT, etc) or if you want to use completely different SceneManagers.

4.4.2 Simple bootstrap for beginners

If you're a user that doesn't want to deal with compositor nodes, you're a beginner, or you're in a rush, there is an utility function that will help you set up a basic workspace and a compositor node to render the whole scene:

```
const IdString workspaceName( "MyOwnWorkspace" );
CompositorManager2 *compositorManager = mRoot->getCompositorManager2();
if( !compositorManager->hasWorkspaceDefinition( workspaceName ) )
        compositorManager->createBasicWorkspaceDef( workspaceName, ColourValue( 0.6f, 0.0f, 0.6f ) );
compositorManager->addWorkspace( mSceneMgr, mWindow, mCamera, workspaceName, true );
```

The workspace created by the utility function is equivalent to the following compositor script:

```
compositor_node MyOwnWorkspace_Node
{
        in 0 renderwindow
```

```
		target renderwindow
		{
			pass clear
			{
				colour_value 0.6 0 0.6 1
			}
			pass render_scene
			{
				rq_first  0
				rq_last			max
			}
		}
}

workspace MyOwnWorkspace
{
		connect_output MyOwnWorkspace_Node 0
}
```

4.4.3 Advanced C++ users

Advanced C++ users who want to deal with the CompositorManager2 directly, may find the information in this section useful.

The CompositorManager2 uses a C++ pattern where there is an object Definition and an instance. For example; there is a class called *CompositorPassSceneDef* and a class called *CompositorPassScene*. The former is the definition, while the latter is the instance.

All instances share the same definition and have only read-access to them. Modifying the shared definition while there are instances active is undefined and could happen anything ranging from what the user expected, to glitches, crashes, or memory leaks. Only by analyzing the code it is possible to determine which changes are likely to be “safe” (like changing the visibility mask) and which ones require the instance to be destroyed and recreated.

The syntax of the compositor scripts translate almost 1:1 to definitions, rather than instances. Probably the most notable difference is that *NodeDef*s contain *CompositorTargetDef*, and these contain *CompositorPassDef*; while the instances, Targets and Passes are joined together, thus Nodes contain *CompositorPasses* directly.

Because the CompositorManager2 is still very new, we admit real time changes to nodes (especially channel connections) can be a bit troublesome to deal with unless destroying everything and recreating it, which could be suboptimal for live editing nodes.

We would love to hear your developer feedback on the forums regarding live editing the nodes and further improve the Compositor.

5 INSTANCING

5.1 What is instancing?

Instancing is a rendering technique to draw multiple instances of the same mesh using just one render call. There are two kinds of instancing:

- **Software:** Two larges vertex & index buffers are created and the mesh vertices/indices are duplicated N number of times. When rendering, invisible instances receive a transform matrix filled with 0s. This technique can take a lot of VRAM and has limited culling capabilities.
- **Hardware:** The hardware supports an extra param which allows Ogre to tell the GPU to repeat the drawing of vertices N number of times; thus taking considerably less VRAM. Because N can be controlled at runtime, individual instances can be culled before sending the data to the GPU.

Hardware techniques are almost always superior to Software techniques, but Software are more compatible, where as Hardware techniques require D3D9 or GL3, and is not supported in GLES2

All instancing techniques require shaders. It is not possible to use instancing with FFP (Fixed Function Pipeline)

5.2 Instancing 101

A common question is why should I use instancing. The big reason is performance. There can be 10x improvements or more when used correctly. Here's a guide on when you should use instancing:

1. You have *a lot* of Entities that are repeated and based on the same Mesh (i.e. a rock, a building, a tree, loose leaves, enemies, irrelevant crowds or NPCs)
2. These Entities that repeat a lot also share the same material (or just a few materials, i.e. 3 or 4)
3. Your game is CPU bottleneck.

If these three requirements are all met in your game, chances are instancing is for you. There will be minimal gains when using instancing on an Entity that repeats very little, or if each instance actually has a different material, or it could run even slower if the Entity never repeats.

If your game is not CPU bottleneck'ed (i.e. it's GPU bottleneck'ed) then instancing won't make a noticeable difference.

5.2.1 Instances per batch

As explained in the previous section, instancing groups all instances into one draw call. However this is half the truth. Instancing actually groups a certain number of instances into a batch. One batch = One draw call.

If the technique is using 80 instances per batch; then rendering 160 instances is going to need 2 draw calls (two batches); if there are 180 instances, 3 draw calls will be needed (3 batches).

What is a good value for instances-per-batch setting? That depends on a lot of factors, which you will have to profile. Normally, increasing the number should improve performance because the system is most likely CPU bottleneck. However, past certain number, certain trade offs begin to show up:

- Culling is first performed at batch level, then for HW techniques culling is also done at per instance level. If the batch contains too many instances, its Aabb will grow too large; thus the hierarchy culling will always pass and Ogre won't be able skip entire batches.
- If the instance per batch is at 10.000 and the application created 10.001 instances; a lot of RAM & VRAM will be wasted because it's set for 20.000 instances; HW techniques will spent an excessive amount of CPU time parsing the 9.999 inactive instances; and SW techniques will saturate the Bus bandwidth sending null matrices for the inactive instances to the GPU.

The actual value will depend a lot on the application and whether all instances are often on screen or frustum culled and whether the total number of instances can be known at production time (i.e. environment props). Normally numbers between 80 and 500 work best, but there have been cases where big values like 5.000 actually improved performance.

5.3 Techniques

Ogre supports 4 different instancing techniques. Unfortunately, each of them requires a different vertex shader, since their approaches are different. Also their compatibility and performance varies.

5.3.1 ShaderBased

This is the most compatible technique. It is a Software Instancing technique. World matrices are passed through constant registers, and thus the maximum number of instances per batch is 80; which quickly goes down if the object is skeletally animated. This technique does not play very well with skeletal animation because of that, unless the number of bones is very low (3 or less).

See ***material Examples/Instancing/ShaderBased*** for an example on how to write the vertex shader. Files:

- ShaderInstancing.material
- ShaderInstancing.vert (GLSL)
- ShaderInstancing.cg (Cg, works with HLSL)

5.3.2 VTF (Software)

VTF stands for “Vertex Texture Fetch”. It is a Software Instancing technique. Unlike ShaderBased, world matrices are passed to the vertex shader through a texture. Such feature has only been supported since Vertex Shader 3.0 and is not supported on Radeon X1xxx cards and is quite slow on GeForce 6 & 7. However it's very fast on any modern GPU (GeForce 8, 9, 200, 300, 400, 500, 600, 700; all Radeon HD series, Intel HD 3000 and above)

The advantage of VTF over ShaderBased is that it supports a very high max number of instances per batch; even if it's skeletally animated.

Take note that you will need to set a texture_unit (preferrably the first one, for compatibility) including the shadow caster besides the texture (eg. diffuse, specular, normal maps) so that Ogre gets where to put the vertex texture.

See ***material Examples/Instancing/VTF*** for an example on how to write the vertex shader and setup the material. Files:

- VTFInstancing.material
- VTFInstancing.vert (GLSL)
- VTFInstancing.cg (Cg, also works with HLSL)

5.3.3 HW VTF

This is the same technique as VTF; but implemented through hardware instancing. It is probably one of the best and most flexible techniques.

The vertex shader has to be slightly different from SW VTF version. See ***material Examples/Instancing/HW_VTF*** for an example on how to write the vertex shader and setup the material. Files:

- HW_VTFInstancing.material
- HW_VTFInstancing.vert (GLSL)
- HW_VTFInstancing.cg (Cg, works with HLSL)

5.3.3.1 HW VTF LUT

LUT is a special feature of HW VTF; which stands for **L**ook **U**p **T**able. It has been particularly designed for drawing large animated crowds.

The technique is a trick that works by animating a limited number of instances (i.e. 16 animations) storing them in a look up table in the VTF, and then repeating these animations to all instances uniformly, giving the appearance that all instances are independently animated when seen in large crowds.

See ***material Examples/Instancing/HW_VTF_LUT***. Files:

- Same as HW VTF (different macros defined)

To enable the use of LUT, SceneManager::createInstanceManager's flags must include the flag IM_VTFBONEMATRIXLOOKUP and specify HW VTF as technique.

```
mSceneMgr->createInstanceManager( "InstanceMgr", "MyMesh.mesh",
                                  ResourceGroupManager::AUTODETECT_RESOURCE_GROUP_NAME,
                                  InstanceManager::HWInstancingVTF,
                              numInstancesPerBatch, IM_USEALL|IM_VTFBONEMATRIXLOOKUP );
```

5.3.4 HW Basic

HW Basic is probably the fastest instancing technique[6], but is surely more compatible than HW VTF.

The world matrix data is passed to the vertex shader using three *TEXCOORD*s (*attribute* in GLSL jargon) instead of a vertex texture. The other big difference with HW VTF, besides how data is being passed, is that HW Basic doesn't support skeletal animations at all, making it the preferred choice for rendering inanimate objects like trees, falling leaves, buildings, etc.

See ***material Examples/Instancing/HWBasic*** for an example. Files:

- HWInstancing.material
- HWBasicInstancing.vert (GLSL)
- HWBasicInstancing.cg (Cg, works with HLSL)

5.4 Custom parameters

Some instancing techniques allow passing custom parameters to vertex shaders. For example a custom colour in an RTS game to identify player units; a single value for randomly colourizing vegetation, light parameters for rendering deferred shading's light volumes (diffuse colour, specular colour, etc)

At the time of writing only HW Basic supports passing the custom parameters. All other techniques will ignore it.[7]

To use custom parameters, call InstanceManager::setNumCustomParams to tell the number of custom parameters the user will need. **This number cannot be changed after creating the first batch** (call createInstancedEntity)

Afterwards, it's just a matter of calling InstancedEntity::setCustomParam with the param you wish to send.

For HW Basic techniques, the vertex shader will receive the custom param in an extra *TEXCOORD.*

```
InstanceManager *instanceMgr; //Assumed to be valid ptr
instanceMgr->setNumCustomParams( 2 );

InstancedEntity *instancedEntity = instanceMgr->createInstancedEntity( "myMaterial" );
instancedEntity->setCustomParam( 0, Vector4( 1.0f, 1.0f, 1.2f, 0.0f ) );
instancedEntity->setCustomParam( 1, Vector4( 0.2f, 0.0f, 0.7f, 1.0f ) );
```

5.5 Supporting multiple submeshes

Multiple submeshes means different instance managers, because instancing can only be applied to the

6 Whether it is actually faster than HW VTF depends on the GPU architecture

7 In theory all other techniques could implement custom parameters but for performance reasons only HW VTF is well suited to implement it. Thought yet remains to be seen whether it should be passed to the shader through the VTF, or through additional TEXCOORDs.

same submesh.

Nevertheless, it is actually quite easy to support multiple submeshes. The first step is to create the InstanceManager setting the *subMeshIdx* parameter to the number of submesh you want to use:

```
std::vector<InstanceManager*> instanceManagers;
MeshPtr mesh = MeshManager::getSingleton().load( "myMesh.mesh" );
for( uint16 i=0; i<mesh->getNumSubMeshes(); ++i )
{
	InstanceManager *mgr =
		mSceneMgr->createInstanceManager( "MyManager" + StringConverter::toString( i ),
								"myMesh.mesh",
						ResourceGroupManager::AUTODETECT_RESOURCE_GROUP_NAME,
								InstanceManager::HWInstancingVTF, numInstancePerBatch,
								flags, i );
	instanceManagers.push_back( mgr );
}
```

The second step lies in sharing the transform with one of the submeshes (which will be named 'master'; i.e. the first submesh) to improve performance and reduce RAM consumption when creating the Instanced Entities:

```
SceneNode *sceneNode; //Asumed to be valid ptr
std::vector<InstancedEntity*> instancedEntities;
for( size_t i=0; i<instanceManagers.size(); ++i )
{
	InstancedEntity *ent = instanceManagers[i]->createInstancedEntity( "MyMaterial" );

	if( i != 0 )
		instancedEntities[0]->shareTransformWith( ent );

	sceneNode->attachObject( ent );
	instancedEntities.push_back( ent );
}
```

Note that it is perfectly possible that each InstancedEntity based on a different “submesh” uses a different material. Selecting the same material won't cause the InstanceManagers to get batched together (though the RenderQueue will try to reduce state change reduction, like with any normal Entity).

Because the transform is shared, animating the master InstancedEntity (in this example, instancedEntity[0]) will cause all other slave instances to follow the same animation.

To destroy the instanced entities, use the normal procedure:

```
SceneNode *sceneNode; //Asumed to be valid ptr
std::vector<InstancedEntity*> instancedEntities;
for( size_t i=0; i<instanceManagers.size(); ++i )
{
	instanceManagers[i]->destroyInstancedEntity( instancedEntities[i] );
}
mSceneMgr->getRootSceneNode()->removeAndDestroyChild( sceneNode );
```

5.6 Defragmenting batches

5.6.1 What is batch fragmentation?

There are two kinds of fragmentation:

1. “Deletion” Fragmentation is when many instances have been created, spanning multiple batches; *and many of them got later removed* but they were all from different batches. If there were 10 instances per batch, 100 instances created, then later 90 removed; it is possible that now there are 10 batches with one instance each (which equals 10 drawcalls); instead of being just 1 batch with 10 instances (which equals 1 drawcall).
2. “Culling” Fragmentation is also when many instances of different batches are all sparsed across the whole scene. If they were defragmented, they would've been put together in the same batch (all instances sorted by proximity to each other should be in the same batch) to take advantage of hierachy culling optimizations.

Defragmented batches can dramatically improve performance:

Suppose there 50 instances per batch, and 100 batches total (which means 5000 instanced entities of the same mesh with same material), and they're all moving all the time.

Normally, Ogre first updates all instances' position, then their AABBs; and while at it, computes the AABB for each batch that encloses all of its instances.

When frustum culling, we first cull the batches, then we cull their instances[8] (that are inside those culled batches). **This is the typical hierachial culling optimization**. We then upload the instances transforms to the GPU.

After moving many instances around the whole world, they will make the batch' enclosing aabb bigger and bigger. Eventually, every batch' aabb will be so large, that wherever the camera looks, all 100 batches will end up passing the frustum culling test; thus having to resort to cull all 5000 instances individually.

5.6.2 Prevention: Avoiding fragmentation

If you're creating static objects that won't move (i.e. trees), create them sorted by proximity. This helps both types of fragmentation:

1. When unloading areas (i.e. open world games), these objects will be removed all together, thus whole batches will no longer have active instances.
2. Batches and instances are often assigned by order of creation. Those instances will belong to the same batch and thus maximizing culling efficiency.

8 Only HW instancing techniques cull per instance. SW instancing techniques send all of their instances, zeroing matrices of those instances that are not in the scene.

5.6.3 Cure: Defragmenting on the fly

There are cases where preventing fragmentation, for example units in an RTS game. By design, all units may end up scattering and moving from one extreme of the scene to the other after hours of gameplay; additionally, lots of units may be in an endless loop of creation and destroying, but if the loop for a certain type of unit is broken; it is possible to end up with the kind of “Deletion” Fragmentation too.

For this reason, the function *InstanceManager::defragmentBatches(bool optimizeCulling)* exists.

Using it as simple as calling the function. **The sample *NewInstancing* shows how to do this interactively**. When *optimizeCulling* is true, both types of fragmentation will be attempted to be fixed. When false, only the “deletion” kind of fragmentation will be fixed.

Take in mind that when *optimizeCulling* = true it takes significantly more time depending on the level of fragmentation and could cause framerate spikes, even stalls. Do it sparingly and profile the optimal frequency of calling.

5.7 Troubleshooting

Q: My mesh doesn't show up.

A: Verify you're using the right material, the vertex shader is set correctly, and it matches the instancing technique being used.

Q: My animation plays quite differently than when it is an Entity, or previewed in Ogre Meshy

A: Your rig animation must be using more than one weight per bone. You need to add support for it in the vertex shader, and make sure you didn't create the instance manager with the flags IM_USEONEWEIGHT or IM_FORCEONEWEIGHT.

For example, to modify the HW VTF vertex shader, you need to sample the additional matrices from the VTF:

```
float2 uv0          :      TEXCOORD0;
// Up to four weights per vertex. Don't use this shader on a model with 3 weights per vertex, or 2 or 1
float4 m03_0        :      TEXCOORD1;  //m03.w is always 0
float4 m03_1        :      TEXCOORD2;
float4 m03_2        :      TEXCOORD3;
float4 m03_3        :      TEXCOORD4;
float4 mWeights     :      TEXCOORD5;

float2 mOffset      :      TEXCOORD6;


float3x4 worldMatrix[4];
worldMatrix[0][0] = tex2D( matrixTexture, m03_0.xw + mOffset );
worldMatrix[0][1] = tex2D( matrixTexture, m03_0.yw + mOffset );
worldMatrix[0][2] = tex2D( matrixTexture, m03_0.zw + mOffset );

worldMatrix[1][0] = tex2D( matrixTexture, m03_1.xw + mOffset );
worldMatrix[1][1] = tex2D( matrixTexture, m03_1.yw + mOffset );
worldMatrix[1][2] = tex2D( matrixTexture, m03_1.zw + mOffset );

worldMatrix[2][0] = tex2D( matrixTexture, m03_2.xw + mOffset );
```

```
worldMatrix[2][1] = tex2D( matrixTexture, m03_2.yw + mOffset );
worldMatrix[2][2] = tex2D( matrixTexture, m03_2.zw + mOffset );

worldMatrix[3][0] = tex2D( matrixTexture, m03_3.xw + mOffset );
worldMatrix[3][1] = tex2D( matrixTexture, m03_3.yw + mOffset );
worldMatrix[3][2] = tex2D( matrixTexture, m03_3.zw + mOffset );

float4 worldPos = float4( mul( worldMatrix[0], inPos ).xyz, 1.0f ) *
mWeights.x;
worldPos += float4( mul( worldMatrix[1], inPos ).xyz, 1.0f ) * mWeights.y;
worldPos += float4( mul( worldMatrix[2], inPos ).xyz, 1.0f ) * mWeights.z;
worldPos += float4( mul( worldMatrix[3], inPos ).xyz, 1.0f ) * mWeights.w;

float4 worldNor = float4( mul( worldMatrix[0], inNor ).xyz, 1.0f ) *
mWeights.x;
worldNor += float4( mul( worldMatrix[1], inNor ).xyz, 1.0f ) * mWeights.y;
worldNor += float4( mul( worldMatrix[2], inNor ).xyz, 1.0f ) * mWeights.z;
worldNor += float4( mul( worldMatrix[3], inNor ).xyz, 1.0f ) * mWeights.w;
```

As you can witness, a HW VTF vertex shader with 4 weights per vertex needs a lot of texture fetches. Fortunately they fit the texture cache very well; nonetheless it's something to keep watching out.

Instancing is meant for rendering large number of objects in a scene. If you plan on rendering thousands or tens of thousands of animated objects with 4 weights per vertex, don't expect it to be fast; no matter what technique you use to draw them.

Try convincing the art department to lower the animation quality or just use IM_FORCEONEWEIGHT for Ogre to do the downgrade for you. There are many plugins for popular modeling packages (3DS Max, Maya, Blender) out there that help automatizing this task.

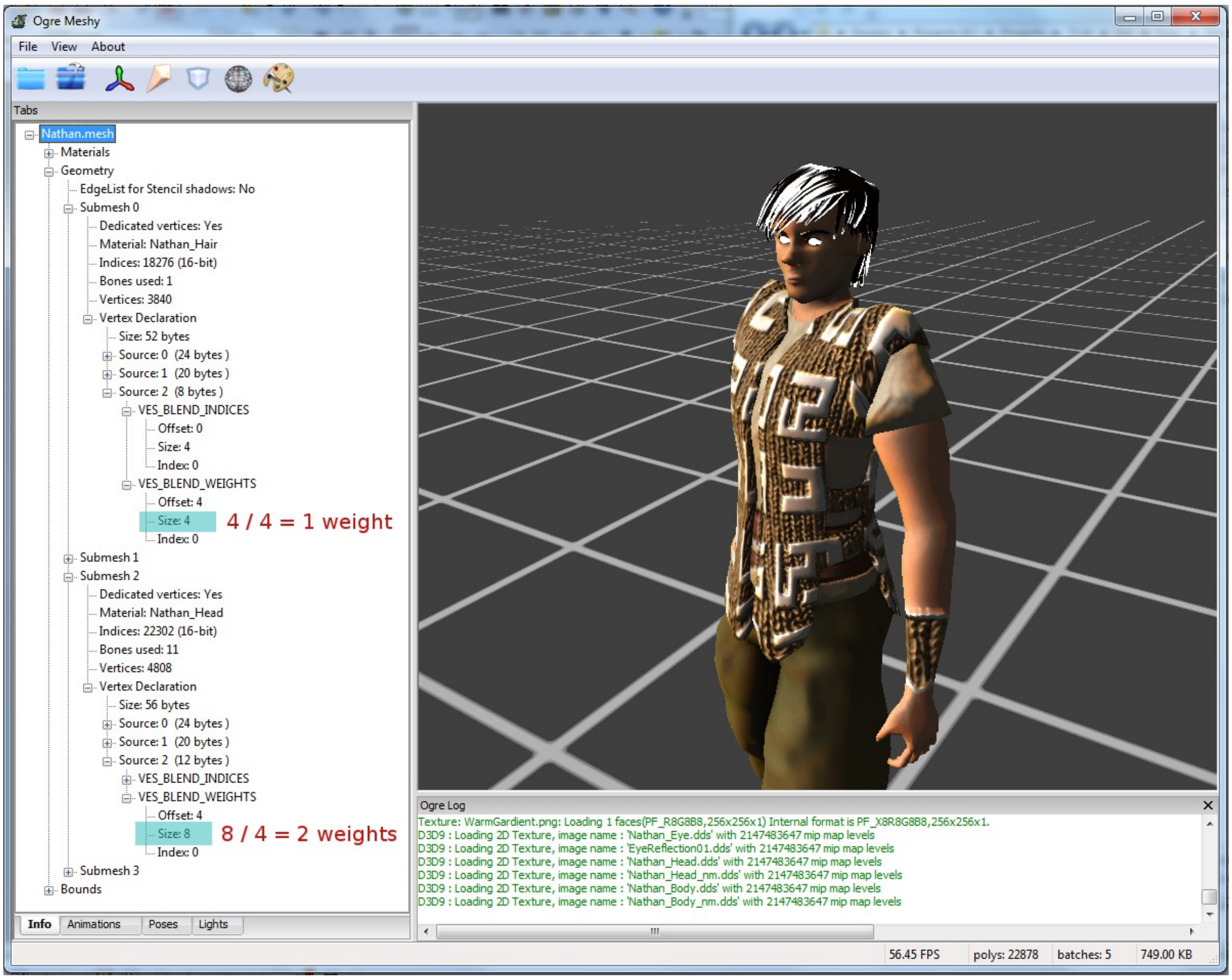
Q: The instance doesn't show up, or when playing animations the mesh deforms very weirdly or other very visible artifacts occur

A: Your rig uses more than one weight per vertex. Either create the instance manager with the flag IM_FORCEONEWEIGHT, or modify the vertex shader to support the **exact** amount of weights per vertex needed (see previous questions).

Q: How do I find how many weights per vertices is using my model?

A: The quickest way is by looking at the size of VES_BLEND_WEIGHTS and divide it by 4[9].

9 One weight is one float. One float is 4 bytes; hence number of weights * 4 is the size of the vertex element.



```
Nathan.mesh
  Materials
  Geometry
    EdgeList for Stencil shadows: No
    Submesh 0
      Dedicated vertices: Yes
      Material: Nathan_Hair
      Indices: 18276 (16-bit)
      Bones used: 1
      Vertices: 3840
      Vertex Declaration
        Size: 52 bytes
        Source: 0  (24 bytes )
        Source: 1  (20 bytes )
        Source: 2  (8 bytes )
          VES_BLEND_INDICES
            Offset: 0
            Size: 4
            Index: 0
          VES_BLEND_WEIGHTS
            Offset: 4
            Size: 4        4 / 4 = 1 weight
            Index: 0
    Submesh 1
    Submesh 2
      Dedicated vertices: Yes
      Material: Nathan_Head
      Indices: 22302 (16-bit)
      Bones used: 11
      Vertices: 4808
      Vertex Declaration
        Size: 56 bytes
        Source: 0  (24 bytes )
        Source: 1  (20 bytes )
        Source: 2  (12 bytes )
          VES_BLEND_INDICES
          VES_BLEND_WEIGHTS
            Offset: 4
            Size: 8        8 / 4 = 2 weights
            Index: 0
    Submesh 3
  Bounds
```

```
Texture: WarmGardient.png: Loading 1 faces(PF_R8G8B8,256x256x1) Internal format is PF_X8R8G8B8,256x256x1.
D3D9 : Loading 2D Texture, image name : 'Nathan_Eye.dds' with 2147483647 mip map levels
D3D9 : Loading 2D Texture, image name : 'EyeReflection01.dds' with 2147483647 mip map levels
D3D9 : Loading 2D Texture, image name : 'Nathan_Head.dds' with 2147483647 mip map levels
D3D9 : Loading 2D Texture, image name : 'Nathan_Head_nm.dds' with 2147483647 mip map levels
D3D9 : Loading 2D Texture, image name : 'Nathan_Body.dds' with 2147483647 mip map levels
D3D9 : Loading 2D Texture, image name : 'Nathan_Body_nm.dds' with 2147483647 mip map levels
```

```
56.45 FPS    polys: 22878    batches: 5    749.00 KB
```

In the picture above, the Ogre Meshy viewer is being used to quickly display the mesh' information. It can be seen that the Hair uses 1 weight per vertex, while the Head needs 2 weights per vertex.

6 THREADING

Ogre 2.0 uses synchronous threading for some of its operations. This means the main thread wakes up the worker threads, and waits for all worker threads to finish. It also means users don't have to be worried that Ogre is using CPU cores while the application is outside a renderOneFrame call.

6.1 Initializing

The number of worker threads must be provided by the user when creating the SceneManager:

```
const size_t numThreads = 4;
InstancingTheadedCullingMethod
            threadedCullingMethod = INSTANCING_CULLING_THREADED;

mSceneMgr = mRoot->createSceneManager(ST_GENERIC, numThreads,
                                      threadedCullingMethod,
                                      "ExampleSMInstance");
```

The other threading parameter besides the number of threads, is the threading strategy used for Instancing, which can be single threaded or multithreaded.

6.1.1 Ideal number of threads

The threading model is synchronous, and meant to be used for tasks that take roughly the same amount of time in each thread (which is a very important assumption!). The ideal number of worker threads is the number of logical cores exposed by the CPU (excluding hyperthreading cores).

Spawning more threads than cores will oversubscribe the system and won't run faster. In fact it should only slow it down.

If you plan to use a whole core for your own computations that will run in parallel while renderOneFrame is working (i.e. one thread for physics) and take a significant cpu time from that core; then in this case the ideal number of threads becomes number_of_logical_cores – 1

Whether increasing the number of threads to include hyperthreading cores improves performance or not remains to be tested.

6.1.2 More info about InstancingThreadedCullingMethod

There are two Instancing techniques that perform culling of their own:

- HW Basic
- HW VTF

Frustum culling is highly parallelizable & scalable. However, we first cull InstanceBatches & regular entities, then ask the culled InstanceBatches to perform their culling to the InstancedEntities they own.

This results performance boost for skipping large amounts of instanced entities when the whole batch isn't visible. However, this also means threading frustum culling of instanced entities got harder.

There were four possible approaches:

- Ask all existing batches to frustum cull. Then use only the ones we want. Sheer brute force. Scales very well with cores, but sacrifices performance unnecessary when only a few batches are visible. This approach is not taken by Ogre.
- Sync every time an InstanceBatchHW or InstanceBatchHW_VTF tries to frustum cull to delegate the job on worker threads. Considering there could be hundreds of InstanceBatches, this would cause a huge amount of thread synchronization overhead & context switches. This approach is not taken by Ogre.
- Each thread after having culled all InstancedBatches & Entities, will parse the culled list to ask all MovableObjects to perform culling of their own. Entities will ignore this call (however they add to a small overhead for traversing them and calling a virtual function) while InstanceBatchHW & InstanceBatchHW_VTF will perform their own culling from within the multiple threads. This approach scales well with cores and only visible batches. However load balancing may be an issue for certain scenes: eg. an InstanceBatch with 5000 InstancedEntities in one thread, while the other three threads get one InstanceBatch each with 50 InstancedEntities. The first thread will have considerably more work to do than the other three. This approach is a good balance when compared to the first two. **This is the approach taken by Ogre when INSTANCING_CULLING_THREADED is on**
- Don't multithread instanced entitites' frustum culling. Only the InstanceBatch & Entity's frustum culling will be threaded. **This is what happens when INSTANCING_CULLING_SINGLE is on**.

Whether INSTANCING_CULLING_THREADED improves or degrades performance depends highly on your scene.

When to use INSTANCING_CULLING_SINGLETHREAD?

If your scene doesn't use HW Basic or HW VTF instancing techniques, or you have very few Instanced entities compared to the amount of regular Entities.

Turning threading on, you'll be wasting your time traversing the list from multiple threads in search of InstanceBatchHW & InstanceBatchHW_VTF

When to use INSTANCING_CULLING_THREADED?

If your scene makes intensive use of HW Basic and/or HW VTF instancing techniques. Note that threaded culling is performed in SCENE_STATIC instances too. The most advantage is seen when the instances per batch is very high and when doing many PASS_SCENE, which require frustum culling multiple times per frame (eg. pssm shadows, multiple light sources with shadows, very advanced compositing, etc)

Note that unlike the number of threads, you can switch between methods at any time at runtime.

6.2 What tasks are threaded in Ogre

The following tasks are partitioned into multiple threads:

- **Frustum culling:** The pool of all visible Entities, InstanceBatches, etc are frustum culled in multiple threads and added to a culled list, one per thread. When all threads are done, the main thread collects the results from all lists.
- **Culling the receiver's box:** Very specific to shadow nodes. When a render_scene pass uses (for example) render queues 4 to 8, but the shadow node users render queues 0 to 8; the shadow node needs receiver's aabb data from RQs 0 to 3; which aren't available. It is very similar to frustum culling; except that the cull list isn't produced, only the aabb is calculated. Since aabb merges are associative: $A \cup B \cup D \cup C = (A \cup B) \cup (D \cup C)$ we can join the results from all threads after they've done. In fact, we even use this associative property to process them using SIMD.
- **Node transform updates:** Updating all scene nodes' derived position and orientation by inheriting from their parent's derived position & orientation. We have to wait for every parent level depth due to data dependencies.
- **Updating all bounds:** Updating the World AABB by applying the node's transform to the local aabbs. The World AABB is needed for correct frustum culling, among other things.
- **Frustum culling instanced entities:** See previous section.

6.3 Using Ogre's threading system for custom tasks

While often users may want to user their own threading system; it is possible to ask Ogre to process their own task using its worker threads. Users need to inherit from *UniformScalableTask* and call SceneManager::executeUserScalableTask.

The following example prints a message to the console from the multipler worker threads:

```
#include <Threading/OgreUniformScalableTask.h>

class MyThreadedTask : public Ogre::UniformScalableTask
{
public:
	virtual void execute( size_t threadId, size_t numThreads )
	{
		printf( "Hello world from thread %i", threadId );
	}
};

int main()
{
	/** assumes Ogre is initialized and sceneMgr is a valid ptr **/
	Ogre::SceneManager *sceneMgr;
```

```
        MyThreadedTask myThreadedTask;
        sceneMgr->executeUserScalableTask( myThreadedTask, true );

        return 0;
}
```

Parameter *threadId* is guaranteed to be in range [0; numThreads) while parameter *numThreads* is the total number of worker threads spawned by that SceneManager.

executeUserScalableTask will block until all threads are done. If you do not wish to block; you can pass false to the second argument and then call *waitForPendingUserScalableTask* to block until done:

```
int main()
{
        /** assumes Ogre is intialized and sceneMgr is a valid ptr **/
        SceneManager *sceneMgr;

        MyThreadedTask myThreadedTask;
        sceneMgr->executeUserScalableTask( myThreadedTask, false );

        doSomeWork();
        printf( "I am going to sleep now" );

        sceneMgr->waitForPendingUserScalableTask();

        printf( "All worker threads finished. Resuming execution of main thread" );

        return 0;
}
```

Attention!

You **must** call *waitForPendingUserScalableTask* after calling *executeUserScalableTask*(myThreadedTask, false) before *executeUserScalableTask* can be called again.

Otherwise deadlocks are bound to happen and Ogre makes no integrity checks. Queuing or scheduling of multiples tasks is not supported. This system is for synchronous multithreading, not for asynchronous tasks.

6.4 Thread safety of SceneNodes

In Ogre 1.x; SceneNodes weren't thread safe at all, not even setPosition or _getDerivedPosition.

In Ogre 2.x, the following **operations are not thread safe**:

- **Creating or destroying nodes/entities**. Don't create a SceneNode while there are scene nodes being used in other threads. It can screw the unique ID and their assignment into the global vector we use to keep track of created nodes. Furthermore The node's memory manager may ran out of memory in its pool and reconstruct the Transform's SoA pointers (it's similar to how std::vector invalidates all iterators when resizing). If that happens, all SceneNodes will be in an inconsistent state. Inversely, if too many nodes have been removed, the manager may decide it's time for a cleanup, in which case many SceneNodes can become in an inconsistent state until the cleanup finishes. How large the pool reserve is can be tweaked, and how often the manager performs can also be tweaked (NodeMemoryManager), though. If the user knows what he's doing the race condition might be possible to avoid. Note other SceneManager implementations may have to fulfill their own needs and introduce race conditions of their own we can't predict.

- **Changing parent / child relationships**. Attaching/detaching a node from another one causes its SoA memory to migrate to a different node memory manager, which can trigger a cleanup and/or one of the managers rans out of memory and has to reconstruct.
- **Calling _getDerivedPositionUpdated & Co (all functions that end in “Updated”)**. These functions will update the derived transforms all way up to the ultimate parent (i.e. root). However in SIMD builds, these updates are performed on 4 nodes at a time (the actual number is not 4, but rather depends on ARRAY_PACKED_REALS). Calling this function could only be thread safe if all all four nodes are in the same thread AND their parents are also on the same thread (parents may not share the same block, thus worst case scenario 4 * 4 = 16 parent nodes have to be in the same thread, not to mention their parents too 4 * 4 * 4 = 64) AND the children of these parents are not calling _getDerivedPositionUpdated too from a different thread.

The following operations are thread-safe:

- **Calling getPosition & Co (getOrientation, getScale, getInheritOrientation, etc), _getDerivedPosition & Co (_getDerivedOrientation, etc)** unless you're calling getPosition and setPosition to the same Node from different threads.
- **Calling setPosition, setOrientation, setScale**. Note on SIMD builds, 4 Nodes can easily share the same 64 byte line, thus it is advisable that all 4 Nodes to be sent to the same thread to reduce the number of false cache sharing performance hits. Calling setPosition to the same Node from different threads is not supported.
- **Calling _setDerivedPosition, _setDerivedOrientation, _setDerivedScale** (which assumes the derived transforms are up to date)

With Ogre 2.0; it is now possible to transfer the position & orientation from a physics engine to Ogre Scene Nodes using a parallel for. Ogre 1.x limitations forced this update to be done in a single thread.

7 PERFORMANCE HINTS

DO:

- ✔ **Sort your creation order of Entities by frequency of creation & destruction (LIFO):** Entities that will be created & destroyed often should be done last. Entities that will never be destroyed should be created first, followed by Entities that will rarely be removed/destroyed.
- ✔ **Use Instancing whenever possible:** InstancedObjects are lightweight, have less RenderQueue and API draw call overhead
- ✔ **Reduce the number of bones in a rig to the minimum necessary.** Specially if there are going to be many copies of it
- ✔ When destroying an entire level or exiting the game, it may be advisable to disable cleanups. Unless your destroy procedure is respecting LIFO order, multiple unnecessary cleanups may be triggered, slowing the shutdown routine. If a level is being destroyed but the game is still running, re-enable cleanups and perform one explicitly after having destroyed all objects involved.

DON'T:

- ✗ Don't update objects created as SCENE_STATIC very often. Updating just one SceneNode or Entity will force to update many of them, possibly all; thus nullifying the performance benefits from SCENE_STATIC .
- ✗ Don't create objects that never move, rotate or scale as SCENE_DYNAMIC
- ✗ Don't create a Shadow Node that uses Render Queues that any of the previously executed render_scenes didn't use. eg. Setting the shadow node's pass to rq_first 0 & rq_last 5; while the render_scene of the regular pass is just rq_first 1 rq_last 4
- ✗ Don't specify more threads than the number of available cores. If your code fully uses (i.e.) 2 cores all for yourself, then substract 2 to the number of threads (don't oversubscribe).[10]

10 Note: The minimum number of threads is 1. One CPU = One Thread. The SceneManager delegates its work on worker threads and goes to sleep.