MANUAL DE OGRE3D

Alumno: Velázquez Muñoz, Mario

Profesor: Juan Peralta Donate

Universidad Carlos III de Madrid

TABLA DE CONTENIDO

TABLA DE CÓDIGO

TABLA DE ILUSTRACIONES

0. INFORMACIÓN SOBRE EL DOCUMENTO

Este documento es una traducción del documento de manual de Ogre correspondiente a la versión 1.7.0 del proyecto. El documento original y actualizado se puede encotrar en la página oficial de Ogre (http://www.ogre3d.org/docs/manual/).

Hay que tener en cuenta los posibles cambios de funcionalidad entre versiones de Ogre, por lo que se recomienda revisar el manual original en caso de haberse producido algún cambio de versión de Ogre.

En algunas partes del documento se hace referencia a demos o códigos de ejemplos, que no vienen incluidos en esta traducción y que se pueden localizar en la página oficial de Ogre, ya sea en la Wiki o en los apartados de demos (www.ogre3d.org).

1.INTRODUCCIÓN

Este capítulo pretende dar una visión general de los componentes principales de Ogre y el por qué de su estructura.

1.1 ORIENTACIÓN A OBJETOS. MÁS QUE UN RUMOR

El nombre (Ogre) lo delata. Es Object-Oriented Graphics Rendering Engine (Motor de renderización de gráficos orientado a objetos), y esto es exactamente lo que es. Pero, ¿por qué esta decisión?

Bien, actualmente los motores gráficos son como cualquier otro gran sistema software. Comienzan siendo pequeños, pero pronto se hinchan en complejas y monstruosas bestias que no se pueden comprender. Es difícil manejar sistemas de este tamaño y mucho más complicado hacer cambios formalmente, y esto es muy importante en un campo en el que las nuevas técnicas y enfoques parecen aparecer cada semana. Diseñar sistemas alrededor de archivos gigantes llenos de llamadas a funciones de C provoca que en un momento se deba de parar, incluso si se ha creado por una sola persona, se hará muy difícil localizar partes de código y mucho más difícil trabajar con ello y fijarlo.

La orientación a objetos es un enfoque muy popular direccionado para resolver estos problemas. Intensifica la descomposición del código en funciones separadas, agrupa funciones y datos en clases que son diseñadas para representar conceptos reales. Permite evitar la complejidad, incluyendo paquetes con un interfaz simple de conceptos, para permitir de forma sencilla el reconocimiento de los mismos y tener una sensación de 'bloques de construcción' que pueden unirse y usarse posteriormente. Se pueden organizar estos bloques de forma que algunos de ellos parezcan lo mismo desde fuera, pero que tienen diferentes formas de alcanzar sus objetivos, de nuevo reduciendo la complejidad para los desarrolladores, ya que ellos sólo tienen que aprenderse el interfaz.

Se creía que la programación orientada a objetos con C++ para una herramienta de gráficos 3D en tiempo real, haría los programas poco eficientes, pero se ha demostrado que no.

Los beneficios que aporta la programación orientada a objetos para Ogre son:

Abstracción

Normalmente los interfaces omiten las pequeñas diferencias entre las implementaciones de una API 3D y un sistema operativo.

Encapsulación

Hay un montón de controladores de estado y acciones de especificación de contexto para realizar en un motor gráfico. La encapsulación permite poner el código y los datos cercanos, permitiendo hacer el código claro y sencillo de comprender, y más fiable, ya que la duplicación está permitida.

Polimorfismo

La posibilidad de que los métodos cambien dependiendo del tipo de objeto que se está usando, incluso si sólo se conoce un interfaz.

1.2 MULTI-TODO

El objetivo no es hacer un motor 3D que use un API, o corra en una plataforma, con un tipo de escena (los niveles interiores son los más populares). Con Ogre se puede ampliar cualquier tipo de escenario, cualquier plataforma y cualquier API 3D.

Por lo tanto, todas las partes 'visibles' de Ogre son completamente independientes de la plataforma, API 3D y tipo de escenario. No hay dependencias en tipos Windows, no se asume nada sobre el tipo de escenario que se está creando, y los principales aspectos 3D están basados en el núcleo de textos matemáticos, mejor que en una implementación particular de una API.

Ahora, por supuesto, Ogre tiene que llegar hasta el meollo de la cuestión de los detalles de la plataforma, la API y el escenario, pero lo hace en las subclases especialmente diseñadas para el entorno en cuestión, pero que aún exponen la misma interfaz que las versiones abstractas.

Por ejemplo, hay una clase 'Win32Window' que maneja todos los detalles sobre la prestación de Windows en una plataforma Win32 – Sin embargo, el diseñador de la aplicación sólo tiene que manipularla a través de la interfaz de la superclase 'RenderWindow', que será el mismo en todas las plataformas.

Asimismo, la clase 'SceneManager' se ocupa de la disposición de objetos en el escenario y su secuencia de representación. Las aplicaciones sólo tienen que utilizar esta interfaz, pero hay una clase 'BspSceneManager', que optimiza el manejo del escenario para los niveles interiores, lo que significa que obtiene gran rendimiento y un interfaz fácil de aprender. Lo que todas las aplicaciones tienen que hacer es indicar el tipo de escena que van a crear y dejar a Ogre elegir la aplicación más apropiada.

La naturaleza orientada a objetos de Ogre hace que todo esto sea posible. Actualmente Ogre se ejecuta en Windows, Linux y Mac OSX usando plugins para impulsar la prestación de APIs de renderizado (en la actualidad Direct3D u OpenGL). Las aplicaciones utilizan Ogre en el nivel abstracto, lo que garantiza que operan de forma automática en todas las plataformas y renderizan subsistemas que Ogre ofrece sin necesidad de plataforma o código API específico.

2. LOS OBJETOS PRINCIPALES

INTRODUCCIÓN

Este apartado muestra un resumen de los objetos principales que se usarán en Ogre y para qué se utilizan.

UN MUNDO DE ESPACIO DE NOMBRES

Ogre usa una propiedad de C++ llamada Espacio de Nombres (Namespaces). Esto permite poner clases, enumeraciones, estructuras, cualquier cosa; dentro del ámbito de un espacio de nombres, evitando de una forma fácil colisión por nombres, es decir, situaciones en las que 2 cosas se llaman de la misma forma. Como Ogre está diseñado para ser utilizado dentro de otras aplicaciones, así se está seguro de que la colisión de nombres no será un problema. Algunas personas establecen prefijos a unas clases/tipos con un pequeño código porque algunos compiladores no soportan espacios de nombres, pero con los compiladores de C++ no ocurrirá este problema. Se recomienda no utilizar compiladores que no soporten espacios de nombres, para que no se puedan dar estos problemas.

Esto significa que cada clase, tipo, etc, debe ser prefijado con 'Ogre::', por ejemplo, 'Ogre::Camera', 'Ogre::Vector3', etc,que significa que si en otras partes de la aplicación se ha utilizado un tipo de Vector3 no se produzcan enfrentamientos de colisión por nombre. Para evitar teclear un montón de código extra, se puede agregar una declaración'using namespace Ogre;' que significa quese usa este espacio de nombres y no será necesario escribir el prefijo 'Ogre::' a menos que haya ambigüedad (en la situación en la que se tiene otra definición con el mismo nombre).

VISTA LEJANA (DESDE 10000 PIES)

Abajo se muestra un diagrama de algunos de los objetos principales y dónde se 'asientan' en el gran escenario de elementos. No se trata de todas las clases de Ogre, son sólo unos pocos ejemplos de las más significativas, para dar una idea de cómo están asociadas unas con otras.

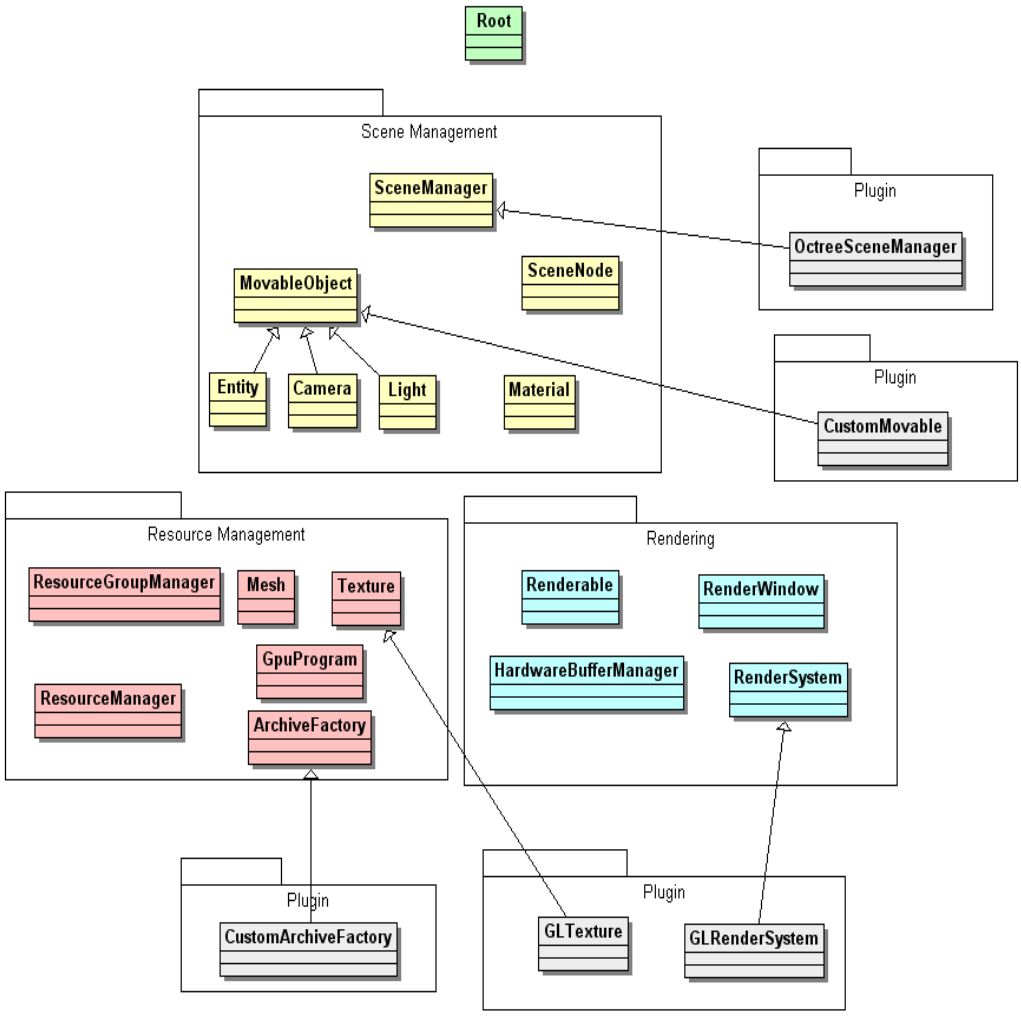

Ilustración 1: Esquema de componentes general de Ogre.

En la parte superior del diagrama está el objeto raíz (Root). Esta es su 'forma' en el sistema de Ogre, y es donde se tienden a crear los objetos de nivel superior que se tienen que abordar, como los directores de escena (scene managers), sistemas de renderizado y ventanas de renderización, cargar plugins, y todas las cosas fundamentales. Si no se sabe por dónde empezar, la raíz (Root) sirve para casi todo, aunque a menudo sólo dará otro objeto que realmente hará el detalle del trabajo, ya que Root en sí es más que un organizador y el objeto facilitador.

La mayoría del resto de las clases de Ogre entran en una de las 3 siguientes funciones:

Gestión de escenario (Scene Management)

Da información sobre el contenido del escenario, cómo está estructurado, cómo se ve desde las cámaras, etc. Los objetos de este área son responsables de ofrecer una interfaz declarativa natural para el mundo que se está construyendo, es decir, no se dice a Ogre 'establece estos estados y luego renderiza 3 polígonos', sin que se le dice 'Quiero un objeto aquí, aquí y aquí, con estos materiales, representados a partir de este punto de vista', y se deja a Ogre manos a la obra.

Gestión de Recursos (Resource Management)

Todo el renderizado necesita recursos, se trata la geometría, las texturas, las fuentes, lo que sea. Es importante gestionar la carga, la reutilización y la descarga de estos elementos con cuidado, eso es lo que las clases en este área realizan.

Renderización (Rendering)

Por último, se ponen las imágenes en la pantalla - se trata del nivel más bajo del canal de renderizado, el sistema específico de representación de objetos de la API, como buffers, renderiza los estados y el producto y empuja hacia abajo todo el canal. Las clases del subsistema de manejo de la escena usan este para conseguir su mayor nivel de información sobre la escena en la pantalla.

Hay un gran número de plugins esparcidos por todo el diagrama. Ogre está diseñado para ser ampliado, y los plugins son la forma habitual de hacerlo. Muchas de las clases en Ogre pueden ser heredadas y ampliadas, se puede por ejemplo cambiar la organización del escenario a través de un SceneManager (Gestor de Escenario) personalizado y añadir un nuevo sistema de renderización de la aplicación (por ejemplo, Direct3D u OpenGL), o proporcionar una forma de cargar los recursos de otra fuente (por ejemplo desde una ubicación web o una base de datos). Una vez más esto es sólo un puñado pequeño de cosas que se pueden hacer utilizando plugins, pero como se puede ver, se puede conectar casi cualquier aspecto del sistema. De esta manera, Ogre no es sólo una solución para un problema estrictamente definido, si no que puede extenderse a casi cualquier cosa que se necesite hacer.

2.1 EL OBJETO RAÍZ (ROOT)

Es el punto de entrada al sistema Ogre. Este objeto debe ser el primero en ser creado, y el último en ser destruido. En las aplicaciones de ejemplo se eligió hacer una instancia de root en el objeto inicial de la aplicación, que asegura que sea creado tan pronto como el objeto de la aplicación se crea, y se elimina cuando se elimina el objeto de la aplicación.

El objeto de raíz permite configurar el sistema, por ejemplo a través de showConfigDialog(), que es un método muy práctico que realiza toda la detección de opciones de renderizado del sistema y muestra un diálogo al usuario para personalizar la resolución, profundidad de color, las opciones de pantalla completa, etc. También establece las opciones que el usuario selecciona para inicializar el sistema directamente después.

El objeto raíz es también un método para la obtención de punteros a otros objetos del sistema, tales como el SceneManager, RenderSystem y varios administradores de recursos. Véase más abajo para más detalles.

Por último, si se ejecuta Ogre en modo de representación permanente, es decir, si se desea refrescar siempre todos los objetos de representación lo más rápido posible (norma para los juegos y demos, pero no para otras utilidades), el objeto raíz tiene un método llamado startRendering, que cuando se le llama entra en un bucle continuo de prestación que sólo terminará cuando todas las ventanas de representación sean cerradas, o cualquier objeto FrameListener indique que quiere detener el ciclo (ver más abajo para más detalles de los objetos FrameListener).

2.2. EL OBJETO RENDERSYSTEM

El objeto RenderSystem es en realidad una clase abstracta que define la interfaz de la subyacente API 3D. Es responsable del envío de las operaciones de renderizado a la API y del establecimiento de todas las diferentes opciones de renderizado. Esta clase es abstracta porque toda la implementación es

específica de representación de la API - hay subclases de la API específicas para cada representación de la API (por ejemplo, D3DRenderSystem para Direct3D). Después de que el sistema ha sido inicializado a través de Root::initialise, el objeto RenderSystem para el renderizado de la API seleccionada está disponible a través del método Root::getRenderSystem().

Sin embargo, una aplicación típica normalmente no debería tener que manipular el objeto RenderSystem directamente - todo lo necesario para la representación de los objetos y los ajustes de personalización deberían estar disponibles en el SceneManager, Material y otras clases orientadas al escenario. Sólo si se desea crear ventanas de representación múltiple (ventanas completamente independientes, sin visores varios como un efecto de pantalla dividida que se realiza a través de la clase RenderWindow) o acceder a otras funciones avanzadas se necesita el acceso al objeto RenderSystem.

Se puede asumir que el SceneManager maneja las llamadas al RenderSystem en el momento adecuado.

2.3. EL OBJETO SCENEMANAGER

Aparte del objeto raíz (Root), esta es probablemente la parte más crítica del sistema desde el punto de vista de aplicación. Ciertamente, será el objeto más utilizado por la aplicación. El SceneManager está a cargo de los contenidos de la escena que se renderizan por el motor gráfico. Es responsable de organizar el contenido usando cualquier técnica que considere más favorable, para crear y gestionar todas las cámaras, los objetos móviles (entidades), las luces y los materiales (propiedades de la superficie de los objetos), y para la gestión de la 'geometría del mundo', que es la geometría estática extendida que normalmente se usa para representar las partes inmóviles de una escena.

Se acude al SceneManager cuando se quiere crear una cámara para el escenario. Es también donde se va para recuperar o eliminar una luz de la escena. No es necesario para la aplicación mantener las listas de objetos, el SceneManager mantiene un conjunto nombrado de todos los objetos de la escena que se pueden acceder, en caso de que se necesite. Se puede comprobar en esta documentación principal los métodos getCamera, getLight, getEntity, etc.

El SceneManager también envía a la escena el objeto RenderSystem cuando es el momento de renderizar la escena. Nunca se tendrá que llamar al método SceneManager::_renderScene directamente, sin embargo - se le llama automáticamente cada vez que a un objeto de representación se le pide que se actualice.

La mayoría de la interacción con el SceneManager se da durante la configuración del escenario. Es probable llamar a un gran número de métodos (tal vez impulsado por un archivo de entrada que contiene los datos de la escena), a fin de crear la escena. También se puede modificar el contenido de la escena de forma dinámica durante el ciclo de renderizado si se crean sus objetos FrameListener propios (véase más adelante).

Dado que los tipos de escena diferentes requieren enfoques algorítmicos muy diferentes a la hora de decidir qué objetos son enviados al RenderSystem con el fin de alcanzar el rendimiento máximo, se ha diseñado la clase SceneManager para que sea heredada en subclases para cada uno de los tipos de escena diferentes. El objeto SceneManager por defecto renderiza una escena, pero tiene poca o ninguna organización de la escena y no se debe esperar que los resultados sean de alto rendimiento en el caso de las grandes escenas. La intención es que se creen especialidades para cada tipo de escena de tal manera que optimice la organización de la escena para un mejor rendimiento, asumiendo que se puede hacer para ese tipo de escena. Un ejemplo es el BspSceneManager que optimiza la prestación de los grandes niveles de interior basado en una partición binaria del Espacio (BSP) de árbol.

La aplicación que utiliza Ogre no tiene que saber qué subclases están disponibles. La aplicación simplemente llama Root::createSceneManager(..) pasando como un parámetro un número de tipos de escena (por ejemplo, ST_GENERIC, ST_INTERIOR, etc.). Ogre usará automáticamente la mejor subclase de SceneManager disponible para ese tipo de escena, o por defecto la SceneManager de base si no hay una especial disponible. Esto permite a los desarrolladores de Ogre añadir nuevas especialidades de escena más tarde y optimizar así los tipos anteriormente no optimizados de escena, sin que las aplicaciones de usuario cambien su código.

2.4 EL OBJETO RESOURCEGROUPMANAGER

La clase ResourceGroupManager es en realidad un 'concentrador'de carga de recursos reutilizables, como las texturas y mallas. Es el lugar en el que se definen los grupos de recursos, que pueden ser cargados y descargados cuando se quiera. Sirviendo los recursos hay una serie de ResourceManagers que gestionan los distintos tipos de recursos, como TextureManager o MeshManager. En este contexto, los recursos son conjuntos de datos que deben ser cargados desde algún lugar para proporcionar a Ogre los datos que necesita.

Los ResourceManagers garantizan que los recursos sólo se cargan una vez y son compartidos en el motor de Ogre. Asimismo, gestionan los requisitos de memoria de los recursos a tratar. También pueden buscar en un número de ubicaciones los recursos que necesitan, incluyendo las rutas de búsqueda múltiples y archivos comprimidos (ficheros ZIP).

La mayoría de las veces no se va a interactuar con los gestores de recursos directamente. Los administradores de recursos son llamados en otras partes del sistema Ogre cuando sea necesario, por ejemplo cuando se solicita una textura que se añade a un material, el TextureManager será llamado.Se puede llamar al administrador de recursos adecuado directamente para pre-cargar los recursos (por ejemplo, si se desea evitar el acceso a disco más adelante), pero la mayoría del tiempo se puede dejar a Ogre decidir cuándo hacerlo.

Una cosa que se querría hacer es decir a los administradores de recursos dónde buscar los recursos. Para ello, se llama a través de Root::getSingleton().addResourceLocation, que en realidad pasa la información al ResourceGroupManager.

Debido a que hay sólo 1 instancia de cada administrador de recursos en el motor, si se quiere obtener una referencia a un administrador de recursos se utiliza la siguiente sintaxis:

```
TextureManager::getSingleton().someMethod()
MeshManager::getSingleton().someMethod()
```

Código 1: Instancias generales.

2.5 EL OBJETO MESH

Un objeto de malla (Mesh) representa un modelo discreto, un conjunto de geometría que es autónomo y por lo general bastante pequeño, a escala mundial. Los objetos de malla se supone que representan los objetos móviles y no se utilizan para geometrías extensas que se suelen utilizar para crear fondos.

Los objetos de malla son un tipo de recurso, y son gestionados por el administrador de recursos MeshManager. Normalmente se cargan desde archivos propios de Ogre con formato '.mesh'. Los archivos de malla se crean normalmente con la exportación desde una herramienta de modelización. Véase la sección 4.1 Exportadores y se puede manipular a través de 4. Herramientas de Malla.

También se pueden crear objetos de malla manualmente llamando al método MeshManager::createManual. De esta forma se puede definir la geometría del mismo, pero esto queda fuera del alcance de este manual.

Los objetos de malla son la base de los distintos objetos móviles en el mundo, que se llaman 2.6 Entidades.

Los objetos de malla también pueden ser animados usando animación del esqueleto. Véase la sección 8.1 Animación Esquelética.

2.6 ENTIDADES

Una entidad es una instancia de un objeto móvil en la escena. Podría ser un coche, una persona, un perro, una estrella, lo que sea. La única hipótesis es que no necesariamente tiene una posición fija en el mundo.

Las entidades se basan en las mallas discretas, es decir, las colecciones de geometrías, que son autónomas y por lo general bastante pequeñas, a escala mundial, que son representados por el objeto de malla (Mesh). Varias entidades pueden basarse en la misma malla, ya que a menudo se desea crear varias copias del mismo tipo de objeto en una escena.

Se crea una entidad llamando al método SceneManager::createEntity, dándole un nombre y especificando el nombre del objeto de malla en que se basará (por ejemplo 'muscleboundhero.mesh'). El SceneManager se asegurará de que la malla se cargue mediante una llamada al administrador de recursos MeshManager. Sólo se cargará una copia de la malla.

Las entidades no se consideran parte de la escena hasta que se adjuntan a un SceneNode (véase la sección de abajo). Adjuntando las entidades a SceneNodes, se pueden crear complejas relaciones jerárquicas entre las posiciones y orientaciones de las entidades. A continuación se modifican las posiciones de los nodos que indirectamente afectan a las posiciones de las entidades.

Cuando se carga una malla, automáticamente viene con una serie de materiales definidos. Es posible tener más de un material adjunto a una malla - las diferentes partes de la malla pueden utilizar diferentes materiales. Cualquier entidad creada a partir de la malla usará de forma automática los materiales por defecto. Sin embargo, se puede cambiar esto en función de cada entidad, de forma que se puede crear una serie de entidades basadas en la misma malla, pero con diferentes texturas, etc.

Para entender cómo funciona esto, se debe saber que todos los objetos de malla se componen en realidad de objetos SubMesh, cada uno de los cuales representa una parte de la malla, y usa un material. Si se utiliza una malla de un material, sólo tendrá una SubMesh.

Cuando una entidad se crea en base a una malla, se compone de (posiblemente) objetos subEntity, cada uno se complementa con cada uno de los objetos SubMesh de la malla original. Se puede acceder a los objetos subEntity utilizando el método Entity::getSubEntity. Una vez que se tiene una referencia a una sub entidad, se puede cambiar el material que utiliza llamando al método setMaterialName. De esta manera se puede hacer una entidad desviada de los materiales por defecto y así crear una versión de aspecto individual de la misma.

2.7 MATERIALES

El objeto material (Material) controla cómo se renderizan los objetos en la escena. Especifica qué propiedades de superficie base tienen los objetos como la reflectancia de los colores, brillo, cómo están

presentes las capas de textura, qué imágenes son de ellos y la forma en que se mezclan, qué efectos especiales se aplican según el medio, modo de selección que utiliza, cómo se filtran las texturas, etc.

Los materiales o bien se pueden configurar mediante programación, llamando SceneManager:: createMaterial y ajustar la configuración, o se pueden especificar en un 'script' que se carga en tiempo de ejecución. Véase la sección 3.1 Scripts de Material para más información.

Básicamente todo lo relacionado con la apariencia de un objeto, aparte de la forma es controlada por la clase Material.

La clase SceneManager gestiona la lista maestra de los materiales disponibles para la escena. La lista se puede agregar a la aplicación llamando al método SceneManager::createMaterial, o por la carga de una malla (Mesh) (que a su vez carga las propiedades del material). Cada vez que los materiales se añaden a la SceneManager, comienzan con un conjunto predeterminado de propiedades, las cuales son definidas por Ogre como las siguientes:

- Ambient reflectance (Reflectancia ambiente) = ColourValue::White(completa)
- Diffuse reflectance (Reflectancia difusa) = ColourValue::White (completa)
- Specular reflectance (Reflectancia especular) = ColourValue::Black (nada)
- Emmissive (Emisiva) = ColourValue::Black (nada)
- Shininess (Brillo) = 0 (no brillante)
- No texture layers (Sin capas de textura) (y por lo tanto sin texturas)
- SourceBlendFactor = SBF_ONE, DestBlendFactor = SBF_ZERO (opaco)
- Depth buffer cheking (Control de buffer de profundidad) on
- Depth buffer writing (Escritura en buffer de profundidad) on
- Depth buffer comparison function (Función de comparación de buffer de profundidad) = CMPF_LESS_EQUAL
- Culling mode (Modo selectivo) = CULL_CLOCKWISE
- Ambient lighting in scene (iluminación ambiental en la escena) ColourValue = (0,5, 0,5, 0,5) (gris medio)
- Dynamic lightting enabled (Iluminación dinámica)
- Gourad shading mode (Modo de sombreado Gourad)
- Solid polygon mode (Modo de polígonos sólidos)
- Bilinear texture filtering (Filtrado de Textura bilineal)

Se puede modificar esta configuración llamando a SceneManager::getDefaultMaterialSettings() y hacer los cambios necesarios al material que se devuelve.

Las entidades tienen automáticamente material asociado a ellas si utilizan un objeto de malla, ya que el objeto de malla normalmente establece sus materiales requeridos en el momento de la carga. También se puede personalizar el material utilizado por una entidad tal como se describe en 2.6 Entidades. Basta con crear un nuevo material, configurarlo (se puede copiar la configuración de otro material) y apuntar las entradas subEntity a él, utilizando subEntity::setMaterialName().

2.8. OVERLAYS

Los Overlays (recubrimientos) permiten renderizar elementos 2D y 3D en la parte superior del contenido normal de la escena para crear efectos como heads-up Displays (HUD), sistemas de menú, los paneles de estado, etc. El marco de estadísticas, el cual viene de serie con Ogre es un ejemplo de una superposición. Los Overlays pueden contener elementos 2D o 3D. Los elementos 2D se utilizan para el HUD, y los elementos 3D se pueden utilizar para crear cabinas o cualquier otro objeto 3D que se desea que se presente en la parte superior del resto de la escena.

Se pueden crear superposiciones ya sea a través del método SceneManager::createOverlay, o se puede definir en una secuencia de comandos en un script .overlay. En realidad, este último es probable que sea el más práctico, porque es más fácil de modificar (sin necesidad de recompilar el código). Se pueden definir tantas superposiciones como se quiera: todas ellas empiezan la vida ocultas, y se muestran llamando al método 'show()'. También se pueden mostrar varias superposiciones de una vez, y su orden Z es determinado por el método Overlay::setZOrder().

CREACIÓN DE ELEMENTOS 2D

La clase abstracta OverlayElement maneja los detalles de elementos 2D que son añadidos a las superposiciones. Todos los elementos que se pueden agregar a los overlays derivan de esta clase. Es posible (y se recomienda) para los usuarios de Ogre definir sus propias subclases personalizadas de OverlayElement con el fin de proporcionar sus propios controles de usuario. Las principales características comunes de todos los OverlayElements son cosas como el tamaño, posición, nombre básico del material, etc. Las subclases extienden este comportamiento a fin de incluir propiedades y comportamientos más complejos.

Una subclase importante de OverlayElement es OverlayContainer. OverlayContainer es lo mismo que un OverlayElement, excepto que puede contener otros OverlayElements, que los reúne (lo que les permite moverse juntos por ejemplo) y les proporciona un local de origen de coordenadas para facilitar la alineación.

La tercera clase importante es OverlayManager. Siempre que una aplicación desee crear un elemento 2D para añadir una superposición (o un contenedor), se debe llamar a OverlayManager:: createOverlayElement. El tipo de elemento que se desea crear es identificado por una cadena, la razón es que permite a los plugins registrar nuevos tipos de OverlayElement para que se puedan crear sin necesidad de vincular específicamente a las bibliotecas. Por ejemplo, para crear un panel (un área rectangular de fricción que puede contener otros OverlayElements) podría llamarse a OverlayManager:: getSingleton ().CreateOverlayElement ("Panel", "myNewPanel");

AÑADIENDO ELEMENTOS 2D AL RECUBRIMIENTO

Sólo los OverlayContainers (contenedores de recubrimiento) se pueden añadir directamente a un recubrimiento. La razón es que cada nivel del contenedor establece el orden Z de los elementos contenidos en el, así que si se anidan varios contenedores, los contenedores del interior tienen un orden Z superior a los externos, para asegurarse que se muestran correctamente. Para agregar un contenedor (como panel) al recubrimiento, simplemente hay que llamar a Overlay::add2D.

Si se desea agregar elementos secundarios al contenedor, se llama a OverlayContainer::addChild. Los elementos secundarios pueden ser OverlayElements o instancias a OverlayContainer. Hay que recordar que la posición de un elemento secundario es relativa a la esquina superior izquierda de la de los padres.

UNA PALABRA SOBRE LAS COORDENADAS 2D

Ogre permite colocar y dimensionar elementos basados en 2 sistemas de coordenadas: **relativo** y basado en **píxeles**.

Modo Pixel

Este modo es útil cuando se desea especificar el tamaño exacto para los recubrimientos, y no importa si los objetos se hacen más pequeños en la pantalla si se aumenta la resolución de la

pantalla (de hecho es posible desear esto). En este modo la única forma de poner algo en el centro o en la derecha o inferior de la pantalla de forma fiable en cualquier resolución es utilizar las opciones de adaptación, mientras que en el modo relativo se puede hacer sólo mediante el uso correcto y relativo de coordenadas. Este modo es muy sencillo, la parte superior izquierda de la pantalla es (0,0) y la parte inferior derecha de la pantalla depende de la resolución. Como se mencionó anteriormente, se pueden utilizar las opciones de adaptación para hacer que los orígenes de las coordenadas vertical y horizontal estén en la derecha, abajo o al centro de la pantalla para colocar los elementos de píxeles en esos lugares sin saber la resolución.

Modo Relativo

Este modo es útil cuando se desea que los elementos en el recubrimiento tengan el mismo tamaño en la pantalla, sin importar la resolución. En modo relativo, la parte superior izquierda de la pantalla (0,0) y la parte inferior derecha es (1,1). Así si se coloca un elemento en (0,5, 0,5), la esquina superior izquierda se coloca exactamente en el centro de la pantalla, no importa la resolución en la que se ejecuta la aplicación. El mismo principio se aplica a los tamaños, si ajusta el ancho de un elemento a 0,5, este cubre la mitad de la anchura de la pantalla. Hay que tener en cuenta que debido a la proporción de la pantalla que es normalmente 1,3333: 1 (ancho: alto), un elemento con dimensiones (0,25, 0,25) no será cuadrado, pero ocupará exactamente 1/16 de la pantalla en términos del área. Si se desea buscar áreas cuadrados, se tendrá que compensar mediante la relación de aspecto típico, por ejemplo usando (0.1875, 0.25) en su lugar.

TRANSFORMANDO RECUBRIMIENTOS

Otra característica interesante de los recubrimientos es ser capaz de girar, desplazarse y escalarse como un todo. Se puede utilizar esta característica para hacer sistemas de menú con zoom in/out, dejándolos caer desde fuera de la pantalla y otros buenos efectos. Ver Overlay::scroll, Overlay::rotate y Overlay::scale para obtener más información.

SCRIPTS DE RECUBRIMIENTOS

Los recubrimientos también pueden definirse en scripts. Véase la sección 3.4 Scripts de Recubrimiento para más detalles.

SISTEMAS DE INTERFAZ GRÁFICA DE USUARIO

Los recubrimientos son sólo diseñados para elementos no interactivos en pantalla, aunque se pueden utilizar como una interfaz gráfica de usuario rudimentaria. Para una solución de interfaz gráfica de usuario mucho más completo, se recomienda CEGUI (http://www.cegui.org.uk), como se demuestra en el demo Demo_Gui.

3. SCRIPTS

Ogre maneja muchas de sus características a través de scripts con el fin de hacer más fácil la configuración. Los scripts son archivos de texto plano que se pueden editar en cualquier editor de texto estándar, y su modificación tiene efecto inmediatamente en las aplicaciones Ogre, sin necesidad de recompilar. Esto hace mucho más rápido el prototipado. Éstos son los elementos sobre los que se pueden hacer scripts:

3.1 SCRIPTS DE MATERIALES

Los scripts de materiales ofrecen la posibilidad de definir materiales complejos en un script que pueden ser reutilizados fácilmente. Aunque se podrían instalar todos los materiales para una escena en el código usando los métodos de las clases Material y TextureLayer, en la práctica esto es un poco difícil de manejar. En su lugar se pueden almacenar definiciones de materiales en los archivos de texto que luego pueden ser cargados siempre que sea necesario.

CARGANDO SCRIPTS

Los scripts de materiales se cargan cuando se inicializan los grupos de recursos: Ogre busca en todos los lugares de recursos asociados con el grupo (véase el Root::addResourceLocation) para archivos con la extensión '. Material' y los analiza. Si desea manejar los archivos manualmente, se puede utilizar MaterialSerializer::parseScript.

Es importante darse cuenta de que los materiales no se cargan completamente por este proceso de parseo: sólo se carga la definición, sin texturas ni otros recursos. Esto es debido a que es común tener una gran biblioteca de materiales, pero sólo utilizar un subconjunto relativamente pequeño de ellos en el escenario. Cargar todo el material completamente de cada script, provocaría una sobrecarga de memoria innecesaria. Se puede acceder al material de una 'carga aplazada' de una forma normal utilizando (MaterialManager::getSingleton().getByName()), pero se debe llamar el método de 'carga' antes de intentar utilizarlo. Ogre hace esto por el usuario cuando se usan los métodos normales de asignación de material de las entidades, etc.

Otro factor importante es que los nombres de los materiales deben ser únicos en todos los scripts cargados por el sistema, ya que los materiales son siempre identificados por su nombre.

FORMATO

Se pueden definir varios materiales en un único script. El formato del script es pseudo-C++, con secciones delimitadas por llaves ('{','}'), y comentarios indicados por un inicio de línea con '//' (nota, no están permitidos los comentarios anidados). El formato general se muestra en el ejemplo a continuación (hay que tener en cuenta que, para empezar, sólo se tiene en cuenta la función fija de materiales que no utilicen programas de vértice, geometría o fragmento, estos se explican más adelante):

```
// This is a comment
material walls/funkywall1
```

```
{
    // first, preferred technique
    technique
    {
        // first pass
        pass
        {
            ambient 0.5 0.5 0.5
            diffuse 1.0 1.0 1.0

            // Texture unit 0
            texture_unit
            {
                texture wibbly.jpg
                scroll_anim 0.1 0.0
                wave_xform scale sine 0.0 0.7 0.0 1.0
            }
            // Texture unit 1 (this is a multitexture pass)
            texture_unit
            {
                texture wobbly.png
                rotate_anim 0.25
                colour_op add
            }
        }
    }

    // Second technique, can be used as a fallback or LOD level
technique
    {
        // .. and so on
    }
}
```

Código 2: Ejemplo general de formato de script.

A cada material en el script se le debe dar un nombre, que es la línea de 'material <blah>' antes de la primera apertura '{'. Este nombre debe ser único de forma global. Se pueden incluir caracteres de la ruta de acceso (como en el ejemplo) para dividir lógicamente el material, y también para evitar nombres duplicados, pero el motor no considera el nombre como jerárquico, sólo como una cadena. Si se incluyen espacios en el nombre, debe estar entre comillas dobles.

NOTA: ':' es el delimitador para especificar una copia del material en el script, de forma que no se puede utilizar como parte del nombre del material.

Un material puede heredar de otro material previamente definido con el uso de dos puntos: después del nombre del material, seguido del nombre del material de referencia del que heredar. Se puede, de hecho, heredar sólo partes de un material de otros, todo esto está cubierto en la sección 3.1.11 Herencias de scripts. También se pueden utilizar variables en el script que pueden ser sustituidas en las versiones heredadas, Véase la sección 3.1.13 Variables de script.

El material puede hacerse con varias técnicas (véase la sección 3.1.1 Técnicas) - una técnica es una forma de lograr el efecto que se busca. Se puede proporcionar más de una técnica a fin de dar enfoques de reserva para que cuando una tarjeta no tiene la capacidad de renderizar la técnica preferida, o cuando se desea definir en un menor nivel de detalle el material con el fin de conservar la potencia de renderizado cuando los objetos están más distantes.

Cada técnica se compone de muchos pasos (véase la sección 3.1.2 pasadas), esto es que un renderizado completo de un objeto se puede realizar en varias veces con diferentes valores para producir efectos complejos. Ogre también puede dividir los pasos que se han definido en muchos pasos en tiempo de ejecución, si se define un pase que utiliza demasiadas unidades de textura para la tarjeta en que se ejecutan actualmente (teniendo en cuenta que sólo se puede hacer si no se está utilizando un programa de fragmento). Cada paso tiene una serie de atributos de alto nivel, tales como 'ambiente' para establecer la cantidad y color de la luz ambiente que se refleja en el material. Algunas de estas opciones no se aplican si se están utilizando programas de vértice, véase la sección 3.1.2 Pases para más detalles.

Dentro de cada pase, puede haber cero o más unidades de textura en uso (véase la sección 3.1.3 unidades de textura). Estas definen la textura a ser utilizada, y, opcionalmente, algunas operaciones de fusión (que utilizan multitexturas) y efectos de textura.

También se pueden referenciar programas de vértice y fragmento (o 'shaders' de vértices y pixel, si se quiere usar esa terminología), en un pase con un determinado conjunto de parámetros. Los programas en sí mismos se declaran en scripts .programseparados (véase la sección 3.1.4 Declaración de Programas de Vértices/Geometría/Fragmentos) y se utilizan como se describe en 3.1.9 Uso de programas de Vértice /Geometría/Fragmento en un pase.

ATRIBUTOS DE ALTO NIVEL DEL MATERIAL

La sección más externa de una definición de material no tiene un montón de atributos propios (la mayoría de los parámetros configurables están dentro de las secciones hijas). Sin embargo, sí tiene algunos, y aquí están:

lod_strategy

Establece el nombre de la estrategia LOD (nivel de detalle) a utilizar. Por defecto se establece a 'Distance' que indica que el LOD cambia según la distancia de la cámara. También soporta 'PixelCount' que cambia el LOD según una estimación los píxeles del espacio del monitor afectados.

Formato: lod_strategy <nombre>
Valor por defecto: lod_strategy Distance

lod_values

Este atributo define los valores utilizados para controlar la transición LOD del material. Estableciendo este atributo, se indica que se quiere que este material altere la técnica que utiliza basándose en algunas medidas, como la distancia de la cámara, o la aproximación de cobertura del espacio de pantalla. El significado exacto de estos valores se determina por la opción elegida de lod_strategy – es una lista de distancias para la estrategia 'Distance', y una lista de cuentas de píxeles para la estrategia 'PixelCount', por ejemplo. Se debe dar una lista de valores, en orden desde el valor más alto de LOD al valor más bajo, cada uno indicando el punto en el que el material cambiará a la siguiente LOD. Implícitamente todos los materiales activan el índice 0 para valores menores que la primera entrada, por lo que no se tiene que especificar '0' al inicio de la lista. Hay que asegurarse de que hay por lo menos una técnica con un valor lod_index por cada valor en la lista (así si se especifican 3 valores, hay que tener técnicas para los índices 0, 1, 2 y 3). Hay que tener en cuenta que siempre se debe tener una técnica con lod_index 0.

Formato: lod_values <valor0><valor1><valor2> ...
Valor por defecto: Ninguno

Ejemplo: lod_strategy Distance lod_values 300.0 600.5 1200

El ejemplo anterior provoca que el material utilice la mejor técnica de lod_index 0 hasta una distancia de 300 unidades, la mejor de lod_index 1 desde 300 hasta 600 unidades de distancia, lod_index 2 desde 600 hasta 1200 unidades, y lod_index 3 por encima de 1200 unidades.

receive_shadows

Este atributo controla si los objetos que usan este material pueden tener sombras proyectadas sobre ellos.

Formato: receive_shadows on|off
Defecto: on

Si recibe o no un objeto una sombra es la combinación de una serie de factores, véase la sección 7. Sombras para los detalles completos; sin embargo esto permite hacer un material con la opción de recepción de sombras si fuera necesario. Hay que tener en cuenta que los materiales transparentes nunca reciben sombras, por lo que esta opción sólo tiene un efecto sobre materiales sólidos.

transparency_casts_shadows

Este atributo controla si los materiales transparentes pueden lanzar ciertos tipos de sombra.

Formato: transparency_casts_shadows <on|off>
Defecto: off

Si proyecta o no un objeto una sombra es la combinación de una serie de factores, véase la sección 7. Sombras para los detalles completos; sin embargo, esto permite hacer sombras de materiales transparentes, cuando de otro modo, no. Por ejemplo, al usar sombras de textura, los materiales transparentes no son renderizados normalmente en la textura de la sombra, porque no deben bloquear la luz. Este atributo cambia esto.

set_texture_alias

Este atributo asocia un alias de la textura con un nombre de textura.

Formato: set_texture_alias <nombre alias><nombre textura>

Este atributo puede ser usado para fijar las texturas utilizadas en los estados de unidad de textura que fueron heredados de otro material. (Ver la sección 3.1.12 de Alias de textura)

3.1.1 TÉCNICAS

La sección 'technique' en el script de material encapsula un método único de renderización de un objeto. La forma más sencilla de definición de material sólo contiene una técnica, sin embargo, desde que el hardware de PC varía muy considerablemente en sus capacidades, sólo se debe hacer si se está seguro de que todas las tarjetas que usarán la aplicación van a soportar las capacidades que la técnica requiere. Además, puede ser útil para definir la manera más sencilla de renderizar un material si desea utilizar el LOD de material, de manera que los objetos más distantes utilicen una técnica simple.

Cuando un material se utiliza por primera vez, es 'compilado'. Lo que implica el análisis de las técnicas que se han definido y el marcado de las que son soportables usando la actual API de renderizado y la

tarjeta gráfica. Si no hay técnicas sostenibles, el material se renderizará en blanco. En la compilación se examina una serie de cosas, tales como:

- El número de entradas texture_unit en cada paso. Hay que tener en cuenta que si el número de entradas texture_unit excede el número de unidades de textura en la tarjeta gráfica actual, la técnica todavía puede ser soportable tánto tiempo como el programa de fragmento no se utilice. En este caso, Ogre dividirá el paso que tiene demasiadas entradas en múltiples pases para la tarjeta de menor capacidad, y la mezcla multitextura se convertirá en una mezcla multipaso (Véase la sección colour_op_multipass_fallback).
- Si se utilizan programas de vértice, geometría o fragmento, y si es así, la sintaxis que utilizan (por ejemplo, vs_1_1, ps_2_x, arbfp1 etc)
- Otros efectos como el mapeo del cubo y la mezcla dot3
- Si el vendedor o el nombre del dispositivo de la tarjeta gráfica actual coincide con algunas normas especificadas por el usuario

En un script de materiales, las técnicas deben ser enumeradas en orden de preferencia, es decir, las técnicas anteriores son preferibles a las técnicas siguientes. Esto normalmente significa que se deben listar las más avanzadas y las más exigentes primero en el script, y la lista de técnicas menos exigentes después.

Para ayudar a identificar claramente para qué se utiliza cada técnica, la técnica puede ser nombrada, pero es opcional. Las técnicas que no se citan en el script tomarán un nombre que es el número de índice de técnica. Por ejemplo: la primera técnica en un material es el índice de 0, su nombre sería "0" si no se le dio un nombre en el script. El nombre de la técnica debe ser único en el material o bien la técnica del final es el resultado de la combinación de todas las técnicas con el mismo nombre en ese material. Un mensaje de advertencia se publica en Ogre.log si esto ocurre. Nombrar a las técnicas puede ayudar cuando se hereda un material y se modifica una técnica existente: (Véase la sección 3.1.11 Herencia de scripts)

Formato: technique name

Las técnicas tienen un pequeño número de atributos propios:

- scheme
- lod_index (y también lod_distances en el material de los padres)
- shadow_caster_material
- shadow_receiver_material
- gpu_vendor_rule
- gpu_device_rule

Scheme

Establece el 'plan o esquema' perteneciente a la técnica. Los esquemas de materiales se utilizan para controlar el nivel superior conmutando entre conjuntos de técnicas. Por ejemplo, podría usarse para definir los niveles de complejidad 'alto', 'medio' y 'bajo' de los materiales para permitir a un usuario elegir una relación rendimiento/calidad. Otra posibilidad es que si se tiene un full-HDR habilitado para máquinas punteras, renderizar todos los objetos usando sombreadores sin restricción, y un renderizador simple para otras máquinas, puede ser implementado utilizando esquemas. El esquema activo suele ser controlado a nivel de visualización, y el activo por defecto a 'Default'.

Formato: scheme <nombre>
Ejemplo: scheme hdr
Defecto: scheme Default

lod_index

Ajusta el nivel de detalle de (LOD) de esta técnica.

Formato: lod_index <número>
Nota: Los valores válidos son 0 (nivel de detalle más alto) a 65535, aunque esto es poco probable. No se debe dejar huecos en los índices de LOD entre técnicas.
Ejemplo: lod_index 1

Todas las técnicas deben pertenecer a un índice de nivel de detalle (LOD), por defecto todas ellas pertenecen al índice 0, es decir, el límite de detalle más alto. El aumento de los índices denota niveles de detalle más bajos. Se puede (y suele) asignar más de una técnica para el mismo índice LOD, lo que significa que es Ogre quien escoge la mejor técnica de los enumerados con el mismo índice LOD. Para facilitar la lectura, se aconseja que se ordene la lista de técnicas según el LOD, a continuación, en orden de preferencia, aunque el último es el único requisito (Ogre determina cuál es el 'mejor' por lo cual aparece en primer lugar). Se debe siempre tener al menos una técnica en lod_index 0.

La distancia a la que el nivel LOD se aplica se determina por el atributo lod_distances de la materia que contiene, Vea la sección lod_distances para más detalles.

Por defecto: lod_index 0

Las técnicas también contienen una o más pasadas (debe de haber por lo menos uno), Ver sección 3.1.2 Pasadas.

shadow_caster_material

Cuando se utiliza, véase la sección 7.2 Sombras basadas en texturas, se puede especificar un material alternativo al representar el objeto usando este material en la textura de la sombra. Esto es como una versión más avanzada de la utilización de shadow_caster_vertex_program, sin embargo se debe de tener en cuenta para el momento de renderizado que se espera hacer la sombra en una pasada, es decir, sólo el primer paso es respetado.

shadow_receiver_material

Cuando se utiliza, véase la sección 7.2 Sombras basadas en texturas, se puede especificar un material alternativo para utilizar cuando se realiza una pasada de recepción de sombra. Hay que tener en cuenta que este pase de 'receptor' explícito sólo se realiza cuando **no** se estén usando Sombras de textura integrada - es decir, la renderización de sombra se hace por separado (ya sea como un pase modular, o un pase de luz enmascarada). Esto es como una versión más avanzada de la utilización de shadow_receiver_vertex_program y shadow_receiver_fragment_program, sin embargo hay que tener en cuenta que para este momento se espera un renderizado de la sombra de una pasada, es decir, sólo el primer paso es respetado.

gpu_vendor_rule y gpu_device_rule

A pesar de que Ogre hace un buen trabajo de detección de las capacidades de las tarjetas gráficas y el establecimiento de la compatibilidad con las técnicas, a veces existe un determinado comportamiento que no necesariamente es detectable y puede que requiera asegurarse de que los materiales vayan por un camino particular para un tipo de uso o evitar este comportamiento. Estas reglas sirven para especificar la concordancia para que una técnica sea considerada sólo en las tarjetas de un proveedor en

particular, o que se ajustan a un patrón de nombre de dispositivo, o se admiten sólo si se considera que no cumplen tales reglas.

El formato de las normas son las siguientes:

gpu_vendor_rule include|exclude <nombre_proveedor>
gpu_device_rule include|exclude <patrón_dispositivo> [case_sensitive]

La regla 'include' significa que la técnica sólo se admite si una de las normas del include se corresponde con la indicada (si no hay reglas include, no pasará nada). La regla 'exclude' significa que la técnica se considera incompatible si se cumple alguna de las reglas exclude. Se pueden proporcionar tantas reglas como se desee, aunque <nombre_proveedor> y <patrón_dispositivo> obviamente debe ser único. La lista válida de valores de <nombre_proveedor> es actualmente 'nvidia','ati','intel','s3','matrox' y '3dlabs'. <patrón_dispositivo> puede ser cualquier cadena, y se pueden utilizar comodines ('*') si es necesario para que coincida con variantes. He aquí un ejemplo:

gpu_vendor_rule include nvidia
gpu_vendor_rule include intel
gpu_device_rule exclude *950*

Estas normas, si son todas incluidas en una técnica, significa que la técnica sólo se considerará compatible con tarjetas gráficas realizadas por NVIDIA e Intel, y mientras que el nombre del dispositivo no tiene '950' en el mismo.

Hay que tener en cuenta que estas normas pueden marcar una técnica 'incompatible' cuando debería ser considerada'compatible' a juzgar por las capacidades del hardware. Incluso si una técnica pasa estas normas, es aún objeto de las pruebas habituales de soporte de hardware.

3.1.2 PASADAS

Una pasada es una renderización simple de la geometría en cuestión, una sola llamada a la API de renderización con un conjunto determinado de propiedades de renderización. Una técnica puede tener entre una y 16 pasadas, aunque es evidente que cuantas más pasadas se utilicen, más costoso será el renderizado de la misma.

Para ayudar a identificar claramente para qué se usa cada pasada, la pasada puede ser nombrada, pero es opcional. Las pasadas que no se nombran en el script tendrán un nombre que es el número de índice de pasada. Por ejemplo: la primera pasada en una técnica tiene índice 0 por lo que su nombre sería "0" si no se le dio un nombre en el script. El nombre debe de ser único dentro de la técnica, o bien la pasada final es el resultado de la combinación de todas las pasadas con el mismo nombre en la técnica. Se muestra un mensaje de advertencia en Ogre.log si esto ocurre. Nombrar las pasadas puede ayudar en la herencia de un material y la modificación de una pasada existente: (Véase la sección 3.1.11 Herencia de scripts).

Las pasadas tienen un conjunto de atributos globales (que se describen más adelante), cero o más entradas anidadas texture_unit (véase la sección 3.1.3 Unidades de Textura) y, opcionalmente, una referencia a un programa de vértice y/o fragmento (véase la sección 3.1.9 Usando Programas de Vertex/Geometría/Fragmento de una pasada).

Éstos son los atributos que se pueden utilizar en una sección de 'pasada' de un script .material:

- ambient
- diffuse

- specular
- emissive
- scene_blend
- separate_scene_blend
- scene_blend_op
- separate_scene_blend_op
- depth_check
- depth_write
- depth_func
- depth_bias
- iteration_depth_bias
- alpha_rejection
- alpha_to_coverage
- light_scissor
- light_clip_planes
- illumination_stage
- transparent_sorting
- normalise_normals
- cull_hardware
- cull_software
- lighting
- shading
- polygon_mode
- polygon_mode_overrideable
- fog_override
- colour_write
- max_lights
- start_light
- iteration
- point_size
- point_sprites
- point_size_attenuation
- point_size_min
- point_size_max

DESCRIPCIONES DE ATRIBUTO

ambient

Establece las propiedades de reflejo del color ambiente en la pasada. Este atributo no tiene ningún efecto si se utiliza un asm, CG, o un programa de sombreado HLSL. Con GLSL, el sombreador puede leer el estado material de OpenGL.

Formato: ambient (<rojo><verde><azul> [<alfa>] | vertexcolour)
Nota: los valores de color válidos van entre 0.0 y 1.0.
Ejemplo: ambient 0.0 0.8 0.0

El color base de una pasada se determina por la cantidad de rojo, verde y azul que se refleja en cada vértice. Esta propiedad determina la cantidad de luz ambiente (luz de rumbo mundial) que se refleja. Es posible también hacer la reflexión ambiente del color definido en el vértice de la malla utilizando la palabra clave vertexcolour en lugar de los valores de color. El valor por defecto es blanco completo, que significa que los objetos son completamente iluminados. Esto se puede reducir si se desea ver efectos de luz difusa o especular, o cambiar la mezcla de colores para hacer que el objeto tenga un color de base

distinto del blanco. Este ajuste no tiene efecto si se desactiva la iluminación dinámica con el atributo 'lighting off', o si cualquier capa de la textura tiene el atributo 'colour_op replace'.

Por defecto: ambient 1.0 1.0 1.0 1.0

diffuse

Establece las propiedades de reflectancia de color difusa de la pasada. Este atributo no tiene ningún efecto si se utiliza un asm, CG, o un programa de sombreado HLSL. Con GLSL, el sombreador puede leer el estado material de OpenGL.

Formato: diffuse (<rojo><verde><azul> [<alfa>] | vertexcolour)
Nota: los valores de color válidos son entre 0.0 y 1.0.
Ejemplo: diffuse 1.0 0.5 0.5

El color base de una pasada se determina por la cantidad de rojo, verde y azul se refleja en cada vértice. Esta propiedad determina la cantidad de luz difusa (la luz de las instancias de la clase Light (luz) en el escenario) que se refleja. También es posible hacer la reflectancia de color difusa según se define en el vértice de la malla utilizando la palabra clave vertexcolour en lugar de los valores de color. El valor por defecto es blanco completo, significando que los objetos reflejan la luz blanca máxima que proceda de los objetos Light (luz). Este ajuste no tiene efecto si se desactiva la iluminación dinámica con el 'lighting off', o si cualquier capa de la textura tiene un atributo 'colour_op replace'.

Por defecto: diffuse 1.0 1.0 1.0 1.0

specular

Establece propiedades de la reflectancia de color especular de esta pasada. Este atributo no tiene ningún efecto si se utiliza un asm, CG, o un programa de sombreado HLSL. Con GLSL, el sombreador puede leer el estado material de OpenGL.

Formato: specular (<rojo><verde><azul> [<alfa>] | vertexcolour) <brillo>
Nota: los valores de color válidos son entre 0.0 y 1.0. El brillo puede ser cualquier valor mayor que 0.
Ejemplo: specular 1.0 1.0 1.0 12.5

El color base de una pasada se determina por la cantidad de rojo, verde y azul que se refleja en cada vértice. Esta propiedad determina la luz especular que se refleja (realzándolo de las instancias de la clase Light (luz) en el escenario). También es posible hacer la pista de reflectancia difusa de color según se define en el vértice de la malla utilizando la palabra clave vertexcolour en lugar de los valores de color. El valor predeterminado es no reflejar la luz especular. El color de las luces especulares está determinado por los parámetros de color y el tamaño del realce por el parámetro de brillo. Cuanto mayor sea el valor del parámetro de brillo, más intenso es el resalte, es decir, el radio es menor. Hay que tener cuidado de utilizar los valores de brillo en el rango de 0 a 1, ya que esto hace que el color especular se aplique a toda la superficie que tiene el material aplicado. Cuando el ángulo de visión de la superficie cambia, se produce un feo parpadeo que se da cuando el brillo está en el rango de 0 a 1. Los valores de brillo entre 1 y 128, funcionan mejor en DirectX y OpenGL. Este ajuste no tiene efecto si se desactiva la iluminación dinámica con el atributo 'lighting off', o si cualquier capa de la textura tiene un atributo 'colour_op replace'.

Por defecto: specular 0.0 0.0 0.0 0.0 0.0

emissive

Establece la cantidad de auto-iluminación que tiene un objeto. Este atributo no tiene ningún efecto si se utiliza un asm, CG, o un programa de sombreado HLSL. Con GLSL, el sombreador puede leer el estado material de OpenGL.

Formato: emissive (<rojo><verde><azul> [<alfa>] | vertexcolour)
Nota: los valores de color válidos son entre 0.0 y 1.0.
Ejemplo: emissive 1.0 0.0 0.0

Si un objeto es auto-iluminado, no necesita fuentes externas de luz, ambientales o de otro tipo. Es como si el objeto tiene su propia luz ambiente personal. A diferencia de lo que su propio nombre indica, este objeto no actúa como una fuente de luz para otros objetos en el escenario (si se quiere, se tiene que crear una luz centrada en el objeto). También es posible hacer la traza de color emisiva del color definido en el vértice de la malla utilizando la palabra clave vertexcolour en lugar de los valores de color. Este ajuste no tiene efecto si se desactiva la iluminación dinámica con el atributo 'lighting off', o si cualquier capa de la textura tiene un atributo 'colour_op replace'.

Por defecto: emissive 0.0 0.0 0.0 0.0

scene_blend

Establece el tipo de mezcla que esta pasada tiene con el contenido actual del escenario. Considerando las operaciones de mezcla de texturas vistas en las entradas texture_unit tiene que ver con la mezcla entre las capas de textura, esta mezcla resulta de combinar la salida de esta pasada como un todo con el contenido existente del objetivo de la representación. Esta combinación permite, por tanto transparencia de objetos y otros efectos especiales. Hay 2 formatos, uno con tipos de mezcla predefinidos y otros personalizados, creados usando factores de origen y destino.

Formato 1: scene_blend add|modulate|alpha_blend|colour_blend
Ejemplo: scene_blend add

Esta es la forma más simple, donde los modos de fusión más comúnmente utilizados se enumeran usando un único parámetro. Los parámetros válidos de <blend_type> son:

add
: El color de la salida del render, se añade al escenario. Bueno para explosiones, bengalas, luces, fantasmas, etc. Equivalente a 'scene_blend one one'.

modulate
: El color de la salida de render se multiplica con el contenido del escenario. En general, los colores y la oscuridad del escenario, bueno para vidrio ahumado, objetos semitransparentes, etc. Equivalente a 'scene_blend dest_colour zero'.

colour_blend
: Colorea el escenario basado en los brillos de los colores de entrada, pero no se oscurecen. Equivalente a 'scene_blend src_colour one_minus_src_colour'.

alpha_blend
: El valor alfa de la salida del renderizado se utiliza como una máscara. Equivalente a 'scene_blend src_alpha one_minus_src_alpha'.

Formato 2: scene_blend <src_factor><dest_factor>

Ejemplo: scene_blend one one_minus_dest_alpha

Esta versión del método permite un control completo sobre la operación de fusionado, especificando los factores de la mezcla origen y destino. El color resultante que se escribe en el render es (textura * factorOrigen) + (scene_pixel * factorDestino). Los valores válidos para ambos parámetros son:

uno
De valor constante de 1.0.

zero
De valor constante de 0.0.

dest_colour
El color del pixel existente.

src_colour
El color del píxel de textura (Texel).

one_minus_dest_colour
1 - (dest_colour).

one_minus_src_colour
1 - (src_colour).

dest_alpha
El actual valor del píxel alfa.

src_alpha
El valor alfa Texel.

one_minus_dest_alpha
1 - (dest_alpha).

one_minus_src_alpha
1 - (src_alpha).

Por defecto: scene_blend one zero (opaco)

Ver también separate_scene_blend.

separate_scene_blend

Esta opción funciona de manera exactamente igual que scene_blend, excepto que permite especificar las operaciones a realizar entre el pixel renderizado y la memoria intermedia de la estructura por separado, por colores y componentes alfa. Por naturaleza, esta opción sólo es útil cuando se representa a los objetivos que tienen un canal alfa que se usará para su posterior procesamiento, tales como un renderizado de la textura.

Formato 1: separate_scene_blend <simple_colour_blend><simple_alpha_blend>
Ejemplo: separate_scene_blend add modulate

Este ejemplo podría agregar componentes de color, pero se multiplican los componentes alfa. Los modos disponibles de mezcla son como en el scene_blend. La forma más avanzada también está disponible:

Formato 2: separate_scene_blend <colour_src_factor><colour_dest_factor><alpha_src_factor><alpha_dest_factor>
Ejemplo: separate_scene_blend one one_minus_dest_alpha one one

Una vez más las opciones disponibles en el segundo formato son las mismas que en el segundo formato de scene_blend.

scene_blend_op

Esta directiva cambia la operación que se aplica entre los dos componentes de la ecuación de fusión de la escena, por defecto es 'add' (FactorOrigen * origen + FactorDestino * destino). Se puede cambiar este por 'add', 'subtract', 'reverse_subtract', 'min' o 'max'.

Formato: scene_blend_op add|subtract|reverse_subtract|min|max
Por defecto: scene_blend_op add

separate_scene_blend_op

Esta directiva es una scene_blend_op, excepto que se establece la operación para el color y alfa por separado.

Formato: separate_scene_blend_op <colorOp><AlfaOp>
Por defecto: separate_scene_blend_op add add

depth_check

Establece si el renderizado de esta pasada se hace con la comprobación del búfer de profundidad o no.

Formato: depth_check on|off

Si el control de profundidad de búfer está activado, cuando un píxel está a punto de ser escrito en el búfer de marco, la profundidad de este se revisa para ver si el pixel está delante de todos los otros píxeles escritos en ese punto. Si no, el píxel no se escribe. Si el control de profundidad de búfer está desactivado, los píxeles se escriben, sin importar lo que se ha renderizado antes. Ver también depth_func para ver configuraciones de chequeo de profundidad más avanzadas.

Por defecto: depth_check on

depth_write

Establece si en esta pasada se renderiza escribiendo con búfer de profundidad o no.

Formato: depth_write on|off

Si la escritura con búfer de profundidad está activada, cuando un píxel se escribe en el búfer de estructura, el búfer de profundidad se actualiza con el valor de la profundidad de ese nuevo pixel, lo que afecta a las operaciones futuras de renderizado si hay píxeles detrás de este. Si la escritura de profundidad está apagada, los píxeles se escriben sin actualizar el búfer de profundidad. La escritura en profundidad normalmente debería estar activada, pero puede apagarse cuando se representan fondos estáticos o cuando se representa una colección de objetos transparentes al final de un escenario de modo que se superponen unos a otros correctamente.

Por defecto: depth_write on

depth_func

Establece la función que se utiliza para comparar valores de profundidad cuando la comprobación de profundidad está activada.

Formato: depth_func <función>

Si se habilita la comprobación de la profundidad (ver depth_check) se presenta una comparación entre el valor de la profundidad del píxel a ser escrito y el contenido actual del búfer. Esta comparación es normalmente less_equal, es decir, si el píxel es escrito está más cerca (o en la misma distancia) que el contenido actual. Las funciones posibles son:

always_fail
Nunca escribe un píxel en el objetivo a renderizar.

always_pass
Siempre escribe un píxel en el objetivo a renderizar.

less
Escribir si (nueva_Z < Z_existente).

less_equal
Escribir si (nueva_Z <= Z_ existente).

equal
Escribir si (nueva_Z == Z _existente).

not_equal
Escribir si (nueva_Z! = Z_ existente).

greater_equal
Escribir si (nueva_Z> = Z_ existente).

greater
Escribir si (nueva_Z> Z_ existente).

Por defecto: depth_func less_equal

depth_bias

Establece la polarización aplicada al valor de la profundidad de esta pasada. Puede ser utilizada para hacer que aparezcan polígonos coplanares en la parte superior de los demás, por ejemplo para calcomanías.

Formato: depth_bias <constant_bias> [<slopescale_bias>]

El valor final de la polarización de profundidad es constant_bias * minObservableDepth + maxSlope * slopescale_bias. La escala de la pendiente de la polarización es relativa al ángulo del polígono con la cámara, lo que lo convierte en un valor adecuado, pero esto se omite en algunas tarjetas viejas. La polarización constante se expresa como un factor del valor de profundidad mínima, de forma que un valor de 1 empujará la profundidad una 'muesca' si se quiere. Ver también iteration_depth_bias.

iteration_depth_bias

Establece una polarización adicional derivada de la cantidad de veces que una pasada ha sido iterada. Funciona igual que depth_bias salvo que se aplica un factor de polaricación adicional al valor por defecto de depth_bias, multiplicando el valor dado por el número de veces que este paso ha sido iterado antes, a través de una de las variantes de la iteración. Así que en la primera pasada obtiene el valor depth_bias, la segunda vez se recibe depth_bias + iteration_depth_bias, la tercera vez se recibe depth_bias + iteration_depth_bias * 2, y así sucesivamente. El valor predeterminado es cero.

Formato: iteration_depth_bias <bias_per_iteration>

alpha_rejection

Establece la forma en que la pasada usará alfa para rechazar totalmente los píxeles del pipeline.

Formato: alpha_rejection <función><valor>
Ejemplo: alpha_rejection greater_equal 128

El parámetro función puede ser cualquiera de las opciones enumeradas en el atributo depth_function del material. El parámetro de valor, teóricamente, puede ser cualquier valor entre 0 y 255, pero es mejor limitarse a 0 o 128 por la compatibilidad de hardware.

Por defecto: alpha_rejection always_pass

alpha_to_coverage

Establece si esta pasada utilizará 'alfa para la cobertura', una forma de ejemplizar los bordes alfa de la textura para que se combinen a la perfección con el fondo. Este servicio se ofrece únicamente en las tarjetas de alrededor de 2006 en adelante, pero es seguro activarlo de todos modos - Ogre lo ignorará si el hardware no lo soporta. El uso común del alfa para la cobertura es para la renderización de texturas de follaje, alambrada y de este estilo.

Formato: alpha_to_coverage on|off
Por defecto: alpha_to_coverage off

light_scissor

Establece si al representar esta pasada, la renderización estará limitada a un espacio de pantalla recortado en rectángulo que representa la cobertura de la luz que se utiliza en esta pasada, derivados de sus rangos de atenuación.

Formato: light_scissor on|off
Por defecto: light_scissor off

Esta opción sólo es útil si esta pasada es una pasada de iluminación aditiva, y es por lo menos la segunda en la técnica. Es decir, las zonas que no están afectadas por la luz en curso nunca tendrán que ser renderizadas. Si hay más de una luz que se pasa en la pasada, a continuación, el recorte se define como el rectángulo que abarca todas las luces en el espacio de la pantalla. Las luces direccionales son ignoradas, ya que son infinitas.

Esta opción no necesita ser especificada si se está utilizando un modo estándar de sombras aditivas, es decir, SHADOWTYPE_STENCIL_ADDITIVE o SHADOWTYPE_TEXTURE_ADDITIVE, ya que es el comportamiento por defecto para usar un recorte por cada sombra aditiva a pasar. Sin embargo, si no

se están utilizando las sombras, o si se está usando Sombras de Textura Integradas las pasadas se especifican de manera personalizada, entonces esto podría ser de utilidad.

light_clip_planes

Establece si al renderizar esta pasada, la configuración de triángulos se limitará a un recorte del volumen cubierto por la luz. Las luces direccionales son ignoradas, las luces puntuales cortan un cubo del tamaño del rango de atenuación o luz, y los focos cortan a una pirámide que limita el ángulo del foco y elrango de atenuación.

Formato: light_clip_planes on|off
Por defecto: light_clip_planes off

Esta opción sólo funcionará si hay una única luz no-direccional en esta pasada. Si hay más de una luz, o sólo hay luces direccionales, entonces, no se producirá el recorte. Si no hay ninguna luz, los objetos no se renderizarán.

Cuando se usa un modo estándar de sombra aditiva, es decir, SHADOWTYPE_STENCIL_ADDITIVE o SHADOWTYPE_TEXTURE_ADDITIVE, se tiene la opción de habilitar un recorte para todas las pasadas de luz llamando a SceneManager::setShadowUseLightClipPlanes independientemente de la configuración de esta pasada, ya que la renderización se hace a lo largo de toda la luz de todos modos. Esta opción está desactivada por defecto ya que con planos de corte no siempre es más rápido – ya que depende de la cantidad de volúmenes de luz cubiertos en la escena. En general, las luces pequeñas son las más probables de que se vea algún beneficio en el lugar del recorte. Si no se están utilizando las sombras, o si se está usando Sombras de Texturas Integradas donde en el pase se especifican de manera personalizada, se debería especificar la opción de pase utilizando este atributo.

Una nota específica sobre OpenGL: los planos de corte de usuario son completamente ignorados cuando se utiliza un programa de vértice ARB. Esto significa que los planos de corte de luz no ayudan mucho si se utilizan programas de vértice ARB en GL, aunque Ogre realiza alguna optimización propia, en el que si se ve que el volumen del recorte está completamente fuera de la pantalla, no se llevará a cabo el renderizado. Cuando se utiliza GLSL, el recorte de usuario puede ser utilizado, pero se tiene que utilizar un sombreador glClipVertex, consultar la documentación GLSL para más información. En Direct3D, los planos de corte de usuario siempre se respetan.

illumination_stage

Cuando se utiliza un modo de iluminación aditiva (SHADOWTYPE_STENCIL_ADDITIVE o SHADOWTYPE_TEXTURE_ADDITIVE), la escena se representa en 3 etapas distintas, ambiente (o pre-iluminación), por la luz (una vez por luz, con sombra) y la calcomanía (o post-iluminación). Normalmente Ogre busca una manera de clasificar las pasadas de forma automática, pero hay algunos efectos que no se pueden lograr sin controlar manualmente la iluminación. Por ejemplo, los efectos especulares están silenciados por la secuencia típica porque todas las texturas se guardan hasta la etapa 'calcomanía', que silencia el efecto especular. En su lugar, se podrían hacer texturas dentro de la etapa pre-iluminación, si es posible por el material y por lo tanto añadir el especular sobre la textura después de la etiqueta, y no posterior a la representación de luz.

Si se asigna una etapa de iluminación a una pasada, se tiene que asignar a todas las pasadas de la técnica, de lo contrario se ignorará. También hay que tener en cuenta que, si bien se puede tener más de una pasada en cada grupo, no se pueden alternar, es decir, todas las pasadas ambiente se darán

antes de las pasadas de pre-iluminación, que también tendrán que ser antes que todos los pases calcomanía. Dentro de sus categorías las pasadas conservarán orden.

Formato: illumination_stage ambient|per_light|decal
Por defecto: nada (autodetectado)

normalise_normals

Establece si esta pasada renderizará todos los vértices normales siendo automáticamente re-normalizados.

Formato: normalise_normals on|off

Escalar objetos causa en los vértices normales el cambio de magnitud, que puede despistar a los cálculos de iluminación. Por defecto, el SceneManager lo detecta y automáticamente re-normaliza los vértices normales de cualquier objeto a escala, pero esto tiene un coste. Si se prefiere controlar manualmente, se llama a SceneManager::setNormaliseNormalsOnScale(false) y luego se utiliza esta opción en los materiales que son sensibles a que los vértices normales se cambien de tamaño.

Por defecto: normalise_normals off

transparent_sorting

Establece si las texturas transparentes deben ser ordenadas por profundidad o no.

Formato: transparent_sorting on|off|force

Por defecto, todos los materiales transparentes se ordenan de tal manera que los más lejanos de la cámara se representan en primer lugar. Este suele ser el comportamiento deseado, pero en algunos casos, esta profundidad de clasificación puede ser innecesaria e indeseable. Por ejemplo, si es necesario para garantizar que el orden de representación no cambie de un frame a otro. En este caso se debe de poner el valor a 'off' para prevenir la ordenación.

También se puede utilizar la palabra clave 'force' para forzar la ordenación transparente, sin importar las otras circunstancias. Normalmente la ordenación sólo se utiliza cuando la pasada es también transparente, y tiene escritura o lectura de profundidad que indica que no se puede renderizar sin ordenación. Utilizando 'force', se le dice a Ogre que ordene esta pasada sin importar las otras circunstancias presentes.

Por defecto: transparent_sorting on

cull_hardware

Establece el modo de sacrificio hardware en esta pasada.

Formato: cull_hardware clockwise|anticlockwise|none

Una forma típica de sacrificio de triángulos por el motor de renderizado del hardware se basa en el 'serpenteado de vértices' de los triángulos. El serpenteado de vértices se refiere a la dirección en la que los vértices se pasan o se indexan en el proceso de renderización, visto desde la cámara, y se sacrificarán según las agujas del reloj o en sentido contrario. Si se establece la opción de en sentido horario, todos los triángulos cuyos vértices son vistos en sentido horario desde la cámara serán sacrificados por el hardware. En la otra opción, se realizará en sentido contrario (obviamente), y la opción de nada (none)

apaga el sacrificio por hardware, de forma que todos los triángulos son renderizados (útil para la creación de pasadas de 2 caras).

Por defecto: cull_hardware clockwise

Nota: Este es el mismo estado por defecto que OpenGL, pero contrario del estado por defecto de Direct3D (porque Ogre utiliza un sistema de coordenadas de mano derecha, como OpenGL).

cull_software

Establece el modo de sacrificio software para esta pasada.

Formato: cull_software back|front|none

En algunas situaciones, el motor también sacrificará geometría en software antes de enviarlo al renderizador hardware. Este valor sólo tiene efecto sobre un SceneManager que lo utilice (es muy útil en grandes grupos del mundo de la geometría plana en lugar de en la geometría móvil, ya que sería costoso), pero si se usa, se puede sacrificar la geometría antes de ser enviada al hardware. En este caso, el sacrificio se basa en si la parte 'trasera' (back) o 'frontal' (front) del triangulo mira hacia la cámara - esta definición se basa en la cara normal (un vector que sobresale de la parte frontal del polígono perpendicular a la cara). Ogre considera caras normales a estar en el lado contrario a las agujas del reloj de la cara, 'cull_software back' es el equivalente a 'cull_hardware clockwise', por lo que ambos son la opción predeterminada. Sin embargo, la denominación es diferente para reflejar la forma en que el sacrificio se realiza, ya que en la mayoría de las veces las caras normales son pre-calculadas y no tiene que ser de la forma que Ogre espera - se podría establecer 'cull_hardware none' de forma que el sacrificio se hará completamente en el software basado en normales propias de su cara, si se tiene la SceneManager que los utiliza correctamente.

Por defecto: cull_software back

lighting

Establece si habrá o no iluminación dinámica para esta pasada. Si la iluminación está apagada, todos los objetos renderizados mediante el pase estarán totalmente iluminados. **Este atributo no tiene efecto si se utiliza un programa de vértice**.

Formato: lighting on|off

Poner la iluminación dinámica apagada hace redundante cualquier propiedad de ambiente, difusa, especular, emisiva y sombreada de esta pasada. Cuando el iluminado se enciende, los objetos se iluminan en función de sus vértices normales de luz difusa y especular, y en el mundo de ambiente y de emisión.

Por defecto: lighting on

shading

Establece el tipo de sombreado que debe utilizarse para la representación de la iluminación dinámica de esta pasada.

Formato: shading flat|gouraud|phong

Cuando la iluminación dinámica está activada, el efecto es el de generar valores de los colores en cada vértice. Si son interpolados estos valores en la cara (y cómo) depende de este ajuste.

flat
: No se lleva a cabo la interpolación. Cada cara es sombreada con un solo color determinado a partir del primer vértice en la cara.

gouraud
: El color en cada vértice es linealmente interpolado en la cara.

phong
: Los vértices normales son interpolados en la cara, y estos se utilizan para determinar el color de cada píxel. Da un efecto de iluminación más natural, pero es más caro y funciona mejor en los altos niveles de Tesselation. No está soportado por todo el hardware.

Por defecto: shading gouraud

polygon_mode

Establece cómo deben ser rasterizados los polígonos, es decir, si deben ser rellenados, o simplemente se dibujan como líneas o puntos.

Formato: polygon_mode solid|wireframe|points

solid
: La situación normal - Los polígonos están rellenos.

wireframe
: Se dibujan solo los contornos de los polígonos.

points
: Sólo los puntos de cada polígono se representan.

Por defecto: polygon_mode solid

polygon_mode_overrideable

Establece si el modo de polígono (polygon_mode) establecido en este paso puede ser degradado por la cámara, si la cámara está en un modo de polígono bajo. Si se establece en 'false', este paso siempre será renderizado en el modo propio del polígono seleccionado, no importa lo que la cámara tenga establecido. El valor predeterminado es 'true'.

Formato: polygon_mode_overrideable true|false

fog_override

Le dice a la pasada si se debe reemplazar la configuración de niebla de la escena, y forzar a usar su propia configuración o no. Muy útil para cosas que no quieren verse afectadas por la niebla cuando el resto de la escena está nublada, o viceversa. Hay que tener en cuenta que esto sólo afecta a la función fija de niebla - los parámetros de niebla original de la escena se siguen enviando a los sombreadores que utilizan el 'fog_params parameter binding' (esto le permite desactivar la niebla de función fija y calcularla en el shader, si se desea deshabilitar el shader de niebla se puede hacer a través de los parámetros de sombreado).

Formato: fog_override true|false [<tipo><color><densidad><inicio><fin>]
Por defecto: fog_override false

Si se especifica 'true' para el primer parámetro y se suministran el resto de los parámetros, se está diciendo a la pasada que utilice estas opciones de niebla en preferencia a los ajustes de escena. Si se especifica 'true', pero no se proporcionan más parámetros, se le está diciendo a esta pasada que no se utilizará niebla, sin importar lo que dice la escena. He aquí una explicación de los parámetros:

tipo

none = sin niebla, el equivalente de usar 'fog_override true'.

linear = niebla lineal de la <start> y <end> distancias.

exp = La niebla aumenta de forma exponencial según la distancia a la cámara (fog = 1 / e ^ (distancia *densidad)), <density> es el párametro de control.

exp2 = La niebla aumenta de forma exponencial al cuadrado (niebla = 1 / e ^ (distancia * Densidad) ^ 2),el uso <density> es el párametro de control.

color

Secuencia de 3 valores de punto flotante de 0 a 1 que indica la intensidad de rojo, verde y azul.

densidad

El parámetro de densidad utilizado en la 'exp' o 'exp2' para los tipos de niebla. No se utiliza en el modo lineal, pero debe aparecer como un marcador de posición.

inicio

La distancia que parte la niebla lineal de la cámara. Debe estar presente en otros modos, a pesar de que no se utiliza.

fin

La distancia desde la cámara al final de la niebla lineal. Debe estar presente en otros modos, a pesar de que no se utiliza.

Ejemplo: fog_everride true exp 1 1 1 0.002 100 10000

colour_write

Establece si se renderiza con escritura de color o no en esta pasada.

Formato: colour_write on|off

Si la escritura de color está apagada, se pintan en la pantalla pixeles no visibles en esta pasada. Se podría pensar es inútil, pero si se renderiza con la escritura de color apagada, y con otros valores muy mínimos, puede utilizar este paso para inicializar el buffer de profundidad antes de renderizar otras pasadas que rellenarán los datos de color. Esto le puede dar incremento en el rendimiento de algunas de las nuevas tarjetas, especialmente cuando se utilizan programas de fragmento complejo, porque si la verificación de la profundidad falla, entonces el programa de fragmento nunca se ejecuta.

Por defecto: colour_write on

start_light

Establece la primera luz que se considerará para el uso con esta pasada.

Formato: start_light <número>

Se puede utilizar este atributo para compensar el punto de partida de las luces de esta pasada. En otras palabras, si se establece start_light a 2 entonces, la primera luz que será procesada en la pasada será la tercera luz de la lista aplicable. Se podría utilizar esta opción para utilizar diferentes pasadas para procesar el primer par de luces de frente a la segunda pareja de luces, por ejemplo, o para usarlo en combinación con la opción de repetición (iteration) para iniciar la iteración en un punto determinado de la lista (por ejemplo, renderizando las primeras 2 luces en la primera pasada, y luego tal vez iterando 2 luces).

Por defecto: start_light 0

max_lights

Establece el número máximo de luces que serán consideradas para su uso con esta pasada.

Formato: max_lights <número>

El número máximo de luces que se pueden utilizar en el renderizado de materiales de función fija está activado por el sistema de representación, y normalmente se establece en 8. Cuando se utiliza el pipeline programable (Véase la sección 3.1.9 Uso de programas de Vértice/Geometría/Fragmento en una pasada) este límite depende del programa que está ejecutando, o, si se utiliza 'iteration once_per_light' o una variante (véase la sección iteration), en la práctica sólo limitada por el número de pasadas que se está dispuesto a utilizar. Si no se está utilizando el paso de iteración, el límite de la luz se aplica una vez por esta pasada. Si se está utilizando el paso de iteración, el límite de la luz se aplica en todas las iteraciones de esta pasada - por ejemplo, si se tienen 12 luces en la gama con la configuración de una 'iteration once_per_light', pero el max_lights se establece en 4 para la pasada, la pasada sólo iterará 4 veces.

Por defecto: max_lights 8

Iteration

Establece si se repite o no esta pasada.

Formato 1: iteration once|once_per_light [<tipoLuz>]
Formato 2: iteration <número> [per_light [<tipoLuz >]]
Formato 3: iteration <número> [per_n_lights <num_lights> [<tipoLuz>]]

Ejemplos:

iteration once
El pase sólo se ejecuta una vez, que es el comportamiento por defecto.

iteration once_per_light point
El pase se ejecuta una vez para cada punto de luz.

iteration 5

El estado del renderizado de la pasada será el de instalación y luego se llamará al pintado 5 veces.

iteration 5 per_light point

El estado del renderizado de la pasada será el de instalación y luego se llamará al pintado que se ejecutará 5 veces. Esto se hará para cada punto de luz.

iteration 1 per_n_lights 2 point

El estado del renderizado de la pasada será el de instalación y luego se llamará al pintado una vez por cada 2 luces.

De forma predeterminada, las pasadas se emitirán sólo una vez. Sin embargo, si se utiliza la canalización (pipeline) programable, o si se desea superar los límites normales en el número de luces que son compatibles, es posible que se desee utilizar la opción once_per_light. En este caso, sólo el índice de luz 0 se usa, y el pase se emite varias veces, cada vez con una luz diferente en el índice de luz 0. Es evidente que esto hará la pasada más costosa, pero puede ser la única manera de lograr ciertos efectos, como efectos de iluminación por píxel que tienen en cuenta de 1 a n luces.

Usando un número en lugar de 'once' instruye a la pasada para iterar más de una vez después de que el renderizador haga la instalación. El estado del renderizador no se cambia después de la llamada de la configuración inicial, de forma que puede hacer repetidas llamadas a dibujar muy rápidas y es ideal para pasadas que utilizan shaders programables que deben de iterar más de una vez con el mismo estado del render, por ejemplo, los shaders que hacen piel, desenfoque de movimiento, o filtrado especial.

Si se utiliza once_per_light, también se debe añadir una pasada ambiente a la técnica antes de esta pasada, de lo contrario cuando no hay ninguna luz en el rango de este objeto no se renderiza todo, esto es importante, incluso cuando no se tiene luz ambiente en la escena, porque se desea que las siluetas de los objetos aparezcan.

El parámetro lightType sólo se aplica si se utiliza once_per_light, per_light o per_n_lights y restringe la pasada para que se ejecute para las luces de un solo tipo (ya sea 'punto', 'dirección' o 'in situ'). En el ejemplo, la pasada se llevará a cabo una vez por cada punto de luz. Esto puede ser útil porque cuando se está escribiendo un programa de vértice/fragmento es mucho más fácil si se puede asumir el tipo de luces que se ocupa. Sin embargo, al menos, el punto y las luces direccionales pueden ser tratados de una manera.

Por defecto: iteration once
Ejemplo: Script de material simple de sombreado de pelo que usa una segunda pasada con 10 iteraciones para crecer el pelo:

```
// GLSL simple Fur
vertex_program GLSLDemo/FurVS glsl
{
  source fur.vert
  default_params
  {
    param_named_auto lightPosition light_position_object_space 0
    param_named_auto eyePosition camera_position_object_space
    param_named_auto passNumber pass_number
    param_named_auto multiPassNumber pass_iteration_number
    param_named furLength float 0.15
  }
}

fragment_program GLSLDemo/FurFS glsl
{
```

```
  source fur.frag
  default_params
  {
    param_named Ka float 0.2
    param_named Kd float 0.5
    param_named Ks float 0.0
    param_named furTU int 0
  }
}

material Fur
{
  technique GLSL
  {
    pass base_coat
{
      ambient 0.7 0.7 0.7
      diffuse 0.5 0.8 0.5
      specular 1.0 1.0 1.0 1.5

      vertex_program_ref GLSLDemo/FurVS
      {
      }

      fragment_program_ref GLSLDemo/FurFS
      {
      }

      texture_unit
      {
        texture Fur.tga
        tex_coord_set 0
        filtering trilinear
      }

    }

    pass grow_fur
    {
      ambient 0.7 0.7 0.7
      diffuse 0.8 1.0 0.8
      specular 1.0 1.0 1.0 64
      depth_write off

      scene_blend src_alpha one
      iteration 10

      vertex_program_ref GLSLDemo/FurVS
      {
      }

      fragment_program_ref GLSLDemo/FurFS
      {
      }

      texture_unit
      {
        texture Fur.tga
        tex_coord_set 0
filtering trilinear
      }
```

```
    }
  }
}
```

Código 3: Ejemplo de Script de material de sombreado iterando.

Nota: se pueden usar auto parámetros de programa GPUpass_number y pass_iteration_number para llamara programas de vértice, geometría o fragmento para indicar el número de pasada y el número de iteración.

point_size

Esta configuración permite cambiar el tamaño de los puntos al representar una lista de puntos, o una lista de sprites de puntos. La interpretación de este mandato depende de la opción point_size_attenuation - si está desactivado (por defecto), el tamaño de punto es en píxeles de pantalla, si está encendido, es expresado en la pantalla como coordenadas normalizadas (1.0 es la altura de la pantalla) cuando el punto está en el origen.

NOTA: Algunos controladores tienen un límite superior para el tamaño de los puntos que soportan - esto incluso puede variar entre ¡APIs de la misma tarjeta! No hay que confiar en los tamaños de punto que causan que los puntos sean muy grandes en la pantalla, ya que pueden restringirse en algunas tarjetas. Los tamaños superiores pueden variar desde 64 hasta 256 píxeles.

Formato: point_size <tamaño>
Por defecto: point_size 1.0

point_sprites

Esta configuración especifica si está habilitado o no el renderizado de sprite de punto hardware para esta pasada. Habilitar significa que una lista de puntos se representa como una lista de cuatro puntos en lugar de una lista de puntos. Es muy útil usar esta opción si se está utilizando una cartelera de especificaciones (BillboardSet) y sólo se tendrán que utilizar carteleras orientadas a punto que son todas del mismo tamaño. También puede utilizarse para cualquier otro punto de la lista de renderizado.

Formato: point_sprites on|off
Por defecto: point_sprites off

point_size_attenuation

Define si el tamaño de punto se atenúa con la distancia, y de qué forma. Esta opción es especialmente útil cuando se están utilizando sprites de punto (ver la sección point_sprites), ya que define la forma en que se reducen de tamaño a medida que van más lejos de la cámara. Se puede deshabilitar esta opción para hacer el sprite de punto con un tamaño de pantalla constante (como puntos), o permitir a los puntos que cambien el tamaño con la distancia.

Sólo se tienen que proporcionar los últimos 3 parámetros si se activa la atenuación. La fórmula para la atenuación es que el tamaño del punto se multiplica por 1/(constant + linear * dist + quadratic * d^2), por lo que apagarlo es equivalente a (constant = 1, linear = 0, quadratic = 0) y la atenuación de la perspectiva estándar (constant = 0, linear = 1, quadratic = 0). Estos últimos valores son los asumidos cuando se pone estado 'on'.

Hay que tener en cuenta que el tamaño atenuado resultante es anclado en el tamaño mínimo y máximo, consulte la sección siguiente.

Formato: point_size_attenuation on|off [constant linear quadratic]
Por defecto: point_size_attenuation off

point_size_min

Establece el tamaño mínimo después de la atenuación (point_size_attenuation). Para más detalles sobre las mediciones de tamaño, Ver la sección point_size.

Formato: point_size_min <tamaño>
Por defecto: point_size_min 0

point_size_max

Ajusta el tamaño de punto máximo después de la atenuación (point_size_attenuation). Para más detalles sobre las mediciones de tamaño, Ver la sección point_size. Un valor de 0 significa que el máximo que se establece es el mismo que el tamaño máximo reportado por la tarjeta actual.

Formato: point_size_max <tamaño>
Por defecto: point_size_max 0

3.1.3 UNIDADES DE TEXTURA

Éstos son los atributos que puede utilizar en una sección 'texture_unit' de un script .material:

ATRIBUTOS DE TEXTURA DE CAPA DISPONIBLES

- texture_alias
- texture
- anim_texture
- cubic_texture
- tex_coord_set
- tex_address_mode
- tex_border_colour
- filtering
- max_anisotropy
- mipmap_bias
- colour_op
- colour_op_ex
- colour_op_multipass_fallback
- alpha_op_ex
- env_map
- scroll
- scroll_anim
- rotate
- rotate_anim
- scale
- wave_xform
- transform
- binding_type
- content_type

También se puede utilizar una sección de 'texture_source' anidada con el fin de utilizar un complemento especial como una fuente de datos de textura, ver sección 6. Fuentes Externas de la textura para obtener más información.

DESCRIPCIONES DE LOS ATRIBUTOS

texture_alias

Establece el nombre de alias para esta unidad de textura.

Formato: texture_alias <nombre>
Ejemplo: texture_alias NormalMap

Establecer el alias de la textura es útil si este material va a ser heredado por otros materiales y sólo se cambiarán las texturas en el nuevo material. (Ver la sección 3.1.12 Alias de textura)

Por defecto: Si una texture_unit tiene un nombre, entonces el valor predeterminado de texture_alias será el nombre del texture_unit.

texture

Establece el nombre de la imagen de textura estática que esta capa va a usar.

Formato: texture <texturename> [<tipo>] [unlimited | numMipMaps] [alpha] [<FormatoPixel>] [gamma]
Ejemplo: texture funkywall.jpg

Esta opción es mutuamente excluyente con el atributo anim_texture. Hay que tener en cuenta que el archivo de textura no puede incluir espacios en su nombre.

El parámetro 'tipo' permite especificar el tipo de textura para crear - el valor predeterminado es '2d', pero se puede reemplazar este, aquí está la lista completa:

1d

Textura de 1 dimensión, es decir, una textura que tiene sólo 1 píxel de altura. Este tipo de texturas puede ser útil cuando se necesita codificar una función en una textura y usarla como una búsqueda sencilla, tal vez en un programa de fragmentos. Es importante utilizar esta configuración cuando se utiliza un programa de fragmentos que utiliza coordenadas de texturas de 1 dimensión, puesto que GL requiere que se utilice un tipo de textura compatible (D3D permitirá salirse con la suya, pero hay que planearlo para cruzar la compatibilidad). El ancho de la textura debe seguir siendo una potencia de 2 para una mejor compatibilidad y rendimiento.

2d

Es el tipo predeterminado que se supone si se omite, las texturas tienen anchura y altura, los cuales preferentemente deben ser potencias de 2, y si se puede, que sean cuadrados porque estos se verán mejor en la mayoría del hardware. Estos pueden ser direccionados con las coordenadas de textura 2d.

3d

Una textura tridimensional, es decir, textura de volumen. La textura tiene una anchura, altura, las cuales deben ser potencias de 2, y tiene profundidad. Estos pueden ser abordados con las coordenadas de textura 3d es decir, a través de un sombreado de píxeles.

cubic

Esta textura se compone de 6 texturas 2d que se pegan alrededor del interior de un cubo. Pueden direccionarse con las coordenadas de texturas 3d y son útiles para mapas de reflexión cúbica y mapas normales.

La opción 'numMipMaps' permite especificar el número de mipmaps a generar para esta textura. El valor predeterminado es 'unlimited' que significa que se generan mips de tamaño 1x1. Se puede especificar un número fijo (incluso 0). Hay que tener en cuenta que si se utiliza la misma textura en otros scripts de material, el número de mipmaps generados se ajustará a la especificada en la primera texture_unit utilizada para cargar la textura - para ser consistente con su uso.

La opción 'alpha' permite especificar que sólo se cargará un canal (luminence) de textura como alfa, por defecto es la carga en el canal rojo. Esto puede ser útil si se desea utilizar texturas sólo alfa en un pipeline de función fija.

Valor predeterminado: nada

La opción de <FormatoPixel> permite especificar el formato de píxel deseado de la textura a crear, que puede ser diferente del formato de píxel del archivo de textura que se está cargando. Hay que tener en cuenta que el formato de píxel final se verá limitado por las capacidades del hardware de modo que no se puede conseguir exactamente lo que se pide. Las opciones disponibles son:

PF_L8
Formato de píxel de 8-bits, todos los bits de luminosidad.

PF_L16
Formato de píxel de 16-bits, todos los bits de luminosidad.

PF_A8
Formato de píxel de 8-bits, todos los alfa bits.

PF_A4L4
Formato de píxel de 8-bits, 4 bits alfa, 4 bits de luminosidad.

PF_BYTE_LA
Formato de píxel de 2 bytes, 1 byte de luminosidad, 1 byte alfa

PF_R5G6B5
Formato de píxel de 16-bit, 5 bits de color rojo, 6 bits de color verde, 5 bits de color azul.

PF_B5G6R5
Formato de píxel de 16-bit, 5 bits de color azul, 6 bits de color verde, 5 bits de color rojo.

PF_R3G3B2
Formato de píxel de 8-bits, 3 bits de color rojo, 3 bits de color verde, 2 bits de color azul.

PF_A4R4G4B4
Formato de píxel de 16-bits, 4 bits para alfa, rojo, verde y azul.

PF_A1R5G5B5
Formato de píxel de 16-bits, 1 bit para el alfa, 5 bits para rojo, verde y azul.

PF_R8G8B8
Formato de píxel de 24-bits, 8 bits para rojo, verde y azul.

PF_B8G8R8
Formato de píxel de 24-bits, 8 bits para el azul, verde y rojo.

PF_A8R8G8B8
Formato de píxel de 32-bits, 8 bits de alfa, rojo, verde y azul.

PF_A8B8G8R8
Formato de píxel de 32-bits, 8 bits de alfa, azul, verde y rojo.

PF_B8G8R8A8
Formato de píxel de 32-bits, 8 bits para el azul, verde, rojo y alfa.

PF_R8G8B8A8
Formato de píxel de 32-bits, 8 bits para rojo, verde, azul y alfa.

PF_X8R8G8B8
Formato de píxel de 32-bits, 8 bits para el rojo, 8 bits para el verde, 8 bits para el azul como PF_A8R8G8B8, pero alfa será descartado.

PF_X8B8G8R8
Formato de píxel de 32-bits, 8 bits para el azul, 8 bits para el verde, 8 bits para el rojo, como PF_A8B8G8R8, pero alfa será descartado.

PF_A2R10G10B10
Formato de píxel de 32-bits, 2 bits para el alfa de 10 bits para rojo, verde y azul.

PF_A2B10G10R10
Formato de píxel de 32-bits, 2 bits para el alfa de 10 bits para el azul, verde y rojo.

PF_FLOAT16_R
Formato de píxel de 16-bits, 16 bits (float) para el rojo.

PF_FLOAT16_RGB
Formato de píxel de 48-bits, 16 bits (float) para el rojo, 16 bits (float) para el verde, 16 bits (float) para el azul.

PF_FLOAT16_RGBA
Formato de píxel de 64 bits, 16 bits (float) para el rojo, 16 bits (float) para el verde, 16 bits (float) para el azul, 16 bits (float) para el alfa.

PF_FLOAT32_R
Formato de píxel de 16-bits, 16 bits (float) para el rojo.

PF_FLOAT32_RGB
Formato de píxel de 96-bits, 32 bits (float) para el rojo, 32 bits (float) para el verde, 32 bits (float) para el azul.

PF_FLOAT32_RGBA
Formato de píxel de 128-bits, 32 bits (float) para el rojo, 32 bits (float) para el verde, 32 bits (float) para el azul, 32 bits (float) para el alfa.

PF_SHORT_RGBA
Formato de píxel de 64-bits, 16 bits para rojo, verde, azul y alfa.

La opción 'gamma' informa al procesador que desea que el hardware de gráficos realice la corrección gamma en los valores de la textura como son mostrados para el renderizado. Esto sólo es aplicable para texturas que tienen canales de color de 8-bits (ejemplo: PF_R8G8B8). A menudo, 8-bits por textura de canal se almacenan en el espacio gamma a fin de aumentar la precisión de los colores más oscuros (http://en.wikipedia.org/wiki/Gamma_correction), pero esto puede rechazar la mezcla y los cálculos de filtrado, ya que asume los valores de color de espacio lineal. Para el sombreado de la mejor calidad, es posible que se desee habilitar la corrección de gamma de forma que el hardware convierta los valores

de la textura al espacio lineal de forma automática cuando se realiza el muestreo de la textura, de modo que los cálculos en el pipeline se pueden hacer en un espacio de color lineal fiable. Al representar una muestra final de 8 bits por canal, también se puede desear convertir de nuevo al espacio gamma lo que se puede hacer en el shader (aumentando el poder 1/2.2) o se puede permitir la corrección gamma cuando la textura va a ser renderizada o cuando se renderiza la ventana. Hay que tener en cuenta que la opción 'gamma' en la textura se aplica sobre la carga de la textura de modo que se debe especificar de forma coherente si se utiliza esta textura en múltiples lugares.

anim_texture

Establece las imágenes a utilizar en una capa de la textura animada. En este caso, una capa de la textura de animación significa una que tiene varios marcos (frames), cada uno de los cuales es un archivo de imagen por separado. Hay 2 formatos, uno para los nombres de imagen determinados implícitamente, y uno para los nombres de imágenes determinados de forma explícita.

Formato 1 (corto): anim_texture <nombre_base><num_frames><duración>
Ejemplo: anim_texture flame.jpg 5 2.5

De esta forma se crea una capa de la textura de animación compuesta por 5 cuadros nombrados flame_0.jpg, flame_1.jpg, flame_2.jpg, etc. Con una longitud de 2,5 segundos de animación (2fps). Si la duración se establece en 0, entonces no se lleva a cabo la transición automática y los marcos deben ser modificados manualmente en el código.

Formato 2 (largo): anim_texture <frame1><frame2> ... <duración>
Ejemplo: anim_texture flamestart.jpg flamemore.png lastflame.tga moreflame.jpg flameagain.jpg 2.5

Esto establece la misma duración de animación, pero de 5 archivos de imagen por separado con nombre. El primer formato es más conciso, pero el segundo se da si no se puede hacer que las imágenes se ajusten a la norma de nomenclatura requeridas para ello.

Valor predeterminado: nada

cubic_texture

Establece las imágenes utilizadas en una textura cúbica, es decir, una formada por 6 imágenes individuales que constituyen las caras de un cubo. Este tipo de texturas se utilizan para los mapas de reflexión (si el hardware soporta mapas de reflexión cúbicos) o skyboxes (cajas de cielo). Hay 2 formatos, un formato breve que espera los nombres de la imagen en un formato especial, y uno más flexible, aunque con más permisión en el formato de nombrado de texturas arbitrariamente.

Formato 1 (corto): cubic_texture <nombre_base> combinedUVW|separateUV

El nombre_base en este formato es algo así como 'skybox.jpg', y el sistema espera que se le proporcione skybox_fr.jpg, skybox_bk.jpg, skybox_up.jpg, skybox_dn.jpg, skybox_lf.jpg, y skybox_rt.jpg para las caras individuales.

Formato 2 (largo): cubic_texture <frente><atrás><izquierda><derecha><arriba><abajo> separateUV

En este caso, cada cara se especifica de forma explícita, en caso de que no se quiera ajustar a los estándares de nomenclatura de arriba. Sólo se puede utilizar esto para la versión separateUV ya que la versión combinedUVW requiere un nombre de textura único que se asignará a la textura combinada 3D (véase más abajo).

En ambos casos el parámetro final significa lo siguiente:

combinedUVW

Las 6 texturas se combinan en un mapa de textura única 'cubic' (cúbico), que se direcciona utilizando las coordenadas de textura 3D con componentes U, V y W. Necesario para los mapas de reflexión, ya que nunca se sabe qué cara de la caja se va a necesitar. Hay que tener en cuenta que no todas las tarjetas soportan el mapeo de entornos cúbicos.

separateUV

Las 6 texturas están separadas pero son todas referencias en esta capa de textura de forma única. Una textura en un momento está activa (se almacena como 6 marcos), y que se direcciona usando coordenadas estándar 2D UV. Este tipo es bueno para los skyboxes ya que sólo se representa una cara en cada momento y esto da más garantías al soporte hardware en las tarjetas más antiguas.

Valor predeterminado: none

binding_type

Le dice a esta textura la unidad de unión de texturas, o con la unidad de procesamiento de fragmento o de la unidad de procesamiento de vértices (para 3.1.10 Vertex Texture Fetch).

Formato: binding_type vertex|fragment
Por defecto: binding_type fragment

content_type

Le dice a esta unidad de textura dónde debe obtener su contenido. El valor predeterminado es obtener el contenido de la textura de una textura nombrada, tal como se define en los atributos texture, cubic_texture, anim_texture. Sin embargo, también se puede obtener información de otras fuentes de textura automatizadas. Las opciones son:

named

La opción por defecto, esto se deriva el contenido de textura de un nombre de textura, cargado por los medios ordinarios de un archivo o de haber sido creado manualmente con un nombre determinado.

shadow

Esta opción le permite tirar de una textura de sombra, y sólo es válida cuando se utilizan sombras de textura y uno de los tipos de sombreado 'custom sequence' (Ver la sección 7. Sombras). La textura de la sombra en cuestión será la n-esima más cercana a la luz que proyecta las sombras, a menos que se utilice una pasada de iteración basada en luz o la opción light_start que puede indicar el índice inicial de luz superior. Cuando se utiliza esta opción en las unidades de textura múltiples en una misma pasada, cada una referencia a la textura de la sombra siguiente. El índice de textura de la sombra se restablece en el paso siguiente, en caso de que se quiera tener en cuenta las mismas texturas de sombra de nuevo en otra pasada (por ejemplo, una pasada separada especular/brillo). Al utilizar esta opción, la correcta proyección del tronco de luz es creado por el usuario para su uso en una función fijada, si se usan shaders debe de referenciarse el parámetro texture_viewproj_matrix auto en el sombreador.

compositor

Esta opción permite referenciar una textura desde un compositor, y sólo es válida cuando se renderiza la pasada con una secuencia de compositor. Puede ser una directiva render_scene dentro de un script compositor, o en una pasada general, en un puerto de vista (viewport) al que está asociado el compositor. Hay que tener en cuenta que el orden se tiene en cuenta para lo que se va a realizar (por ejemplo, un apilamiento de texturas puede causar que la textura referenciada sea sobreescrita por alguna otra en el momento en que se referencia),

Los parámetros extra del content_type sólo se requieren para este tipo:

El primero es el nombre del compositor referenciado (requerido).

El segundo es el nombre de la textura a referenciar en el compositor (Requerido).

El tercero es el índice de la textura a coger, en el caso de MRT (Opcional).

Formato: content_type named|shadow|compositor [<Nombre Compositor Referenciado>] [<Nombre Textura Referenciada>] [<Índice MRT Referenciado>]
Por defecto: content_type named
Ejemplo: content_type compositor DepthCompositor OutputTexture

tex_coord_set

Establece qué coordenadas de texturas se van a utilizar para esta capa de la textura. Una malla puede definir varios conjuntos de coordenadas de textura, esto establece qué material se utiliza.

Formato: tex_coord_set <número>
Ejemplo: tex_coord_set 2
Por defecto: tex_coord_set 0

tex_address_mode

Define lo que sucede cuando las coordenadas de textura sobrepasan el 1.0 para esta capa. Se puede utilizar formato simple para especificar el modo de direccionamiento de textura de las 3 posibles coordenadas a la vez, o se pueden utilizar los 2/3 parámetros de formato ampliado para especificar un modo diferente por coordenada de textura.

Formato simple: tex_address_mode <uvw_modo>
Formato extendido: tex_address_mode <u_modo><v_modo> [<w_modo>]

wrap
Cualquier valor más allá de 1.0 vuelve de nuevo a 0.0. La textura se repite.

clamp
Los valores más allá de 1.0 se fijan a 1.0. En las texturas de 'rayas' más allá de 1.0, última línea de píxeles se repite en el resto del espacio de direcciones. Útiles para las texturas que necesitan la cobertura exacta del 0.0 a 1.0 sin el 'filo borroso' que provoca wrap cuando se combina con el filtrado.

mirror
La textura invierte todos los límites, es decir, la textura se repite cada 1.0 U o V.

border
Los valores fuera del intervalo [0.0, 1.0] se establecen en el color del borde, también se puede establecer el atributo tex_border_colour.

Por defecto: tex_address_mode wrap

tex_border_colour

Establece el color del borde del modo de direccionamiento de textura de borde (véase el tex_address_mode).

Formato: tex_border_colour <rojo><verde><azul> [<alfa>]
Nota: los valores de color válidos son entre 0.0 y 1.0.
Ejemplo: tex_border_colour 0.0 1.0 0.3
Por defecto: tex_border_colour 0.0 0.0 0.0 1.0

filtering

Establece el tipo de filtrado de textura que se utiliza al aumentar o reducir una textura. Hay 2 formatos para este atributo, el formato simple en el que sólo se tiene que especificar el nombre de un conjunto predefinido de opciones de filtrado; y el formato complejo, en el que de forma individual se establece la reducción, ampliación, y los filtros de MIP propios.

Formato simple
Formato: filtering none|bilinear|trilinear|anisotropic
Por defecto: filtering bilineal

Con este formato, sólo es necesario proporcionar un único parámetro, que es uno de los siguientes:

none
No se realiza filtrado o mipmapping. Esto es equivalente al formato complejo 'filtering point point none'.

bilinear
Cuadro de 2x2, realiza la filtración cuando se produce aumento o reducción de una textura y un mipmap se recoge en la lista, pero no se realiza el filtrado entre los niveles de los mipmaps. Esto es equivalente al formato complejo 'filtering linear linear point'.

trilinear
Cuadro de 2x2, la filtración cuando se produce aumento o reducción de una textura, y los 2 mipmaps más próximo se filtran juntos. Esto es equivalente al formato complejo 'filtering linear linear linear'.

anisotropic
Este es el mismo que 'trilinear', salvo que el algoritmo de filtrado tiene en cuenta la pendiente del triángulo en relación con la cámara en lugar de simplemente hacer un filtro de 2x2 píxeles en todos los casos. Esto hace que los triángulos en ángulos agudos se vean menos difusos. Equivalente al formato complejo 'filtering anisotropic anisotropic linear'. Hay que tener en cuenta que para hacer alguna diferencia, se debe establecer el atributo max_anisotropy también.

Formato Complejo
Formato: filtering <minification ><magnification><mip>
Por defecto: filtering linear linear point

Este formato le un control completo sobre la reducción (minitication), ampliación (magnification), y los filtros mip. Cada parámetro puede ser uno de los siguientes:

none

Nada - sólo una opción válida para el filtro 'mip', ya que esto apaga el mipmapping completamente. La posición más baja para min y mag es 'point'.

point

Escoge el píxel más cercano en modo min o mag. En el modo mip, este recoge mipmap coincidente más cercano.

linear

Filtra una caja de 2x2 píxeles alrededor del más cercano. En el filtro 'mip' esto permite filtrar entre los niveles del mipmap.

anisotropic

Sólo válido para los modos min y mag, hace que el filtro compense la pendiente del espacio de la cámara de los triángulos. Hay que tener en cuenta que para hacer alguna diferencia, se debe establecer el atributo max_anisotropy también.

max_anisotropy

Establece el máximo grado de anisotropía de que el renderizador trata de compensar la hora de filtrar las texturas. El grado de anisotropía es la relación entre la altura del segmento de la textura visible en una región en comparación con el ancho - así por ejemplo, un plano del suelo, que se extiende en la lejanía y por lo tanto la textura de coordenadas verticales cambian mucho más rápido que las horizontales, tiene una mayor anisotropía que un muro que está de frente (tiene una anisotropía de 1 si la línea de visión es perfectamente perpendicular a ella). Se debe establecer el valor max_anisotropy a algo superior a 1 para empezar a compensar; los valores más altos pueden compensar los ángulos más agudos. El valor máximo es determinado por el hardware, pero normalmente es de 8 o 16.

Para que esto se utilice, se tienen que establecer las opciones de filtrado (filtering) de reducción y/o la ampliación de esta textura.

Formato: max_anisotropy <valor>
Por defecto: max_anisotropy 1

mipmap_bias

Establece el valor de desviación aplicado para el cálculo mipmapping, lo que permite alterar la decisión de qué nivel de detalle de textura utilizar para cualquier distancia. El valor de desviación se aplica después del cálculo de la distancia regular, y ajusta el nivel de mipmap en 1 nivel para cada unidad de desviación. Los valores de desviación negativa fuerzan a los niveles mip mayores a ser utilizados, los valores de desviación positiva fuerzan a los niveles mip menores a ser utilizados. La desviación es un valor de punto flotante para que se puedan utilizar valores entre números enteros para realizar así un ajuste fino.

Para que esta opción pueda ser utilizada, el hardware tiene que soportar desviación de mipmap (expuestos a través de la capacidad de renderización del sistema), y el filtrado (filtering) de reducción ha de ser establecido a point o linear.

Formato: mipmap_bias <valor>
Por defecto: mipmap_bias 0

colour_op

Determina cómo el color de esta capa de la textura se combina con la de abajo (o el efecto de iluminación de la geometría si se trata de la primera capa).

Formato: colour_op replace|add|modulate|alpha_blend

Este método es la forma más sencilla de combinar capas de textura, ya que sólo requiere un parámetro, da la mezcla de los tipos más comunes, y configura automáticamente 2 métodos de mezcla: uno para sí solo está disponible un paso de multitextura hardware, y otra para si no, y la mezcla debe ser alcanzada a través de varias pasadas de renderizado. Es, sin embargo, bastante limitada y no expone operaciones de multitexturas más flexibles, simplemente porque estas no pueden ser soportadas de forma automática en el modo alternativo multipasada. Si se desea utilizar las opciones más elegantes, hay que usar colour_op_ex, pero habrá que asegurarse de que estarán disponibles suficientes unidades multitexturas, o de forma explícita, se debe fijar una alternativa utilizando colour_op_multipass_fallback.

replace
Reemplaza todos los colores con textura con ningún ajuste.

add
Añade componentes de color juntos.

modulate
Multiplica los componentes de color juntos.

alpha_blend
Mezcla basada en el alfa de la textura.

Por defecto: colour_op modulate

colour_op_ex

Esta es una versión ampliada del atributo colour_op que permite un control extremadamente detallado sobre la aplicación de mezcla entre la presente y anteriores capas. El hardware multitextura puede aplicar operaciones de fusión más complejas que el mezclado multipasada, pero está limitado al número de unidades de textura que están disponibles en el hardware.

Formato: color_op_ex <operacion><source1><source2> [<manual_factor>] [<manual_colour1>] [<manual_colour2>]
Ejemplo: colour_op_ex add_signed src_manual src_current 0.5

Ver la nota IMPORTANTE a continuación acerca de los problemas entre mulitpasada y multitexturas que la utilización de este método puede crear. Las operaciones de color de textura determinan cómo el color final de la superficie aparece cuando se representa. Las unidades de textura se utilizan para combinar los valores de color de varias fuentes (por ejemplo, el color difuso de la superficie procedente de los cálculos de iluminación, combinado con el color de la textura). Este método permite especificar la 'operación' a utilizar, es decir, el cálculo, como sumas o multiplicaciones, y qué valores se usarán como argumentos, como un valor fijo o un valor de un cálculo previo.

Opciones de Operación

source1
Utiliza source1 sin modificación.

source2
Utiliza source2 sin modificación.

modulate
Multiplica source1 y source2 juntos.

modulate_x2
Multiplica source1 y source2 juntos, luego por 2 (brillo).

modulate_x4
Multiplica source1 y source2 juntos, luego por 4 (brillo).

add
Añadir source1 y source2 juntos.

add_signed
Añadir source1 y source2 luego restar 0.5.

add_smooth
Añadir source1 y source2, restar el producto.

subtract
Restar source2 de source1.

blend_diffuse_alpha
Usa valores alfa interpolados de los vértices a escala source1, a continuación, agregua source2 escalado por (1-alfa).

blend_texture_alpha
Como blend_diffuse_alpha, pero usa el alfa de la textura.

blend_current_alpha
Como blend_diffuse_alpha pero usa el actual alfa de las etapas previas (igual que para la primera capa blend_diffuse_alpha).

blend_manual
Como blend_diffuse_alpha pero usa un valor constante de alfa especificado <manual>.

dotproduct
El punto producto de source1 y source2.

blend_diffuse_colour
Usa valor interpolado de color de los vértices a escala source1, a continuación, agrega source2 escalado por (1-color).

Opciones de source1 y source2

src_current
El color es construido a partir de las etapas anteriores.

src_texture
El color derivado de la textura asignada a esta capa.

src_diffuse
El color difuso interpolado a partir de los vértices (lo mismo que 'src_current' para la primera capa).

src_specular

El color especular interpolado a partir de los vértices.

src_manual
El color manual se especifica al final del comando.

Por ejemplo 'modulate' toma los resultados de color de la capa anterior, y se multiplica con la nueva textura que se aplica. Teniendo en cuenta que los colores son los valores RGB 0.0-1.0 al multiplicarlos darán lugar a los valores en el mismo rango, 'teñidos' por la multiplicación. Nótese sin embargo que una multiplicación normalmente tiene el efecto de oscurecimiento de las texturas - por esta razón no son alentadoras, las operaciones como modulate_x2. Hay que tener en cuenta que debido a las limitaciones en algunas APIs subyacentes (Direct3D incluido), el argumento 'textura' sólo puede ser utilizado como primer argumento, no como segundo.

Hay que tener en cuenta que el último parámetro sólo es necesario si se decide pasar un valor manualmente en la operación. Por lo tanto sólo se tiene que rellenar este si se utiliza la operación 'blend_manual'.

IMPORTANTE: Ogre trata de utilizar el hardware multitextura para mezclar las capas de textura juntas. Sin embargo, si se queda sin unidades de textura (por ejemplo 2 en la GeForce2, 4 en una GeForce3) tiene que recurrir a la renderización multipasada, es decir, renderizar el mismo objeto varias veces con diferentes texturas. Esto es menos eficiente y hay un rango menor de operaciones de mezcla que se pueden realizar. Por esta razón, si se utiliza este método, realmente se debe establecer el atributo colour_op_multipass_fallback para especificar a qué efecto que desea recurrir si el hardware suficiente no está disponible (el valor predeterminado es 'modulate', que es poco probable que sea lo que se quiere si se está haciendo una mezcla elegante). Si se desea no tener que hacer esto, se puede utilizar el atributo colour_op simple, que permite opciones de fusión menos flexibles, pero que establece una alternativa multipasada de forma automática, ya que sólo permite las operaciones que tienen equivalentes multipasada directos.

Por defecto: nada (colour_op modulate)

colour_op_multipass_fallback

Establece una operación alternativa multipasada para esta capa, si se ha utilizado colour_op_ex y no hay hardware multitexturas disponible.

Formato: colour_op_multipass_fallback <src_factor><dest_factor>
Ejemplo: colour_op_mulitpass_fallback one one_minus_dest_alpha

Debido a que algunos de los efectos que se pueden crear utilizando colour_op_ex sólo son compatibles bajo hardware multitexturas, si el hardware falta, el sistema debe buscar una alternativa en la renderización multipasada, que lamentablemente no es compatible con muchos efectos. Este atributo sirve para que se pueda especificar la operación alternativa que más le convenga.

Los parámetros son los mismos que en el atributo scene_blend; esto se debe a que la renderización es efectivamente la mezcla de la escena, ya que cada capa se renderiza en la parte superior de la última, usando el mismo mecanismo que para hacer un objeto transparente, se renderiza varias veces en el mismo lugar para obtener el efecto multitextura. Si se utiliza el atributo más simple (y menos flexible) colour_op, no es necesario llamar a esto ya que el sistema establece la alternativa automáticamente.

alpha_op_ex

Se comporta de la misma forma que colour_op_ex salvo que determina cómo se combinan los valores de alfa entre las capas de textura en lugar de los valores de color. La única diferencia es que los 2 colores manuales al final de colour_op_ex son valores simples de punto flotante en alpha_op_ex.

env_map

Enciende/apaga efectos de coordenadas de textura que hace que esta capa sea un mapa del entorno.

Formato: env_map off|spherical|planar|cubic_reflection|cubic_normal

Mapas del entorno hacen que un objeto parezca reflejado mediante el uso de la generación de coordenadas de texturas automáticas en función de la relación entre los vértices de los objetos o normales y el ojo.

spherical
: Un mapa de entorno esférico. Requiere una textura única que puede ser una lente de ojo de pez de la escena reflejada, o alguna otra textura que se vea bien como un mapa esférico (una textura brillante de los relieves es popular sobre todo en los simuladores de coches). Este efecto se basa en la relación entre la dirección de la mirada y las normales de vértice del objeto, por lo que funciona mejor cuando hay un montón de cambios normales, es decir, objetos curvos.

planar
: Similar al mapa de entorno esférico, pero el efecto se basa en la posición de los vértices en el visor en lugar de las normales de vértice. Este efecto es por lo tanto útil para la geometría plana (donde un env_map spherical no se vería bien porque los normales son todos iguales) o los objetos sin curvaturas.

cubic_reflection
: Es una forma más avanzada de la cartografía de reflejo que utiliza un grupo de 6 texturas que componen el interior de un cubo, cada una de ellas es una vista de la escena por cada eje. Funciona muy bien en todos los casos, pero tiene un requisito técnico superior de la tarjeta que el mapeo esférico. Requiere que se enlace un cubic_texture a esta unidad de textura y usar la opción 'combinedUVW'.

cubic_normal
: Genera coordenadas de textura 3D que contiene el vector de espacio normal de la cámara de la información normal tomada en los datos de vértice. Una vez más, la plena utilización de esta funcionalidad requiere una cubic_texture con la opción 'combinedUVW'.

Por defecto: env_map off

scroll

Establece un desplazamiento compensado fijo para la textura.

Formato: scroll <x><y>

Este método compensa la textura de esta capa en una cantidad fija. Útil para pequeños ajustes sin alterar las coordenadas de textura en los modelos. Sin embargo, si se desea tener un efecto de desplazamiento animado, ver el atributo scroll_anim.

scroll_anim

Configura un desplazamiento animado para la capa de textura. Útil para la creación de efectos de desplazamiento de velocidad fija en una capa de textura (para diferentes velocidades de desplazamiento, véase wave_xform).

Formato: scroll_anim <xspeed><yspeed>

rotate

Rota una textura un ángulo fijo. Este atributo cambia la orientación de rotación de una textura en un ángulo fijo, útil para los ajustes fijos. Si desea animar la rotación, ver rotate_anim.

Formato: rotate <ángulo>

El parámetro <ángulo> es un ángulo en grados en sentido antihorario.

rotate_anim

Crea un efecto de rotación de animación de esta capa. Útil para crear animaciones de rotación de velocidad fija (para diferentes velocidades, ver wave_xform).

Formato: rotate_anim <revs_por_segundo>

El parámetro es un número de revoluciones por segundo en sentido anti horario.

scale

Ajusta el factor de escala aplicado a esta capa de textura. Útil para ajustar el tamaño de las texturas sin hacer cambios en la geometría. Este es un factor de escala fija, si se desea animar, ver wave_xform.

Formato: ecale <x_scale><y_scale>

Los valores válidos de escala, son los mayores que 0, con un factor de escala de 2 toma la textura el doble de grande que esa dimensión, etc.

wave_xform

Establece una animación de transformación basada en una función de onda. Útil para efectos de transformación de capa de textura avanzados. Se pueden agregar varias instancias de este atributo a una única capa de textura, si así se desea.

Formato: wave_xform <xform_type><wave_type><base><frequency><phase><amplitude>
Ejemplo: wave_xform scale_x sine 1.0 0.2 0.0 5.0

xform_type

scroll_x
Anima el valor de desplazamiento x.

scroll_y
Anima el valor de desplazamiento y.

rotate
Anima el valor de rotación.

scale_x
Anima el valor de escala x.

scale_y
Anima el valor de escala y.

wave_type

sine
Una onda senoidal típica con bucles suaves entre los valores mínimo y máximo.

triangle
Una ola angulada que aumenta y disminuye a una velocidad constante, con un cambio inmediato en los extremos.

square
Máximo para la mitad de la longitud de onda, mínimo para el resto con la transición inmediata.

sawtooth
Constante de aumento gradual del mínimo al máximo en el período con un retorno inmediato del mínimo al final.

inverse_sawtooth
Constante de disminución gradual de máximo a mínimo durante el período, con un retorno inmediato del máximo al final.

base
El valor base, el mínimo si la amplitud (amplitude)> 0, el máximo si amplitud < 0.

frequency
El número de iteraciones de onda por segundo, es decir, la velocidad.

phase
Desplazamiento de la salida de onda.

amplitude
El tamaño de la onda.

El rango de la salida de la onda será {base, base + amplitude}. Así que en el ejemplo anterior, se escala la textura en la dirección x entre 1 (tamaño normal) y 5 a lo largo de una onda senoidal en un ciclo cada 5 segundos (0.2 ondas por segundo).

transform

Este atributo permite especificar una matriz de transformación estática 4x4 para la unidad de textura, sustituyendo así el desplazamiento individual, rotación y escala de los atributos antes mencionados.

Formato: transform m00 m01 m02 m03 m10 m11 m12 m13 m20 m21 m22 m23 m30 m31 m32 m33

Los índices de valor de la matriz 4x4 se expresan en forma de m<fila><columna>.

3.1.4 DECLARACIÓN DE PROGRAMAS DE VÉRTICES/GEOMETRÍAS/FRAGMENTOS

Para utilizar programas de vértices, geometría o de fragmento en materiales (vér la sección 3.1.9 Uso de programas de Vertices/Geometría/Fragmentos en una pasada), primero hay que definirlos. Una

definición de un programa puede ser utilizada por cualquier número de materiales, el único requisito es que el programa debe ser definido antes de ser referenciado en la sección de la pasada de un material.

La definición de un programa puede ser embebido en el script .material (en cuyo caso debe preceder a cualquier referencia a él en la secuencia de comandos), o si se desea utilizar el mismo programa a través de múltiples archivos .material, se puede definir en un archivo de script .program. Se define el programa de la misma forma que un script .program o un script .material, la única diferencia es que todos los scripts .program se garantiza que se analizan antes que **todos** los scripts .material, por lo que se puede garantizar que el programa se ha definido antes que cualquier script .material que pueda usarlo. Igual que los scripts .material, los scripts .program serán leídos desde cualquier lugar que se encuentra en la ruta de acceso de recursos, y se pueden definir muchos programas en un único script.

Los programas de Vértices, geometría y fragmentos pueden ser de bajo nivel (es decir, código ensamblador escrito para la especificación de la sintaxis de bajo nivel dada como vs_1_1 o arbfp1) o de alto nivel como DirectX9 HLSL, Open GL Shader Language o lenguaje Cg de nVidia (Ver la sección programas de alto nivel). Los lenguajes de alto nivel darán un número de ventajas, como ser capaz de escribir código más intuitivo, y la posibilidad de usar múltiples arquitecturas en un único programa (por ejemplo, el mismo programa Cg puede ser utilizado tanto en D3D como GL, mientras que el programa de bajo nivel equivalente requiere técnicas independientes, cada uno para una API diferente). Los programas de alto nivel también permiten utilizar parámetros con nombre en lugar de simplemente los indexados, aunque los parámetros no se definen aquí, se utilizan en la pasada (Pass).

He aquí un ejemplo de una definición de un programa de bajo nivel de vértice:

```
vertex_program myVertexProgram asm
{
    source myVertexProgram.asm
    syntax vs_1_1
}
```

Código 4: Ejemplo programa de bajo nivel.

Como se puede ver, es muy simple, y la definición de un programa de fragmento o de geometría es exactamente lo mismo, sólo con reemplazar vertex_program por fragment_program o geometry_program, respectivamente. Hay que dar al programa un nombre en el encabezado, seguido de la palabra 'asm' para indicar que este es un programa de bajo nivel. Entre las llaves, se especifica que la fuente va a venir a partir de un fichero (y esto se carga de cualquiera de las ubicaciones de los recursos como con otros medios de comunicación), e indicará también la sintaxis que se utiliza. Se podría preguntar por qué la especificación de sintaxis es necesario cuando muchas de las sintaxis de ensamblador tienen una cabecera de identificación - y la razón es que el motor necesita saber qué sintaxis del programa es antes de leerlo, porque durante la compilación del material, se desea saltar rápidamente programas que utilizan una sintaxis insostenible, sin cargar el programa.

Las sintaxis soportadas actualmente son:

vs_1_1

Esta es una de las sintaxis de ensamblador DirectX vertex shader.

Compatible con las tarjetas: ATI Radeon 8500, nVidia GeForce 3.

vs_2_0

Otra de las sintaxis de ensamblador DirectX vertex shader.

Compatible con las tarjetas: ATI Radeon 9600, nVidia GeForce FX serie 5.

vs_2_x

Otra de las sintaxis de ensamblador DirectX vertex shader.

Compatible con las tarjetas: ATI Radeon serie X, nVidia GeForce FX serie 6.

vs_3_0

Otra de las sintaxis de ensamblador DirectX vertex shader.

Compatible con las tarjetas: nVidia GeForce FX Serie 6.

arbvp1

Este es el formato de ensamblador estándar OpenGL para los programas de vértice. Es más o menos equivalente a DirectX vs_1_1.

vp20

Esta es una sintaxis específico de nVidia OpenGL vertex shader, que es un súper conjunto de vs 1.1.

vp30

Otra sintaxis específica de nVidia OpenGL vertex shader. Se trata de un súperconjunto de vs 2.0, que es compatible con nVidia GeForce FX serie 5 o superior.

vp40

Otra sintaxis específica de nVidia OpenGL vertex shader. Se trata de un súperconjunto de vs 3.0, que es compatible con nVidia GeForce FX series 6 o superior.

ps_1_1, ps_1_2, ps_1_3

Sintaxis de ensamblador DirectX pixel shader (es decir, programa de fragmentos).

Compatible con las tarjetas: ATI Radeon 8500, nVidia GeForce 3.

NOTA: para hardware ATI 8500, 9000, 9100, 9200, este perfil también se puede utilizar en OpenGL. De la ATI 8500 a 9200 no son compatibles con arbfp1 pero soportan la extensión atifs en OpenGL que es muy similar en función a ps_1_4 en DirectX. Ogre se ha construido en ps_1_x al compilador atifs que se invoca automáticamente cuando ps_1_x se utiliza en OpenGL en el hardware de ATI.

ps_1_4

Sintaxis de ensamblador DirectX pixel shader (es decir, programa de fragmentos).

Soportado en las tarjetas de: ATI Radeon 8500, nVidia GeForce FX 5 series

NOTA: para ATI 8500, 9000, 9100, 9200, este perfil también se puede utilizar en OpenGL. Las ATI 8500 a 9200 no son compatibles con arbfp1 pero se soporta la extensión atifs en OpenGL que es muy similar en función a ps_1_4 en DirectX. Ogre se ha construido en ps_1_x al compilador atifs que se invoca automáticamente cuando ps_1_x se utiliza en OpenGL en el hardware de ATI.

ps_2_0

Sintaxis de ensamblador DirectX pixel shader (es decir, programa de fragmentos).

Tarjetas compatibles: ATI Radeon 9600, nVidia GeForce FX 5 series.

ps_2_x

Sintaxis de ensamblador DirectX pixel shader (es decir, programa de fragmentos). Esto es básicamente ps_2_0 con un mayor número de instrucciones.

Tarjetas compatibles: ATI Radeon serie X, nVidia GeForce FX Serie 6.

ps_3_0

Sintaxis de ensamblador DirectX pixel shader (es decir, programa de fragmentos).

Tarjetas compatibles: NVIDIA GeForce serie FX6.

ps_3_x

Sintaxis de ensamblador DirectX pixel shader (es decir, programa de fragmentos).

Tarjetas compatibles: NVIDIA GeForce serie FX7.

arbfp1

Este es el formato de ensamblador estándar OpenGL para los programas de fragmento. Es más o menos equivalente a ps_2_0, lo que significa que no todas las tarjetas que soportan los sombreadores de píxeles básicos soportan DirectX arbfp1 (por ejemplo, ni GeForce3 o GeForce4 soportan arbfp1, pero sí soportan ps_1_1).

fp20

Esta es una sintaxis de fragmento específica de nVidia de OpenGL que es un súperconjunto de ps 1.3. Permite utilizar el formato de 'nvparse' para los programas de fragmento básicos. En realidad, usa NV_texture_shader y NV_register_combiners para proporcionar una funcionalidad equivalente a ps_1_1 de DirectX conforme GL, pero sólo para las tarjetas nVidia. Sin embargo, ya que las tarjetas ATI adoptaron arbfp1 un poco antes que nVidia, se trata principalmente para las tarjetas nVidia como la GeForce3 y GeForce4. Se puede encontrar más información acerca de nvparse en http://developer.nvidia.com/object/nvparse.html.

fp30

Otra sintaxis específica de nVidia, OpenGL fragment shader. Se trata de un súperconjunto de ps 2.0, que es compatible con nVidia GeForce FX serie 5 o superior.

fp40

Otra sintaxis específica de nVidia OpenGL fragment shader. Se trata de un súperconjunto de ps 3.0, que es compatible con nVidia GeForce FX serie 6 y superior.

gpu_gp, gp4_gp

Una sintaxis específica de nVidia OpenGL fragment shader.

Tarjetas compatibles: NVIDIA GeForce serie FX8.

Se puede obtener una lista definitiva de las sintaxis compatibles con la tarjeta actual llamando a GpuProgramManager::getSingleton().getSupportedSyntax().

Especificación de constantes con nombre para shaders ensamblador

Los sombreadores (shaders) ensamblador no tienen constantes nombradas (también llamados parámetros uniformes) porque el lenguaje no es compatible con ellos - sin embargo, si por ejemplo, se precompilan shaders de un lenguaje de alto nivel hasta ensamblador para el rendimiento o la oscuridad, es posible que se desee utilizar los parámetros con nombre. Bueno, en realidad se puede – GpuNamedConstants contiene mapas de parámetros con nombre que tiene un método 'save' que se

puede utilizar para escribir el dato a disco, donde se puede hacer referencia más adelante con la directiva manual_named_constants dentro de la declaración del programa ensamblado, por ejemplo:

```
vertex_program myVertexProgram asm
{
    source myVertexProgram.asm
    syntax vs_1_1
    manual_named_constants myVertexProgram.constants
}
```

Código 5: Programa de vértice con nombrado.

En este caso myVertexProgram.constants ha sido creado llamando a highLevelGpuProgram-> getNamedConstants().save("myVertexProgram.constants"); en algún momento a principios de la preparación, desde el programa original de alto nivel. Una vez que se ha utilizado esta directiva, se pueden utilizar parámetros con nombre, aunque el programa ensamblador en sí no tiene conocimiento de ellos.

Parámetros por defecto del programa

Mientras se define un programa de vértice, geometría o fragmento, también se pueden especificar los parámetros por defecto que se utilizarán para los materiales que utilizarán, a menos que se sobrescriban ellos específicamente. Esto se realiza mediante la inclusión de una sección anidada 'default_params', así:

```
vertex_program Ogre/CelShadingVP cg
{
        source Example_CelShading.cg
        entry_point main_vp
        profiles vs_1_1 arbvp1

        default_params
        {
        param_named_auto lightPosition light_position_object_space 0
        param_named_auto eyePosition camera_position_object_space
                param_named_auto worldViewProj worldviewproj_matrix
                param_named shininess float 10
        }
}
```

Código 6: Programa de vértice con parámetros por defecto.

La sintaxis de la definición de parámetros es exactamente la misma que cuando se definen los parámetros cuando se utilizan programas, Ver especificación de parámetros de programa. Definir parámetros por defecto permite evitar volver a enlazar parámetros comunes en varias ocasiones (claramente en el ejemplo anterior, 'shininess' es poco probable que cambie entre los usos del programa) que hace más cortas las declaraciones de materiales.

Declarando parámetros compartidos

A menudo, cada parámetro que se quiere pasar a un shader no es único para el programa, y quizás se desee dar el mismo valor a un número diferente de programas, y un número diferente de materiales usando esos programas. Los juegos de parámetros compartidos se pueden definir en un 'area de contención' para parámetros compartidos que pueden ser referenciados cuando se necesiten en

shaders particulares, mientras que mantiene la definición del valor en un lugar. Para definir un juego de parámetros compartidos, se hace lo siguiente:

```
shared_params YourSharedParamsName
{
        shared_param_named mySharedParam1 float4 0.1 0.2 0.3 0.4
        ...
}
```

Código 7: Definiendo parámetros compartidos.

Como se puede ver, se necesita utilizar la palabra clave 'shared_params' seguido del nombre que se utilizará para identificar ese parámetro compartido. Dentro de las llaves, se puede definir un parámetro por línea, con una sintaxis similar al nombrado de parámetros (param_named). La definición de esas líneas es:

Formato: shared_param_name <nombre_param><tipo_param> [<[tam_array]>][<valores_iniciales>]

El nombre_param debe de ser único en el juego, y el tipo_param puede ser float, float2, float3, float4, int, int2, int3, int4, matrix2x2, matrix2x3, matrix2x4, matrix3x2, matrix3x3, matrix3x4, matrix4x2, matrix4x3 y matrix4x4. La opción tam_array permite definir arrays de tipos de parámetros cómo se desee, y si está presente debe ser un número entre corchetes (y tiene que estar separado de tipo_param por un espacio en blanco). Si se desea, se pueden inicializar los parámetros aportando una lista de valores.

Una vez que se han definido los parámetros compartidos, se pueden referenciar dentro de los bloques default_params y params usando shared_param_ref. También se puede obtener una referencia a ellos en el código a través de GpuProgramManager::getSharedParameters, y actualizar los valores para todas las instancias que los usan.

PROGRAMAS DE ALTO NIVEL

La compatibilidad con programas de vértice de alto nivel y programas de fragmento es proporcionada a través de plugins, esto es para asegurarse de que una aplicación que utiliza Ogre puede utilizar mucho o poco de la funcionalidad del programa de alto nivel como se quiera. Ogre actualmente soporta 3 tipos de programas de alto nivel, Cg (3.1.5 programas Cg) (un API o tarjeta independiente, lenguaje de alto nivel que permite escribir programas para OpenGL y DirectX para muchas tarjetas), lenguaje de sombreado de alto nivel DirectX 9 (3.1.6 DirectX9 HLSL), y Lenguaje de sombreado de OpenGL (3.1.7 OpenGL GLSL). HLSL sólo puede utilizarse con el rendersystem de DirectX, y GLSL sólo puede utilizarse con el rendersystem GL. Cg se puede utilizarse con ambos, aunque la experiencia ha demostrado que los programas más avanzados, en particular los programas de fragmento que realizan una gran cantidad de texturas, pueden producir un mejor código en el sombreador específico rendersystem del lenguaje.

Una forma de soportar tanto HLSL como GLSL es incluir técnicas distintas en el script de material, cada una de ellas hace referencia a programas separados. Sin embargo, si los programas son básicamente los mismos, con los mismos parámetros, y las técnicas son complejas esto puede incrementar las secuencias de comandos con duplicación de material y bastante rapidez. En cambio, si la única diferencia es el idioma de los programas de los vértices y fragmentos se puede utilizar Ogre. 3.1.8 Unificado de programas de alto nivel para recoger de forma automática un programa adecuado para su rendersystem usando una técnica única.

Animación esquelética en programas de vértices

Se puede aplicar animación esquelética en hardware escribiendo un programa de vértices que use índices de fusión por vértice y mezcla de pesos, junto con una serie de matrices del mundo (que será facilitado por Ogre si se enlaza el parámetro automático 'world_matrix_array_3x4'). Sin embargo, es necesario comunicar este soporte a Ogre para que no se realice la animación del esqueleto en el software para el usuario. Para ello, hay añadir el siguiente atributo a la definición del vertex_program:

```
includes_skeletal_animation true
```

Código 8: Animación esquelética en un programa de vértice.

Al hacer esto, cualquier entidad esquelética animada que utilice este material renunciará a la mezcla de animación habitual y espera que el programa de vértice lo haga, tanto para posiciones de los vértices como de normales. Hay que tener en cuenta que TODAS las submallas deben tener asignado un material que implemente esto, y que si se combina la animación del esqueleto con la animación de vértices (Véase la sección 8. Animación), todas las técnicas deben ser aceleradas por hardware.

Animación morfológica en programas de vértices

Se puede implementar una animación morfológica en hardware escribiendo un programa de vértice que combina linealmente entre los fotogramas clave primero y segundo pasados como posiciones y el primer juego de coordenadas libres de texturas, y envolviendo el valor animation_parametric a un parámetro (que indica la distancia para interpolar entre los dos). Sin embargo, es necesario comunicar este soporte a Ogre para que no se realice animación metamórfica en software. Para ello, hay que añadir el siguiente atributo a la definición del vertex_program:

```
includes_morph_animation true
```

Código 9: Animación morfológica en un programa de vértice.

Al hacer esto, cualquier entidad con animación morfológica que utilice este material renunciará a la metamorfosis de software habitual y esperará que el programa de vértice lo haga. Hay que tener en cuenta que si el modelo incluye la animación esquelética y animación metamórfica, ambas deben llevarse a cabo en el programa de vértices, si ninguno se hace con aceleración de hardware. Hay que tener en cuenta que TODAS las submallas deben tener asignado un material que implemente el programa, y si se combina animación del esqueleto con la animación de vértices (Véase la sección 8. Animación), todas las técnicas deben ser aceleradas por hardware.

Animación de posturas en programas de vértices

Se puede implementar animación de postura (mezcla entre varias posturas basadas en el peso) en un programa de vértice tirando de los datos de vértices originales (vinculados a la posición), y como hay muchos buffers de posturas que son definidos en la declaración 'includes_pose_animation', estará en la primera unidad libre de textura hacia arriba. También se debe de utilizar el parámetro animation_parametric para definir el punto de partida de las constantes que contendrán los pesos de las posturas; se iniciará en el parámetro definido y se llenarán las 'n' constantes, donde 'n' es el número máximo de poses que este sombreador puede mezclar, es decir, el parámetro a includes_pose_animation.

```
includes_pose_animation 4
```

Código 10: Animación de posturas en programas de vértice.

Hay que tener en cuenta que TODAS las submallas deben tener asignado un material al que se aplica este programa, y que si se combina animación del esqueleto con la animación de vértices (Véase la sección 8. Animación), todas las técnicas deben ser aceleradas por hardware.

Atracción de textura de vértices en los programas de vértice

Si el programa de vértice hace uso de 3.1.10 Atracción de textura de vétices, se debe declarar con la directiva 'uses_vertex_texture_fetch'. Esto es suficiente para decir a Ogre que el programa utiliza esta característica y que lacompatibilidad del hardware para ello debe estar comprobada.

```
uses_vertex_texture_fetch true
```

Código 11: Atracción de textura en programas de vértice.

Información de adyacencia en programas de geometría

Algunos programas de geometría requieren información de adyacencia de la geometría. Esto significa que un shader (sombreador) de geometría no sólo obtiene la información de la primitiva que opera, también tiene acceso a sus vecinos (en el caso de líneas o triángulos). Esta directiva indicará a Ogre que envíe la información a los shaders de geometría.

```
uses_adjacency_information true
```

Código 12: Información de adyacencia en programas de geometría.

Programas de vértices con sombras

Cuando se usan sombras (véase la sección 7. Sombras), el uso de programas de vértice puede añadir cierta complejidad adicional, ya que Ogre sólo puede ocuparse automáticamente de todo lo que utiliza un pipeline de función fija. Si se utilizan programas de vértice, y también se están utilizando las sombras, puede que se necesiten hacer algunos ajustes.

Si se utilizan **sombras plantilla**, entonces cualquier programa vértice que haga una deformación de vértices puede ser un problema, porque las sombras plantilla se calculan en la CPU, la cual no tiene acceso a los vértices modificados. Si el programa vértice está haciendo animación del esqueleto normal, esto es aceptable (véase la sección anterior), ya que Ogre sabe cómo reproducir el efecto en el software, pero cualquier deformación de los otros vértices no puede repetirse, y se tendrá que aceptar que la sombra no reflejará esta deformación, o se deben desactivar la sombras para ese objeto.

Si se utilizan **sombras de textura**, entonces la deformación de vértices es aceptable, sin embargo, al representar el objeto en una textura de sombra (la pasada caster de sombra), la sombra ha de ser dictada en un color sólido (unido con el color ambiental para sombras modulativas, negro para las sombras aditivas). Por lo tanto, se debe proporcionar un programa de vértice alternativo, para lo que Ogre ofrece una manera de especificarlo que se debe usar cuando se representa un caster, véase la sección programas de sombras y vértices.

3.1.5 PROGRAMAS DE CG

Con el fin de definir programas de Cg, se tiene que cargar el Plugin_CgProgramManager.so/.dll en el arranque, ya sea a través de plugins.cfg o a través de código propio de carga del plugin. Son muy fáciles de definir:

```
fragment_program myCgFragmentProgram cg
{
    source myCgFragmentProgram.cg
    entry_point main
    profiles ps_2_0 arbfp1
}
```

Código 13: Definición de programa de CG.

Existen algunas diferencias entre éste y el programa ensamblador - para empezar, se declara que el programa fragmento es de tipo 'cg' en lugar de 'asm', que indica que se trata de un programa de alto nivel que utiliza Cg. El parámetro 'source' es el mismo, excepto que esta vez se trata de una fuente que referencia un Cg en lugar de un archivo de ensamblador.

Aquí es donde las cosas comienzan a cambiar. En primer lugar, se tiene que definir un 'entry_point', que es el nombre de una función en el programa Cg, que será la primera parte llamada como parte del programa de fragmento. A diferencia de los programas en ensamblador, que funcionan de arriba a abajo, los programas de Cg pueden incluir múltiples funciones y, como tal, se debe especificar la que comenzará a rodar la pelota.

A continuación, en lugar de un parámetro 'sintaxis' fijo, se especifica uno o varios 'profiles', los perfiles son cómo Cg compila un programa hasta el ensamblador de bajo nivel. Los perfiles tienen el mismo nombre que el código sintáctico ensamblador mencionado anteriormente, la principal diferencia es que se puede enumerar más de uno, lo que permite que el programa sea compilado a más bajo nivel de sintaxis para que se pueda escribir un único programa de nivel alto que funciona tanto en D3D como GL. Se aconseja que se introduzcan los perfiles más simples en los programas que pueden ser compilados con el fin de darle la máxima compatibilidad. El orden también es importante, si una tarjeta es compatible con más de una sintaxis a continuación, la que aparece primero será la utilizada.

Por último, hay una última opción llamada 'compile_arguments', donde se pueden especificar los argumentos exactamente igual a como se haría con el compilador de línea de comandos cgc, si se desea.

3.1.6 DIRECTX9 HLSL

DirectX9 HLSL tiene una sintaxis muy similar al lenguaje Cg, pero está ligado a la API DirectX. El único beneficio sobre Cg es que sólo requiere el plugin de sistema de renderizado de DirectX 9, sin complementos adicionales. La declaración de un programa de HLSL DirectX9 es muy similar a Cg. He aquí un ejemplo:

```
vertex_program myHLSLVertexProgram hlsl
{
    source myHLSLVertexProgram.txt
    entry_point main
    target vs_2_0
}
```

Código 14: Declaración programa DirectX9 HLSL.

Como se puede ver, la sintaxis principal es casi idéntica, salvo que en lugar de 'profiles' con una lista de formatos de ensamblador, se tiene un parámetro 'target' que permite especificar un objetivo ensamblador único - obviamente, esto tiene que ser un código sintáctico ensamblador de DirectX.

Nota importante en la ordenación de la matriz: Una cosa a tener en cuenta es que HLSL permite utilizar 2 maneras diferentes para multiplicar un vector por una matriz - mul (v, m) o mul (m, v). La única diferencia entre ellos es que la matriz es efectivamente transpuesta. Se debe usar mul (m, v) con las matrices pasadas desde Ogre - esto está así de acuerdo con los shaders producidos a partir de herramientas como RenderMonkey, y es coherente con Cg también, pero no está de acuerdo con el SDK de DX9 y FX Composer que utilizan mul (v, m) – se tendrá que cambiar los parámetros para mul () en estos shaders.

Hay que tener en cuenta que si se utilizan los tipos float3x4/matrix3x4 en el sombreador, se está obligando a Ogre a una auto definición (como las matrices de los huesos), se debe utilizar la opción column_major_matrices = false (explicado abajo) en la definición del programa. Esto se debe a que Ogre pasa float3x4 con filas principales para ahorrar espacio de constante (3 float4 en lugar de 4 float4 ya que sólo los 3 primeros valores se utilizan) y esto indica a Ogre que pase todas las matrices de este tipo, de modo que se puede usar mul (m, v) de forma sistemática en todos los cálculos. Ogre también manda al shader para compilar en la fila (no se tiene que configurar la opción de compilación /Zpr o la opción #pragma pack (fila-principal), Ogre hace esto automáticamente).Hay que tener en cuenta que los huesos que pasa en formato float4x3 no son compatibles con Ogre, pero no es necesario dadas las anotaciones anteriores.

Opciones avanzadas

preprocessor_defines <defines>

Esto permite definir los símbolos que se pueden utilizar dentro del código de sombreado HLSL para modificar el comportamiento (a través de cláusulas #ifdef o #if). Las definiciones están separadas por ';' o ',' y, opcionalmente, puede tener un operador '=' dentro de ellos para especificar un valor de definición. Los que no tienen un '=' implícitamente tienen una definición de 1.

column_major_matrices true|false

El valor predeterminado para esta opción es 'true' de manera que Ogre pasa matrices auto determinadas de una forma que mul (m, v) funciona. Al establecer esta opción a false hace 2 cosas – adapta las matrices 4x4 auto-envolventes y también establece la opción /Zpr (fila-principal) en la compilación de sombreado. Esto significa que todavía se puede usar mul (m, v), pero el diseño de la matriz es con filas primero. Esto sólo es útil si se necesitan usar las matrices de hueso (float3x4) en un shader, ya que guarda una constante float4 para cada hueso afectado.

Optimisation_level <opt>

Establece el nivel de optimización, que puede ser uno de los siguientes: 'default', 'none', '0', '1', '2', o '3'. Esto se corresponde con el parámetro /O de fxc.exe, excepto que en el modo 'default', la optimización se deshabilita en el modo de pruebas (debug) y se establece a 1 en el modo de lanzamiento (release) (fxc.exe utiliza 1 todo el tiempo). Como es lógico, el modo por defecto es 'default'. Se puede querer cambiar esto si se quiere rebajar la optimización, por ejemplo, si el shader es demasiado complejo, no compilará si no se establece un nivel mínimo de optimización.

3.1.7 OPENGL GLSL

GLSL OpenGL tiene una sintaxis de lenguaje similar al HLSL pero está vinculada a la API de OpenGL. Tiene algunas ventajas sobre Cg ya que sólo requiere el plugin del sistema de renderizado de OpenGL, sin complementos adicionales. Declarar un programa de GLSL OpenGL es similar a Cg, pero más sencillo. He aquí un ejemplo:

```
vertex_program myGLSLVertexProgram glsl
{
    source myGLSLVertexProgram.txt
}
```

Código 15: Declaración programa OpenGL GLSL.

En GLSL, no necesita definirse ningún punto de entrada, ya que es siempre 'main()' y no hay definición de objetivos ya que la fuente GLSL está compilada a código nativo GPU y no es ensamblado directamente.

GLSL soporta el uso de shaders modulares. Esto significa que se pueden escribir funciones GLSL externas que pueden ser utilizadas en múltiples shaders.

```
vertex_program myExteranalGLSLFunction1 glsl
{
    source myExternalGLSLfunction1.txt
}

vertex_program myExteranalGLSLFunction2 glsl
{
    source myExternalGLSLfunction2.txt
}

vertex_program myGLSLVertexProgram1 glsl
{
    source myGLSLfunction.txt
    attach myExteranalGLSLFunction1 myExteranalGLSLFunction2
}

vertex_program myGLSLVertexProgram2 glsl
{
    source myGLSLfunction.txt
    attach myExteranalGLSLFunction1
}
```

Código 16: Declaración de funciones GLSL externas.

Las funciones externas GLSL se adjuntan al programa que las necesita mediante 'attach' e incluyendo los nombres de todos los programas externos necesarios en la misma línea separados por espacios. Esto se puede hacer tanto para programas de vértice como para los programas de fragmentos.

Muestreadores de texturas GLSL

Para pasar los valores de índice de unidades de textura de la secuencia de comandos de material a los muestreadores de la textura en GLSL el usa el tipo de parámetros nombrados 'int'. Véase el siguiente ejemplo: extracto de la fuente GLSL example.frag:

```
varying vec2 UV;
uniform sampler2D diffuseMap;

void main(void)
{
        gl_FragColor = texture2D(diffuseMap, UV);
}
```

Código 17: Paso de muestreadores de textura GLSL.

En el script de material:

```
fragment_program myFragmentShader glsl
{
  source example.frag
}

material exampleGLSLTexturing
{
  technique
  {
    pass
    {
      fragment_program_ref myFragmentShader
      {
        param_named diffuseMap int 0
      }

      texture_unit
      {
        texture myTexture.jpg 2d
      }
    }
  }
}
```

Código 18: Material con texturación GLSL.

Un valor índice de 0 hace referencia a la primera unidad de textura en la pasada, un valor índice de 1 hace referencia a la segunda unidad en el pase y así sucesivamente.

Parámetros de la matriz

Éstos son algunos ejemplos de pasar matrices a GLSL mat2, mat3, mat4 uniformes:

```
material exampleGLSLmatixUniforms
{
  technique matrix_passing
```

```
  {
pass examples
    {
      vertex_program_ref myVertexShader
      {
        // mat4 uniform
        param_named OcclusionMatrix matrix4x4 1 0 0 0  0 1 0 0  0 0 1
0  0 0 0 0
        // or
        param_named ViewMatrix float16 0 1 0 0  0 0 1 0  0 0 0 1  0 0
0 0

        // mat3
        param_named TextRotMatrix float9 1 0 0  0 1 0  0 0 1
      }

      fragment_program_ref myFragmentShader
      {
        // mat2 uniform
param_named skewMatrix float4 0.5 0 -0.5 1.0
      }

    }
  }
}
```

Código 19: Ejemplo paso de matrices a GLSL.

Acceso a los estados de OpenGL en GLSL

GLSL puede acceder a la mayoría de los estados GL directamente por lo que no se necesita pasar estos estados a través de param_named_auto en el script de material. Esto incluye luces, estado material, y todas las matrices utilizadas en la matriz de OpenGL, es decir, el estado de vista del modelo, la matriz de proyección de la visión del mundo, etc.

Enlazando atributos vértice

GLSL soporta de forma nativa el enlace automático de los atributos más comunes de entrada por vértice (por ejemplo, gl_Vertex, gl_Normal, gl_MultiTexCoord0, etc.) Sin embargo, hay algunos que no están ligados automáticamente, por lo que debe declararse en el shader utilizando la sintaxis 'attribute <tipo><nombre>', y los datos de vértices ligados a ella por Ogre.

Además de la construcción de los atributos descritos en la sección 7.3 del manual de GLSL, Ogre soporta una serie de atributos vértice automáticamente ligados a atributos de vértice personalizados. Hay algunos controladores que no se comportan correctamente cuando se mezclan atributos incorporados como gl_Normal y atributos de vértice personalizados, por lo que para una máxima compatibilidad, se deben utilizar todos los atributos personalizados en shaders donde se necesita al menos uno (por ejemplo, para la animación esquelética).

vertex
: Se liga VES_POSITION, declarado como 'attribute vec4 vertex;'.

normal
: Se liga VES_NORMAL, declarado como 'attribute vec3 normal;'.

colour
Se liga VES_DIFFUSE, declarado como 'attribute vec4 colour;'.

secondary_colour
Se liga VES_SPECULAR, declarado como 'attribute vec4 secondary_colour;'.

uv0 - uv7
Se liga VES_TEXTURE_COORDINATES, declarado como 'attribute vec4 uv0;'. Hay que tener en cuenta que uv6 y uv7 comparten atributos con la tangente y binormal, respectivamente, por lo que no pueden estar presentes.

tangent
Se liga VES_TANGENT, declarado como 'attribute vec3 tangent;'.

binormal
Se liga VES_BINORMAL, declarado como 'attribute vec3 binormal;'.

blendIndices
Se liga VES_BLEND_INDICES, declarado como 'attribute vec4 blendIndices;'.

blendWeights
Se liga VES_BLEND_WEIGHTS, declarado como 'attribute vec4 blendWeights;'.

Definiciones de preprocesador

GLSL admite el uso de definiciones de preprocesador en el código - algunas son definidas por la implementación, pero también se puede definir una propia, por ejemplo para usar el mismo código fuente para algunas variantes diferentes de la misma técnica. Para poder utilizar esta función, hay que incluir condiciones de preprocesador en el código GLSL, del tipo #ifdef SYMBOL, #if SYMBOL == 2, etc. Entonces en la definición del programa, se utiliza la opción 'preprocessor_defines', siguiéndolo con una cadena de las definiciones condicionales. Las definiciones están separadas por ';' o ',' y, opcionalmente, pueden tener un operador '=' dentro de ellos para especificar un valor de definición. Los que no tienen un '=' implícitamente tienen una definición de 1. Por ejemplo:

```
// in your GLSL

#ifdef CLEVERTECHNIQUE
        // some clever stuff here
#else
        // normal technique
#endif

#if NUM_THINGS==2
        // Some specific code
#else
        // something else
#endif

// in  your program definition
preprocessor_defines CLEVERTECHNIQUE,NUMTHINGS=2
```

Código 20: Definiciones de preprocesador en GLSL.

De esta manera se puede utilizar el mismo código fuente, pero incluyendo pequeñas variaciones, cada una de ellas está definida con un nombre de programa Ogre diferente, pero basadas en el mismo código fuente.

Especificación del shader de geometría GLSL

GLSL permite que el mismo shader funcione en diferentes tipos primitivos de geometría. Con el fin de enlazar adecuadamente los shaders juntos, se tiene que especificar qué primitivos recibirá como entrada, qué primitivos se emiten y cuántos vértices por ejecución puede generar el shader. La definición GLSL geometry_program requiere tres parámetros adicionales:

input_operation_type
El tipo de operación de la geometría que el shader va a recibir. Puede ser 'point_list', 'line_list', 'line_strip', 'triangle_list', 'triangle_strip' o 'triangle_fan'.

output_operation_type
El tipo de operación de la geometría que el shader emite. Puede ser 'point_list', 'line_strip' o 'triangle_strip'.

max_output_vertices
El número máximo de vértices que el shader puede emitir. Hay un límite máximo para este valor, este se expone en las capacidades del sistema de renderizado. Por ejemplo:

```
geometry_program Ogre/GPTest/Swizzle_GP_GLSL glsl
{
        source SwizzleGP.glsl
        input_operation_type triangle_list
        output_operation_type line_strip
        max_output_vertices 6
}
```

Código 21: Definición de shader de geometría GLSL.

3.1.8 PROGRAMAS DE ALTO NIVEL UNIFICADOS

Como se mencionó anteriormente, a menudo puede ser útil para escribir tanto programas HLSL como programas GLSL centrarse específicamente en cada plataforma, pero si se hace a través de múltiples técnicas del material, puede causar una definición de material hinchado cuando la única diferencia es el idioma del programa. Hay otra opción. Se pueden 'envolver' múltiples programas en una definición de programa 'unificada', que automáticamente tendrá que elegir uno de una serie de los programas 'delegados' en función del sistema de renderizado y soporte de hardware.

```
vertex_program myVertexProgram unified
{
        delegate realProgram1
        delegate realProgram2
        ... etc
}
```

Código 22: Definición de programa unificado.

Esto funciona tanto para los programas de vértices como para los programas de fragmento, y se pueden enumerar tantas delegaciones como se quiera - el primer programa compatible por el sistema de renderizado actual y el hardware será utilizado como el programa real. Esto es casi como un sistema mini-técnica, pero para un programa único y con un propósito mucho más estricto. Sólo se puede utilizar éste donde los programas toman las mismas entradas, en particular, texturas y otros estados de pasadas/ejemplos. Cuando la única diferencia entre los programas es el lenguaje (o, posiblemente, el objetivo en HLSL – se pueden incluir varios programas HLSL con diferentes objetivos en un programa unificado también si se desea, o cualquier número de otros programas de alto nivel), esta puede

convertirse en una característica muy poderosa. Por ejemplo, sin esta característica, he aquí cómo habría que definir un material programable soportado por HLSL y GLSL:

```
vertex_program myVertexProgramHLSL hlsl
{
        source prog.hlsl
        entry_point main_vp
        target vs_2_0
}
fragment_program myFragmentProgramHLSL hlsl
{
        source prog.hlsl
        entry_point main_fp
        target ps_2_0
}
vertex_program myVertexProgramGLSL glsl
{
        source prog.vert
}
fragment_program myFragmentProgramGLSL glsl
{
        source prog.frag
        default_params
        {
                param_named tex int 0
        }
}
material SupportHLSLandGLSLwithoutUnified
{
        // HLSL technique
        technique
        {
                pass
                {
                        vertex_program_ref myVertexProgramHLSL
                        {
                                param_named_auto worldViewProj
world_view_proj_matrix
                                param_named_auto lightColour
light_diffuse_colour 0
                                param_named_auto lightSpecular
light_specular_colour 0
                                param_named_auto lightAtten
light_attenuation 0
                        }
                        fragment_program_ref myFragmentProgramHLSL
                        {
                        }
                }
        }
        // GLSL technique
        technique
        {
                pass
                {
                        vertex_program_ref myVertexProgramHLSL
                        {
                                param_named_auto worldViewProj
world_view_proj_matrix
                                param_named_auto lightColour
light_diffuse_colour 0
```

```
                                param_named_auto lightSpecular
light_specular_colour 0
                                param_named_auto lightAtten
light_attenuation 0
                        }
                        fragment_program_ref myFragmentProgramHLSL
                        {
                        }
                }
        }
}
```

Código 23: Definición de material programable soportado por HLSL y GLSL.

Y esto es un ejemplo muy pequeño. Todo lo que se añada a la técnica HLSL, tiene que duplicarse en la técnica de GLSL también. Aquí mostramos cómo hacerlo con las definiciones de programa unificado:

```
vertex_program myVertexProgramHLSL hlsl
{
        source prog.hlsl
        entry_point main_vp
        target vs_2_0
}
fragment_program myFragmentProgramHLSL hlsl
{
        source prog.hlsl
        entry_point main_fp
        target ps_2_0
}
vertex_program myVertexProgramGLSL glsl
{
        source prog.vert
}
fragment_program myFragmentProgramGLSL glsl
{
        source prog.frag
        default_params
        {
                param_named tex int 0
        }
}
// Unified definition
vertex_program myVertexProgram unified
{
        delegate myVertexProgramGLSL
        delegate myVertexProgramHLSL
}
fragment_program myFragmentProgram unified
{
        delegate myFragmentProgramGLSL
        delegate myFragmentProgramHLSL
}
material SupportHLSLandGLSLwithUnified
{
        // HLSL technique
        technique
        {
                pass
                {
                        vertex_program_ref myVertexProgram
                        {
```

```
                                param_named_auto worldViewProj
world_view_proj_matrix
                                param_named_auto lightColour
light_diffuse_colour 0
                                param_named_auto lightSpecular
light_specular_colour 0
                                param_named_auto lightAtten
light_attenuation 0
                        }
                        fragment_program_ref myFragmentProgram
                        {
                        }
                }
        }
}
```

Código 24: Definición de material con unificaciones.

En tiempo de ejecución, cuando se utilizan myVertexProgram o myFragmentProgram, Ogre selecciona automáticamente un programa real para delegar basado en lo que es compatible con el hardware/rendersystem actual. Si no se admite ninguno de los delegados, la referencia técnica de todo el programa unificado se marca como no compatible y se pasa a comprobar la siguiente técnica en el material, al igual que normalmente. Como los materiales se hacen más grandes, y se considera que se necesita específicamente soporte HLSL y GLSL (o hay necesidad de escribir con versiones múltiples de interfaces compatibles de un programa por cualquier otra razón), los programas unificados realmente pueden ayudar a reducir la duplicación.

3.1.9 USANDO PROGRAMAS DE VÉRTICE/GEOMETRÍA/FRAGMENTO EN UNA PASADA

Dentro de la sección de una pasada en un script de material, se puede hacer referencia a un programa de vértice, geometría y/o fragmento que se ha definido en un script .program (véase la sección 3.1.4 Declaración de programas de vértice/geometría/fragmento). Los programas se definen por separado para usarlos en la pasada, ya que los programas son muy propensos a ser reutilizados entre muchos materiales por separado, probablemente a través de muchos scripts .material diferentes, así que este método permite definir el programa de una sola vez y utilizarlo muchas veces.

Así como nombrar el programa en cuestión, también se puede proporcionar parámetros al mismo. He aquí un ejemplo sencillo:

```
vertex_program_ref myVertexProgram
{
        param_indexed_auto 0 worldviewproj_matrix
        param_indexed      4 float4  10.0 0 0 0
}
```

Código 25: Declaración programa de vértices.

En este ejemplo se declara un programa de vértices llamado 'myVertexProgram' (que será definido en otro lugar) para la pasada, y se le pasan 2 parámetros, uno es un parámetro 'auto', que significa que no tiene que suministrar un valor como tal, sólo un código reconocido (en este caso es la matriz de mundo/vista/proyección que se mantiene actualizada automáticamente por Ogre). El segundo parámetro es un parámetro especificado manualmente, un elemento float4. Los índices se describen más adelante.

La sintaxis del enlace a un programa de vértices y un programa de fragmentos o geometría son idénticos, la única diferencia es que 'fragment_program_ref' y 'geometry_program_ref' se utilizan, respectivamente, en lugar de 'vertex_program_ref'.

Para muchas situaciones los programas de vértices, geometría y fragmentos se asocian entre sí en una pasada, pero esto no es inamovible. Se podría tener un programa de vértices que puede ser utilizado por varios programas de fragmentos diferentes. Otra situación que se plantea es que se pueden mezclar pipelines fijas y pipelines programables (shaders) juntas. Se podrían utilizar las pipelines no-programables de funciones fijas de vertices y luego ofrecer un fragment_program_ref en una pasada, es decir, no habría sección vertex_program_ref en la pasada. El programa de fragmento referenciado en la pasada debe cumplir los requisitos definidos en el API relacionados con el fin de leer los resultados de los vértices de pipelines fijas. También se puede tener un programa de vértices que devuelve un pipeline de funciones fijas de fragmentos.

Los requisitos para leer o escribir en el pipeline de funciones fija son similares entre la cada API de renderizado (DirectX y OpenGL), pero cómo está hecho actualmente cada tipo de shader (vértice, geometría o fragmento) depende del lenguaje de sombreado. Para HLSL (API DirectX) y asm asociados consultar MSDN en http://msdn.microsoft.com/library/. Para GLSL (OpenGL), consulte la sección 7.6 de la especificación de GLSL 1.1 disponible en http://developer.3dlabs.com/documents/index.htm. Construir en diferentes variables previstas en GLSL permite que un programa pueda leer/escribir en diferentes pipelines de función fija. Para Cg se puede consultar la sección de perfiles de lenguaje en CgUsersManual.pdf que viene con el Toolkit de Cg disponible en http://developer.nvidia.com/object/cg_toolkit.html. Para HLSL y Cg es la variación de definiciones lo que permite a los diversos programas de sombreado para leer/escribir en variaciones de pipeline de función fija.

ESPECIFICACIÓN DE PARÁMETROS

Los parámetros pueden espedificarse utilizando uno de los 4 comandos que se muestran a continuación. Se utiliza la misma sintaxis si se está definiendo un parámetro para este uso particular del programa, o cuando se especifican los parámetros predeterminados del programa. Los parámetros establecidos en el uso específico del programa sobrescriben los valores predeterminados.

- param_indexed
- param_indexed_auto
- param_named
- param_named_auto

param_indexed

Este comando establece el valor de un parámetro indexado.

Formato: param_indexed <índice><tipo><valor>
Ejemplo: param_indexed 0 float4 10.0 0 0 0

El 'índice' es simplemente un número que representa la posición en la lista de parámetros en el que el valor debe ser escrito, y se debe obtener de la definición del programa. El índice es relativo a la forma en que las constantes se almacenan en la tarjeta, que es en bloques de 4 elementos. Por ejemplo, si se ha definido un parámetro float4 en el índice 0, el índice siguiente sería 1. Si ha definido una matrix4x4 en el índice 0, el índice utilizable siguiente sería 4, ya que una matriz de 4x4 ocupa 4 índices.

El valor de 'tipo' puede ser float4, matrix4x4, float<n>, int4, int<n>. Hay que tener en cuenta los parámetros 'int' están disponibles sólo en algunas sintaxis de programas más avanzados, hay que comprobar la documentación de D3D o GL en programas de fragmento para más detalles. Por lo general los más útiles serán float4 y matrix4x4. Teniendo en cuenta que si se utiliza un tipo que no sea múltiplo de 4, los valores que resten hasta el múltiplo de 4 se completarán con ceros (las GPUs siempre utilizan los bancos de 4 floats por constante, incluso si sólo se usa uno).

'valor' es simplemente un espacio o una lista delimitada por tabuladores de valores que se pueden convertir en el tipo que se ha especificado.

param_indexed_auto

Este comando indica a Ogre que actualice automáticamente un parámetro dado con un valor derivado. Esto libera de la escritura de código para actualizar los parámetros del programa cuando todos los fotogramas están siempre cambiando.

Formato: param_indexed_auto <índice><código_valor><params_extra>
Ejemplo: param_indexed_auto 0 worldviewproj_matrix

'índice' tiene el mismo significado que en param_indexed; nota: en este momento no se tiene que especificar el tamaño del parámetro porque el motor ya lo sabe. En el ejemplo, la matriz world/view/projection se utiliza implícitamente como una matrix4x4.

'código_valor' es uno de una siguiente lista de valores reconocidos:

world_matrix
La matriz actual del mundo.

inverse_world_matrix
La inversa de la matriz actual del mundo.

transpose_world_matrix
La transposición de la matriz del mundo.

inverse_transpose_world_matrix
La transposición inversa de la matriz del mundo.

world_matrix_array_3x4
Un array de matrices del mundo, cada una representada por sólo una matriz de 3x4 (3 filas de 4 columnas) por lo general para hacer sacrificio de hardware. Se deben hacer suficientes entradas disponibles en el programa de vértices para el número de huesos en uso, es decir, una serie de numBones * 3 float4.

view_matrix
La matriz de vista actual.

inverse_view_matrix
La inversa de la matriz de vista actual.

transpose_view_matrix
La transposición de la matriz de vista.

inverse_transpose_view_matrix
La transposición de la matriz inversa de la vista.

projection_matrix
La matriz de proyección actual.

inverse_projection_matrix
La inversa de la matriz de proyección.

transpose_projection_matrix
La transpuesta de la matriz de proyección.

inverse_transpose_projection_matrix
La transposición inversa de la matriz de proyección.

worldview_matrix
Las matrices del mundo actual y de vista concatenadas.

inverse_worldview_matrix
El inverso de las matrices del mundo actual y de vista concatenadas.

transpose_worldview_matrix
El transpuesto de las matrices del mundo y de vista.

inverse_transpose_worldview_matrix
La transposición inversa de las matrices del mundo actual y de vista concatenadas.

viewproj_matrix
Las matrices de visión actual y de proyección concatenadas.

inverse_viewproj_matrix
La inversa de las matrices de vista y de proyección.

transpose_viewproj_matrix
La transposición de las matrices de visión y proyección.

inverse_transpose_viewproj_matrix
La transposición inversa de las matrices de vista y proyección.

worldviewproj_matrix
Las matrices del mundo actual, vista y proyección concatenadas.

inverse_worldviewproj_matrix
El inverso de las matrices del mundo, visión y proyección.

transpose_worldviewproj_matrix
La transposición de las matrices del mundo, visión y proyección.

inverse_transpose_worldviewproj_matrix
La transposición inversa de las matrices del mundo, visión y proyección.

texture_matrix
La matriz transformada de una unidad de textura determinada, que normalmente se ve en el pipeline de función fija. Esto requiere un índice en el campo 'params_extra', y se refiere a la 'n esíma' unidad de textura de la pasada en cuestión. Nota: si el índice dado excede el número de unidades de textura para esta pasada, entonces el parámetro se establece en Matrix4::IDENTITY.

render_target_flipping

El valor usado para ajustar la posición y transformada si se evita la transformación de la matriz de proyección. Es -1 si el objetivo del renderizado requiere voltear la textura, +1 de la otra manera.

light_diffuse_colour

El color difuso de una luz dada, lo que exige un índice en el campo 'params_extra', y se refiere a la 'enésima' luz más cercana que pudiera afectar a este objeto (es decir, 0 se refiere a la luz más cercana - las luces direccionales son siempre las primeras en la lista y siempre están presentes). Nota: si no hay luces cerca, a continuación, el parámetro se establece en negro.

light_specular_colour

El color especular de la luz dada, exige un índice en el campo 'params_extra', y se refiere a la 'enésima' luz más cercana que pudiera afectar a este objeto (es decir, 0 se refiere a la luz más cercana). Nota: si no hay luces cerca, entonces, el parámetro se establece en negro.

light_attenuation

Un float4 que contiene las 4 variables de atenuación de luz de una luz determinada. Esto requiere un índice en el campo 'params_extra', y se refiere a la 'enésima' luz más cercana que pudiera afectar a este objeto (es decir, 0 se refiere a la luz más cercana). Nota: si no hay luces cerca, entonces, el parámetro se establece en todas las variables a cero. El orden de los parámetros es: rango, atenuación constante, atenuación lineal, atenuación cuádrica.

spotlight_params

Un float4 que contiene los 3 parámetros de foco y un valor de control. El orden de los parámetros es coseno (ángulo interno / 2), coseno (ángulo externo / 2), caída, y el valor final de w es 1.0f. Para objetos sin foco el valor es float4 (1,0,0,1). Esto requiere un índice en el campo 'params_extra', y se refiere a la 'enésima' luz más cercana que pudiera afectar a este objeto (es decir, 0 se refiere a la luz más cercana). Si hay menos luces que este, los detalles son como sin foco.

light_position

La posición de la luz en el espacio mundial. Esto requiere un índice en el campo 'params_extra', y se refiere a la 'enésima' luz más cercana que pudiera afectar a este objeto (es decir, 0 se refiere a la luz más cercana). Nota: si no hay luces tan cerca, entonces, el parámetro se establece a ceros. Hay que tener en cuenta que esta característica funciona con todo tipo de luces, incluso las luces direccionales, ya que el parámetro se establece como un vector 4D. Los puntos de luces serán (pos.x, pos.y, pos.z, 1.0f), mientras que las luces direccionales serán (-dir.x,-dir.y,-dir.z, 0.0f). Operaciones como productos punto funcionarán de forma coherente en ambos casos.

light_direction

La dirección de la luz en el espacio mundial. Esto requiere un índice en el campo 'params_extra', y se refiere a la 'enésima' luz más cercana que pudiera afectar a este objeto (es decir, 0 se refiere a la luz más cercana). Nota: si no hay luces tan cerca, entonces, el parámetro se establece en a ceros. OBSOLETA - esta característica sólo funciona con luces direccionales, y se recomienda que se utilice light_position en su lugar ya que devuelve un vector genérico 4D.

light_position_object_space

La posición de la luz en el espacio objeto (es decir, cuando el objeto está en (0,0,0)). Esto requiere un índice en el campo 'params_extra', y se refiere a la 'enésima' luz más cercana que pudiera afectar a este objeto (es decir, 0 se refiere a la luz más cercana). Nota: si no hay luces tan cerca, entonces, el parámetro se establece a ceros. Hay que tener en cuenta que esta

característica funciona con todo tipo de luces, incluso las luces direccionales, ya que el parámetro se establece como un vector 4D. Los puntos de luces serán (pos.x, pos.y, pos.z, 1.0f), mientras que las luces direccionales serán (-dir.x,-dir.y,-dir.z, 0.0f). Operaciones como productos punto funcionarán de forma coherente en ambos casos.

light_direction_object_space

La dirección de la luz en el espacio objeto (es decir, cuando el objeto está en (0,0,0)). Esto requiere un índice en el campo 'params_extra', y se refiere a la 'enésima' luz más cercana que pudiera afectar a este objeto (es decir, 0 se refiere a la luz más cercana). Nota: si no hay luces tan cerca, entonces, el parámetro se establece a ceros. OBSOLETA, a excepción de focos - para las luces direccionales, se recomienda que se utilice light_position_object_space ya que devuelve un vector genérico 4D.

light_distance_object_space

La distancia de la luz emitida desde el centro del objeto - esto es una aproximación útil para los cálculos de la distancia de vértice de objetos relativamente pequeños. Esto requiere un índice en el campo 'params_extra', y se refiere a la 'enésima' luz más cercana que pudiera afectar a este objeto (es decir, 0 se refiere a la luz más cercana). Nota: si no hay luces tan cerca, entonces, el parámetro se establece a ceros.

light_position_view_space

La posición de la luz en el espacio de vista (es decir, cuando la cámara está en (0,0,0)). Esto requiere un índice en el campo 'params_extra', y se refiere a la 'enésima' luz más cercana que pudiera afectar a este objeto (es decir, 0 se refiere a la luz más cercana). Nota: si no hay luces tan cerca, entonces, el parámetro se establece a ceros. Hay que tener en cuenta que esta característica funciona con todo tipo de luces, incluso las luces direccionales, ya que el parámetro se establece como un vector 4D. Los puntos de luces serán (pos.x, pos.y, pos.z, 1.0f), mientras que las luces direccionales serán (-dir.x,-dir.y,-dir.z, 0.0f). Operaciones como productos punto funcionarán de forma coherente en ambos casos.

light_direction_view_space

La dirección de la luz en el espacio de vista (es decir, cuando la cámara está en (0,0,0)). Esto requiere un índice en el campo 'params_extra', y se refiere a la 'enésima' luz más cercana que pudiera afectar a este objeto (es decir, 0 se refiere a la luz más cercana). Nota: si no hay luces tan cerca, entonces, el parámetro se establece a ceros. OBSOLETA, a excepción de focos - para las luces direccionales, se recomienda que se utilice light_position_view_space ya que devuelve un vector genérico 4D.

light_power

La escala de 'potencia' para una luz determinada, útil en la prestación de HDR. Esto requiere un índice en el campo 'params_extra', y se refiere a la 'enésima' luz más cercana que pudiera afectar a este objeto (es decir, 0 se refiere a la luz más cercana).

light_diffuse_colour_power_scaled

Como light_diffuse_colour, excepto que los canales RGB del color de la pasada han sido pre-escalados por la escala de potencia de la luz dada por la light_power.

light_specular_colour_power_scaled

Como light_specular_colour, excepto que los canales RGB del color de la pasada han sido pre-escalados por la escala de potencia de la luz dada por la light_power.

light_number

Al renderizar, hay generalmente una lista de las luces disponibles para su uso por todas las pasadas de un objeto dado, y las luces pueden o no ser referenciadas en una o más pasadas. A veces puede ser útil saber dónde está una luz en la lista general de luz (como se ve desde una pasada). Por ejemplo, si se utiliza iteración once_per_light, la pasada siempre ve la luz como índice 0, pero en cada iteración la luz actual referenciada es diferente. Este enlace permite pasar a través del índice actual de la luz en la lista general. Sólo se tiene que dar el parámetro del número de luz relativa a la pasada y se trazará un mapa al índice general de la lista.

light_diffuse_colour_array

Es como light_diffuse_colour, excepto que esta rellena un array de parámetros con un número de luces, y el campo 'params_extra' se refiere al número de la luz 'enésima' más cercana para ser procesada. Este parámetro no es compatible con las opciones de pass_iteration basadas en luz pero puede ser utilizado para una sola pasada de iluminación.

light_specular_colour_array

Es como light_specular_colour, excepto que esta rellena un array de parámetros con un número de luces, y el campo 'params_extra' se refiere al número de la luz 'enésima' más cercana para ser procesada. Este parámetro no es compatible con las opciones de pass_iteration basadas en luz pero puede ser utilizado para una sola pasada de iluminación.

light_diffuse_colour_power_scaled_array

Es como light_diffuse_colour_power_scaled, excepto que esta rellena un array de parámetros con un número de luces, y el campo 'params_extra' se refiere al número de la luz 'enésima' más cercana para ser procesada. Este parámetro no es compatible con las opciones de pass_iteration basadas en luz pero puede ser utilizado para una sola pasada de iluminación.

light_specular_colour_power_scaled_array

Como light_specular_colour_power_scaled, excepto que esta rellena un array de parámetros con un número de luces, y el campo 'params_extra' se refiere al número de la luz 'enésima' más cercana para ser procesada. Este parámetro no es compatible con las opciones de pass_iteration basadas en luz pero puede ser utilizado para una sola pasada de iluminación.

light_attenuation_array

Como light_attenuation, excepto que esta rellena un array de parámetros con un número de luces, y el campo 'params_extra' se refiere al número de la luz 'enésima' más cercana para ser procesada. Este parámetro no es compatible con las opciones de pass_iteration basadas en luz pero puede ser utilizado para una sola pasada de iluminación.

spotlight_params_array

Como spotlight_params, excepto que esta rellena un array de parámetros con un número de luces, y el campo 'params_extra' se refiere al número de la luz 'enésima' más cercana para ser procesada. Este parámetro no es compatible con las opciones de pass_iteration basadas en luz pero puede ser utilizado para una sola pasada de iluminación.

light_position_array

Como light_position, excepto que esta rellena un array de parámetros con un número de luces, y el campo 'params_extra' se refiere al número de la luz 'enésima' más cercana para ser procesada. Este parámetro no es compatible con las opciones de pass_iteration basadas en luz pero puede ser utilizado para una sola pasada de iluminación.

light_direction_array

Como light_direction, excepto que esta rellena un array de parámetros con un número de luces, y el campo 'params_extra' se refiere al número de la luz 'enésima' más cercana para ser

procesada. Este parámetro no es compatible con las opciones de pass_iteration basadas en luz pero puede ser utilizado para una sola pasada de iluminación.

light_position_object_space_array
Como light_position_object_space, excepto que esta rellena un array de parámetros con un número de luces, y el campo 'params_extra' se refiere al número de la luz 'enésima' más cercana para ser procesada. Este parámetro no es compatible con las opciones de pass_iteration basadas en luz pero puede ser utilizado para una sola pasada de iluminación.

light_direction_object_space_array
Como light_direction_object_space, excepto que esta rellena un array de parámetros con un número de luces, y el campo 'params_extra' se refiere al número de la luz 'enésima' más cercana para ser procesada. Este parámetro no es compatible con las opciones de pass_iteration basadas en luz pero puede ser utilizado para una sola pasada de iluminación.

light_distance_object_space_array
Como light_distance_object_space, excepto que esta rellena un array de parámetros con un número de luces, y el campo 'params_extra' se refiere al número de la luz 'enésima' más cercana para ser procesada. Este parámetro no es compatible con las opciones de pass_iteration basadas en luz pero puede ser utilizado para una sola pasada de iluminación.

light_position_view_space_array
Como light_position_view_space, excepto que esta rellena un array de parámetros con un número de luces, y el campo 'params_extra' se refiere al número de la luz 'enésima' más cercana para ser procesada. Este parámetro no es compatible con las opciones de pass_iteration basadas en luz pero puede ser utilizado para una sola pasada de iluminación.

light_direction_view_space_array
Como light_direction_view_space, excepto que esta rellena un array de parámetros con un número de luces, y el campo 'params_extra' se refiere al número de la luz 'enésima' más cercana para ser procesada. Este parámetro no es compatible con las opciones de pass_iteration basadas en luz pero puede ser utilizado para una sola pasada de iluminación.

light_power_array
Como light_power, excepto que esta rellena un array de parámetros con un número de luces, y el campo 'params_extra' se refiere al número de la luz 'enésima' más cercana para ser procesada. Este parámetro no es compatible con las opciones de pass_iteration basadas en luz pero puede ser utilizado para una sola pasada de iluminación.

light_count
El número total de luces activas en esta pasada.

light_casts_shadows
Establece un parámetro entero a 1 si la luz proyecta sombras, 0 en caso contrario, requiere un parámetro índice de la luz.

ambient_light_colour
El color de la luz ambiental fijada actualmente en la escena.

surface_ambient_colour
Las propiedades reflectantes del color ambiente de la pasada (ver sección ambiente). Esto permite el acceso a la función fija de propiedad de pipeline fácilmente.

surface_diffuse_colour

Las propiedades reflectantes del color difuso de la pasada (ver sección difusa). Esto permite el acceso a la función fija de propiedad de pipeline fácilmente.

surface_specular_colour
Las propiedades reflectantes de color especular de la pasada (ver la sección especular). Esto permite el acceso a la función fija de propiedad de pipeline fácilmente.

surface_emissive_colour
La cantidad de la auto-iluminación de la pasada (ver la sección emisivo). Esto permite el acceso a la función fija de propiedad de pipeline fácilmente.

surface_shininess
El brillo de la pasada, que afecta el tamaño de las luces especulares (Ver la sección especular). Esto permite unirse a la función fija de propiedad de pipeline fácilmente.

derived_ambient_light_colour
El color de la luz ambiente derivado, con componentes 'r', 'g', 'b' con productos de la surface_ambient_colour y ambient_light_colour, respectivamente, y un componente 'a' con el componente alpha de la superficie ambiente.

derived_scene_colour
El color de la escena derivados, con componentes 'r', 'g' y 'b' con la suma de derived_ambient_light_colour y surface_emissive_colour, respectivamente, y un componente 'a' con el componente alpha de la superficie difusa.

derived_light_diffuse_colour
El color difuso de la luz derivada, con componentes 'r', 'g' y 'b' con productos de la surface_diffuse_colour, light_diffuse_colour y light_power, respectivamente, y un componente 'a' con el componente alfa difuso de la superficie. Esto requiere un índice en el campo 'params_extra', y se refiere a la 'enésima' luz más cercana que pudiera afectar a este objeto (es decir, 0 se refiere a la luz más cercana).

derived_light_specular_colour
El color especular de la luz derivada, con componentes 'r', 'g' y 'b' con productos de la surface_specular_colour y light_specular_colour, respectivamente, y un componente 'a' con el componente alpha especular de la superficie. Esto requiere un índice en el campo 'params_extra', y se refiere a la 'enésima' luz más cercana que pudiera afectar a este objeto (es decir, 0 se refiere a la luz más cercana).

derived_light_diffuse_colour_array
Como derived_light_diffuse_colour, excepto que esta rellena un array de parámetros con un número de luces, y el campo 'params_extra' se refiere al número de la luz 'enésima' más cercana para ser procesada. Este parámetro no es compatible con las opciones de pass_iteration basadas en luz pero puede ser utilizado para una sola pasada de iluminación.

derived_light_specular_colour_array
Como derived_light_specular_colour, excepto que esta rellena un array de parámetros con un número de luces, y el campo 'params_extra' se refiere al número de la luz 'enésima' más cercana para ser procesada. Este parámetro no es compatible con las opciones de pass_iteration basadas en luz pero puede ser utilizado para una sola pasada de iluminación.

fog_colour
El color de la niebla actualmente fijado en la escena.

fog_params
Los parámetros de la niebla actualmente fijados en la escena. Empaquetados como (exp_density, linear_start, linear_end, 1.0 / (linear_end - linear_start)).

camera_position
La posición actual de las cámaras en el espacio mundial.

camera_position_object_space
La posición actual de las cámaras en el espacio objeto (es decir, cuando el objeto está en (0,0,0)).

lod_camera_position
La posición actual de la cámara LOD en el espacio mundial. Una cámara de LOD es una cámara independiente asociada con la cámara de representación que permite que los cálculos LOD sean calculados por separado. El ejemplo clásico es basar el LOD del renderizado de textura de sombra sobre la posición de la cámara principal, en lugar de la cámara de sombra.

lod_camera_position_object_space
La actual posición de la cámara LOD en el espacio objeto (es decir, cuando el objeto está en (0,0,0)).

time
La hora actual, factorizada por el parámetro opcional (o 1.0f si no es suministrado).

time_0_x
Valor de tiempo en float, que se repite sobre la base de "tiempo de ciclo" dado en el campo 'params_extra'.

costime_0_x
Coseno de time_0_x.

sintime_0_x
Seno de time_0_x.

tantime_0_x
Tangente de time_0_x.

time_0_x_packed
Vector de 4 elementos de time0_x, sintime0_x, costime0_x, tantime0_x.

time_0_1
Como time0_x pero escalado en [0 .. 1].

costime_0_1
Como costime0_x pero a escala [0 .. 1].

sintime_0_1
Como sintime0_x pero a escala [0 .. 1].

tantime_0_1
Como tantime0_x pero a escala [0 .. 1].

time_0_1_packed
Como time0_x_packed pero todos los valores a escala 0 .. 1].

time_0_2pi

Como time0_x pero a escala [0 .. 2 * Pi].

costime_0_2pi
Como costime0_x pero a escala [0 .. 2 * Pi].

sintime_0_2pi
Como sintime0_x pero a escala [0 .. 2 * Pi].

tantime_0_2pi
Como tantime0_x pero a escala [0 .. 2 * Pi].

time_0_2pi_packed
Como time0_x_packed pero a escala [0 .. 2 * Pi].

frame_time
El tiempo por marco (frame) actual, factorizado por el parámetro opcional (o 1.0f, si no se suministra).

fps
Los marcos por segundo actuales.

viewport_width
El ancho de vista actual en píxeles.

viewport_height
La altura vista actual en píxeles.

inverse_viewport_width
1.0/el ancho de vista actual en píxeles.

inverse_viewport_height
1.0/la altura vista actual en píxeles.

viewport_size
Vector de 4 elementos de viewport_width, viewport_height, inverse_viewport_width, inverse_viewport_height.

texel_offsets
Proporciona detalles de la compensación de coordenadas de textura específicas del sistema de renderizado (rendersystem) requeridas para mapear gráficos en píxeles. float4 (HorizontalOffset, VerticalOffset, HorizontalOffset / viewport_width, VerticalOffset / viewport_height).

view_direction
Vector de dirección de la vista en espacio de objeto.

view_side_vector
Vista local del eje X.

view_up_vector
Vista local del eje Y.

fov
Campo vertical de vista, en radianes.

near_clip_distance
Distancia de clip cercana, en unidades del mundo.

far_clip_distance
Distancia clip de lejana, en unidades del mundo (puede ser 0 para ver la proyección infinita).

texture_viewproj_matrix
Aplicables a programas de vértice que se han especificado como programas de vértices alternativos de 'receptor de sombra', o cuando una unidad de textura se marca como sombra content_type, lo que proporciona detalles de la matriz de vista/proyección para el proyector de sombra actual. La entrada opcional de 'params_extra' especifica la luz que el proyector referencia (para el caso de la sombra de content_type donde más de una textura de sombra puede estar presentada en una simple pasada), donde 0 es el valor predeterminado y se refiere a la primera luz que se hace referencia en esta pasada.

texture_viewproj_matrix_array
Como texture_viewproj_matrix, a excepción de que se pasa un array de matrices, hasta el número que se especifica como el valor de 'params_extra'.

texture_worldviewproj_matrix
Como texture_viewproj_matrix excepto que también incluye la matriz del mundo.

texture_worldviewproj_matrix_array
Como texture_worldviewproj_matrix, a excepción de que es pasado un array de matrices, hasta el número que se especifica como el valor de 'params_extra'.

spotlight_viewproj_matrix
Proporciona una matriz de vista/proyección que coincide con la puesta en marcha de un foco dado (requiere de una entrada 'params_extra' para indicar que el índice de luz, que debe ser foco). Puede ser utilizado para proyectar una textura de un foco determinado.

spotlight_worldviewproj_matrix
Como spotlight_viewproj_matrix excepto que también incluye la matriz del mundo.

scene_depth_range
Proporciona información sobre el rango de profundidad, como se ve desde la cámara actual que se utiliza para representar. Dado como Float4 (mindepth, maxdepth, depthRange, 1 / depthRange).

shadow_scene_depth_range
Proporciona información sobre el rango de profundidad, como se ve desde la cámara de sombra en relación con la luz seleccionada. Requiere un parámetro índice de la luz. Dado como float4 (mindepth, maxdepth, depthRange, 1 / depthRange).

shadow_colour
El color de la sombra (para las sombras modulativas) tal como se establece a través de SceneManager::setShadowColour.

shadow_extrusion_distance
La distancia de extrusión de sombra, como se determina por el alcance de una luz no direccional o conjunto a través de SceneManager::setShadowDirectionalLightExtrusionDistance para las luces direccionales.

texture_size
Proporciona tamaño de textura de la unidad de textura seleccionada. Requiere un parámetro unidad de índice de textura. Dado como float4 (ancho, alto, profundidad, 1). Para textura 2D, la profundidad se fija a 1, para textura 1D, altura y profundidad se fija a 1.

inverse_texture_size

Proporciona tamaño de textura inversa de la unidad de textura seleccionada. Requiere un parámetro de índice de unidad de textura. Dado como float4 (1/ancho, 1/altura, 1/profundidad, 1). Para textura 2D, la profundidad se fija a 1, para textura 1D, la altura y la profundidad fija a 1.

packed_texture_size

Proporciona empaquetado el tamaño de textura de la unidad de textura seleccionada. Requiere un parámetro de índice de la unidad de textura. Dado como float4 (anchura, altura, 1/ancho, 1/altura). Para una textura 3D, la profundidad se ignora, para una textura 1D, la altura se pone a 1.

pass_number

Establece el número de índice de la pasada activa en un parámetro de la GPU. La primera pasada en una técnica tiene un índice 0, el segundo un índice de 1 y así sucesivamente. Esto es útil para los shaders multipasada (es decir, sombreado de piel o de desenfoque) que necesitan saber qué pasada es. Al establecer el parámetro auto en una lista de paramétros de programas predeterminados en una definición de programa, no hay ningún requisito para establecer el parámetro del número de pasada en cada pasada y perder la pista. (Ver ejemplo_piel)

pass_iteration_number

Útil para los programas de GPU que necesitan saber cuál es el número actual de iteración de la pasada. La primera iteración de una pasada es el número 0. El número de última iteración es uno menos que el que está establecido para el número de iteración de la pasada. Si una pasada tiene su atributo iteración a 5, entonces, el número de la última iteración (5ª de ejecución de la pasada) es 4. (Ver la sección iteración)

animation_parametric

Útil para la animación de vértices hardware. Para la animación morfológica, establece el valor paramétrico (0..1) representando la distancia entre el fotograma clave en la primera posición (ligado a las posiciones) y la segunda posición de fotogramas clave (ligada a la primera textura de coordenada libre) para que el programa vértice pueda interpolar entre ellos . Para plantear la animación, indica un grupo de hasta 4 valores de los parámetros de peso para la aplicación de una secuencia de hasta 4 poses (cada uno de ellos vinculado a x, y, z, w de la constante), una para cada pose. Las posiciones originales se mantienen en el búfer en la posición habitual, y los offset a tomar esas posiciones para la pose donde el peso == 1.0 en las primeras 'n' coordenadas de textura; 'n' viene determinado por el valor pasado a includes_pose_animation. Si son necesarias más de 4 poses simultáneas, entonces se necesitará más de una constante de sombreado para mantener el valor de los parámetros, en cuyo caso se debe usar este enlace más de una vez, referenciando una entrada constante diferente; el segundo contendrá las paramétricas de las poses 5-8, el tercero para poses 9-12, y así sucesivamente.

Custom

Permite mapear un parámetro personalizado en un individuo Renderable (véase Renderable::setCustomParameter) a un parámetro en un programa de la GPU. Se requiere que se rellene el campo 'params_extra' con el índice que se utiliza en la llamada Renderable::setCustomParameter, y esto garantizará que cada vez que este Renderable se utilice, tendrá su parámetro personalizado mapeado. Es muy importante que este parámetro se haya definido en todos los Renderables que se les asigna el material que contiene esta asignación automática, de lo contrario el proceso fracasará.

param_named

Este atributo es igual que param_indexed, pero utiliza un parámetro nombrado en lugar de un índice numérico. Esto sólo se puede utilizar con programas de alto nivel que incluyen los nombres de parámetros, si se está usando un programa en ensamblador, entonces no se tiene más remedio que utilizar los índices. Hay que tener en cuenta que también se pueden utilizar los parámetros indexados para programas de alto nivel, pero es menos portable ya que si se reordenan los parámetros en el programa de alto nivel, los índices van a cambiar.

Formato: param_named <nombre><tipo><valor>
Ejemplo: param_named shininess floatT 10.0 0 0 0

El 'tipo' es necesario porque el programa no se compila y se carga cuando se analiza el script de materiales, por lo que en este momento no se tiene idea de qué tipo son los parámetros. Los programas son sólo compilados y cargados cuando se utilizan, para ahorrar memoria.

param_named_auto

Este es equivalente a param_indexed_auto, para su uso con programas de alto nivel.

Formato: param_named_auto <nombre><código_valor><params_extra>
Ejemplo: param_named_auto worldViewProj WORLDVIEWPROJ_MATRIX

Los códigos de valores permitidos y el significado de params_extra se detallan en el param_indexed_auto.

PROGRAMAS DE SOMBRAS Y VÉRTICES

Cuando se usan sombras (véase la sección 7. Sombras), el uso de programas de vértice puede añadir cierta complejidad adicional, ya que Ogre sólo puede ocuparse automáticamente de todo cuando se utiliza un pipeline de función fija. Si se utiliza un programa de vértice, y también se están utilizando sombras, puede que se tengan que hacer algunos ajustes.

Si se utilizan **sombras plantilla**, entonces cualquier programa vértice que haga deformación de vértices puede ser un problema, porque las sombras plantilla se calculan sobre la CPU, que no tiene acceso a los vértices modificados. Si el programa vértice está haciendo animación del esqueleto normal, esto es aceptable (véase la sección anterior), ya que Ogre sabe cómo reproducir el efecto en el software, pero cualquier otra deformación de los vértices no puede repetirse, y se tendrá que aceptar que la sombra no refleja la deformación, o se deben desactivar las sombras para ese objeto.

Si se utilizan **sombras de textura**, entonces la deformación de vértices es aceptable, sin embargo, al representar el objeto en la textura de la sombra (la pasada lanzadora de sombra), la sombra ha de ser renderizada en un color sólido (en relación con el color del entorno). Por lo tanto, se debe proporcionar un programa de vértice alternativo, por lo que Ogre ofrece una manera de especificar uno que se debe usar cuando se representa el proyector. Básicamente se vincula un programa de vértice alternativo, utilizando exactamente la misma sintaxis que el original enlace del programa de vértice:

```
shadow_caster_vertex_program_ref myShadowCasterVertexProgram
{
        param_indexed_auto 0 worldviewproj_matrix
        param_indexed_auto 4 ambient_light_colour
}
```

Código 26: Vinculación programa de vértice alternativo.

Al renderizar un proyector de sombra, Ogre utilizará automáticamente el programa alternativo. Se puede obligar a los mismos o diferentes parámetros para el programa - lo más importante es vincular **ambiend_light_colour**, ya que esto determina el color de la sombra en las sombras de textura modulativa. Si no se proporciona un programa alternativo, Ogre caerá de nuevo en un material de función fija que no refleja ninguna deformación de vértices que se realice en el programa de vértice.

Además, al representar los receptores de la sombra con texturas de sombra, Ogre necesita proyectar la textura de la sombra. Esto se hace automáticamente en el modo de función fija, pero si los receptores utilizan programas de vértice, es necesario tener un programa receptor de sombra que realice la deformación de vértices comunes, pero también genera coordenadas de textura proyectivas. El programa adicional se vincula a la pasada de la siguiente forma:

```
shadow_receiver_vertex_program_ref myShadowReceiverVertexProgram
{
        param_indexed_auto 0 worldviewproj_matrix
        param_indexed_auto 4 texture_viewproj_matrix
}
```

Código 27: Otra vinculación de programa de vértice alternativo.

A los efectos de redacción de este programa alternativo, hay una vinculación automática de parámetros de 'texture_viewproj_matrix' que proporciona el programa con los parámetros de proyección de textura. El programa vértice debe hacer su procesamiento de vértices normal, y generar las coordenadas de textura usando esta matriz y establecerlo en las coordenadas de textura a 0 y 1, ya que algunas técnicas de sombra usan 2 unidades de textura. El color de salida de los vértices de este programa vértice siempre debe ser de color blanco, a fin de no afectar el color final de la sombra renderizada.

Al utilizar sombras de textura aditivas, la renderización de la pasada de sombra es en realidad la renderización de la iluminación, de modo que si realiza cualquier iluminación de programa de fragmento también tiene que tirar de un programa de fragmento personalizado. Se utiliza el shadow_receiver_fragment_program_ref para ello:

```
shadow_receiver_fragment_program_ref myShadowReceiverFragmentProgram
{
        param_named_auto lightDiffuse light_diffuse_colour 0
}
```

Código 28: Llamada a programa de fragmento alternativo.

Se deben pasar las coordenadas de sombra proyectada de los vértices del programa personalizado. En cuanto a las texturas, la unidad de textura 0 siempre será la textura de la sombra. Cualquier otra textura que se vincule a la pasada se realizará también, pero se moverá por 1 unidad para hacer espacio para la textura de la sombra. Por lo tanto, el programa de fragmento receptor de sombra es probable que sea el mismo que la pasada de iluminación de desnudos del material normal, excepto que se inserte un sampler de textura extra en el índice 0, que se va a utilizar para ajustar el resultado (la modulación de los componentes difusa y especular).

3.1.10 FETCHING DE TEXTURAS DE VÉRTICES

INTRODUCCIÓN

Las generaciones más recientes de las tarjetas de vídeo permiten realizar una lectura de una textura en el programa de vértice en lugar de sólo en el programa de fragmento, como es tradicional. Esto permite, por ejemplo, leer el contenido de una textura y desplazar vértices basándose en la intensidad del color que contiene.

DECLARAR EL USO DE LA BÚSQUEDA DE TEXTURA DE VÉRTICES

Dado que el soporte de hardware para la búsqueda de textura de vértices no es omnipresente, se debe utilizar la directiva uses_vertex_texture_fetch (Ver sección fetching de textura de vértices en programas de vértice) al declarar programas de vértices que utilizan texturas vértice, de modo que si no es soportado, se puede activar una técnica alternativa. Esto no es estrictamente necesario para shaders específicos de DirectX, ya que la obtención de la textura de vértices sólo se admite en vs_3_0, que se puede establecer como una sintaxis necesaria en la definición del shader, pero para OpenGL (GLSL), hay tarjetas que soportan GLSL pero no texturas de vértices, por lo que se debe ser explícito acerca de su necesidad.

DIFERENCIAS DE FIJADO DE TEXTURAS DEL SISTEMA DE RENDERIZADO

Lamentablemente, el método de fijado de texturas para que estén disponibles para un programa de vértices no está bien estandarizado. En el momento de la escritura, Shader Model 3.0 (SM3.0), hardware bajo DirectX9, incluye 4 muestras separadas fijadas para los propósitos de las texturas vértice. OpenGL, por otro lado, es capaz de acceder a texturas vértice en GLSL (y en ensamblador a través de NV_vertex_program_3, aunque esto es menos popular), pero las texturas se comparten con el pipeline de fragmento. Se espera que DirectX pase al modelo GL con la llegada de DirectX10, ya que una arquitectura de sombreado unificado implica el intercambio de recursos de textura entre las dos etapas. Como está ahora sin embargo, se estanca con una situación incoherente.

Para reflejar esto, se debe usar el atributo binding_type en una unidad de textura para indicar qué unidad se está dirigiendo con la textura – 'fragmento' (por defecto) o 'vértice'. Para los sistemas de renderizado que no tienen fijados separados, esto realmente no hace nada. Pero para los que lo hacen, se asegurará que la textura se enlaza a la unidad de procesamiento correcta.

Hay que tener en cuenta que si bien DirectX9 tiene enlazado separado para pipelines de vértice y de fragmento, ligar una textura a la unidad de procesamiento de vértices aún utiliza un 'slot' (espacio) que luego no se encuentra disponible para su uso en el pipeline de fragmentos. No se ha conseguido encontrar esto documentado en ninguna parte, pero las muestras de nVidia sin duda evitan ligar a una textura con el mismo índice en ambas unidades de vértice y fragmento, y cuando se intentó hacerlo, la textura no aparecía correctamente en la unidad de fragmento, mientras que lo hizo tan pronto como se trasladó a la unidad siguiente.

LIMITACIONES DE FORMATO DE TEXTURA

De nuevo, como en el momento de la escritura, los tipos de textura que se pueden utilizar en un programa de vértice están limitados a 1 - o 4-componentes, con precisión total en formatos de punto

flotante. En el código equivale a PF_FLOAT32_R o PF_FLOAT32_RGBA. No se soportan otros formatos. Además, las texturas deben ser texturas 2D regulares (sin mapas de cubo o volumen) y el mipmapping y filtrado no es compatible, aunque si se desea se puede realizar el filtrado en el programa de vértices por muestreo en múltiples ocasiones.

LIMITACIONES DEL HARDWARE

Como en el momento de la escritura (a principios del tercer trimestre de 2006), ATI no soporta recuperación (fetching) de textura en su listado actual de tarjetas (Radeon X1n00). nVidia lo soporta, en sus series 6n00 y 7n00. ATI soporta una alternativa llamada 'Renderizado al búfer de vértice', pero esto no está estandarizado en este momento y es muy diferente en su implementación, por lo que no puede ser considerado como un reemplazo exacto. Esta es la situación, aunque las tarjetas Radeon X1n00 dicen soportar vs_3_0 (que requiere recuperación de textura de vértices).

3.1.11 HERENCIA DE SCRIPTS

Cuando se crean nuevos objetos script que tienen sólo ligeras variaciones de otro objeto, es bueno evitar copiar y pegar entre los scripts. La herencia de script permite hacer esto; en esta sección vamos a utilizar scripts de material como un ejemplo, pero a este se le aplican todos los scripts analizados con los compiladores de Scripts en Ogre 1.6 y en adelante.

Por ejemplo, para hacer un material nuevo que se basa en otros previamente definidos, agregar dos puntos ':' después del nombre del nuevo material, seguido del nombre del material que se va a copiar.

Formato: material <NuevoMaterialHijo> : <ReferenciaMaterialPadre>

La única salvedad es que el material de los padres debe haber sido definido/analizado antes del script de material hijo que se está analizando. La forma más fácil de conseguir esto es colocar los scripts padres al principio del archivo de scripts de materiales, o usar la directiva de 'importación' (véase la sección 3.1.14 Directiva de importación de Script). Hay que tener en cuenta que la herencia es en realidad una copia - después de que los scripts se cargan en Ogre, los objetos ya no mantienen la copia de la estructura heredada. Si un material padre se ha modificado a través de código en tiempo de ejecución, los cambios no tienen efecto en el material hijo que se han copiado de él en el script.

La copia de material con scripts alivia la monotonía de copiar/pegar, sólo con la capacidad de identificar las técnicas específicas, pasadas, y modificación de las unidades de textura, hace más fácil la copia de material. Las técnicas, pasadas, unidades de textura pueden ser identificadas directamente en el material hijo sin necesidad de diseños anteriores de técnicas, pasadas, unidades de textura mediante la asociación de un nombre con ellos, las técnicas y las pasadas pueden tener un nombre y las unidades de textura se pueden enumerar dentro del script de material. También se pueden utilizar variables, ver la sección 3.1.13 Variables de Script.

Los nombres pueden llegar a ser útiles en los materiales que copian de otros materiales. Con el fin de anular valores deben estar en la técnica, pasada, unidad de textura correcta, etc. El script podría ser establecido mediante la secuencia de técnicas, pasadas, unidades de textura en el material del hijo, pero si sólo un parámetro necesita cambiar, es decir la 5ª pasada, entonces las primeras cuatro pasadas antes de la quinta tienen que ser colocados en el script:

Un ejemplo:

```
material test2 : test1
{
```

```
technique
  {
    pass
    {
    }

    pass
    {
    }

    pass
    {
    }

    pass
    {
}

    pass
    {
          ambient 0.5 0.7 0.3 1.0
    }
  }
}
```

Código 29: Ejemplo básico herencia de Scripts.

Este método es tedioso para materiales que tienen pequeñas variaciones con respecto a sus padres. Una forma fácil de solucionar esto es nombrando la pasada directamente sin tener que listar las pasadas anteriores:

```
material test2 : test1
{
  technique 0
  {
    pass 4
    {
        ambient 0.5 0.7 0.3 1.0
    }
  }
}
```

Código 30: Ejemplo reducido herencia de Scripts.

El nombre de la pasada padre debe ser conocido y debe de estar en la técnica correcta para que funcione correctamente. El mejor método es especificar el nombre de la técnica y el nombre de la pasada. Si la técnica/pasada del padre no se nombra se usan los valores de sus índices como se ha hecho en el ejemplo.

AÑADIENDO NUEVAS TÉCNICAS, PASADAS, A LOS MATERIALES COPIADOS:

Si se necesita añadir una nueva técnica o pasada a un material copiado, entonces se utiliza un nombre único para la técnica o la pasada que no existe en el material base. Usando un índice para el nombre que es uno más que el último índice en el que el padre hace lo mismo. La nueva técnica/pasada se añade al final de las técnicas/pasadas copiadas del material padre.

Nota: si a las pasadas o técnicas no se les da un nombre, se tomará un nombre por defecto en función de su índice. Por ejemplo, la primera pasada tiene el índice 0 por lo que su nombre será 0.

IDENTIFICANDO UNIDADES DE TEXTURA PARA INVALIDAR VALORES

A un estado de unidad de textura específico (TUS) se le puede dar un nombre único dentro de una pasada de un material para que se le pueda identificar más adelante en materiales clonados que necesiten reemplazar estados de la unidad de textura específicos en la pasada sin declarar unidades de textura anteriores. Usando un nombre único para una unidad de textura en una pasada de un material clonado, se añade una nueva unidad de textura al final de la lista de unidad de textura para la pasada.

```
material BumpMap2 : BumpMap1
{
  technique ati8500
  {
    pass 0
    {
      texture_unit NormalMap
      {
        texture BumpyMetalNM.png
      }
    }
  }
}
```

Código 31: Identificando unidades de textura.

HERENCIA DE SCRIPTS AVANZADA

A partir de Ogre 1.6, los objetos de scripts pueden heredar de otros más generales. El concepto anterior de la herencia, la copia de material, se limitaba exclusivamente a la parte nivel-superior de los objetos de material. Ahora, cualquier nivel de objeto puede tomar el beneficio de la herencia (por ejemplo, las técnicas, pasadas, y los objetivos del compositor).

```
material Test
{
    technique
    {
        pass : ParentPass
        {
        }
    }
}
```

Código 32: Herencia de Scripts avanzada.

Hay que tener en cuenta que la pasada se hereda de ParentPass. Esto permite la creación de de jerarquías de herencia de grano fino.

Junto con el sistema de herencia más generalizado viene una nueva palabra clave importante: 'abstract' (abstracto). Esta palabra clave se utiliza en una declaración de nivel alto de objeto (no dentro de cualquier otro objeto), que indica que no es algo que en realidad el compilador debe tratar de compilar, sino que es sólo para el propósito de la herencia. Por ejemplo, un material declarado con la palabra clave abstract nunca se convertirá en un material utilizable real en el marco material. Los objetos que no pueden estar a un nivel superior en el documento (como una pasada), pero que gustaría que se declararan como tal para ser heredados, deben declararse con la palabra clave abstract.

```
abstract pass ParentPass
{
    diffuse 1 0 0 1
}
```

Código 33: Pasada abstracta.

Esto declara el objeto ParentPass que fue heredado de en el ejemplo anterior. Se debe notar que la palabra clave 'abstract' informa al compilador que no debe tratar de convertir en este objeto en cualquier tipo de recurso de Ogre. Si se trata de hacerlo, entonces es obvio que fallará, ya que una pasada por su cuenta no es válida.

La opción de concordancia final se basa en comodines. Utilizando el carácter '*', se puede hacer un régimen de compensación de gran alcance y anular varios objetos a la vez, incluso si no se sabe los nombres exactos o las posiciones de los objetos en el objeto heredado.

```
abstract technique Overrider
{
   pass *color*
{
diffuse 0 0 0 0
   }
}
```

Código 34: Uso de comodines.

Esta técnica, cuando se incluye en un material, se anulan todos las pasadas que coincidan con el comodín '*color*' (color tiene que aparecer en alguna parte del nombre) y convertir sus propiedades difusas en negro. No importa su posición o su nombre exacto en la técnica heredada.

3.1.12 ALIAS DE TEXTURA

Los alias de textura son útiles para cuando sólo las texturas utilizadas en unidades de textura deben ser especificadas para un material clonado. En el material de origen, es decir, el material original para ser clonado, a cada unidad de textura se le puede dar un nombre de alias de textura. El material clonado en el script puede especificar qué texturas debe utilizar para cada alias de textura. Hay que tener en cuenta que los alias de textura son una versión más específica de 3.1.13 Variables de scripts que puede utilizarse para configurar fácilmente otros valores.

Usando alias textura dentro de las unidades de textura:

Formato:texture_alias <nombre>
Por defecto: <nombre> se pondrá por defecto texture_unit <nombre> si se establece

```
texture_unit DiffuseTex
{
  texture diffuse.jpg
```

```
}
```

Código 35: Ejemplo unidad de textura, para hacer alias.

Texture_alias queda por defecto a DiffuseTex.

Ejemplo: El material base a ser clonado:

```
material TSNormalSpecMapping
{
  technique GLSL
  {
    pass
    {
      ambient 0.1 0.1 0.1
      diffuse 0.7 0.7 0.7
      specular 0.7 0.7 0.7 128

      vertex_program_ref GLSLDemo/OffsetMappingVS
      {
        param_named_auto lightPosition light_position_object_space 0
        param_named_auto eyePosition camera_position_object_space
        param_named textureScale float 1.0
      }

      fragment_program_ref GLSLDemo/TSNormalSpecMappingFS
      {
        param_named normalMap int 0
        param_named diffuseMap int 1
        param_named fxMap int 2
      }

      // Normal map
      texture_unit NormalMap
      {
        texture defaultNM.png
        tex_coord_set 0
        filtering trilinear
      }

      // Base diffuse texture map
      texture_unit DiffuseMap
      {
        texture defaultDiff.png
        filtering trilinear
        tex_coord_set 1
      }

      // spec map for shinnines
      texture_unit SpecMap
      {
        texture defaultSpec.png
        filtering trilinear
        tex_coord_set 2
      }

    }

  }

  technique HLSL_DX9
```

```
  {
    pass
    {

      vertex_program_ref FxMap_HLSL_VS
      {
        param_named_auto worldViewProj_matrix worldviewproj_matrix
        param_named_auto lightPosition light_position_object_space 0
        param_named_auto eyePosition camera_position_object_space
      }

      fragment_program_ref FxMap_HLSL_PS
      {
        param_named ambientColor float4 0.2 0.2 0.2 0.2
      }

      // Normal map
      texture_unit
      {
        texture_alias NormalMap
        texture defaultNM.png
        tex_coord_set 0
        filtering trilinear
      }

      // Base diffuse texture map
      texture_unit
      {
        texture_alias DiffuseMap
        texture defaultDiff.png
        filtering trilinear
        tex_coord_set 1
      }

      // spec map for shinnines
      texture_unit
      {
        texture_alias SpecMap
        texture defaultSpec.png
        filtering trilinear
        tex_coord_set 2
}

    }

  }
}
```

Código 36: Material base a clonar.

Hay que tener en cuenta que las técnicas GLSL y HLSL usan las mismas texturas. Para cada tipo de uso de textura se da un alias de textura que describe la textura que se utiliza. Así que la primera unidad de textura en la técnica de GLSL tiene el mismo alias que el TUS en la técnica de HLSL ya que es la misma textura utilizada. Se usa igual para las unidades de textura segunda y tercera. Para fines de la demostración, la técnica de GLSL hace uso de la denominación texture_unit y por lo tanto el nombre de texture_alias no tiene que ser establecido, puesto que da por defecto el nombre de la unidad de textura. Entonces, ¿Por qué no utilizar la opción predeterminada todo el tiempo ya que requiere menos escritura? Para la mayoría de las situaciones que se pueda, cuando se clona un material que a continuación, se desea cambiar el alias, se debe utilizar el comando texture_alias en el script. No se

puede cambiar el nombre de un texture_unit en un material clonado, la forma texture_alias proporciona un mecanismo para asignar un nombre de alias.

Ahora se quiere clonar el material, pero sólo se desean cambiar las texturas utilizadas. Se puede copiar y pegar todo el material, pero si se decide cambiar el material de base más tarde entonces también se tendrá que actualizar el material copiado en el script. Con set_texture_alias, la copia de un material es muy fácil. set_texture_alias se especifica en la parte superior de la definición del material. Todas las técnicas con el alias de textura especificado serán afectadas por la sentencia set_texture_alias.

Formato: set_texture_alias <nombre alias><nombre textura>

```
material fxTest : TSNormalSpecMapping
{
  set_texture_alias NormalMap fxTestNMap.png
  set_texture_alias DiffuseMap fxTestDiff.png
  set_texture_alias SpecMap fxTestMap.png
}
```

Código 37: Creando alias de textura.

Las texturas de ambas técnicas en el material hijo automáticamente se reemplazarán con la nueva cuando se quiera utilizar.

El mismo proceso se puede realizar en el código mientras se configura la textura de los nombres de alias para que no haya necesidad de revisar la técnica/pasada/TUS para cambiar una textura. Simplemente se llama a myMaterialPtr->applyTextureAliases(myAliasTextureNameList) que actualizará todas las texturas en todas las unidades de textura que coinciden con los nombres de alias en el contenedor de referencia del mapa que se pasa como parámetro.

No se tienen que proporcionar todas las texturas en el material copiado.

```
material fxTest2 : fxTest
{
  set_texture_alias DiffuseMap fxTest2Diff.png
  set_texture_alias SpecMap fxTest2Map.png
}
```

Código 38: Creando alias de textura, otro ejemplo.

El material fxTest2 sólo cambia los mapas difuso y de especificación del material fxTest y utiliza el mismo mapa normal.

Otro ejemplo:

```
material fxTest3 : TSNormalSpecMapping
{
set_texture_alias DiffuseMap fxTest2Diff.png
}
```

Código 39: Creando alias de textura, último ejemplo.

fxTest3 acabará con las texturas por defecto para el mapa normal y de especificaciones en el material TSNormalSpecMapping, pero tendrá un mapa difuso diferente. Así que el material base puede definir las texturas por defecto a usar y entonces los materiales hijos podrán sobrescribir texturas específicas.

3.1.13 VARIABLES DE SCRIPTS

Una nueva característica muy poderosa en Ogre 1.6 son las variables. Las variables permiten parametrizar datos en materiales para que puedan ser más generalizados. Esto permite una mayor reutilización de los scripts objetivizando puntos personales específicos. Usando variables junto con la herencia se permiten grandes cantidades de invalidación y reutilización fácil de objetos.

```
abstract pass ParentPass
{
   diffuse $diffuse_colour
}

material Test
{
   technique
   {
       pass : ParentPass
       {
         set $diffuse_colour "1 0 0 1"
       }
   }
}
```

Código 40: Variables de Script en pasadas.

El objeto ParentPass declara una variable llamada “diffuse_colour”, que luego se reemplaza en la pasada del material Test. La palabra clave “set” se utiliza para establecer el valor de esa variable. La asignación de variables sigue reglas de ámbito léxico, lo que significa que el valor de “1 0 0 1” sólo es válido dentro de esa definición de pasada. La asignación de variables en los ámbitos externos prórroga en los ámbitos internos.

```
material Test
{
    set $diffuse_colour "1 0 0 1"
    technique
    {
        pass : ParentPass
        {
        }
    }
}
```

Código 41: Variables de Script en técnica y pasada.

La asignación $diffuse_colour se realiza a través de la técnica y la pasada.

3.1.14 DIRECTIVA IMPORT DE SCRIPTS

Las importaciones son una característica introducida para eliminar la ambigüedad de las dependencias de los scripts. Cuando se utilizan scripts que heredan unos de otros, pero que se definen en archivos separados a veces se producen errores porque los scripts se cargan en orden incorrecto. Usando importaciones se elimina este problema. El script que se hereda puede de forma explícita importar la definición de sus padres, y se asegurará de que no se producirán errores porque la definición de los padres no fue encontrada.

```
import * from "parent.material"
material Child : Parent
{
}
```

Código 42: Importación de Scripts.

El material “Parent” está definido en parent.material y la importación asegura que esas definiciones se encontrarán correctamente. También se pueden importar objetivos específicos dentro de un archivo.

```
import Parent from "parent.material"
```

Código 43: Importación de partes de Scripts.

Si hay otras definiciones en el fichero parent.material, no se importarán.

Hay que tener en cuenta, sin embargo, que la importación en realidad no hace que los objetos importados del script sean plenamente analizados y creados, sólo hace que las definiciones estén disponibles para la herencia. Esto tiene una ramificación específica para la definición de programas de vértice/fragmento, que deben ser cargadas antes de que los parámetros puedan ser especificados. Se debe continuar para poner definiciones de programas comunes en archivos .program para garantizar que son plenamente analizados antes de ser referenciados en múltiples archivos .material. El comando ‘import’ sólo asegura que se puedan resolver las dependencias entre las definiciones equivalentes del script (por ejemplo, material a material).

3.2 SCRIPTS DE COMPOSITOR

El marco de trabajo del Compositor es una subsección de la API de Ogre que permite definir con facilidad efectos de post-procesamiento de pantalla completa. Los scripts del Compositor ofrecen la posibilidad de definir los efectos del compositor en un script que puede ser reutilizado y modificado fácilmente, en lugar de tener que usar el API para definirlos. Todavía es necesario utilizar el código para crear instancias de un compositor contra uno de sus visores visibles, pero este es un proceso mucho más simple que la definición del propio compositor.

FUNDAMENTOS DEL COMPOSITOR

La realización del post-procesamiento de efectos, implica en primer lugar, la representación de la escena en una textura, ya sea como complemento o en lugar de la ventana principal. Una vez que la escena está en una textura, se puede llevar la imagen de escena a un programa de fragmento y realizar operaciones en él a través del renderizando del cuadrado de pantalla completa. El objetivo de este post-proceso de renderizado puede ser el resultado principal (por ejemplo una ventana), o se puede renderizar otra textura para que pueda realizar múltiples circunvoluciones de etapa en la imagen. Se puede incluso hacer renderizado de “ping-pong”, de ida y vuelta entre un par de texturas de renderizado para realizar circunvoluciones que requieren muchas iteraciones, sin utilizar una textura diferente para cada etapa. Eventualmente, se querrá renderizar el resultado a la salida final, que se puede hacer con un cuadrado de pantalla completa. Esto puede sustituir toda la ventana (por lo tanto la ventana principal no tiene que hacer la escena en sí), o podría ser un efecto combinatorio.

Se puede discutir la forma de aplicar estas técnicas de manera eficiente, una serie de definiciones que se requieren:

Compositor

Definición de un efecto de pantalla completa que se puede aplicar a una ventana gráfica de usuario. Esto es lo que se está definiendo la hora de escribir scripts de compositor que se detallan en esta sección.

Compositor Instance (Instancia del Compositor)

Una instancia de un compositor que se aplica a una vista única. Se pueden crear estos basados en las definiciones del compositor, véase la sección 3.2.4 Aplicando un Compositor.

Compositor Chain (Cadena de Compositores)

Es posible habilitar más de una instancia del compositor en un visor al mismo tiempo, con un compositor tomando los resultados del anterior como entrada. Esto se conoce como una cadena de compositores. Cada vista que tiene al menos un compositor que se le ha unido, tiene una cadena de compositor. Véase la sección 3.2.4 Aplicando un Compositor.

Target (Objetivo)

Este es un RenderTarget, es decir, el lugar donde se envía el resultado de una serie de operaciones de renderizado. Un objetivo puede ser el resultado final (y esto es implícito, no hay que declararlo), o puede ser una textura de renderizado intermedia, que se declara en el script con la línea de la textura. Un objetivo que no es el objetivo de salida tiene un tamaño definido y el formato de píxel que se puede controlar.

Output Target (Objetivo Producto)

Como un objetivo, pero este es el resultado final y único de todas las operaciones. El tamaño y el formato de píxel de este objetivo no pueden ser controlados por el compositor, ya que se define por la aplicación usándolo, por lo que no se declara en el script. Sin embargo, se declara una pasada objetivo para ello, ver abajo.

Target Pass (Pasada Objetivo)

Un objetivo, puede ser renderizado muchas veces en el curso de un efecto de composición. En particular, si se usa una circunvolución 'ping-pong' entre un par de texturas, se tendrá más de una pasada objetivo por objetivo. Las pasadas objetivo son declaradas en el script utilizando una línea target o una línea target_output, siendo este último la pasada objetivo de salida final, de los cuales sólo puede haber uno.

Pass (Pasada)

Dentro de una pasada objetivo, hay una o más pasadas individuales, que realizan unas acciones muy específicas, tales como el procesamiento de la escena original (o tirar el resultado del compositor anterior en la cadena), lo que hace un cuadrado de pantalla completa, o limpieza de uno o más búferes. Por lo general dentro de una pasada objetivo única se va a utilizar la evaluación pasada de 'renderizado de escenario' o una pasada de 'renderizado cuadrático', pero no ambos. La limpieza se puede utilizar con cualquier tipo.

CARGANDO SCRIPTS

Los Scripts del Compositor se cargan cuando se inicializan los grupos de recursos: Ogre busca en todos los lugares de recursos asociados con el grupo (véase el Root::addResourceLocation) para los archivos con la extensión '. compositor' y los analiza. Si se desea analizar los archivos manualmente, hay que utilizar CompositorSerializer::parseScript.

FORMATO

Se pueden definir varios compositores en un único script. El formato de secuencia de comandos es pseudo-C++, con secciones delimitadas por llaves ('{','}'), y comentarios indicados al iniciar una línea con '//'. El formato general se muestra a continuación en el siguiente ejemplo:

```
// This is a comment
// Black and white effect
compositor B&W
{
    technique
    {
        // Temporary textures
        texture rt0 target_width target_height PF_A8R8G8B8

        target rt0
        {
            // Render output from previous compositor (or original
scene)
            input previous
        }

        target_output
        {
            // Start with clear output
            input none
            // Draw a fullscreen quad with the black and white image
            pass render_quad
            {
                // Renders a fullscreen quad with a material
                material Ogre/Compositor/BlackAndWhite
                input 0 rt0
}
        }
    }
}
```

Código 44: Ejemplo de Compositor.

A cada compositor en la escritura se le debe dar un nombre, que está en la línea 'compositor <nombre>' antes de la primera apertura '{'. Este nombre debe ser único a nivel global. Se pueden incluir caracteres de ruta de acceso (como en el ejemplo) para dividir la lógica de compositores, y también para evitar nombres duplicados, pero el motor no considera el nombre como jerárquica, sino como una cadena. Los nombres pueden incluir espacios, pero deben de estar rodeados de comillas dobles, por ejemplo: compositor "Mi Nombre".

Los principales componentes de un compositor son las técnicas, las pasadas objetivo y las pasadas, que se tratan en detalle en las secciones siguientes.

3.2.1 TÉCNICAS

Una técnica de compositor es como una técnica de materiales en que se describe un método para lograr el efecto que se busca. Una definición de compositor puede tener más de una técnica si se desea proporcionar algunas de reserva por si no fueran soportadas por el hardware que se prefiere utilizar. Las técnicas son evaluadas por el hardware para el soporte basándose en 2 cosas:

Compatibilidad del material

Todas las pasadas que renderizan un cuadrado a pantalla completa usan un material, para que la técnica sea compatible, todos los materiales referenciados deben tener al menos una técnica de material compatible. Si no lo hacen, la técnica del compositor se marca como no compatible y no se utilizará.

Compatibilidad con el formato de textura

Este es un poco más complicado. Cuando se solicita una textura en una técnica, se solicita un formato de píxel. No todos los formatos son compatibles de forma nativa con el hardware, especialmente los formatos de punto flotante. Sin embargo, en este caso, el hardware suele rebajar el formato de textura que se pidió a uno que para el hardware es compatible - con efectos del compositor, sin embargo, es posible que se desee utilizar un enfoque diferente si este es el caso. Por lo tanto, al evaluar las técnicas, el compositor buscará primero compatibilidades nativas para el formato de píxel exacto que se ha pedido, y se saltará a la siguiente técnica, si no es compatible, lo que permite definir otras técnicas, con formatos simples de píxeles que utilizan un enfoque diferente. Si no se encuentran las técnicas que se admiten de forma nativa, se intenta de nuevo, esta vez permitiendo que el hardware rebaje el formato de textura y por tanto debe encontrar por lo menos alguna compatible por lo que se ha pedido.

Al igual que con las técnicas de materiales, las técnicas de compositor se evalúan en el orden en que se definen en el script.

Formato: technique { }

Las técnicas pueden tener los siguientes elementos anidados:

4. texture (textura)
5. texture_ref
6. scheme
7. compositor_logic
8. target (objetivo)
9. target_output (objetivo de salida)

texture (Textura)

Declara una textura de renderizado para ser usada en posteriores target_output (pasadas objetivo).

Formato: texture <Nombre><Ancho><Alto><Formato Pixel> [<Formato2 Pixel MRT>] [<FormatoN Pixel MRT>] [pooled] [gamma] [no_fsaa] [<scope>]

He aquí una descripción de los parámetros:

Nombre

Un nombre para dar a la textura de renderización, el cual debe ser único dentro de este compositor. Este nombre se utiliza para hacer referencia a la textura en la pasada objetivo,

cuando la textura se renderiza, y en pasadas, cuando la textura se utiliza como entrada a un material para renderizar un cuadrado a pantalla completa.

Ancho, Alto

Las dimensiones de la textura de renderizado. Pueden especificar un ancho y altura fijos, o se puede solicitar que la textura se base en las dimensiones físicas de la ventana de vista a la que se adjunta al compositor. Las opciones para este último son 'target_width', 'target_height', 'target_width_scaled <factor>' y 'target_height_scaled <factor>' - donde 'factor' es la cantidad por la que desea multiplicar el tamaño del objetivo principal para obtener las dimensiones.

Formato Pixel

El formato de píxel de la textura de renderizado. Esto afecta a la cantidad de memoria que tendrá, qué canales de color estarán disponibles, y qué precisión se tendrá dentro de esos canales. Las opciones disponibles son PF_A8R8G8B8, PF_R8G8B8A8, PF_R8G8B8, PF_FLOAT16_RGBA, PF_FLOAT16_RGB, PF_FLOAT16_R, PF_FLOAT32_RGBA, PF_FLOAT32_RGB, y PF_FLOAT32_R.

Pooled

Si está presente, esta directiva hace que esta textura sea 'pooled' (agrupada) entre las instancias de compositor, que puede ahorrar algo de memoria.

No_fsaa

Si está presente, esta directiva deshabilita el uso de anti-aliasing en esta textura. FSAA sólo se utiliza si la textura está sometida a una pasada render_scene y FSAA está activa en el puerto de vista original o en el compositor en el que está basado; esta opción permite sobreescribirlo y deshabilitar el FSAA si se desea.

Scope

Si está presente, esta directiva establece el ámbito para la textura para ser accesible para otros compositores, utilizando la directiva texture_ref. Hay tres opciones: 'local_scope' (ámbito por defecto) que indica que sólo el compositor que define la textura puede acceder a ella. 'chain_scope' que indica que los compositores después del actual en la cadena pueden referenciar su textura, y 'global_scope' que indica que la aplicación al completo puede acceder a la textura. Esta directiva afecta también a la creación de texturas (las texturas globales se crean una vez y por ello no pueden utilizarse con la directiva 'pooled', y no se puede contar con el tamaño en el puerto de vista).

Ejemplo: texture rt0 512 512 PF_R8G8B8A8
Ejemplo: texture rt1 target_width target_height PF_FLOAT32_RGB

De hecho, se puede repetir este elemento si se desea. Si se hace, significa que la textura de renderizado se convierte en un Objetivo de Renderizado Múltiple (MRT), cuando la GPU escribe múltiples texturas a la vez. Es imperativo que si se utiliza MRT, los shaders que renderizan la textura, rendericen a todos los objetivos. No hacerlo puede causar resultados indefinidos. También es importante señalar que aunque se pueden utilizar diferentes formatos de pixel por cada objetivo en una MRT, cada uno debe tener la misma profundidad de bits total, ya que la mayoría de las tarjetas no son compatibles con las resoluciones de bits independientes. Si se intenta utilizar esta característica en las tarjetas que no soporten el número de MRT que se ha solicitado, la técnica será omitida (por lo que se debería escribir una técnica de reserva).

Ejemplo: texture mrt_output target_width target_height PF_FLOAT16_RGBA PF_FLOAT16_RGBA chain_scope

texture_ref

Declara una referencia de una textura de otro compositor para ser utilizada en este compositor.

Formtato: texture_ref <NombreLocal><Compositor Referenciado><Nombre Textura Referenciada>

Descripción de los parámetros:

NombreLocal
Un nombre para dar a la textura referenciada, que debe de ser único dentro de este compositor. Este nombre se utiliza para referenciar la textura en pasadas objetivo, cuando se renderiza la textura, y en pasadas, cuando la textura se utiliza como entrada en la renderización cuadrática de pantalla completa de un material.

Compositor Referenciado
El nombre del compositor del que se referencia la textura

Nombre Textura Referenciada
El nombre de la textura en el compositor referenciado.

Hay que estar seguro, que la textura referenciada tiene un ámbito correcto para ser refenciada (ya sea de cadena o global) y está establecida en un lugar correcto durante la creación de la cadena (si tiene un ámbito de cadena, en el que el compositor debe de estar presente en la cadena y estar en una posición posterior al compositor referenciado).

Ejemplo: texture_ref GBuffer GBufferCompositor mrt_output

scheme

Da a la técnica del compositor un nombre de esquema, permitiendo automáticamente intercambiar entre diferentes técnicas para este compositor cuando se inicializa desde un puerto de vista llamando a CompositorInstance::setScheme.

Formato: material_scheme <Nombre>

compositor_logic

Este parámetro conecta entre el compositor y el código que se requiere para funcionar correctamente. Cuando se crea una instancia de este compositor, la lógica de compositor será notificada y tendrá la posibilidad de preparar las operaciones del compositor (por ejemplo, añadir un listener).

Formato: compositor_logic <Nombre>

El registro de lógicas de compositor se realiza a través del nombre con CompositorManager::registerCompositorLogic.

3.2.2 PASADAS OBJETIVO

Un pasada objetivo es la acción de renderizar un objetivo determinado, ya sea una textura de renderizado o el resultado final. Se puede actualizar la misma textura de renderizado varias veces añadiendo más de una pasada objetivo al script compositor - esto es muy útil para renderizados de 'ping-pong' entre un par de texturas de renderizado para realizar circunvoluciones complejas que no se pueden hacer en un solo renderizado, tales como imágenes borrosas.

Hay dos tipos de pasadas objetivo, el que actualiza una textura de renderizado:

Formato: target <Nombre> { }

... y el que define la salida del renderizado final:

Formato: target_output { }

Los contenidos de ambas son idénticos, la única diferencia real es que sólo se puede tener una entrada de target_output, mientras que se pueden tener muchas entradas de objetivo. Éstos son los atributos que se pueden utilizar en una sección 'target' o 'target_output' de un script .compositor:

- Input
- only_initial
- visibility_mask
- lod_bias
- material_scheme
- shadows
- pass

DESCRIPCIÓN DE LOS ATRIBUTOS

input

Establece el modo de entrada del objetivo, lo que indica la pasada objetivo que se transmite antes de que cualquiera de sus propias pasadas sean renderizadas.

Formato: input (none | previous)
Por defecto: input none

none

El objetivo no tendrá nada de entrada, todos los contenidos del objetivo deben ser generados usando sus propias pasadas. Nota: esto no significa que el objetivo será vacío, simplemente no hay datos a tirar. Por ello, para que realmente esté en blanco, se necesitaría una pasada 'limpia' dentro de este objetivo.

previous

El objetivo se tirará en el contenido anterior del visor. Este será o bien la escena original si este es el primer compositor de la cadena, o será la salida del compositor anterior en la cadena de compositores, si el visor tiene múltiples compositores habilitado.

only_initial

Si se establece, esta pasada objetivo sólo se ejecutará una vez inicialmente después de que el efecto haya sido habilitado. Esto podría ser útil para llevar a cabo renderizados de una vez al frente, después de que el contenido estático sea utilizado por el resto del compositor.

Formato: only_initial (on | off)
Por defecto: only_initial off

visibility_mask

Establece la máscara de visibilidad de cualquier pasada render_scene realizada en esta pasada objetivo. Esta es una máscara de bits (aunque debe especificarse como decimal, no hexadecimal) y se mapea con SceneManager::setVisibilityMask.

Formato: visibility_mask <máscara>
Por defecto: visibility_mask 4294967295

lod_bias

Establece la tendencia LOD de la escena para cualquier pasada render_scene realizada en esta pasada objetivo. El valor por defecto es de 1.0, por debajo que significa menor calidad, el aumento mayor calidad.

Formato: lod_bias <valor>
Por defecto: lod_bias 1.0

shadows

Establece si las sombras deben ser renderizadas durante cualquier pasada render_scene realizada en esta pasada objetivo. El valor predeterminado es ‘on’.

Formato: shadows (on | off)
Por defecto: shadows on

material_scheme

Si se establece, indica el esquema de material a utilizar para cualquier pasada render_scene. Útil para realizar efectos especiales de renderizado.

Formato: material_scheme <nombre proyecto>
Valor predeterminado: Ninguno

3.2.3 PASADAS DE COMPOSITOR

Una pasada es una acción de renderizado única que se realiza en una pasada objetivo.

Formato: 'pass' (render_quad | clear | stencil | render_scene | render_custom) [custom name] { }

Hay cuatro tipos de pasada:

clear

Este tipo de pasada establece el contenido de uno o más búferes en el objetivo a un valor fijo. Así que esto puede borrar el búfer de color a un color preciso, el buffer de profundidad a un determinado conjunto de contenidos, llena el buffer de plantilla con un valor, o cualquier combinación de los anteriores.

stencil

Este tipo de pasadas configuran las operaciones de plantilla para las siguientes pasadas. Se puede configurar la plantilla de la función de comparación, las operaciones y los valores de referencia para que se puedan realizar sus propios efectos de plantillas.

render_scene

Este tipo de pasada realiza una renderización periódica de la escena. Se utilizará visibility_mask, lod_bias, y material_scheme del objetivo de las pasadas padres.

render_quad

Este tipo de pasada renderiza un cuadrado en todo el objetivo de renderizado, utilizando un material determinado. Sin duda, se va a querer tirar en los resultados de otra pasada objetivo esta operación para realizar efectos de pantalla completa.

render_custom

Este tipo de pasada es una retrollamada al código de usuario para la pasada de compositor especificada en el nombre personalizado (y registrado vía CompositorManager::registerCustomCompositionPass) y permite al usuario crear operaciones de renderizado personalizadas para efectos avanzados. Este es el único tipo de pasada que requiere el parámetro de nombre personalizado 'custom name'.

Éstos son los atributos que se pueden utilizar en una sección 'pass' de un script .compositor:

ATRIBUTOS DE PASADA DISPONIBLES

- material
- input
- identifier
- first_render_queue
- last_render_queue
- material_scheme
- clear
- stencil

material

Para pasadas de tipo 'render_quad', establece el material utilizado para renderizar el cuadrado. Se tendrán que usar los shaders en este material para realizar efectos de pantalla completa, y utilizar el atributo de entrada (input) para mapear otros objetivos de textura en los enlaces de textura necesitados por este material.

Formato: material <Nombre>

input

Para pasadas de tipo 'render_quad', esta es la forma de asignar una o más texturas de renderizado locales (Ver compositor_texture) en el material que se está utilizando para renderizar el cuadrado de pantalla completa. Para ligar más de una textura, se repete este atributo con los índices de muestras diferentes.

Formato: input <muestra><Nombre> [<índiceMRT>]

Muestra
La textura de muestra a establecer, debe ser un número en el intervalo [0, OGRE_MAX_TEXTURE_LAYERS-1].

Nombre
El nombre de la textura de renderizado local a ligar, como se ha declarado en compositor_texture y renderizado en una o más pasadas objetivo.

ÍndiceMRT
Si la textura local que se está haciendo referencia es un objetivo de procesamiento múltiple (MRT), identifica la superficie de la MRT que se desea hacer referencia (0 es la primera superficie, 1, el segundo, etc).

Ejemplo: input 0 rt0

Identifier

Asocia un identificador numérico con la pasada. Esto es útil para el registro de un oyente con el compositor (CompositorInstance::addListener), y para ser capaz de identificar qué pasada es la que está siendo procesada cuando se dan acontecimientos en relación con ella. Se admiten números entre 0 y 2^32.

Formato: identifier <número>
Ejemplo: identifier 99945
Por defecto: identifier 0

first_render_queue

Para pasadas de tipo 'render_scene', establece el primer identificador de cola de renderización que se incluye en el renderizador. Por defecto es el valor de RENDER_QUEUE_SKIES_EARLY.

Formato: first_render_queue <id>
Por defecto: first_render_queue 5

last_render_queue

Para pasadas de tipo 'render_scene', establece el último identificador de cola de renderización que está incluido en el renderizador. Por defecto es el valor de RENDER_QUEUE_SKIES_LATE.

Formato: last_render_queue <id>
Por defecto: last_render_queue 95

material_scheme

Si se establece, indica el esquema de material a utilizar sólo para esta pasada. Es útil para realizar efectos de renderizado de casos especiales.

Esto reescribirá el esquema si se establece en el ámbito del objetivo.

Formato: material_scheme <nombre esquema>
Por defecto: nada

SECCIÓN CLEAR

Para pasadas de tipo 'clear', esta sección define los parámetros de vaciado de buffer.

Formato: clear { }

Éstos son los atributos que se pueden utilizar en una sección 'clear' de un script .compositor:

1. Buffers
2. colour_value
3. depth_value
4. stencil_value

buffers

Establece los búferes limpiados en esta pasada.

Formato: buffers [colour] [depth] [stencil]
Por defecto: buffers colour depth

colour_value

Ajusta el color utilizado para rellenar el búfer de color en esta pasada, si el búfer de color está limpio (buffers).

Formato: colour_value <rojo><verde><azul><alfa>
Por defecto: colour_value 0 0 0 0

depth_value

Establece el valor de la profundidad utilizado para llenar el búfer de profundidad en esta pasada, si el búfer de profundidad está limpio (buffers).

Formato: depth_value <profundidad>
Por defecto: depth_value 1.0

stencil_value

Establece el valor plantilla utilizado para llenar el búfer de plantilla para esta pasada, si el búfer de plantilla está limpio (buffers).

Formato: stencil_value <valor>
Por defecto: stencil_value 0.0

SECCIÓN STENCIL

Para pasadas de tipo 'stencil', esta sección define los parámetros de operación de plantilla.

Formato: stencil { }

Éstos son los atributos que se pueden utilizar en una sección 'stencil' de un script .compositor:

- check
- comp_func
- ref_value
- mask
- fail_op
- depth_fail_op
- pass_op
- two_sided

check

Activa o desactiva la comprobación de plantilla, permitiendo así el uso del resto de las características de esta sección. El resto de las opciones en esta sección no hacen nada si la plantilla de verificación está desactivada.

Formato: check (on | off)

comp_func

Establece la función utilizada para realizar la siguiente comparación:

(ref_value & mask) comp_func (Stencil Buffer Value & mask)

Lo que sucede como resultado de esta comparación será una de las 3 acciones en el búfer de plantillas, dependiendo de si la prueba falla, no falla pero el verificado de búfer de profundidad falla, o no falla y la verificación de búfer de profundidad tampoco. Para establecer las acciones en el fail_op, depth_fail_op y pass_op respectivamente. Si la plantilla de verificación falla, no se escriben colores o profundidades en el búfer de marco (frame).

Formato: comp_func (always_fail | always_pass | less | less_equal |not_equal| greater_equal| greater)
Por defecto: comp_func always_pass

ref_value

Establece el valor de referencia utilizado para comparar con el búfer de plantilla como se describe en comp_func.

Formato: ref_value <valor>

Por defecto: ref_value 0.0

mask

Establece la máscara utilizada para comparar con el buffer de plantilla como se describe en comp_func.

Formato: mask <valor>
Por defecto: mask 4294967295

fail_op

Establece qué hacer con el valor del búfer de plantilla si el resultado de la comparación de plantilla (comp_func) y comparación en profundidad fracasan.

Formato: fail_op (keep | zero | replace | increment | decrement | increment_wrap | decrement_wrap | invert)
Por defecto: depth_fail_op keep

Estas acciones significan:

keep
Deja el búfer de plantilla sin cambios.

zero
Establece el valor de plantilla a cero.

replace
Establece el valor de plantilla al valor de referencia.

increment
Agrega un valor a la plantilla, hasta el valor máximo.

decrement
Resta un valor a la plantilla, hasta 0.

increment_wrap
Agrega un valor a la plantilla, volviendo a 0 en el máximo.

decrement_wrap
Resta un valor a la plantilla, volviendo al máximo por debajo de 0.

invert
Invierte el valor de plantilla.

depth_fail_op

Establece qué hacer con el valor del búfer de plantilla si el resultado de la comparación de plantilla (comp_func) pasa, pero la comparación en profundidad da un error.

Formato: depth_fail_op (keep | zero | replace | increment | decrement | increment_wrap | decrement_wrap | invert)
Por defecto: depth_fail_op keep

pass_op

Establece qué hacer con el valor de búfer de plantilla si el resultado de la comparación de plantilla (comp_func) y la comparación en profundidad pasan satisfactoriamente.

Formato: pass_op (keep |zero | replace | increment | decrement | increment_wrap | decrement_wrap | invert)
Por defecto: pass_op keep

two_sided

Activa o desactiva operaciones de plantilla de dos caras, lo que significa que el inverso de las operaciones se aplica a las caras posteriores de los polígonos.

Formato: two_sided (on | off)
Por defecto: two_sided off

3.2.4. APLICANDO UN COMPOSITOR

Añadir una instancia de compositor a un visor es muy simple. Todo lo que se necesita hacer es:

```
CompositorManager::getSingleton().addCompositor(viewport,
compositorName);
```

Código 45: Aplicando un Compositor.

Donde viewport es un puntero al visor, y compositorName es el nombre del compositor a crear una instancia. De esta manera, una nueva instancia de un compositor se añadirá a una nueva cadena de compositor del visor. Se puede llamar al método varias veces para agregar más compositores a la cadena en este visor. De forma predeterminada, cada compositor que se añade está desactivado, pero se puede cambiar este estado llamando a:

```
CompositorManager::getSingleton().setCompositorEnabled(viewport,
compositorName, enabledOrDisabled);
```

Código 46: Habilitando un Compositor.

Para más información definiendo y usando compositors, ver Demo_Compositor en el área de ejemplos, junto con el script Examples.compositor en el área media.

3.3 SCRIPTS DE PARTÍCULAS

Los scripts de partículas permiten definir sistemas de partículas para ser instanciados en el código sin tener que codificar la configuración en el código fuente, lo que permite un cambio muy rápido en los cambios que se realicen. Los sistemas de partículas que se definen en los scripts se utilizan como plantillas, y los múltiples sistemas actuales pueden ser creados a partir de ellos en tiempo de ejecución.

CARGANDO SCRIPTS

Los scripts de sistemas de partículas se cargan en el momento de la inicialización del sistema: por defecto se ven en todos los lugares de recursos comunes (véase Root::addResourceLocation) para los archivos con la extensión '.particle' y se analizan. Si se desea analizar los archivos con una extensión diferente, se utiliza el método ParticleSystemManager::getSingleton().parseAllSources con una extensión propia, o si se desea analizar un archivo individual, se usa ParticleSystemManager::getSingleton().parseScript.

Una vez que se han analizado los scripts, el código es libre de instanciar sistemas basados en ellos utilizando el método SceneManager::createParticleSystem() que puede tomar el nombre para el nuevo sistema, y el nombre de la plantilla en la que se basará (este nombre de la plantilla está en el script).

FORMATO

Varios sistemas de partículas se pueden definir en un único script. El formato del script es pseudo-C ++, con secciones delimitadas por llaves ({}), y comentarios indicados por comenzar una línea con '//' (nota, no está permitido anidar comentarios). El formato general se muestra a continuación en un ejemplo típico:

```
// A sparkly purple fountain
particle_system Examples/PurpleFountain
{
    material Examples/Flare2
    particle_width 20
    particle_height 20
    cull_each false
    quota 10000
billboard_type oriented_self

    // Area emitter
    emitter Point
    {
        angle 15
        emission_rate 75
        time_to_live 3
        direction 0 1 0
        velocity_min 250
        velocity_max 300
        colour_range_start 1 0 0
        colour_range_end 0 0 1
    }

    // Gravity
    affector LinearForce
    {
        force_vector 0 -100 0
        force_application add
    }

    // Fader
    affector ColourFader
{
        red -0.25
        green -0.25
blue -0.25
    }
```

```
}
```

Código 47: Ejemplo de Script de sistema de partículas.

Cada sistema de partículas en el script debe tener un nombre, que es la línea antes de la primera apertura '{', en el ejemplo de esto es 'Examples/PurpleFountain'. Este nombre debe ser único global. Se pueden incluir caracteres de ruta de acceso (como en el ejemplo) que lógicamente divide los sistemas de partículas, y evita nombres duplicados, pero el motor no considera el nombre como jerárquica, sino como una cadena.

Un sistema puede tener atributos de alto nivel establecidos mediante los comandos de scripts disponibles, tales como 'quota' para establecer el número máximo permitido de partículas en el sistema. Emisores (que crean partículas) y affectors (que modifican las partículas) que se añaden como definiciones anidadas dentro del script. Los parámetros disponibles en las secciones emisor y affector dependen totalmente del tipo de emisor/affector.

Para obtener una descripción detallada de los atributos fundamentales del sistema de partículas, consulte la siguiente lista:

Atributos del sistema de partículas disponibles

1. quota
2. material
3. particle_width
4. particle_height
5. cull_each
6. billboard_type
7. billboard_origin
8. billboard_rotation_type
9. common_direction
10. common_up_vector
11. renderer
12. sorted
13. local_space
14. point_rendering
15. accurate_facing
16. iteration_interval
17. nonvisible_update_timeout

Ver también: 3.3.2. Emisores de partículas, 3.3.5. Affectors de partículas

3.3.1 ATRIBUTOS DEL SISTEMA DE PARTÍCULAS

Esta sección describe los atributos que se pueden establecer en cada sistema de partículas utilizando scripts. Todos los atributos tienen valores por defecto de forma que todos los ajustes son opcionales en el script.

quota

Establece el número máximo de partículas a la vez que este sistema puede contener. Cuando se agote este límite, los emisores no podrán emitir más partículas hasta que algunas sean destruidas (por ejemplo, a través de que se agote su time_to_live). Se debe de tener en cuenta que casi siempre se quiere que esto cambie, ya que por defecto se tiene un valor muy bajo (las agrupaciones de partículas siempre aumentan de tamaño, nunca disminuyen).

Formato: quota <max_particulas>
Ejemplo: quota 10000
Valor por defecto: 10

material

Establece el nombre del material que todas las partículas en este sistema van a utilizar. Todas las partículas en un sistema utilizan el mismo material, aunque cada partícula puede tintar este material mediante el uso de su propiedad de color.

Formato: material <nombre_material>
Ejemplo: material Examples/Flare
Valor por defecto: ninguno (material en blanco)

particle_width

Define la anchura de las partículas en coordenadas del mundo. Hay que tener en cuenta que esta propiedad es absoluta cuando billboard_type (ver abajo) se establece a 'point' o 'perpendicular_self', pero es escalado por la longitud del vector de dirección cuando billboard_type es 'oriented_common', 'oriented_self' o 'perpendicular_common'.

Formato: particle_width <ancho>
Ejemplo: particle_width 20
Valor por defecto: 100

particle_height

Establece la altura de las partículas en coordenadas del mundo. Hay que tener en cuenta que esta propiedad es absoluta cuando billboard_type (ver abajo) se establece a 'point' o 'perpendicular_self', pero es escalado por la longitud del vector de dirección cuando billboard_type es 'oriented_common', 'oriented_self' o 'perpendicular_common'.

Formato: particle_height <altura>
Ejemplo: particle_height 20
Valor por defecto: 100

cull_each

Todos los sistemas de partículas son sacrificados por la caja que contiene todas las partículas en el sistema. Esto es normalmente suficiente para sistemas de partículas bastante limitados a nivel local donde la mayoría de las partículas están juntas, ya sean visibles o no visibles. Sin embargo, para los sistemas en que las partículas están repartidas en un área más amplia (por ejemplo, un sistema de lluvia), es posible que se desee realmente sacrificar cada partícula individualmente para ahorrar tiempo, ya que es mucho más probable que sólo un subconjunto de las partículas sean visibles. Para ello, se establece el parámetro cull_each a true.

Formato: cull_each <true|false>
Ejemplo: cull_each true
Valor por defecto: false

renderer

Los sistemas de partículas no se renderizan a sí mismos, lo hacen a través de clases ParticleRenderer. Estas clases son registradas con un manager a fin de ofrecer sistemas de partículas con un 'aspecto' particular. Ogre viene configurado por defecto con un renderizador basado en carteleras, pero se pueden añadir más a través de plugins. Los renderizadores de partículas se registran con un nombre único, y se puede utilizar ese nombre en este atributo para determinar el renderizador a usar. El valor predeterminado es 'billboard'.

Los renderizadores de partículas pueden tener atributos, que se pueden pasar mediante el establecimiento de ellos en el sistema de partículas de raíz.

Formato: renderer <nombre_renderizador>
Valor por defecto: billboard

sorted

De forma predeterminada, las partículas no están ordenadas. Al establecer este atributo a 'true', las partículas serán ordenadas con respecto a la cámara, las más lejanas en primer lugar. Esto puede hacer que algunos efectos de renderizado vayan mejor con un gasto de clasificación pequeño.

Formato: order <true|false>
Valor por defecto: false

local_space

De forma predeterminada, las partículas son emitidas en el espacio mundial, de manera que si se transforma el nodo al que está conectado el sistema, no afectará a las partículas (sólo a los emisores). Esto tiende a dar el comportamiento normal esperado, que es el modelarlo como partículas del mundo real que viajan de forma independiente con respecto a los objetos emisores. Sin embargo, para crear algunos de los efectos es posible que se desee que las partículas permanezcan unidas al espacio local donde está el emisor y le sigan directamente. Esta opción permite hacer eso.

Formato: local_space <true|false>
Valor por defecto: false

billboard_type

Este es en realidad un atributo del renderizador de partículas 'billboard' (por defecto), y es un ejemplo de transmisión de atributos a un renderizador de partículas que se declaró directamente en la declaración del sistema. Las partículas son renderizadas usando el renderizador predeterminado de cartelera (billboard), que son rectángulos formados por 2 triángulos que giran para hacer frente a la dirección dada. Sin embargo, hay más de 1 manera de orientar una cartelera. En el enfoque clásico la cartelera está directamente frente a la cámara: este es el comportamiento por defecto. Sin embargo, con esta colocación sólo se ve bien para partículas que están representando algo vagamente esférico como una luz de bengala. Para efectos más lineales como un rayo láser, realmente se quiere que la partícula tenga una orientación propia.

Formato: billboard_type
<point|oriented_common|oriented_self|perpendicular_common|perpendicular_self>
Ejemplo: billboard_type oriented_self
Valor por defecto: point

Las opciones para estos parámetros son:

point

Colocación predeterminada, se aproxima a partículas esféricas y la cartelera siempre se enfrenta a la cámara.

oriented_common

Las partículas se orientan en torno a un vector común, generalmente de dirección fija (véase common_direction), que actúa como su eje local Y. El cartel sólo gira en torno a este eje, dando a la partícula cierto sentido de dirección. Bueno para tormentas, campos estelares, etc. donde las partículas viajan en una sola dirección - esto es ligeramente más rápido que oriented_self (véase más abajo).

oriented_self

Las partículas se orientan en torno a su propio vector de dirección, que actúa como su eje Y local. Como la dirección de los cambios de las partículas, por lo que el cartel se reorienta para hacer frente a esta forma. Bueno para rayos láser, fuegos artificiales y otras partículas 'no uniformes' que debe parecer que están viajando en su propia dirección.

perpendicular_common

Las partículas son perpendiculares a un vector común, generalmente de dirección fijada (véase common_direction), que actúa como eje Z local, y su eje Y local coplanario con dirección común y el vector común de subida (ver common_up_vector). La cartelera no gira para hacer frente a la cámara, se puede utilizar el material de doble cara para garantizar que las partículas no sean sacrificadas por las caras traseras. Bueno para aureolas, anillos, etc, donde las partículas van perpendiculares a la tierra - esto es ligeramente más rápido que perpendicular_self (véase más abajo).

perpendicular_self

Las partículas son perpendiculares a su vector de dirección propio, que actúa como su eje Z local, y su eje Y local coplanar con su vector de dirección propio y el vector común de subida (ver common_up_vector). La cartelera no gira para hacer frente a la cámara, se puede utilizar material de doble cara para garantizar que las partículas no sean sacrificadas por las caras traseras. Bueno para pilas de anillos, etc, donde las partículas van perpendiculares a su dirección de viaje.

billboard_origin

Especifica el punto que actúa como punto de origen para todas las partículas de la cartelera, controla el ajuste fino donde una partícula de cartel aparece en relación con su posición.

Formato:
billboard_origin<top_left|top_center|top_right|center_left|center|center_right|bottom_left|bottom_center|bottom_right>
Ejemplo: billboard_origin top_right
Valor por defecto: center

Las opciones para estos parámetros son:

top_left

El origen del Billboard es la esquina superior izquierda.

top_center

El origen del Billboard es el centro del borde superior.

top_right

El origen del Billboard es la esquina superior derecha.

center_left
El origen del Billboard es el centro del borde izquierdo.

center
El origen del Billboard es el centro.

center_right
El origen del Billboard es el centro del borde derecho.

bottom_left
El origen del Billboard es la esquina inferior izquierda.

bottom_center
El origen del Billboard es el centro del borde inferior.

bottom_right
El origen del Billboard es la esquina inferior derecha.

billboard_rotation_type

De forma predeterminada, las partículas de cartel rotarán las coordenadas de textura de acuerdo con la rotación de las partículas. Sin embargo, rotar las coordenadas de textura tiene algunas desventajas, por ejemplo, las esquinas de las texturas se pierden después de girar, y las esquinas de la cartelera se llenarán con zonas textura no deseadas utilizando el modo de dirección wrap o con muestreo de sub-textura. Esta configuración permite especificar otros tipos de rotación.

Formato: billboard_rotation_type <vertex|texcoord>
Ejemplo: billboard_rotation_type vertex
Valor por defecto: texcoord

Las opciones para este parámetro son:

vertex
Las Partículas Billboard girarán en torno a los vértices de su dirección de orientación de acuerdo con la rotación de la partícula. Rotar vértices garantiza que las esquinas de textura coincidan exactamente con las esquinas de la cartelera (Billboard), con la ventaja ya mencionada, pero debería tomar más tiempo para generar los vértices.

texcoord
Las partículas de Billboard rotarán las coordenadas de textura de acuerdo con la rotación de la partícula. Rotar coordenadas de textura es más rápido que rotar vértices, pero tiene algunas desventajas mencionadas anteriormente.

common_direction

Sólo es necesario si billboard_type se establece en oriented_common o perpendicular_common, este vector es el vector de dirección común utilizado para orientar todas las partículas en el sistema.

Formato: common_direction <x><y><z>
Ejemplo: common_direction 0 -1 0
Valor por defecto: 0 0 1

Ver también: 3.3.2 Emisores de partículas, 3.3.5 Affectors de partículas

common_up_vector

Sólo es necesario si billboard_type se establece en perpendicular_self o perpendicular_common, este vector es el vector común utilizado para orientar todas las partículas en el sistema.

Formato: common_up_vector <x><y><z>
Ejemplo: common_up_vector 0 1 0
Valor por defecto: 0 1 0

Ver también: 3.3.2 Emisores de partículas, 3.3.5 Affectors de partículas

point_rendering

Esto es en realidad un atributo del renderizador de partícula de 'Billboard' (por defecto), y establece si la cartelera utilizará renderizado de punto en lugar de cuadrados generados manualmente.

Por defecto, una cartelera (BillboardSet) se renderiza mediante la generación de la geometría de un cuadrado de textura en la memoria, teniendo en cuenta el tamaño y la configuración de orientación, y cargándolo en la tarjeta de vídeo. La alternativa es usar la renderización punto de hardware, lo que significa que sólo se tiene que enviar una posición por la cartelera en lugar de 4 y el tipo de hardware ordena cómo esto se renderiza basado en el estado del renderizador.

Usar el punto de renderizado es más rápido que la generación de cuadrados manualmente, pero es más restrictivo. Se aplican las siguientes restricciones:

1. Sólo es compatible el tipo de orientación 'point'.
2. El tamaño y la apariencia de cada partícula son controlados por la pasada del material (point_size, point_size_attenuation, point_sprites).
3. El tamaño por partícula no es compatible (se deriva de la anterior).
4. La rotación por partícula no es compatible, y esto sólo puede ser controlado mediante la rotación de la unidad de textura en la definición del material.
5. Sólo el origen 'center' es compatible.
6. Algunos controladores tienen un límite superior para el tamaño de los puntos que soportan - ¡Esto puede variar entre APIs de la misma tarjeta! No hay que confiar en tamaños de punto que causan sprites de punto muy grandes en la pantalla, ya que pueden obtener restricciones en algunas tarjetas. Los tamaños superiores pueden variar desde 64 hasta 256 píxeles.

Es casi seguro que se desee habilitar en la pasada de material los dos puntos de atenuación y sprites si se utiliza esta opción.

accurate_facing

Esto es en realidad un atributo del renderizador de partículas 'Billboard' (por defecto), y establece si la cartelera utilizará un cálculo más lento pero más preciso para hacer frente de la cartelera a la cámara. Por defecto se usa la dirección de la cámara, que es más rápido, pero significa que las carteleras no se quedan en la misma orientación en la que gira la cámara. La opción 'accurate_facing true' hace el cálculo basado en un vector de cada cartel a la cámara, lo que significa que la orientación es constante, incluso mientras que la cámara rota.

Formato: accurate_facing on | off
Por defecto: accurate_facing off 0

iteration_interval

Normalmente, los sistemas de partículas se actualizan en base de la tasa de fotogramas; sin embargo esto puede dar resultados variables con más intervalos de porcentajes de fotogramas extremos, sobre todo en menores tasas de fotogramas. Se puede utilizar esta opción para que la frecuencia de actualización de un intervalo fijado, con unas tasas de fotogramas más bajas, la actualización de las partículas se repite en el intervalo fijado hasta que el plazo de tiempo se agote. Un valor de 0 significa el valor predeterminado de iteración de fotograma de tiempo.

Formato: iteration_interval <secs>
Ejemplo: iteration_interval 0.01
Valor por defecto: iteration_interval 0

nonvisible_update_timeout

Establece cuándo el sistema de partículas debe de dejar de actualizar después de que no haya sido visible durante un tiempo. Por defecto, los sistemas de partículas visibles se actualizan todo el tiempo, incluso cuando no están a la vista. Esto significa que están garantizados para ser coherentes cuando entran en la vista. Sin embargo, esto tiene un coste, la actualización de sistemas de partículas puede ser costosa, especialmente si son perpetuas.

Esta opción le permite establecer un 'tiempo de espera' en el sistema de partículas, de modo que si no es visible por esta cantidad de tiempo, no se actualizará hasta que sea visible. Un valor de 0 desactiva el tiempo de espera y siempre actualiza.

Formato: nonvisible_update_timeout <secs>
Ejemplo: nonvisible_update_timeout 10
Por defecto: nonvisible_update_timeout 0

3.3.2 EMISORES DE PARTÍCULAS

Los emisores de partículas se clasifican por 'type' (tipo) por ejemplo, los emisores de punto 'point' emiten desde un solo punto mientras que los emisores de caja 'Box' emiten al azar desde una zona. Se pueden añadir a Ogre nuevos emisores mediante la creación de plugins. Se agrega un emisor a un sistema anidando otra sección dentro de él, encabezada con la palabra clave 'emisor' seguida del nombre del tipo de emisor (mayúsculas y minúsculas). Ogre actualmente soporta emisores 'Point' (punto), 'Box' (Caja), 'Cylinder' (Cilindro), 'Ellipsoid' (Elipsoide), 'HollowEllipsoid' (Elipsoide con hueco) y 'Ring' (anillo).

También es posible 'emitir emisores' - es decir, han generado nuevos emisores sobre la base de la posición de las partículas. Vea la sección Emitiendo Emisores

Atributos Universales del Emisor de partículas

1. angle
2. colour
3. colour_range_start
4. colour_range_end
5. direction
6. emission_rate
7. position
8. velocity
9. velocity_min
10. velocity_max
11. time_to_live

12. time_to_live_min
13. time_to_live_max
14. duration
15. duration_min
16. duration_max
17. repeat_delay
18. repeat_delay_min
19. repeat_delay_max

Ver también 3.3 Scripts de partículas, 3.3.5 Affectors de partículas

3.3.3 ATRIBUTOS DE EMISOR DE PARTÍCULAS

Esta sección describe los atributos comunes de todos los emisores de partículas. Los emisores específicos también pueden tener sus propios atributos extra.

angle

Establece el ángulo máximo (en grados) que las partículas emitidas pueden desviarse de la dirección del emisor (ver direction). Al establecer esta a 10 permite a las partículas desviarse hasta 10 grados en cualquier dirección fuera de la dirección del emisor. Un valor de 180 significa que se emiten en cualquier dirección, mientras que 0 significa que emiten siempre exactamente en la dirección del emisor.

Formato: angle <grados>
Ejemplo: angle 30
Valor por defecto: 0

colour

Establece un color estático para todas las partículas emitidas. Ver también los atributos colour_range_start y colour_range_end para el establecimiento de un rango de colores. El formato del parámetro de color es “r g b a”, donde cada componente es un valor de 0 a 1, y el valor alfa es opcional (se supone 1 si no se especifica).

Formato: color <r><g><b> [<a>]
Ejemplo: color 1 0 0 1
Valor por defecto: 1 1 1 1

colour_range_start y colour_range_end

Como el atributo ‘colour’, con excepción de que estos 2 atributos deben especificarse juntos, e indican el rango de colores disponibles para las partículas emitidas. El color real será elegido al azar entre esos 2 valores.

Formato: como colour
Ejemplo (genera colores al azar entre los colores rojo y azul):
colour_range_start 1 0 0
colour_range_end 0 0 1
Valor por defecto: Ambos con 1 1 1 1

direction

Establece la dirección del emisor. Esto es relativo al SceneNode al que el sistema de partículas se adjunta, lo que significa que al igual que otros objetos móviles, cambiando la orientación del nodo también se mueve el emisor.

Formato: direction <x><y><z>
Ejemplo: direction 0 1 0
Valor por defecto: 1 0 0

emission_rate

Establece cuantas partículas por segundo deben ser emitidas. El emisor específico no tiene que emitir estas en una ráfaga continua - Este es un parámetro relativo y el emisor puede optar por emitir todos segundos valores de las partículas de cada medio segundo, por ejemplo, el comportamiento depende del emisor. La tasa de emisión también estará limitada por el ajuste del 'quota' en el sistema de partículas.

Formato: emission_rate <particulas_por_segundo>
Ejemplo: emission_rate 50
Valor por defecto: 10

position

Establece la posición del emisor en relación con el SceneNode al que el sistema de partículas está conectado.

Formato: position <x><y><z>
Ejemplo: position 10 0 40
Valor por defecto: 0 0 0

velocity

Establece una velocidad constante de todas las partículas en el momento de emisión. Véase también los atributos velocity_min y velocity_max que le permiten establecer un rango de velocidades en lugar de uno fijo.

Formato: velocity <unidades_del_mundo_por_segundo>
Ejemplo: velocity 100
Valor por defecto: 1

velocity_min y velocity_max

Como 'velocity', salvo que estos atributos establecen un rango de velocidad. Cada partícula se emite con una velocidad al azar dentro de este rango.

Formato: como velocity
Ejemplos:
velocity_min 50
velocity_max 100
Valor por defecto: ambos a 1

time_to_live

Establece el número de segundos que cada partícula 'vivirá' antes de ser destruida. Nota: es posible que los affectors de partículas modifiquen esto en vuelo, pero este es el valor dado a las partículas en las emisiones. Véase también los atributos time_to_live_min y time_to_live_max que permiten establecer un rango de vida en lugar de uno fijo.

Formato: time_to_live <segundos>
Ejemplo: time_to_live 10
Valor por defecto: 5

time_to_live_min y time_to_live_max

Como time_to_live, excepto que esta establece un rango de ciclos de vida y cada partícula obtiene un valor aleatorio entre este rango en la emisión.

Formato: como time_to_live
Ejemplo:
time_to_live_min 2
time_to_live_max 5
Valor por defecto: ambos a 5

duration

Establece el número de segundos que el emisor está activo. El emisor se puede iniciar de nuevo, ver repeat_delay. Un valor de 0 significa duración infinita. Véase también los atributos duration_min y duration_max que permiten establecer un rango de duración en vez de uno fijo.

Formato: duration <segundos>
Ejemplo:duration 2.5
Valor por defecto: 0

duration_min y duration_max

Como la duration, a excepción de que estos atributos establecen un rango de tiempo variable entre el mínimo y máximo cada vez que el emisor se inicia.

Formato: como duration
Ejemplo:
duration_min 2
duration_max 5
Valor por defecto: ambos a 0

repeat_delay

Establece el número de segundos a esperar antes de la repetición de emisión cuando se detuvo por una duración limitada. Véase también los atributos repeat_delay_min y repeat_delay_max que permiten establecer un rango de repeat_delays en lugar de un valor fijo.

Formato: repeat_delay <segundos>
Ejemplo:repeat_delay 2.5
Valor por defecto: 0

repeat_delay_min y repeat_delay_max

Como repeat_delay, excepto que estas establecen un rango de demoras repetidas y cada vez que el emisor se inicia obtiene un valor aleatorio entre ambos valores.

Formato: como repeat_delay
Ejemplo:
repeat_delay 2
repeat_delay 5
Valor por defecto: ambos a 0

Ver también: 3.3.4 Emisores de partículas estándar, 3.3 Scripts de partículas, 3.3.5 Affectors de partículas

3.3.4 EMISORES DE PARTÍCULAS ESTÁNDAR

Ogre viene preconfigurado con unos pocos emisores de partículas. Los nuevos se pueden agregar mediante la creación de plugins: ver el proyecto Plugin_ParticleFX como un ejemplo de cómo podría hacerse esto (que es donde se implementan estos emisores).

- Emisor Point (Punto)
- Emisor Box (Caja)
- Emisor Cylinder (Cilindro)
- Emisor Ellipsoid (Elipsoide)
- Emisor Hollow Ellipsoid (Elipsoide con hueco)
- Emisor Ring (Anillo)

Emisor punto

El emisor emite partículas desde un único punto, que es su posición. El emisor no tiene atributos adicionales por encima de los atributos estándar.

Para crear un emisor punto, hay que incluir una sección como ésta dentro del script del sistema de partículas:

```
emitter Point
{
    // Settings go here
}
```

Código 48: Script de emisor de partículas de punto.

Hay que tener en cuenta que el nombre del emisor 'Punto' es sensible a mayúsculas y minúsculas.

Emisor caja

Este emisor emite partículas desde un lugar al azar dentro de una caja de 3 dimensiones. Sus atributos adicionales son:

width (anchura)
Define la anchura de la caja (este es el tamaño de la caja a lo largo de su eje X local, que depende del atributo 'direction', que forma el eje local Z de la caja).

Formato: width <unidades>
Ejemplo: width 250
Valor por defecto: 100

height (altura)

Define la altura de la caja (este es el tamaño de la caja a lo largo de su eje Y local, que depende del atributo 'direction', que forma el eje Z local de la caja).

Formato: height <unidades>
Ejemplo: height 250
Valor por defecto: 100

depth (profundidad)

Establece la profundidad de la caja (este es el tamaño de la caja a lo largo de su eje Z local, que es el mismo que el atributo 'direction').

Formato: depth <unidades>
Ejemplo: depth 250
Valor por defecto: 100

Para crear un emisor de caja, se incluye una sección como ésta dentro de un script del sistema de partículas:

```
emitter Box
{
    // Settings go here
}
```

Código 49: Script de emisor de partículas de caja.

Emisor Cilindro

Este emisor emite partículas en una dirección aleatoria dentro de un área cilíndrica, donde el cilindro está orientado a lo largo del eje Z. El emisor tiene exactamente los mismos parámetros que el emisor de caja, así que no hay parámetros adicionales que considerar aquí - la anchura y la altura determinan la forma del cilindro a lo largo de su eje (si son diferentes, es un cilindro de elipsoide), la profundidad determina la longitud del cilindro.

Emisor Elipsoide

El emisor emite partículas dentro de una zona con forma de elipsoide, es decir, una esfera o área esférica aplastada. Los parámetros son idénticos a los del emisor de caja, excepto que las dimensiones describen los puntos más separados a lo largo de cada uno de los ejes.

Emisor Elipsoide con hueco

Este emisor es como el emisor elipsoide, salvo que hay una zona hueca en el centro del elipsoide de la que no se emiten partículas. Por lo tanto, tiene 3 parámetros adicionales a fin de definir esta área:

inner_width

La anchura de la zona interior que no emite partículas.

inner_height

La altura de la zona interior que no emite partículas.

inner_depth

La profundidad de la zona interior que no emite partículas.

Emisor Anillo

El emisor emite partículas de un área en forma de anillo, es decir, como el emisor elipsoide con hueco, excepto que sólo en 2 dimensiones.

inner_width
La anchura de la zona interior que no emite partículas.

inner_height
La altura de la zona interior que no emite partículas.

Ver también: 3.3 Scripts de partículas, 3.3.2 Emisores de partículas

Emitiendo emisores

Es posible generar nuevos emisores a la expiración de partículas, por ejemplo para efectos de de fuegos artificiales. Esto es controlado a través de las siguientes directivas:

emit_emitter_quota
Este parámetro es un parámetro de nivel de sistema y dice al sistema cuantos emisores emitidos pueden estar en uso en cualquier momento. Esto es sólo para permitir el proceso de asignación de espacio.

name
Este parámetro es un parámetro a nivel de emisor, da un nombre a un emisor. Esto puede ser contemplado en otro emisor como el tipo nuevo de emisor para lanzar cuando una partícula emitida muere.

emit_emitter
Este es un parámetro a nivel de emisor, y si se especifica, significa que cuando las partículas emitidas por este emisor mueren, generan un nuevo emisor del tipo mencionado.

3.3.5 AFFECTORS DE PARTÍCULAS

Los Affectors de partículas modifican las partículas durante su ciclo de vida. Están clasificados por 'tipo' por ejemplo, affectors de 'LinearForce' aplican una fuerza a todas las partículas, mientras que affectors 'ColourFader' alteran el color de las partículas en vuelo. Nuevos affectors se pueden añadir a Ogre mediante la creación de plugins. Para agregar un affector a un sistema hay que anidar otra sección en el mismo, encabezada con la palabra clave 'affector' seguida del nombre del tipo de affector (sensible a mayúsculas y minúsculas). Ogre actualmente soporta 'ColourFader' y 'LinearForce'.

Los Affectors de partículas en realidad no tienen atributos universales, sino que todos son específicos para el tipo de affector.

Ver también: 3.3.6 Affectos de partículas estándar, 3.3 Scripts de partículas, 3.3.2 Emisores de partículas

3.3.6 AFFECTOR DE PARTÍCULASESTÁNDAR

Ogre viene preconfigurado con unos pocos affectors de partículas. Los nuevos se pueden agregar mediante la creación de plugins: ver el proyecto Plugin_ParticleFX como un ejemplo de cómo se podría hacer esto (que es donde se implementan estos affectors).

- Affector de fueza lineal
- Affercor ColourFader
- Affercor ColourFader2
- Affector escalador
- Affector rotador
- Affector ColourInterpolator
- Affector ColourImage
- Affector DeflectorPlane
- Affector DirectionRandomiser

Affector de fuerza lineal

Este affector aplica un vector de fuerza a todas las partículas para modificar su trayectoria. Puede ser utilizado para gravedad, viento, o cualquier otra fuerza lineal. Sus atributos adicionales son:

force_vector

Establece el vector de fuerza que debe aplicarse a cada partícula. La magnitud de este vector determina lo fuerte que es la fuerza.

Formato: force_vector <x><y><z>
Ejemplo: force_vector 50 0 -50
Valor por defecto: 0 -100 0 (un efecto de gravedad)

force_application

Establece la forma en que el vector de la fuerza se aplica al momento de la partícula.

Formato: force_application <add|average>
Ejemplo: force_application average
Valor por defecto: add

Las opciones son:

average
El momento resultante es la media del vector de fuerza y el movimiento actual de la partícula. Es auto-estabilizado, pero la velocidad a la que la partícula cambia de dirección no es lineal.

add
El momento resultante es el movimiento actual de la partícula, más el vector de fuerza. Esta es la aceleración de la fuerza tradicional, pero potencialmente puede resultar en una velocidad sin límites.

Para crear un affector de fuerza lineal, hay que incluir una sección como ésta dentro del script del sistema de partículas:

```
affector LinearForce
{
    // Settings go here
}
```

Código 50: Script de affector de fuerza lineal.

Hay que tener en cuenta que el nombre del tipo de affector ('LinearForce') es sensible a mayúsculas y minúsculas.

Affector ColourFader

Este affector modifica el color de las partículas en vuelo. Sus atributos adicionales son:

red

Establece el ajuste para realizarlo en el componente rojo del color de las partículas por segundo.

Formato: red <valor_delta>
Ejemplo: red -0.1
Valor por defecto: 0

green

Establece el ajuste para realizarlo en el componente verde del color de las partículas por segundo.

Formato: green <valor_delta>
Ejemplo: green -0.1
Valor por defecto: 0

blue

Establece el ajuste para realizarlo en el componente azul del color de las partículas por segundo.

Formato: blue <valor_delta>
Ejemplo: blue -0.1
Valor por defecto: 0

alpha

Establece el ajuste para realizarlo en el componente alfa del color de las partículas por segundo.

Formato: alpha <valor_delta>
Ejemplo: alpha -0.1
Valor por defecto: 0

Para crear un affector colourfader, hay que incluir una sección como ésta dentro del script del sistema de partículas:

```
affector ColourFader
{
    // Settings go here
}
```

Código 51: Script de affector de Color.

Affector ColoutFader2

Este affector es similar a la Affector ColourFader, excepto que presenta dos estados de los cambios de color en lugar de sólo uno. El segundo estado de cambio de color se activa una vez que pasa una cantidad especificada de tiempo en la vida de las partículas.

red1

Establece el ajuste para realizarlo en el componente rojo del color de las partículas por segundo para el primer estado.

Formato: red1 <valor_delta>
Ejemplo: red1 -0.1
Valor por defecto: 0

green1

Establece el ajuste para realizarlo en el componente verde del color de las partículas por segundo para el primer estado.

Formato: green1 <valor_delta>
Ejemplo: green1 -0.1
Valor por defecto: 0

blue1

Establece el ajuste para realizarlo en el componente azul del color de las partículas por segundo para el primer estado.

Formato: blue1 <valor_delta>
Ejemplo: blue1 -0.1
Valor por defecto: 0

alpha1

Establece el ajuste para realizarlo en el componente alfa del color de las partículas por segundo para el primer estado.

Formato: alpha1 <valor_delta>
Ejemplo: alpha1 -0.1
Valor por defecto: 0

red2

Establece el ajuste para realizarlo en el componente rojo del color de las partículas por segundo para el segundo estado.

Formato: red2 <valor_delta>
Ejemplo: red2 -0.1
Valor por defecto: 0

green2

Establece el ajuste para realizarlo en el componente verde del color de las partículas por segundo para el segundo estado.

Formato: green2 <valor_delta>
Ejemplo: green2 -0.1
Valor por defecto: 0

blue2

Establece el ajuste para realizarlo en el componente azul del color de las partículas por segundo para el segundo estado.

Formato: blue2 <valor_delta>
Ejemplo: blue2 -0.1
Valor por defecto: 0

alpha2
Establece el ajuste para realizarlo en el componente alfa del color de las partículas por segundo para el segundo estado.

Formato: alpha2 <valor_delta>
Ejemplo: alpha2 -0.1
Valor por defecto: 0

state_change
Establece el momento del tiempo de vida restante de una partícula para que cambie al estado 2.

Formato: state_change <segundos>
Ejemplo: state_change 2
Valor por defecto: 1

Para crear un affector ColourFader2, hay que incluir una sección como ésta en el script del sistema de partículas:

```
affector ColourFader2
{
    // Settings go here
}
```

Código 52: Script de affector de Color 2.

Affector escalador

Este affector escala las partículas en vuelo. Sus atributos extra son:

rate
Especifica la cantidad por la que escalar las partículas en las direcciones x e y por segundo.

Para crear un affector escalador, hay que incluir una sección como ésta en el script del sistema de partículas:

```
affector Scaler
{
    // Settings go here
}
```

Código 53: Script de affector escalador.

Affector rotador

Este affector rota las partículas en vuelo. Se hace rotando la textura. Sus atributos extra son:

rotation_speed_range_start
El inicio de un rango de velocidades de rotación para ser asignadas a las partículas emitidas.

Formato: rotation_speed_range_start <grados_por_segundo>
Ejemplo: rotation_speed_range_start 90
Valor por defecto: 0

rotation_speed_range_end
El fin de un rango de velocidades de rotación para ser asignadas a las partículas emitidas.

Formato: rotation_speed_range_end <grados_por_segundo>
Ejemplo: rotation_speed_range_end 180
Valor por defecto: 0

rotation_range_start
El inicio de un rango de ángulos de rotación para ser asignados a las partículas emitidas.

Formato: rotation_range_start <grados>
Ejemplo: rotation_range_start 0
Valor por defecto: 0

Rotation_range_end
El fin de un rango de ángulos de rotación para ser asignados a las partículas emitidas.

Formato: rotation_range_end <grados>
Ejemplo: rotation_range_end 360
Valor por defecto: 0

Para crear un affector rotador, se incluye una sección como ésta en el script del sistema de partículas:

```
affector Rotator
{
    // Settings go here
}
```

Código 54: Script de affector rotador.

Affector ColourInterpolator

Similar a los affectors ColourFader y ColourFader2, este affector modifica el color de las partículas en vuelo, excepto que tiene un número variable de fases definidas. Hace un intercambio en el color de las partículas en varias etapas en la vida de una partícula e interpola entre ellas. Sus atributos adicionales son:

time0
El punto en el tiempo de la fase 0.

Formato: time0 <0-1 basado en el tiempo de vida>
Ejemplo: time0 0
Valor por defecto: 1

colour0
El color en la fase 0.

Formato: colour0 <r><g><b> [<a>]
Ejemplo: colour0 1 0 0 1
Valor por defecto: 0.5 0.5 0.5 0.0

time1
El punto en el tiempo de la fase 1.

Formato: time1 <0-1 basado en el tiempo de vida>
Ejemplo: time1 0.5
Valor por defecto: 1

colour1
El color en la fase 1.

Formato: colour1 <r><g><b> [<a>]
Ejemplo: colour1 0 0 0 1
Valor por defecto: 0.5 0.5 0.5 0.0

time2
El punto en el tiempo de la fase 2.

Formato: time2 <0-1 basado en el tiempo de vida>
Ejemplo: time2 0
Valor por defecto: 1

colour2
El color en la fase 2.

Formato: colour2 <r><g><b> [<a>]
Ejemplo: colour2 1 0 0 1
Valor por defecto: 0.5 0.5 0.5 0.0

[...]

El número de estados es variable. El número máximo de estados es 6; donde time5 y colour5 son los últimos parámetros posibles. Para crear un affector de interpolación de color, hay que incluir una sección como ésta en el script del sistema de partículas:

```
affector ColourInterpolator
{
    // Settings go here
}
```

Código 55: Script de affector de interpolador de color.

Affector ColourImage

Este es otro affector que modifica el color de las partículas en vuelo, pero en lugar de usar colores definidos programados, los colores se toman de un fichero de imagen especificado. El rango de valores de colores comienza desde el lado izquierdo de la imagen y se mueve hasta el lado derecho durante el ciclo de vida de la partícula, sólo la dimensión horizontal de la imagen se utiliza. Sus atributos extra son:

image
Indica la imagen a utilizar.

Formato: image <nombre_imagen>
Ejemplo: image rainbow.png
Valor por defecto: ninguno

Para crear una affector ColourImage, hay que incluir una sección como ésta dentro del script del sistema de partículas:

```
affector ColourImage
{
    // Settings go here
}
```

Código 56: Script de affector de color con imagen.

Affector DeflectorPlane

Este affector define un plano que desvía partículas que colisionan con él. Sus atributos son:

plane_point
Un punto en el plano desviador. Junto con el vector normal se define el plano.

Por defecto: plane_point 0 0 0

plane_normal
El vector normal del plano desviador. Junto con el punto se define el plano.

Por defecto: plane_normal 0 1 0

bounce
La cantidad de rebote cuando una partícula se desvía. 0 significa que no hay desviación y 1 es el 100 por ciento de la reflexión.

Por defecto: bounce 1.0

Affector DirectionRandomiser

Este affector aplica una aleatoriedad al movimiento de las partículas. Sus atributos extra son:

randomness
La cantidad de aleatoriedad a introducir en cada dirección del eje.

Ejemplo: randomness 5
Por defecto: randomness 1

scope
El porcentaje de partículas afectadas en cada ejecución del affector.

Ejemplo: scope 0.5
Por defecto: scope 1.0

keep_velocity
Determina si la velocidad de las partículas es inalterable.

Ejemplo: keep_velocity true
Por defecto: keep_velocity false

3.4 SCRIPTS DE SUPERPOSICIÓN

Los scripts de superposición ofrecen la posibilidad de definir superposiciones en un script que se pueden reutilizar fácilmente. Mientras que se podrían establecer todas las superposiciones de una escena en el código usando los métodos de las clases SceneManager, Overlay y OverlayElement, en la práctica es un poco difícil de manejar. En su lugar se pueden almacenar definiciones de superposición en archivos de texto que luego pueden ser cargados siempre que sea necesario.

CARGANDO SCRIPTS

Los scripts de superposición se cargan en tiempo de inicialización del sistema: por defecto el sistema ve todos los lugares de recursos comunes (véase Root::addResourceLocation) para los archivos con la extensión '.overlay' y los analiza. Si se desea analizar los archivos con una extensión diferente, hay que utilizar el método OverlayManager::getSingleton().parseAllSources con la extensión propia, o si se desea analizar un archivo individual, se utiliza OverlayManager::getSingleton().parseScript.

FORMATO

Se pueden definir varias superposiciones en un único script. El formato del script es pseudo-C++, con secciones delimitadas por llaves ({}), los comentarios se indican comenzando una línea con '//' (nota, no se pueden anidar comentarios), y la herencia a través del uso de plantillas. El formato general se muestra a continuación en un ejemplo típico:

```
// The name of the overlay comes first
MyOverlays/ANewOverlay
{
zorder 200

    container Panel(MyOverlayElements/TestPanel)
    {
        // Center it horzontally, put it at the top
        left 0.25
        top 0
        width 0.5
        height 0.1
        material MyMaterials/APanelMaterial

        // Another panel nested in this one
container Panel(MyOverlayElements/AnotherPanel)
        {
left 0
             top 0
             width 0.1
             height 0.1
             material MyMaterials/NestedPanel
}
    }
}
```

Código 57: Script de superposición.

El ejemplo anterior define un recubrimiento único llamado 'MyOverlays/ANewOverlay', con 2 paneles en él, uno anidado en el otro. Utiliza indicadores relativos (el valor predeterminado si no se encuentra ninguna opción es metrics_mode).

Cada superposición en el script debe tener un nombre, que es la línea antes de la primera apertura '{'. Este nombre debe ser único global. Se pueden incluir caracteres de la ruta de acceso (como en el ejemplo) para dividir lógicamente las superposiciones, y también para evitar nombres duplicados, pero el motor no trata el nombre como una estructura jerárquica, sino como una cadena. Dentro de las llaves están las propiedades de la plantilla, y todos los elementos anidados. La superposición sólo tiene una propiedad 'zorder' que determina la 'altura' en la pila de superposiciones si más de una se muestra al mismo tiempo. Las superposiciones con valores más altos de zorder se muestran en la parte superior.

AÑADIENDO ELEMENTOS A LA SUPERPOSICIÓN

Dentro de una superposición, se puede incluir cualquier número de elementos en 2D o 3D. Para ello, se define un bloque anidado encabezado por:

'element'
Si desea definir un elemento en 2D que no puede tener hijos.

'container'
Si desea definir un objeto contenedor 2D (que puede tener en sí contenedores o elementos anidados).

Los bloques de elemento y contenedor son bastante idénticos, aparte de su capacidad para almacenar bloques anidados.

BLOQUES 'CONTAINER' / 'ELEMENT'

Estos están delimitados por llaves. El formato de la cabecera anterior a la primera llave es el siguiente:

[container | element] <nombre_tipo> (<nombre_instancia>) [: <nombre_plantilla>]
{...

nombre_tipo
Debe resolverse el nombre de un tipo de OverlayElement que ha sido registrado con el OverlayManager. Los plugins se registran con el OverlayManager para anunciar su capacidad de crear elementos, y en este momento anuncian el nombre del tipo. Ogre viene preconfigurado con los tipos 'Panel', 'BorderPanel' y 'TextArea'.

nombre_instancia
Debe ser un nombre único entre todos los demás elementos o contenedores que permita identificar el elemento. Hay que tener en cuenta que se puede obtener un puntero a cualquier elemento llamado llamando OverlayManager:: getSingleton ().GetOverlayElement (nombre).

nombre_plantilla
Plantilla opcional en la que basar este elemento. Ver plantillas.

Las propiedades que pueden ser incluidas dentro de las llaves dependen del tipo propio. Sin embargo, las siguientes son siempre válidas:

1. metrics_mode
2. horz_align
3. vert_align
4. left
5. top
6. width

7. height
8. material
9. caption

PLANTILLAS

Se pueden utilizar plantillas para crear numerosos elementos con las mismas propiedades. Una plantilla es un elemento abstracto y no se añade a un recubrimiento. Actúa como una clase base que los elementos pueden heredar y obtener sus propiedades por defecto. Para crear una plantilla, la palabra clave 'template' debe ser la primera palabra en la definición del elemento (antes de contenedor o elemento). El elemento de plantilla se crea en el ámbito de aplicación más alto - no es especificado en una superposición. Se recomienda que se definan las plantillas en una superposición distinta aunque esto no es esencial. Teniendo las plantillas definidas en un archivo separado permitirá un aspecto diferente para ser sustituido fácilmente.

Los elementos pueden heredar una plantilla de una manera similar a la herencia C++ - utilizando el operador ':' en la definición de elemento. El operador ':' se coloca después del corchete de cierre del nombre (separados por un espacio). El nombre de la plantilla a heredar se coloca después del operador ':' (también separados por un espacio).

Una plantilla puede contener plantillas hija que se crean cuando la plantilla es una subclase e instanciada. Usar la palabra clave de la plantilla para los hijos de una plantilla es opcional pero recomendado para una mayor claridad, ya que los hijos de una plantilla siempre van a ser plantillas de sí mismos.

```
template container BorderPanel(MyTemplates/BasicBorderPanel)
{
left 0
    top 0
    width 1
    height 1

// setup the texture UVs for a borderpanel

// do this in a template so it doesn't need to be redone everywhere
    material Core/StatsBlockCenter
    border_size 0.05 0.05 0.06665 0.06665
    border_material Core/StatsBlockBorder
    border_topleft_uv 0.0000 1.0000 0.1914 0.7969
    border_top_uv 0.1914 1.0000 0.8086 0.7969
    border_topright_uv 0.8086 1.0000 1.0000 0.7969
    border_left_uv 0.0000 0.7969 0.1914 0.2148
    border_right_uv 0.8086 0.7969 1.0000 0.2148
    border_bottomleft_uv 0.0000 0.2148 0.1914 0.0000
    border_bottom_uv 0.1914 0.2148 0.8086 0.0000
border_bottomright_uv 0.8086 0.2148 1.0000 0.0000
}
template container Button(MyTemplates/BasicButton) :
MyTemplates/BasicBorderPanel
{
left 0.82
    top 0.45
    width 0.16
    height 0.13
    material Core/StatsBlockCenter
    border_up_material Core/StatsBlockBorder/Up
    border_down_material Core/StatsBlockBorder/Down
```

```
}
template element TextArea(MyTemplates/BasicText)
{
    font_name Ogre
    char_height 0.08
    colour_top 1 1 0
    colour_bottom 1 0.2 0.2
    left 0.03
    top 0.02
    width 0.12
    height 0.09
}

MyOverlays/AnotherOverlay
{
zorder 490
    container BorderPanel(MyElements/BackPanel) :
MyTemplates/BasicBorderPanel
{
left 0
        top 0
        width 1
        height 1

        container Button(MyElements/HostButton) :
MyTemplates/BasicButton
{
left 0.82
            top 0.45
            caption MyTemplates/BasicText HOST
}

container Button(MyElements/JoinButton) : MyTemplates/BasicButton
{
left 0.82
            top 0.60
            caption MyTemplates/BasicText JOIN
}
    }
}
```

Código 58: Script de definición de plantillas de recubrimiento.

El ejemplo anterior utiliza plantillas para definir un botón. Hay que tener en cuenta que el botón de plantilla se hereda de la plantilla borderPanel. Esto reduce el número de atributos necesarios para crear una instancia de un botón.

También hay que tener en cuenta que la instancia de un botón necesita un nombre de la plantilla para el atributo de título. Así las plantillas también pueden ser utilizadas por los elementos que necesitan creación dinámica de elementos hijo (el botón se crea un TextAreaElement en este caso para su título).

Véase la sección 3.4.1 Atributos OverlayElement, 3.4.2 OverlayElements Estándar

3.4.1 ATRIBUTOS OVERLAYELEMENT

Estos atributos son válidos dentro de las llaves de un bloque ‘container’ o ‘element’ en un script de superposición. Cada uno debe estar en su propia línea. El orden no es importante.

metrics_mode

Establece las unidades que se utilizarán para el tamaño y la posición de este elemento.

Formato: metrics_mode <pixels|relative>
Ejemplo: metrics_mode pixels

Esto puede ser usado para cambiar la forma de interpretar todos los atributos de medición en el resto de este elemento. En modo relativo (relative), se interpreta como un valor paramétrico de 0 a 1, como proporción de la anchura/altura de la pantalla. En el modo de píxeles, se mide según los píxeles.

Por defecto: metrics_mode relative

horz_align

Establece la alineación horizontal de este elemento, en términos de donde está el origen horizontal.

Formato: horz_align <left|center|right>
Ejemplo: horz_align center

Esto puede ser usado para cambiar el lugar donde se considera el origen para los propósitos de cualquier atributo de posición horizontal de este elemento. Por defecto, el origen se considera el borde izquierdo de la pantalla, pero se puede cambiar este al centro o la derecha para alinear los elementos. Hay que tener en cuenta que la fijación de la alineación al centro o a la derecha no significa que automáticamente fuerce a los elementos a aparecer en el centro o el borde derecho, sólo hay que tratar ese punto como el origen y ajustar las coordenadas adecuadamente. Esto es más flexible porque se puede elegir la posición del elemento en cualquier lugar en relación con el origen. Por ejemplo, si el elemento era de 10 píxeles de ancho, se puede usar la propiedad 'left' de -10 para alinear exactamente hasta el borde derecho, o -20 para dejar un hueco pero hacer que se pegue al borde derecho.

Hay que tener en cuenta que se puede utilizar esta propiedad en los modos relativos y de píxeles, pero es más útil en el modo de pixel.

Por defecto: horz_align left

vert_align

Establece la alineación vertical de este elemento, en términos de donde está el origen vertical.

Formato: vert_align <top|center|bottom>
Ejemplo: vert_align center

Esto puede ser usado para cambiar el lugar donde se considera el origen para los propósitos de cualquier atributo de posición vertical de este elemento. Por defecto, el origen se considera que es el borde superior de la pantalla, pero se puede cambiar este al centro o a abajo para alinear los elementos. Hay que tener en cuenta que la fijación de la alineación al centro o al final no implica automáticamente forzar a los elementos a que aparezcan en el centro o el borde inferior, sólo hay que tratar ese punto como el origen y ajustar sus coordenadas adecuadamente. Esto es más flexible porque se puede elegir la posición del elemento en cualquier lugar en relación con el origen. Por ejemplo, si el elemento es de 50 píxeles de alto, se usaría un 'top' de -50 para alinear exactamente hasta el borde inferior, o -70 para dejar un hueco pero hacer que se pegue al borde inferior.

Hay que tener en cuenta que se puede utilizar esta propiedad en los modos relativos y de píxeles, pero es más útil en el modo de pixel.

Por defecto: vert_align top

left

Establece la posición horizontal del elemento relativa a la de los padres.

Formato: left <valor>
Ejemplo: left 0.5

Las posiciones están en relación con el padre (la parte superior izquierda de la pantalla si el padre es un recubrimiento, la parte superior izquierda del padre en otro caso) y se expresa en términos de una proporción de tamaño de la pantalla. Por lo tanto, 0,5 es la mitad del camino a través de la pantalla.

Por defecto: left 0

top

Establece la posición vertical del elemento relativo al de los padres.

Formato: top <valor>
Ejemplo: Top 0.5

Las posiciones están en relación con el padre (la parte superior izquierda de la pantalla si el padre es un recubrimiento, la parte superior izquierda del padre en otro caso) y se expresan en términos de una proporción de tamaño de la pantalla. Por lo tanto, 0.5 es la mitad del camino en la pantalla.

Por defecto: top 0

width

Establece el ancho del elemento como una proporción del tamaño de la pantalla.

Formato: width <valor>
Ejemplo: width 0.25

Los tamaños son relativos al tamaño de la pantalla, por lo que 0.25 es un cuarto de la pantalla. Las dimensiones no son relativas al padre, lo que es común en los sistemas de ventanas donde la parte superior e izquierda son relativas, pero el tamaño es absoluto.

Por defecto: width 1

height

Define la altura del elemento como una proporción del tamaño de la pantalla.

Formato: height <valor>
Ejemplo: height 0.25

Los tamaños son relativos al tamaño de la pantalla, por lo que 0.25 es un cuarto de la pantalla. Las dimensiones no son relativas al padre, lo que es común en los sistemas de ventanas donde la parte superior e izquierda son relativas, pero el tamaño es absoluto.

Por defecto: height 1

material

Establece el nombre del material a utilizar en este elemento.

Formato: material <nombre>
Ejemplo: material Examples/TestMaterial

Esto establece el material base que este elemento va a utilizar. Cada tipo de elemento puede interpretar esto de manera diferente, por ejemplo el elemento del Ogre 'Panel' considera este como el fondo del panel, mientras que 'BorderPanel' lo interpreta esto como el material para el área del centro solamente.Los materiales deben ser definidos en scripts .material.

Hay que tener en cuenta que el uso de un material en un elemento de superposición deshabilita automáticamente la iluminación y la comprobación de profundidad sobre este material. Por lo tanto, no se debe utilizar el mismo material que se utiliza para los objetos reales en 3D de una imagen superpuesta.

Por defecto: ninguno

caption

Establece un título de texto para el elemento.

Formato: caption <string>
Ejemplo: caption Esto es un título

No todos los elementos soportan títulos, de modo que cada elemento es libre de ignorar esto si quiere. Sin embargo, un título de texto en general es tan común a muchos elementos que se incluye en la interfaz genérica para que sea más fácil de usar. Esta es una característica común en los sistemas de interfaz gráfica de usuario.

Por Defecto: en blanco

rotation

Establece la rotación del elemento.

Formato: rotation <angulo_en_grados><eje_x><eje_y><eje_z>
Ejemplo: rotation 30 0 0 1
Valor predeterminado: ninguno

3.4.2 OVERLAYELEMENTS ESTÁNDAR

Aunque las clases OverlayElement y OverlayContainer de Ogre están diseñadas para ser ampliadas por los desarrolladores de aplicaciones, hay algunos elementos que vienen de serie con Ogre. Estos incluyen:

- Panel
- BorderPanel
- TextArea

Esta sección describe cómo definir sus atributos personalizados en un script .overlay, pero también se pueden cambiar estas propiedades personalizadas en el código si se desea. Se puede hacer esto llamando a setParameter(nombre, valor). Es posible que se desee utilizar la clase StringConverter para convertir tipos a cadenas y viceversa.

Panel (container)

Este es el contenedor más estándar que se puede utilizar. Es un área rectangular que puede contener otros elementos (o contenedores) y puede o no puede tener un fondo, que puede ser rellenado como se quiera. El material de base es determinado por el atributo de material, pero sólo se muestra si la transparencia está apagada.

Atributos:

transparent <true | false>
Si se establece a 'true' el panel es transparente y no se renderiza a sí mismo, es utilizado como un nivel de agrupamiento por sus hijos.

tiling <capa><celda_x><celda_y>
Establece el número de veces que la textura o texturas de la materia son puestas en el panel en las direcciones X e Y. <capa> es la capa de textura, desde 0 hasta el número de capas de textura en el material menos uno. Con el establecimiento recubrimiento por capa, se pueden crear algunos buenos escenarios multitextura para los paneles, esto funciona especialmente bien cuando se anima alguna de las capas.

uv_coords <arribaIzquierda_u><arribaIzquierda_v><abajoDerecha_u><abajoDerecha_v>
Establece las coordenadas de textura a utilizar para este panel.

BorderPanel (container)

Esta es una versión ligeramente más avanzada que Panel, donde en lugar de sólo un panel plano individual, el panel tiene una frontera independiente que cambia de tamaño con el panel. Esto se hace tomando un enfoque muy similar a la utilización de tablas de HTML para el contenido: el panel se renderiza como 9 áreas cuadrados, el área central se renderiza con el material principal (como en el Panel) y las otras 8 áreas (las 4 esquinas y los 4 lados) se renderizan con un material de frontera por separado. La ventaja de que las esquinas se rendericen por separado de los bordes es que el borde de las texturas puede ser diseñado de manera que se puede estirar sin distorsionarlas, de forma que cualquier textura puede servir para paneles de cualquier tamaño.

Atributos:

border_size <izquierda><derecha><arriba><abajo>
El tamaño de la frontera en cada borde, como proporción del tamaño de la pantalla. Esto permite tener las fronteras de diferente tamaño en cada extremo si se quiere, o se puede utilizar el mismo valor 4 veces para crear una frontera de tamaño constante.

border_material <nombre>
El nombre del material a utilizar para la frontera. Se trata normalmente de un material diferente al utilizado para la zona centro, porque la zona centro suele ser de baldosas que significa que no se pueden poner zonas fronterizas. Se deben poner todas las imágenes que se necesite para todas las esquinas y los lados en una textura única.

border_topleft_uv <u1><v1><u2><v2>

[también border_topright_uv, border_bottomleft_uv, border_bottomright_uv], las coordenadas de textura que se utilizarán para las zonas de la esquina de la frontera. 4 coordenadas son necesarias, 2 para la esquina superior izquierda del cuadrado, 2 para la parte inferior derecha del cuadrado.

border_left_uv <u1><v1><u2><v2>

[también border_right_uv, border_top_uv, border_bottom_uv], las coordenadas de textura que se utilizarán para las zonas de borde de la frontera. 4 coordenadas son necesarias, 2 para la esquina superior izquierda, 2 para la parte inferior derecha. Hay que tener en cuenta que se debe diseñar la textura para que los bordes izquierdo y derecho puedan ser alargados/aplastados en el eje vertical y los bordes superior e inferior se pueden estirar/aplastar en el eje horizontal, sin efectos perjudiciales.

TextArea (element)

Este es un elemento genérico que se puede utilizar para renderizar texto. Utiliza las fuentes que se pueden definir en el código utilizando las clases FontManager y Font, o que hayan sido predefinidos en los archivos .fontdef. Ver la sección de definiciones de fuente para más información.

Atributos:

font_name <nombre>

El nombre de la fuente a utilizar. Esta fuente debe ser definida en un archivo .fontdef para asegurarse de que está disponible en tiempo de script.

char_height <altura>

La altura de las letras como proporción de la altura de la pantalla. El grosor de los caracteres puede variar debido a que Ogre es compatible con fuentes proporcionales, pero se basa en esta altura constante.

colour <rojo><verde><azul>

Un color sólido para renderizar el texto. A menudo las fuentes se definen en blanco y negro, por lo que esto permite colorearlo bien y el uso de la misma textura para diferentes áreas de texto de múltiples colores. El color de todos los elementos debe ser expresado en valores entre 0 y 1. Si se utilizan fuentes predibujadas con color, entonces no se necesita este atributo.

colour_bottom <rojo><verde><azul> / colour_top <rojo><verde><azul>

Como una alternativa a un color sólido, se puede colorear el texto de manera diferente en la parte superior e inferior para crear un efecto de degradado de color que puede ser muy eficaz.

Alignment <left | center | right>

Establece la alineación horizontal del texto. Esto es diferente del parámetro horz_align.

space_width <ancho>

Establece el ancho de un espacio en relación a la pantalla.

3.5 SCRIPTS DE DEFINICIÓN DE FUENTES

Ogre utiliza fuentes basadas en texturas para renderizar el TextAreaOverlayElement. También se puede utilizar el objeto Font para otros propóstios, si se desea. La forma final de una fuente es un objeto Material generado por la fuente, y un conjunto de información de coordenadas de textura 'glyph' (carácter).

Hay 2 formas de obtener una fuente en Ogre:

1. Diseñando una textura de fuente utilizando un paquete de arte o herramienta generadora de tipografías.
2. Preguntar a Ogre para generar una textura de fuente basada en una fuente de tipo verdadero.

El primero da flexibilidad y mejor rendimiento (en términos de tiempos de arranque), pero el último es conveniente si se desea utilizar una fuente rápidamente sin tener que generar la textura de la misma. Se sugiere el último caso en el uso de prototipos y el cambio a la primera para la solución definitiva.

Todas las definiciones de fuente se mantienen en los archivos .fontdef, que son procesados por el sistema en tiempo de inicio. Cada archivo .fontdef puede contener varias definiciones de fuente. El fomato básico de una entrada en el archivo .fontdef es:

```
<font_name>
{
    type <image | truetype>
    source <image file | truetype font file>
    ...
    ... custom attributes depending on type
}
```

Código 59: Script de definición de Fuentes.

USO DE UNA TEXTURA FUENTE EXISTENTE

Ogre soporta plenamente texturas de fuente de color, o, alternativamente, puede guardarlos en monocromo/escala de grises y utilizar la función de la coloración TextArea. Las texturas de fuente siempre deben tener un canal alfa, preferiblemente un canal alfa de 8 bits soportado por ficheros de TGA y PNG, porque puede dar lugar a bordes mucho más agradables. Para utilizar una textura existente, aquí están los ajustes que se necesitan:

type image
Esto dice a Ogre que se desea una fuente pre-dibujada.

source <nombre_fichero>
Este es el nombre del archivo de imagen que se desea cargar. Esto se carga desde la ubicación de los recursos estándar TextureManager y pueden ser de cualquier tipo que Ogre soporta, aunque JPEG no se recomienda debido a la falta de alfa y la compresión con pérdida. Se recomienda el formato PNG, que tiene tanto una buena compresión sin pérdidas como un canal alfa de 8 bits.

glyph <caracter><u1><v1><u2><v2>
Proporciona las coordenadas de textura para el carácter especificado. Se debe repetir este procedimiento con cada carácter que se tiene en la textura. Los 2 primeros números son el X e Y de la esquina superior izquierda, la siguientes son las X e Y de la esquina inferior derecha. Hay

que tener en cuenta que realmente se debería usar una altura común para todos los caracteres, pero el ancho puede variar debido a las fuentes proporcionales.

'carácter' es o bien un carácter ASCII para carácter no extendido de 7 bits ASCII o de glifos extendido, un valor decimal Unicode, que se identifica por el carácter 'u' precediendo al número - por ejemplo, 'u0546' denota el valor de Unicode 546.

Nota para usuarios de Windows: Se recomienda usar BitmapFontBuilder (http://www.lmnopc.com/bitmapfontbuilder/), una herramienta gratuita que generará una textura y grosor de los caracteres de exportación, se puede encontrar una herramienta para la conversión de la salida binaria de esta en líneas 'glifo' en la carpeta Herramientas.

GENERACIÓN DE UNA TEXTURA DE FUENTE

También se pueden generar texturas de fuente sobre la marcha utilizando fuentes TrueType. No se recomienda el uso intensivo de este trabajo en la producción debido a que la renderización de la textura puede tomar varios segundos por fuente a los tiempos de carga. Sin embargo, es una forma muy agradable de conseguir rápidamente la salida de texto en una fuente de su elección.

Éstos son los atributos que se necesita para hacerlo:

type truetipe
Dice a Ogre que generará la textura de una fuente

source <archivo ttf>
El nombre del archivo ttf a cargar. Este será buscado en los lugares de recursos comunes y en cualquier ubicación de recursos que se agreguen al FontManager.

size <tamaño_en_puntos>
El tamaño en el que se genera la fuente, en puntos estándar. Nota: Esto sólo afecta a lo grande que son los caracteres que están en la textura de la fuente, no lo grandes que son en la pantalla. Se debe adaptar esto dependiendo de lo grande en que se espera renderizar las fuentes, la generación de una gran textura se traducirá en caracteres borrosos cuando se escala muy pequeña (debido al mipmapping), y a la inversa, generando una fuente pequeña se traducirá en caracteres como bloques si los se renderizan los caracteres en tamaño grande.

resolution <ppp>
La resolución en puntos por pulgada, que se utiliza en conjunción con el tamaño de punto para determinar el tamaño final. 72/96 ppp es normal.

antialias_colour <true|false>
Esta es una propiedad opcional, que por defecto es 'false'. El generador hará antialias de la fuente por defecto con el componente alfa de la textura, que se verá bien si se utiliza mezcla alfa para hacer el texto (este es el supuesto valor predeterminado por TextAreaOverlayElement por ejemplo). Si, en cambio,se desea utilizar una mezcla basada en el color como añadir (add) o modular (modulate) en el código, se debe establecer este valor en 'true' de forma que los valores de color son anitaliased también. Si se establece en "true" y se usa mezcla alfa, se podrá encontrar que los bordes de la fuente son suavizadas con demasiada rapidez con resultado de aspecto 'delgado' de las fuentes, porque no sólo se hace mezcla alfa de los bordes, el color se desvanece demasiado. Se recomienda dejar esta opción en el valor predeterminado en caso de duda.

nn code_points-nn [nn-nn] ..

Esta directiva permite especificar qué puntos de código Unicode deben generarse como glifos (glyphs) en la textura de la fuente. Si no se especifica este, por defecto los puntos de código 33-166 serán generados por defecto, por el cual se abarcan los glifos básicos Latin 1. Si se utiliza este parámetro, se debe especificar una lista separada por espacios de las gamas de puntos incluidos en el código de la forma 'inicio-fin'. Los números deben estar en decimal.

También se pueden crear nuevas fuentes en tiempo de ejecución mediante la clase FontManager si se desea.

4. HERRAMIENTAS DE MALLA

Hay un número de herramientas de malla disponible con Ogre para ayudar a manipular mallas.

4.1 Exportadores
Para obtener datos de los modeladores e incluirlos en Ogre.

4.2 XmlConverter
Para convertir mallas y esqueletos a/desde XML.

4.3 MeshUpgrader
Para actualizar mallas binarias desde una versión de Ogre a otra.

4.1 EXPORTADORES

Los exportadores son plugins para herramientas de modelado 3D que escriben mallas y animación esquelética para los formatos de archivo que Ogre puede utilizar para el renderizado en tiempo real. Los archivos de los exportadores escritos terminan en .mesh y .skeleton respectivamente.

Cada exportador tiene que ser escrito específicamente para el modelador en cuestión, aunque todos ellos utilizan un conjunto común de servicios prestados por las clases MeshSerializer y SkeletonSerializer. Asimismo, normalmente requieren la posesión de la herramienta de modelado.

Todos los exportadores aquí se pueden construir a partir del código fuente, o se pueden descargar versiones precompiladas del sitio web de Ogre.

UNA NOTA ACERCA DE MODELADO/ANIMACIÓN PARA OGRE

Hay algunas reglas al crear un modelo animado para Ogre:

- Se deben tener no más de 4 huesos pesados por vértice. Si tiene más, Ogre eliminará las tareas más pesadas y renormalizará los otros pesos. Este límite es impuesto por las limitaciones de la mezcla de hardware.
- Todos los vértices deben ser asignados a al menos un hueso - asignar los vértices estáticos en el hueso de raíz.
- Por lo menos cada hueso debe tener un fotograma clave en el comienzo y el final de la animación.

Si se crean mallas inanimadas, entonces no hay que preocuparse por lo anterior.

La documentación completa para cada exportador se proporciona junto con el propio exportador, y hay una lista de las herramientas de modelización actualmente soportadas en el Wiki de Ogre en http://www.ogre3d.org/wiki/index.php/Exporters.

4.2 XMLCONVERTER

La herramienta OgreXmlConverter puede convertir archivos binarios .mesh y .skeleton a XML y viceversa - esta es una herramienta muy útil para depurar el contenido de las mallas, o para intercambiar datos de malla fácilmente - muchos de los exportadores de malla de los modeladores exportan a XML, porque es más fácil de hacer, y OgreXmlConverter puede producir un binario de ella. Aparte de la simplicidad, la otra ventaja es que OgreXmlConverter puede generar información adicional para la malla, como delimitador regiones y el nivel de reducción de detalle.

Sintáxis:

```
Usage: OgreXMLConverter sourcefile [destfile]
sourcefile = name of file to convert
destfile   = optional name of file to write to. If you don't
specify this OGRE works it out through the extension
             and the XML contents if the source is XML. For example
             test.mesh becomes test.xml, test.xml becomes test.mesh
             if the XML document root is <mesh> etc.
```

Código 60: Sintáxis de OgreXMLConverter.

Cuando se convierte de XML a .mesh, se pide si (re)generar la información del nivel-de-detalle (LOD) para la malla – se puede elegir omitir esta parte si se quiere, pero hacerlo permitirá hacer una malla reducida en detalle de forma automática cuando se carga en el motor. El motor utiliza un complejo algoritmo para determinar las mejores partes de la malla para reducir el detalle en función de muchos factores, como la curvatura de la superficie, los bordes de la malla y las costuras en los bordes de las texturas y los grupos de suavizado - aprovechando esto se recomienda para hacer mallas más escalables en escenas reales.

4.3 MESHUPGRADER

Esta herramienta permite actualizar mallas cuando el formato binario cambia – a veces, se altera para añadir nuevas funciones, como tal, se necesita mantener sus elementos hasta la fecha. Esta herramienta tiene una sintaxis muy sencilla:

```
OgreMeshUpgrade <oldmesh><newmesh>
```

Código 61: Sintáxis de OgreMeshUpgrader.

Los lanzamientos de Ogre notifican cuando esto es necesario con un nuevo lanzamiento.

5. BÚFERS HARDWARE

Los búfer de vértice, de índice y de píxeles heredan la mayoría de sus características de la clase HardwareBuffer. La premisa general de un búfer de hardware es que es un área de memoria con el que se puede hacer lo que se quiera, no hay un formato (vértice o no) asociado con el búfer - depende totalmente de la interpretación de los métodos que utilizan - de esa manera, un HardwareBuffer es como un espacio de memoria que se puede asignar utilizando 'malloc' - la diferencia es que esta memoria es probable que se encuentre en la GPU o en la memoria AGP.

5.1 EL ADMINISTRADOR DE BÚFER DE HARDWARE

La clase HardwareBufferManager es el centro de fábrica de todos los objetos en el sistema de la nueva geometría. Se pueden crear y destruir la mayoría de los objetos que se utilizan para definir la geometría a través de esta clase. Es un Singleton, por lo que se accede a ella usando HardwareBufferManager::getSingleton() - sin embargo, hay que ser conscientes de que sólo se garantiza que exista después de la inicialización del RenderSystem (después de llamar a Root::inicialise); esto se debe a que los objetos creados son invariables según la API específica, aunque se ocupará de ellos a través de una interfaz común.

Por ejemplo:

```
VertexDeclaration* decl =
HardwareBufferManager::getSingleton().createVertexDeclaration();
```

Código 62: HardwareBuffer para obtener un VertexDeclaration

```
HardwareVertexBufferSharedPtr vbuf =
       HardwareBufferManager::getSingleton().createVertexBuffer(
               3*sizeof(Real), // size of one whole vertex
               numVertices, // number of vertices
               HardwareBuffer::HBU_STATIC_WRITE_ONLY, // usage
               false); // no shadow buffer
```

Código 63: Creación de VertexBuffer con HardwareBuffer

No hay que preocuparse por los detalles del ejemplo, ya que se verá en secciones siguientes. La cosa importante a recordar es que siempre se crean los objetos a través del HardwareBufferManager, no hay que usar 'new' (no funcionará en la mayoría de los casos).

5.2 USO DEL BÚFER

Debido a que la memoria en un búfer de hardware es probable que esté en significativa disputa durante la renderización de una escena, el tipo de acceso que se necesita para el búfer sobre el tiempo que se utiliza es muy importante; si es necesario actualizar el contenido del búfer regularmente, si tiene que ser capaz de leer la información de vuelta de ella, todos estos son factores importantes para cómo la tarjeta gráfica administra el búfer. El método y los parámetros exactos para crear un búfer dependen de si se está creando un búfer de índice o de vértice (véase la sección 5.6 Búfers hardware de vértice y 5.7 Búfers hardware de índice), sin embargo un parámetro de la creación es común a ambos - el 'usage'.

El tipo más óptimo de un búfer de hardware es el que no se actualiza con frecuencia, y nunca se lee. El parámetro de uso de createVertexBuffer o createIndexBuffer puede ser uno de los siguientes:

HBU_STATIC

Significa que no es necesario actualizar el búfer muy a menudo, pero de vez en cuando se puede leer de él.

HBU_STATIC_WRITE_ONLY
Significa que no es necesario actualizar el búfer muy a menudo, y no es necesario leer de él. Sin embargo, se puede leer desde su búfer de sombra, si se pone uno (ver sección 5.3 Búferes de sombra). Esta es la configuración óptima de utilización de búferes.

HBU_DYNAMIC
Significa que se puede esperar actualizar el búfer a menudo, y que puede que se desee leer de él. Esta es la configuración menos óptima.

HBU_DYNAMIC_WRITE_ONLY
Significa que se puede esperar actualizar el búfer a menudo, pero que no se quiere leer de él. Sin embargo, se puede leer desde su búfer de sombra, si se pone uno (ver sección 5.3 Búferes de sombra). Si se utiliza esta opción, y se sustituye la totalidad del contenido del búfer en cada frame, se debe usar la HBU_DYNAMIC_WRITE_ONLY_DISCARDABLE en su lugar, ya que tiene mejores características de rendimiento en algunas plataformas.

HBU_DYNAMIC_WRITE_ONLY_DISCARDABLE
Significa que se espera sustituir la totalidad del contenido del búfer de forma muy regular, lo más probable es que sea en cada fotograma. Al seleccionar esta opción, se libera al sistema de tener que preocuparse por perder el contenido actual del búfer en cualquier momento, porque si lo pierde, se le reemplaza con el siguiente fotograma de todos modos. En algunas plataformas puede suponer una diferencia de rendimiento significativa, por lo que debe tratar de usar esto siempre que se tiene que actualizar un búfer periódicamente. Hay que tener en cuenta que si se crea un búfer de esta manera, se debe usar la marca HBL_DISCARD al cerrar el contenido del mismo para la escritura.

Elegir el uso de los búferes con cuidado es importante para obtener un rendimiento óptimo de la geometría. Si se tiene una situación en la que se tiene que actualizar un búfer de vértices con frecuencia, hay que considerar si realmente se necesitan actualizar **todas** las partes de él, o sólo algunas. Si es esto último, hay que considerar el uso de más de un búfer, con sólo los datos que se necesitan para modificar en el búfer de HBU_DYNAMIC.

Siempre hay que intentar utilizar las formas _WRITE_ONLY. Esto sólo significa que no se puede leer directamente desde el búfer de hardware, que es una buena práctica ya que la lectura de búferes del hardware es muy lenta. Si realmente se necesita para leer datos de nuevo, se usa un búfer de sombra, que se describe en la siguiente sección.

5.3 BÚFERES DE SOMBRA

Como se discutió en la sección anterior, la lectura de datos de un búfer de hardware funciona muy mal. Sin embargo, si se tiene la necesidad de leer el contenido del búfer de vértices, se debe establecer el parámetro 'shadowBuffer' de createVertexBuffer o createIndexBuffer a 'true'. Esto hace que el búfer de hardware sea respaldado con una copia de la memoria del sistema, que se puede leer sin más pega que la lectura de la memoria común. El problema es que al escribir los datos en este búfer, primero actualiza la copia de la memoria del sistema, y luego actualiza el búfer de hardware, como proceso de copia por separado - por lo que esta técnica tiene una sobrecarga adicional al escribir datos. No se debe usar a menos que realmente se necesite.

5.4 BLOQUEANDO BÚFERES

Para poder leer o actualizar un búfer de hardware, se tiene que 'bloquearlo'. Esto realiza 2 funciones - le dice a la tarjeta que se desea tener acceso al búfer (que puede tener un efecto sobre la cola de renderización), y devuelve un puntero que se puede manipular. Hay que tener en cuenta que si se ha pedido que lea el búfer (aunque realmente no se debiera, a menos que se haya establecido el búfer con un búfer de sombra), el contenido del búfer de hardware se ha copiado en la memoria del sistema en algún lugar para que se pueda tener acceso a ella. Por la misma razón, cuando se haya terminado con el bloqueo se debe desbloquear, aunque se haya bloqueado el búfer para escribir, se activa el proceso de cargar la información modificada al hardware de gráficos

Parámetros de bloqueo

Cuando se bloquea un búfer, se llama a uno de los siguientes métodos:

```
// Lock the entire buffer
pBuffer->lock(lockType);
// Lock only part of the buffer
pBuffer->lock(start, length, lockType);
```

Código 64: Bloqueos de búfer hardware.

La primera llamada bloquea el búfer completamente, la segunda bloquea sólo la sección desde 'start' (como offset), para 'lenght' bytes. Esto puede ser más rápido que bloquear el búfer al completo, ya que se transfiere menos, pero no si posteriormente se va a actualizar el resto del búfer, porque hacerlo en trozos pequeños, significa que no se puede utilizar HBL_DISCARD (ver abajo).

El parámetro 'lockType' puede tener un gran efecto sobre el rendimiento de la aplicación, especialmente si no se está utilizando un búfer de sombra.

HBL_NORMAL

Este tipo de bloqueo permite la lectura y la escritura de la memoria intermedia - es también el más óptimo, porque básicamente está diciendo a la tarjeta que podría no estar haciendo nada en absoluto. Si no se está utilizando un búfer de sombra, se requiere el búfer para ser transferido de la tarjeta y viceversa. Si se está utilizando un búfer de sombra el efecto es mínimo.

HBL_READ_ONLY

Esto significa que sólo se quiere leer el contenido del búfer. Es la mejor opción cuando se crea el búfer con un búfer de sombra, porque en ese caso, los datos no tienen que ser descargados de la tarjeta.

HBL_DISCARD

Esto significa que la tarjeta descarta todo el contenido actual del búfer. Implícitamente, esto significa que no se van a leer los datos - también significa que la tarjeta puede evitar los atascos si el búfer se está renderizando actualmente, ya que en realidad se le dará otro completamente distinto. Hay que usar esta medida cuando se cierra un búfer que no se creó con un búfer de sombra. Si se está usando un búfer de sombra importa menos, aunque con un búfer de sombra es preferible para bloquear el búfer completo a la vez, ya que permite el búfer de sombra para el uso HBL_DISCARD cuando se carga el contenido actualizado en el búfer real.

HBL_NO_OVERWRITE

Esto es útil si se está bloqueando sólo una parte del búfery por lo tanto no se puede utilizar HBL_DISCARD. Le dice a la tarjeta que se compromete a no modificar cualquier sección del búfer que ya ha sido utilizada en una operación de renderizado de este fotograma. De nuevo, esto sólo es útil en búferes sin búfer sombra.

Una vez que se haya bloqueado un búfer, se puede usar el puntero devuelto, si se desea (no se molesta en tratar de leer los datos si se ha utilizado HBL_DISCARD, o de escribir los datos si ha utilizado HBL_READ_ONLY). La modificación del contenido depende del tipo de bloqueo, ver la sección 5.6 Búfers de hardware y vértice y 5.7 Búfers de hardware de Índice.

5.5 CONSEJOS ÚTILES DE BÚFER

La interacción de modo de uso en la creación, y opciones de bloqueo en la lectura/actualización es importante para el rendimiento. He aquí algunos consejos:

1. Objetivo para el 'búfer perfecto', creando con HBU_STATIC_WRITE_ONLY, sin búfer de sombra, y el bloqueo de todos por una sola vez con HBL_DISCARD para rellenarlo. Nunca hay que tocarlo de nuevo.

2. Si se necesita actualizar un búfer regularmente, hay que hacer concesiones. Usando HBU_DYNAMIC_WRITE_ONLY a la hora de crear (sin búfer de sombra), y usando HBL_DISCARD para bloquear el búfer completo, o si después no se puede, utilizar HBL_NO_OVERWRITE para bloquear partes del búfer.

3. Si realmente se necesitan leer datos del búfer, hay que crear un búfer de sombra. Asegurándose de que se utiliza HBL_READ_ONLY para bloqueo de la lectura, pues evitará la carga que normalmente se asocia con el desbloqueo del búfer. También se puede combinar esto con cualquiera de los 2 puntos anteriores, obviamente, hay que tratarla como estática si se puede –hay que recordar que parte de la _WRITE_ONLY se refiere al búfer de hardware para que pueda usarse de manera segura con un búfer de sombra que leer.

4. Dividir el búfer de vértice hasta que se encuentre que los patrones de uso de los diferentes elementos de los vértices son diferentes. No tiene sentido tener un búfer actualizable enorme con todos los datos de vértice en él, si todo lo que se necesita es actualizar las coordenadas de textura. Dividir esa parte en su propio búfer y hacer con HBU_STATIC_WRITE_ONLY el resto.

5.6 BÚFER DE HARDWARE DE VÉRTICE

Esta sección cubre búferes del hardware que contengan datos de vértice. Para una discusión general de búfer de hardware, junto con las normas para la creación y de bloqueo, consulte la sección 5. Búfers Hardware.

5.6.1 LA CLASE VERTEXDATA

La clase VertexData agrupa toda la información de vértices relacionados utilizada para renderizar la geometría. El nuevo RenderOperation requiere un puntero a un objeto VertexData, y también se utiliza en Mesh y SubMesh para almacenar las posiciones de vértice, normales, coordenadas de textura, etc. VertexData puede utilizarse solo (con el fin de renderizar geometría no indexada, donde el flujo de vértices define los triángulos), o en combinación con IndexData donde los triángulos son definidos por los índices que hacen referencia a las entradas en VertexData.

Vale la pena notar que no necesariamente se tienen que utilizar VertexData para almacenar aplicaciones de la geometría; todo lo que se requiere es que se puede construir una estructura VertexData cuando se trata de la renderización. Esto es bastante fácil, ya que todos los miembros de VertexData son punteros, por lo que se podría mantener sus búferes de vértices y las declaraciones en estructuras alternativas si se quiere, siempre y cuando se los puede convertir para el renderizado.

La clase VertexData tiene un número de miembros importantes:

vertexStart
: La posición en los búferes fijada para empezar a leer los datos de vértice. Esto permite utilizar un único búfer para muchos renderables diferentes.

vertexCount
: El número de vértices para procesar en este grupo de renderización determinado.

vertexDeclaration
: Un puntero a un objeto VertexDeclaration que define el formato de la entrada de vértice; hay que tener en cuenta que este es creado por VertexData. Ver la sección 5.6.2 Declaraciones de vértices.

vertexBufferBinding
: Un puntero a un objeto VertexBufferBinding que define qué búferes de vértice están ligados a qué fuentes - de nuevo, esta es creada por VertexData. Ver la sección 5.6.3 Vinculaciones de búfer de vértice.

5.6.2 DECLARACIONES DE VÉRTICES

Las declaraciones de vértices definen las entradas de vértices utilizados para renderizar la geometría que se desea que aparezca en la pantalla. Básicamente esto significa que para cada vértice, se desea alimentar un cierto conjunto de datos en el pipeling de gráficos, que (se espera) afectará a lo que aparece cuando se dibujan los triángulos. Las declaraciones de vértices permiten extraer los objetos de datos (que se llaman elementos de vértice, representados por la clase VertexElement) de cualquier número de búferes, ambos compartidos y dedicados a ese elemento en particular. Es trabajo del desarrollador garantizar que el contenido de los búferes tiene sentido cuando se interpretan en la forma en que el VertexDeclaration indica.

Para agregar un elemento a un VertexDeclaration, se llama al método addElement. Los parámetros de este método son:

source
: Esto le dice a la declaración de qué búfer se coge el elemento. Hay que tener en cuenta que esto es sólo un índice, que puede ir desde 0 hasta uno menos que el número de búferes que están enlazados como fuentes de datos de vértice. Ver la sección 5.6.3 Enlaces de búferes de vértices para obtener información sobre cómo un búfer real está vinculado a un índice de la fuente. El almacenamiento de la fuente del elemento vértice de esta manera (en lugar de usar un puntero de búfer) permite volver a enlazar la fuente de un vértice muy fácilmente, sin cambiar el formato de declaración del mismo vértice.

offset
: Le dice a la declaración de en qué medida en bytes se desplaza el elemento desde el inicio de cada vértice en este búfer. Esta será 0 si este es el único elemento que se obtiene a partir de este búfer, pero si hay otros elementos, entonces puede ser mayor. Una buena manera de

pensar para este parámetro es el tamaño de todos los elementos que preceden a este elemento en el búfer.

type

Define el tipo de datos de la entrada de vértice, incluyendo su tamaño. Este es un elemento importante porque como las GPU se hacen más avanzadas, ya no se puede suponer que la posición de entrada siempre requerirá 3 números en coma flotante, debido a canales de vértices programables permiten un control total sobre las entradas y salidas. Esta parte de la definición de elemento cubre el tipo básico y tamaño, por ejemplo, VET_FLOAT3 es de 3 números de punto flotante - el significado de los datos se tratan en el parámetro siguiente.

semantic

Define el significado del elemento - la GPU usa esto para determinar qué utilizar como entrada, y qué canales de vértices programables usar para identificar la semántica para mapear la entrada. Esto puede identificar el elemento como datos de posición, los datos normales, los datos de coordenadas de textura, etc. Ver la referencia de API para los detalles completos de todas las opciones.

index

Este parámetro sólo es necesario si se está suministrando más de un elemento de la misma semántica en una declaración de vértice. Por ejemplo, si se oferta más de un conjunto de coordenadas de textura, primero hay que fijar el índice a 0, y el segundo fijarlo a 1.

Se puede repetir la llamada a addElement para tantos elementos como se tengan en las estructuras de entrada de vértice. También hay métodos útiles en VertexDeclaration para la localización de elementos dentro de una declaración - ver la referencia de API para más detalles.

Consideraciones importantes

Aunque, en teoría, se ha reinado totalmente sobre el formato de los vértices, en realidad, hay algunas restricciones. Antiguos hardware de DirectX imponen una determinada ordenación de los elementos que se extraen de cada búfer, específicamente cualquier hardware antes de DirectX 9, podrá imponer las siguientes restricciones:

- VertexElements debe añadirse en el siguiente orden, y el orden de los elementos dentro de un búfer compartido debe ser el siguiente:
- Posiciones.
- Pesos combinados.
- Normales.
- Colores difusos.
- Colores especulares.
- Las coordenadas de textura (empezando en 0, enumerados en orden, sin espacios).
- No debe haber espacios no utilizados en los búferes que no sean referenciados por cualquier VertexElement.
- No debe haber solapamiento entre las características de búfer y offset de dos VertexElements.

El hardware compatible con OpenGL y DirectX 9 no está obligado a seguir estas limitaciones estrictas, por lo que se puede encontrar, por ejemplo, que si se rompieron las reglas de la aplicación se ejecutará en OpenGL y DirectX en las tarjetas recientes, pero no está garantizado para funcionar con hardware antiguo DirectX a menos que se adhieran a las normas anteriores. Por este motivo que se recomienda atenerse a ellas.

5.6.3 ENLACES DE BÚFER DE VÉRTICES

Estos enlaces sirven para asociar un búfer de vértice con un índice de fuente utilizado en 5.6.2 Declaraciones de vértices.

Creación del búfer de vértice

En primer lugar, veamos cómo crear un búfer de vértices:

```
HardwareVertexBufferSharedPtr vbuf =
        HardwareBufferManager::getSingleton().createVertexBuffer(
                3*sizeof(Real), // size of one whole vertex
                numVertices, // number of vertices
                HardwareBuffer::HBU_STATIC_WRITE_ONLY, // usage
                false); // no shadow buffer
```

Código 65: Creando un búfer de vértices.

Hay que tener en cuenta que se usa 5.1 El administrador de búfer de hardware para crear el búfer de vértices, y se retorna desde el método una clase llamada HardwareVertexBufferSharedPtr, en lugar de un puntero. Esto es debido a que los búferes de vértice son un contador de referencias - que son capaces de utilizar un búfer de un sólo vértice como fuente de múltiples piezas de geometría por lo tanto un puntero normal y que no sería suficiente, porque no se sabe cuando los diferentes usuarios habrán terminado con él. La clase HardwareVertexBufferSharedPtr gestiona su propia destrucción por mantener una cuenta de referencia de la cantidad de veces que se usa - cuando se destruye el último HardwareVertexBufferSharedPtr, el búfer automáticamente se destruye.

Los parámetros para la creación de un búfer de vértices son los siguientes:

vertexSize

El tamaño en bytes de un vértice completo en este búfer. Un vértice puede incluir varios elementos, y de hecho el contenido de los datos de vértice puede ser reinterpretado por las declaraciones de vértice diferente si se desea. Por lo tanto, se debe informar al administrador del búfer lo grande que es todo un vértice, pero no el formato interno de los vértices, ya que se ha reducido a la declaración de interpretar. En el ejemplo anterior, el tamaño se ajusta al tamaño de 3 valores de punto flotante - esto sería suficiente para mantener una posición 3D estándar o normal, o coordenada de una textura 3D, por vértice.

numVertices

El número de vértices en este búfer. Hay que recordar que no todos los vértices tienen que ser utilizados a la vez - esto puede ser beneficioso para crear grandes búferes que se comparten entre muchos trozos de la geometría, porque cambiar los enlaces de búfer de vértice es un intercambio de estado del renderizador, y es mejor si se reducen al mínimo.

usage

Esto indica al sistema cómo se va a utilizar el búfer. Ver la sección 5.2 Uso del búfer.

useShadowBuffer

Le dice al sistema si se desea que este búfer sea respaldado por un sistema de copia de memoria. Ver la sección 5.3 Búfers de sombra.

Enlazando el búfer de vértices

La segunda parte del proceso es enlazar el búfer que se ha creado a un índice fuente. Para hacerlo, se llama a:

```
vertexBufferBinding->setBinding(0, vbuf);
```

Código 66: Enlace de búfer con un índice fuente.

Esto resulta que el búfer de vértice creado sea enlazado al índice de fuente 0, de forma que cualquier elemento vértice cuyos datos sean cogidos del índice fuente 0, recuperarán los datos de este búfer.

Hay más métodos para recuperar búferes de datos enlazados – ver la referencia del API para los detalles completos.

5.6.4 ACTUALIZANDO BÚFERES DE VÉRTICE

La complejidad de la actualización de un búfer de vértices depende enteramente de la forma en que sus contenidos son establecidos. Se puede bloquear un búfer (Ver la sección 5.4 Bloqueando búfers), pero la forma de escribir datos en su vértice depende mucho de lo que contiene.

Se comienza con un ejemplo sencillo de vértice. Digamos que se tiene un búfer que sólo contiene las posiciones de vértice, por lo que sólo contiene series de 3 números de punto flotante por vértice. En este caso, todo lo que se necesita hacer para escribir datos en él es:

```
Real* pReal =
static_cast<Real*>(vbuf->lock(HardwareBuffer::HBL_DISCARD));
```

Código 67: Bloqueo de búfer para escribir datos en el.

... , sólo hay que escribir las posiciones en trozos de 3 reales. Si se tienen otros datos de punto flotante, es un poco más complejo, pero el principio es básicamente el mismo, sólo hay que escribir elementos alternativos. Pero si se tienen elementos de distintos tipos, o se necesita obtener la forma de escribir los datos del vértice desde los elementos por sí mismos, hay algunos métodos útiles en la clase VertexElement para ayudar.

En primer lugar, bloquear el búfer, asignar el resultado a un unsigned char * en lugar de un tipo específico. Entonces, para cada elemento que se abastece de este búfer (que se puede encontrar llamando a VertexDeclaration::findElementsBySource) se llama a VertexElement::baseVertexPointerToElement. Esto da un puntero que apunta a la base de un vértice en un búfer para el comienzo del elemento en cuestión, y permite utilizar un puntero del tipo adecuado para arrancar. He aquí un ejemplo completo:

```
// Get base pointer
unsigned char* pVert = static_cast<unsigned char*>(vbuf-
>lock(HardwareBuffer::HBL_READ_ONLY));
Real* pReal;
for (size_t v = 0; v < vertexCount; ++v)
{
        // Get elements
        VertexDeclaration::VertexElementList elems = decl-
>findElementsBySource(bufferIdx);
        VertexDeclaration::VertexElementList::iterator i, iend;
        for (i = elems.begin(); i != elems.end(); ++i)
        {
```

```
                VertexElement& elem = *i;
                if (elem.getSemantic() == VES_POSITION)
                {
                        elem.baseVertexPointerToElement(pVert, &pReal);
                        // write position using pReal

                }

                ...


        }
        pVert += vbuf->getVertexSize();
}
vbuf->unlock();
```

Código 68: Edición de búfer con datos complejos.

Ver el API para los detalles completos de todos los métodos de ayuda en VertexDeclaration y VertexElement para ayudar en la manipulación de punteros de datos de búferes de vértice.

5.7 BÚFERES DE ÍNDICE HARDWARE

Los búferes de índice se utilizan para renderizar la geometría, a través de la construcción de triángulos de vértices indirectamente, por referencia a su posición en el búfer, mejor dicho, según el orden de lectura secuencial del búfer. Los búferes de índice son más simples que los búferes de vértices, ya que son sólo una lista de índices, sin embargo pueden ser manejados por el hardware y compartidos entre múltiples piezas de geometría de la misma forma que pueden los búferes de vértice, por lo que las normas sobre la creación y bloqueo son las mismas. Ver la sección 5. Búfers de hardware para obtener información.

5.7.1 LA CLASE INDEXDATA

Esta clase resume la información requerida para usar un juego de índices para renderizar geometría. Sus miembros son los siguientes:

indexStart
: Es el primer índice utilizado por esta pieza de la geometría; puede ser útil para compartir un simple búfer de índice entre varias piezas geométricas.

indexCount
: El número de índices utilizados por este renderizable en particular.

indexBuffer
: El búfer de índice que es utilizado como origen de los índices.

Creando un búfer de índice

Los búferes de índice se crean igual que los búferes de vértice. Ver sección 5.1 El Manejador de búfers Hardware. Un ejemplo:

```
HardwareIndexBufferSharedPtr ibuf =
HardwareBufferManager::getSingleton().createIndexBuffer(
                HardwareIndexBuffer::IT_16BIT, // type of index
                numIndexes, // number of indexes
```

```
                HardwareBuffer::HBU_STATIC_WRITE_ONLY, // usage
                false); // no shadow buffer
```

Código 69: Creación de búfer de índice.

De nuevo, hay que notar que el tipo de retorno es una clase, o mejor dicho un puntero; esta referencia cuenta de forma que el búfer es automáticamente destruido cuando no hay más referencias al búfer. Los parámetros de creación del búfer de índice son:

indexType

Hay dos tipos de índices; 16-bit y 32-bit. Ambos realizan la misma tarea, excepto que el último puede direccionar búferes de vértices más grandes. Si el búfer incluye más de 65526 vértices, entonces se necesitarán utilizar índices de 32 bits. Hay que tener en cuenta que se deben de utilizar índices de 32 bits sólo cuando se necesiten, ya que provocan más saturación que vértices de 16 bits, y no son compatibles con algún harware viejo.

numIndexes

El número de índices en el búfer. Como en los búferes de vértices, se debe considerar que se puede utilizar un búfer de índice compartido que sea usado por múltiples piezas de geometría, de forma que se pueden obtener ventajas de rendimiento utilizando menos búferes.

usage

Dice el sistema que se pretende que use el búfer. Ver sección 5.2 Uso del búfer.

useShadowBuffer

Indica al sistema si se desea que este búfer sea respaldado por una copia de sistema de memoria. Ver la sección 5.3 Búferes de sombra.

5.7.2 ACTUALIZANDO BÚFERES DE ÍNDICE

La actualización de búferes de índice sólo puede ser realizado cuando se bloquea el búfer para escritura. Ver la sección 5.4 Bloqueando búferes para detalles. El bloqueo retorna un puntero void, que debe de cambiarse al tipo apropiado; con búferes de índice, es o un unsigned short (para índices de 16 bits) o un unsigned long (para índices de 32 bits). Por ejemplo:

```
unsigned short* pIdx =
static_cast<unsigned short*>(ibuf->lock(HardwareBuffer::HBL_DISCARD));
```

Código 70: Bloqueo para escritura de búfer de índices.

Entonces se puede escribir en el búfer utilizando las semánticas típicas para punteros, recordando desbloquear el búfer cuando se termine la actualización.

5.8 BÚFERES DE PÍXELES HARDWARE

Los búferes de píxeles hardware son un tipo especial de búfer que almacenan datos gráficos en tarjetas de memoria gráfica, generalmente para usar como texturas. Los búferes de pixel pueden representar una imagen de una, dos o tres dimensiones. Una textura puede consistir en múltiples de estos búferes.

Al contrario que los búferes de vértice e índice, los búferes de píxeles no son construidos directamente. Cuando se crea una textura, el búfer de pixel necesario para almacenar estos datos se construye automáticamente.

5.8.1 TEXTURAS

Una textura es una imagen que puede ser aplicada a la superficie de un modelo tridimensional. En Ogre, las texturas están representadas por la clase Texture.

Creando una textura

Las texturas se crean a través del TextureManager. En la mayoría de los casos se crean directamente desde archivos de imagen por el sistema de recursos de Ogre. Si se está leyendo esto, es muy probable que se quiera crear una textura manualmente, de forma que se pueda proveer con una imagen propia. Esto se hace a través de TextureManager::createManual:

```
ptex = TextureManager::getSingleton().createManual(
    "MyManualTexture", // Name of texture
    "General", // Name of resource group in which the texture should
be created
    TEX_TYPE_2D, // Texture type
    256, // Width
    256, // Height
    1, // Depth (Must be 1 for two dimensional textures)
    0, // Number of mipmaps
    PF_A8R8G8B8, // Pixel format
    TU_DYNAMIC_WRITE_ONLY // usage
);
```

Código 71: Creación de una textura de forma manual.

Este ejemplo crea una textura llamada *MyManualTexture* en el grupo de recursos *General*. Es una textura cuadrada de *dos dimensiones*, con una anchura de 256 y altura de 256. No tiene *mapa de bits*, formato interno *PF_A8R8G8B8* y un uso *TU_DYNAMIC_WRITE_ONLY*.

Los diferentes tipos de texturas se discuten en 5.8.3 Tipos de Texturas. Los formatos de pixel se resumen en 5.8.4 Formatos de Pixel.

Usos de texturas

En adicción a los usos de búfer hardware como se indica en la sección 5.2 Uso de búferes, hay algunas marcas específicas de uso para texturas:

TU_AUTOMIPMAP
: Los Mipmaps para esta textura serán generados automáticamente por el hardware gráfico. El algoritmo exacto utilizado no está definido, pero se puede asumir que será un filtro caja de 2x2.

TU_RENDERTARGET
: Esta textura será un objetivo de renderizado, es decir, se utilizará como objetivo para renderizar la textura. Estableciendo esta marca, se ignorarán todos los otros usos de textura excepto TU_AUTOMIPMAP.

TU_DEFAULT
: Es en realidad una combinación de marcas de uso, y es equivalente a TU_AUTOMIPMAP | TU_STATIC_WRITE_ONLY. El sistema de recursos usa estas marcas para texturas que son cargadas desde imágenes.

Obteniendo un PixelBuffer

Una textura puede consistir en múltiples PixelBuffers, uno por cada combo de nivel mipmap y número de cara. Para obtener un PixelBuffer de un objeto Textura, se utiliza el método Texture::getBuffer(face, mipmap):

Face debería ser cero para texturas sin mapa cúbico. Para mapas cúbicos, se define la cara a utilizar, que es una de las caras descritas en la sección 5.8.3 Tipos de texturas.

Mipmap es cero para el nivel mipmap cero, uno para el primero, y así sucesivamente. En texturas que tienen generación de mipmap automática (TU_AUTOMIPMAP), sólo se puede acceder al nivel 0, el resto se tratan en el API de renderizado.

Un simple ejemplo de usar getBuffer:

```
// Get the PixelBuffer for face 0, mipmap 0.
HardwarePixelBufferSharedPtr ptr = tex->getBuffer(0,0);
```

Código 72: Ejemplo para obtener un PixelBuffer.

5.8.2 ACTUALIZANDO BÚFERES DE PIXEL

Los búfers de píxel pueden ser actualizados de dos formas diferentes; una simple, la forma conveniente y una más difícil (pero en algunos casos más rápida). Ambos métodos utilizan objetos PixelBox (ver sección 5.8.5 Cajas de Píxeles) para representar datos de una imagen en memoria.

blitFormMemory

El método fácil para obtener una imagen en un PixelBuffer es usando HardwarePixelBuffer::blitFromMemory. Este obtiene un objeto PixelBox y hace todas las conversiones de formato de píxel necesarias y las escala. Por ejemplo, para crear una textura manual y cargar una imagen en ella, hay que hacer lo siguiente:

```
// Manually loads an image and puts the contents in a manually created
texture
Image img;
img.load("elephant.png", "General");
// Create RGB texture with 5 mipmaps
TexturePtr tex = TextureManager::getSingleton().createManual(
    "elephant",
    "General",
    TEX_TYPE_2D,
    img.getWidth(), img.getHeight(),
    5, PF_X8R8G8B8);
// Copy face 0 mipmap 0 of the image to face 0 mipmap 0 of the
texture.
tex->getBuffer(0,0)->blitFromMemory(img.getPixelBox(0,0));
```

Código 73: Creación de textura manual y carga de imagen.

Bloqueo directo de memoria

El método más avanzado para transferir los datos de una imagen a/desde un PixelBuffer es usar bloqueo. Bloqueando un PixelBuffer se puede acceder directamente al contenido sin importar el formato interno que tenga el búfer dentro de la CPU.

```
/// Lock the buffer so we can write to it
buffer->lock(HardwareBuffer::HBL_DISCARD);
const PixelBox &pb = buffer->getCurrentLock();

/// Update the contents of pb here
/// Image data starts at pb.data and has format pb.format
/// Here we assume data.format is PF_X8R8G8B8 so we can address pixels
as uint32.
uint32 *data = static_cast<uint32*>(pb.data);
size_t height = pb.getHeight();
size_t width = pb.getWidth();
size_t pitch = pb.rowPitch; // Skip between rows of image
for(size_t y=0; y<height; ++y)
{
    for(size_t x=0; x<width; ++x)
{
        // 0xRRGGBB -> fill the buffer with yellow pixels
        data[pitch*y + x] = 0x00FFFF00;
}
}
/// Unlock the buffer again (frees it for use by the GPU)
buffer->unlock();
```

Código 74: Método para tranferir imagen a PixelBuffer por bloqueo de memoria.

5.8.3 TIPOS DE TEXTURAS

Hay cuatro tipos de textura compatibles con el hardware actual, tres de ellos sólo se diferencian en la cantidad de dimensiones que tienen (una, dos o tres). La cuarta es especial. Los diferentes tipos de texturas son:

TEX_TYPE_1D
: Textura de una dimensión, utilizado en combinación con coordenadas de texturas 1D.

TEX_TYPE_2D
: Textura de dos dimensiones, utilizado en combinación con coordenadas de texturas 2D.

TEX_TYPE_3D
: Textura de tres dimensiones, utilizado en combinación con coordenadas de texturas 3D.

TEX_TYPE_CUBE_MAP
: Mapa de cubo (seis texturas de dos dimensiones, una por cada cara del cubo), utilizada en combinación con coordenadas de texturas 3D.

Texturas de mapas de cubos

El tipo de textura de mapa de cubo (TEX_TYPE_CUBE_MAP) es diferente de los demás; una textura de mapa de cubo representa una serie de seis imágenes de dos dimensiones direccionadas por coordenadas de textura 3D.

+X (cara 0)
Representa el plano positivo X (derecha).

-X (cara 1)
Representa el plano negativo X (izquierda).

+Y (cara 2)
Representa el plano positivo Y (arriba).

-Y (cara 3)
Representa el plano negativo Y (abajo).

+Z (cara 4)
Representa el plano positivo Z (frente).

-Z (cara 5)
Representa el plano negativo Z (posterior).

5.8.4 FORMATOS DE PÍXEL

El formato de píxel describe el formato de almacenamiento de los datos de píxel. Define la forma en que los píxeles son codificados en memoria. Las siguientes clases de formato de píxel (PF_*) se definen como:

Formatos de codificación (orden de bits) nativa (PF_A8R8G8B8 y otros formatos con cuenta de bits)
Estos son enteros con órden nativo en memoria (16, 24 y 32 bits). Esto significa una imagen con formato PF_A8R8G8B8 puede verse como un array de enteros de 32 bits, definido como 0xAARRGGBB en hexadecimal. El significado de las letras se describe abajo.

Formatos de byte (PF_BYTE_*)
Estos formatos tienen un byte por canal, y sus canales en memoria están organizados en el orden en que se especifican en el nombre de formato. Por ejemplo, PF_BYTE_RGBA consiste en un bloque de cuatro bytes, uno para rojo, uno para verde, uno para azul y uno para alfa.

Formatos de short (PF_SHORT_*)
Estos formatos tienen un unsigned short (entero de 16 bits) por canal, y sus canales en memoria están organizados en el orden en que están especificados en su nombre de formato. Por ejemplo, PF_SHORT_RGBA consiste en un bloque de cuatro enteros de 16 bits, uno para rojo, uno para verde, uno para azul y uno para alfa.

Formatos de float16 (PF_FLOAT16_*)
Estos formatos tienen un número de 16 bits de punto flotante por canal, y sus canales en memoria se organizan en el orden en que son especificados en el nombre de formato. Por ejemplo, PF_FLOAT16_RGBA consiste en un bloque de cuatro floats de 16 bits, uno para rojo, uno para verde, uno para azul y uno para alfa. Los floats de 16 bits, también llamados semi-float) son muy similares a los floats de 32 bits estándar de punto flotante de precisión simple de la IEEE, excepto que tienen sólo 5 bits de exponente y 10 de parte entera. Hay que tener en cuenta que no es un tipo de datos estándar de C++ por lo que el soporte de CPU para trabajar con ellos no es muy eficiente, pero las GPUs pueden calcular con estos mucho mejor que con flotantes de 32 bits.

Formatos de flotas32 (PF_FLOAT32_*)
Estos formatos tienen un número de punto flotante de 32 bits por canal, y los canales se organizan en memoria en el orden en que se especifica en su nombre de formato. Por ejemplo

PF_FLOAT32_RGBA consiste en bloques de cuatro flotantes de 32 bits, uno para rojo, uno para verde, uno para azul y uno para alfa. El tipo de datos en C++ de estos 32 bits es “float”.

Formatos comprimidos (PF_DXT[1-5])
Formatos de texturas comprimidos S3TC, una buena descripción se puede encontrar en Wikipedia (http://en.wikipedia.org/wiki/3STC).

Canales de color

El significado de los canales R, G, B, A, L y X se define como:

R
Componente de color rojo, normalmente con rango desde 0.0 (sin rojo) hasta 1.0 (rojo completo).

G
Componente de color verde, normalmente con rango desde 0.0 (sin verde) hasta 1.0 (verde completo).

B
Componente de color azul, normalmente con rango desde 0.0 (sin azul) hasta 1.0 (azul completo).

A
Componente alfa, normalmente con rango desde 0.0 (completamente transparente) hasta 1.0 (opaco).

L
Componente luminosidad, normalmente con rango desde 0.0 (negro) hasta 1.0 (blanco). El componente de luminosidad se duplica en los canales R, G y B para conseguir una imagen en escala de grises.

X
Este componente es completamente ignorado.

Si no se define un componente rojo, verde, azul o luminosidad en el formato, por defecto quedan a 0. Para el canal alfa es diferente; si no se define alfa, se pone por defecto a 1.

Lista completa de formatos de píxel

Los formatos de píxel compatibles con la versión actual de Ogre son:

Formatos de byte
PF_BYTE_RGB, PF_BYTE_BGR, PF_BYTE_BGRA, PF_BYTE_RGBA, PF_BYTE_L, PF_BYTE_LA, PF_BYTE_A.

Formatos de short
PF_SHORT_RGBA.

Formatos de Float16
PF_FLOAT16_R, PF_FLOAT16_RGB, PF_FLOAT16_RGBA.

Formatos de Float32
PF_FLOAT32_R, PF_FLOAT32_RGB, PF_FLOAT32_RGBA.

Formatos de orden nativo de 8 bits
PF_L8, PF_A8, PF_A4L4, PF_R3G3B2.

Formatos de orden nativo de 16 bits
PF_L16, PF_R5G6B5, PF_B5G6R5, PF_A4R4G4B4, PF_A1R5G5B5.

Formatos de orden nativo de 24 bit
PF_R8G8B8, PF_B8G8R8.

Formatos de orden nativo de 32 bits
PF_A8R8G8B8, PF_A8B8G8R8, PF_B8G8R8A8, PF_R8G8B8A8, PF_X8R8G8B8, PF_X8B8G8R8, PF_A2R10G10B10 PF_A2B10G10R10.

Formatos comprimidos
PF_DXT1, PF_DXT2, PF_DXT3, PF_DXT4, PF_DXT5.

5.8.5 CAJAS DE PÍXELES

Todos los métodos de Ogre que toman o retornan datos de imágenes, retornan un objeto PixelBox.

Un PixelBox es un primitivo que describe volúmenes (3D), imágenes (2D) o líneas (1D) de píxeles en la memoria de la CPU. Describe la localización y formato de datos de una región de memoria utilizada para datos de imagen, pero que no realiza control de memoria.

Dentro de la memoria apuntada por el miembro *data* de una caja de píxel, los píxeles son almacenados como una sucesión de "depth (profundidad)" trozos (en Z), cada uno contiene "height (altura)" filas (Y) de "width (anchura)" píxeles (X).

Las dimensiones que no se utilizan deben de ser 1. Por ejemplo, una imagen de una dimensión tiene la extensión (width, 1, 1). Una imagen de dos dimensiones tiene la extensión (width, height, 1).

Un PixelBox tiene los siguientes miembros:

data
El puntero al primer componente de los datos de la imagen en memoria.

format
El formato de pixel (ver sección 5.8.4 Formatos de píxel) de los datos de la imagen.

rowPitch
El número de elementos entre el pixel más a la izquierda en una fila y la izquierda del pixel siguiente. Este valor debe siempre ser igual a getWidth() (consecutivo) para formatos comprimidos.

slicePitch
El número de elementos entre el pixel de arriba a la izquierda (profundidad) de un trozo y el pixel de arriba a la izquierda del siguiente. Debe de ser un múltiplo de rowPitch. Este valor debe de ser siempre igual a getWidth()*getHeight() (consecutivo) para formatos comprimidos.

left, top, right, bottom, front, back
Extensión de la caja en espacio entero de tres dimensiones. Hay que tener en cuenta que las esquinas left, top y front están incluidas pero right, bottom y top no. *Left* debe de ser siempre menor o igual que *right*, *top* debe de ser siempre menor o igual que *bottom*, y *front* debe de ser siempre menor o igual que *back*.

También tiene algunos métodos útiles:

getWidth()
Obtiene la anchura de la caja.

getHeight()
Obtiene la altura de la caja. Es uno en imágenes de una dimensión.

getDepth()
Obtiene la profundidad de la caja. Es uno en imágenes de una o dos dimensiones.

setConsecutive()
Establece el rowPitch y el slicePitch de forma que el búfer se pone consecutivo en memoria.

getRowSkip()
Obtiene el número de elementos entre el pixel de más a la derecha de una fila y el pixel más a la izquierda del siguiente trozo. Este es 0 si los trozos son consecutivos.

isConsecutive()
Retorna si este búfer está en memoria de forma consecutiva o no (es decir, los espacios son iguales a las dimensiones).

getConsecutiveSize()
Retorna el tamaño (en bytes) que esta imagen ocuparía y estuviera en memoria de forma consecutiva.

getSubVolume(const Box &def)
Retorna un subvolumen de este PixelBox, como un PixelBox.

Para más información sobre estos métodos, consultar la API.

6. RECURSOS DE TEXTURAS EXTERNOS

INTRODUCCIÓN

En este tutorial se ofrecerá una breve introducción de las clases ExternalTextureSource y ExternalTextureSourceManager, su relación, y cómo funcionan los plugins. Para aquellos interesados en el desarrollo de un Plugin de Recurso de Textura o tal vez que sólo quiera saber más sobre este sistema, ver el plugin ffmpegVideoSystem, del cual se puede encontrar más información en los foros de Ogre.

¿QUÉ ES UN RECURSO DE TEXTURA EXTERNO?

¿Qué es un recurso de textura? Bien, un recurso de textura puede ser cualquier – png, bmp, jpeg, etc. Sin embargo, cargar texturas de archivos de mapa de bits tradicionales actualmente está controlado por otra parte de Ogre. Hay, sin embargo, otros tipos de recursos de los que obtener datos de textura – por ejemplo películas mpeg/avi/ect, flash, recursos de datos generados en tiempo real, definidos por el usuario, etc.

¿Cómo benefician plugins de recursos de textura externos a Ogre? Bueno, la respuesta principal es: añadir soporte para cualquier tipo de fuente de textura no requiere cambiar Ogre para soportarlo... todo lo que está en juego es escribir un nuevo plugin. Además, como el manejador utiliza la clase StringInterface para emitir los comandos/parámetros, no se requiere ningún cambio en el lector de scripts de material. Como resultado de ello, si un plugin necesita un conjunto de parámetros especiales, sólo se crea un nuevo comando en su diccionario de parámetros. - Ver el plugin ffmpegVideoSystem como ejemplo. Para hacer este trabajo, se han añadido dos clases a Ogre: ExternalTextureSource y ExternalTextureSourceManager.

CLASE EXTERNALTEXTURESOURCE

La clase ExternalTextureSource es la clase base de la que los PlugIns de recursos de texturas deben derivar. Otorga un framework genérico (a través de la clase StringInterface) con una funcionalidad limitada. Los parámetros más comunes pueden establecerse a través del interfaz de clase TexturePlugInSource o vía los comandos de StringInterface contenidos dentro de la clase. Aunque esto parece como duplicación de código, no lo es. Usando el interfaz de comandos de string, se hace extremadamente fácil derivar de plugins para añadir nuevos tipos de parámentros que se puedan necesitar.

Los Parametros de Comandos por defecto definidos en la clase base ExternalTextureSource son:

- Nombre parámetro: “filename”. Tipo Argumento: Ogre::String. Establece un nombre de archivo del plugin del que leer.
- Nombre parámetro: “play_mode”. Tipo Argumento: Ogre::String. Establece el modo de estado para usar en el plugin: “play”, “loop”, “pause”.
- Nombre parámetro: “set_T_P_S”. Tipo Argumento: Ogre::String. Usado para establecer los niveles de técnica, pasada y textura para aplicar a esta textura. Como ejemplo: Para establecer una técnica de nivel 1, una pasada de nivel 2 y una textura de nivel 3, se envía el String “1 2 3”.
- Nombre parámetro: “frames_per_second”. Tipo Argumento: Ogre::String. Establece la velocidad de actualización en frames por segundo. (Sólo valores enteros).

CLASE EXTERNALTEXTURESOURCEMANAGER

Es responsable de mantener los Plugins de recursos de texturas cargados. Ayuda en la creación de texturas de recursos de texturas desde scripts. También es el interfaz que se debería usar cuando se trata con plugins de recursos de textura.

Nota: Los prototipos de función mostrados abajo son maquetas – los nombres de parámetro se han simplificado para ilustrar mejor aquí su propósito... Los pasos necesarios para crear una nueva textura a través de ExternalTextureSourceManager son:

- Obviamente, el primer paso es tener el plugin deseado incluido en plugin.cfg para que sea cargado.
- Establecer el plugin deseado como activo a través de AdvancedTextureManager::getSingleton(). setCurrentPlugIn (String type); - type es lo que el plugin registra a manejar (por ejemplo, "video", "Flash", "lo que sea", etc.).
- Nota: Consultar el plugin deseado para ver qué parámetros son necesarios. Establecer pares parámetros/valor a través de AdvancedTextureManager::getSingleton().getCurrentPlugIn() -> setParameter (String param, String Value);.
- Una vez establecidos los parámetros necesarios, una simple llamada a AdvancedTextureManager::getSingleton ().getCurrentPlugIn () -> createDefinedTexture (sMaterialName); creará una textura con el nombre de material determinado.

El controlador también aporta un método para borrar un material de recurso de textura: AdvancedTextureManager::DestroyAdvancedTexture(String sTextureName);. El método de destrucción funciona difundiendo el nombre del material a todos los TextureSourcePlugIns cargados. El PlugIn que creó el material es responsable del borrado, mientras que los otros Plugins ignorarán la petición. Esto significa que no se necesita preocuparse de qué PlugIn creó el material, o de la activación del Plugin. Simplemente con llamar al método del controlador para borrar el material. También, todos los plugins de textura deben manejar la limpieza cuando estén parados.

SCRIPT DE MATERIAL DE RECURSOS DE TEXTURA

Como se ha comentado anteriormente, el proceso de definir/crear un recurso de textura puede realizarse con un archivo de script de material. Aquí hay un ejemplo de la definición de un script de material – Nota: Este ejemplo está basado en los parámetros del plugin ffmpegVideoSystem.

```
material Example/MyVideoExample
{
        technique
        {
                pass
                {
                        texture_unit
                        {
                                texture_source video
                                {
                                        filename mymovie.mpeg
                                        play_mode play
                                        sound_mode on
                                }
                        }
                }
        }
```

```
}
```

Código 75: Script de Material de recursos de textura.

Hay que tener en cuenta que los dos primeros pares parámetro/valor están definidos en la clase base ExternalTextureSource y que el tercer par parámetro/valor no está definido en la clase base… Este parámetro es añadido al diccionario de parámetro usando ffmpegVideoPlugin… Esto muestra que extender la funcionalidad con plugins es extremadamente fácil. También, hay que prestar atención a la línea: texture_source video. Esta línea identifica que esta unidad de textura viene de un plugin de recurso de textura. Requiere un parámetro que determina qué plugin de textura se va a utilizar. En el ejemplo de abajo, el plugin pedido es uno registrado con el nombre "video".

DIAGRAMA SIMPLIFICADO DEL PROCESO

Este diagrama usa ffmpegVideoPlugin como ejemplo, pero todos los plugins trabajarán de la misma manera a cómo se registran/usan aquí. También hay que tener en cuenta que los Plugins TextureSource son cargados/registrados antes de parsear los scripts. Esto no significa que estén inicializados…. Los plugins no se inicializan hasta que se establezcan como ¡activo! Esto sirve para asegurar que un sistema de renderizado está configurado antes de que los plugins puedan hacerle una llamada.

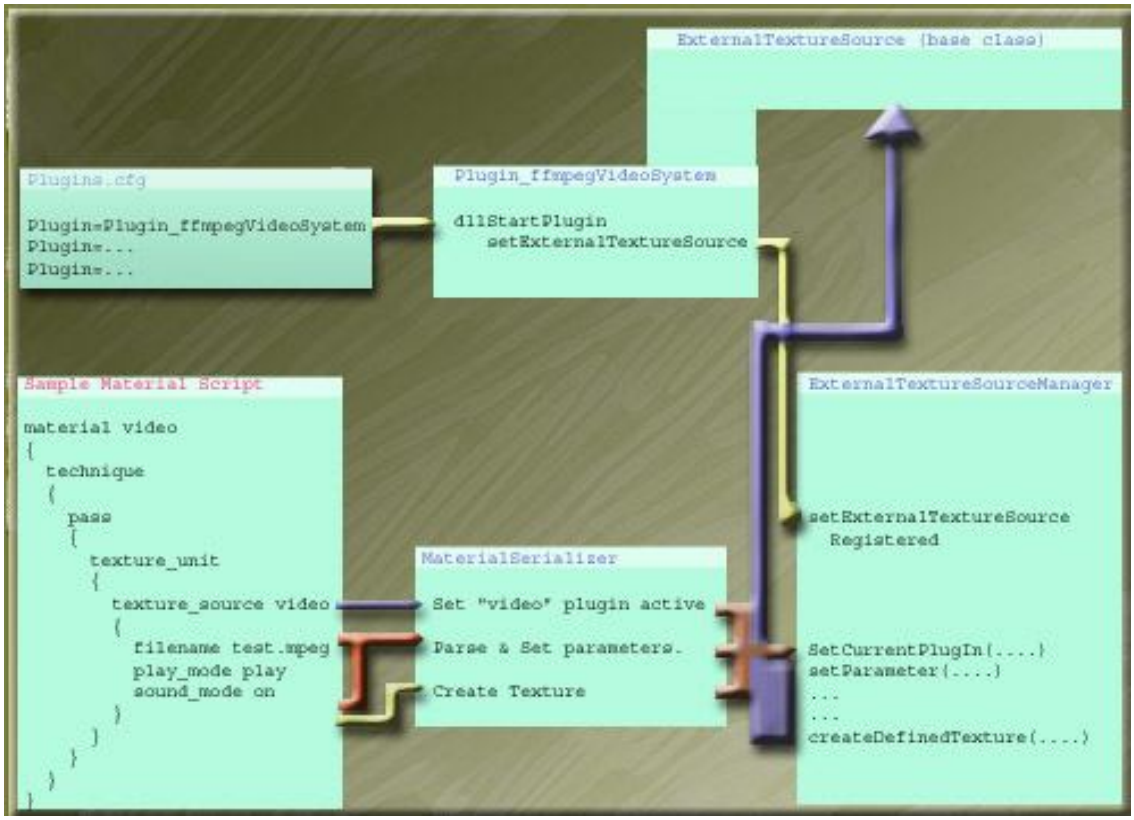

Ilustración 2: Diagrama de proceso de carga y registro de un plugin de recursos de textura.

7. SOMBRAS

Las sombras son claramente una parte importante del renderizado de una escena creíble - proporcionan una sensación más tangible de los objetos de la escena, y ayuda al espectador en la comprensión de la relación espacial entre los objetos. Por desgracia, las sombras son también uno de los aspectos más desafiantes del renderizado 3D, y siguen siendo en gran medida un área activa de investigación. Aunque existen muchas técnicas para hacer las sombras, ninguna es perfecta y todas vienen con ventajas y desventajas. Por esta razón, Ogre proporciona múltiples implementaciones de sombra, con un montón de opciones de configuración, para que se pueda elegir cuál es la técnica más apropiada para la escena.

Las implementaciones de sombra básicamente se dividen en 2 categorías generales: 7.1 Sombras de plantilla y 7.2 Sombras basadas en texturas. Estas describen el método por el cual se genera la forma de la sombra. Además, hay más de una forma de renderizar la sombra en la escena: 7.3 Sombras Modulativas, que oscurece la escena en las zonas de sombra, y el 7.4 Máscara de luz aditiva que por el contrario se acumula la contribución de la luz en las zonas que no están en la sombra. También se tiene la opción de Sombras de textura integrada que dan control completo sobre la aplicación de texturas de sombra, lo que permite complejos shaders de una sola pasada. Ogre soporta todas estas combinaciones.

Habilitando sombras

Las sombras están deshabilitadas por defecto, aquí se indica cómo cambiar esto y configurarlas en sentido general:

1. Habilitar una técnica de sombra en el SceneManager como la **primera** tarea que se hace en la configuración de la escena. Es importante hacer esto lo primero porque la técnica de sombra puede alterar la forma en que se cargan las mallas. Aquí hay un ejemplo:

```
mSceneMgr->setShadowTechnique(SHADOWTYPE_STENCIL_ADDITIVE);
```

Código 76: Habilita una técnica de sobreado.

2. Crear una o más luces. Hay que tener en cuenta que no todos los tipos de luz son necesariamente admitidos por todas la técnicas de sombras, se deberían de chequear las secciones sobre cada técnica para comprobar esto. También se debe de tener en cuenta que ciertas luces pueden no proyectar sombras, se puede deshabilitar esto llamando a setCastShadows(false) en la luz, que por defecto es true.

3. Desabilitar proyección de sombras en objetos que no deberían de proyectarlas. Llamando a setCastShadows(false) en los objetos que no se quiera que proyecten sombras, por defecto todos los objetos tienen este valor a true.

4. Configurar sombras lejanas. Se puede limitar la distancia en la que las sombras se consideran, por razones de rendimiento, llamando a SceneManager::setShadowFarDistance.

5. Apagar la recepción de sombras en materiales que no deberían recibirlas. Se pueden apagar (nota, no para la proyección de sombras, que se hace por objeto) llamando a Material::setReceiveShadows o usando el atributo de material receive_shadows. Esto es útil, por ejemplo, para materiales que pueden ser considerados semi-iluminados. Hay que tener en cuenta que los materiales transparentes son normalmente excluidos de recibir y proyectar sombras, aunque se puede ver la opción transparency_casts_shadows para excepciones.

Exclusión de sombras

Ogre por defecto trata todos los objetos no transparentes como proyectores y receptores de sombras (dependiendo de la técnica de sombra, pueden no estar ambos disponibles a la vez, comprobar la documentación de la técnica escogida primero). Se pueden deshabilitar las sombras de varias formas:

Apagando proyección de sombras en la luz

Llamando a Light::setCastsShadows(false) indicará que las luces no proyectarán sombras.

Apagando recepción de sombra en un material

Llamando a Material::setReceiveShadows(false) prevendrá a cualquier objeto que utilice este material de recibir sombras.

Apagando la proyección de sombras en objetos individuales

Llamando a MovableObject::setCastsShadows(false) se deshabilita la proyección de sombras para el objeto.

Apagando sombras en un grupo en cola de renderizado al completo

Llamando a RenderQueueGroup::setShadowsEnabled(false) apagará tanto la proyección como la recepción de sombras en un grupo de cola de renderizado al completo. Esto es útil porque Ogre tiene que hacer las tareas de iluminación por grupo para poder preservar el orden dentro del grupo. Ogre automáticamente deshabilita sombras en un número de grupos, como RENDER_QUEUE_BACKGROUND, RENDER_QUEUE_OVERLAY, RENDER_QUEUE_SKIES_EARLY y RENDER_QUEUE_SKIES_LATE. Si se eligen más colas de renderizado (y por defecto, no se quieren usar más que estas colas y las colas 'estandar', ¡hay que ignorar esto si no se sabe lo que significa!), hay que estar atento de que cada cola tendrá un coste de configuración de la iluminación, y se deberán de deshabilitar las sombras en las colas adicionales que se usan si se puede.

7.1 SOMBRAS DE PLANTILLA

Las sombras de plantilla son un método por el cual se crea una 'máscara' para la pantalla usando una propiedad llamada búfer de plantilla. Esta máscara puede ser utilizada para excluir áreas de la pantalla para renderizadores posteriores, y así puede ser utilizado para excluir o incluir áreas en sombra. Se habilitan llamando a SceneManager::setShadowTechnique con un parámetro SHADOWTYPE_STENCIL_ADDITIVE o SHADOWTYPE_STENCIL_MODULATIVE. Como la plantilla sólo puede enmascarar áreas para estar 'disponibles' o 'no disponibles', las sombras de plantilla tienen bordes 'duros', esto es que hay una clara división entre luces y sombras – no es posible suavizar estos bordes.

Para generar la plantilla, 'los volúmenes de sombra' son renderizados por extrusión de la silueta de la sombra alejada de la luz. Donde estos volúmenes de sombras intersecten con otros objetos (o el molde, cuando está semi-sombreado es soportado usando esta técnica), se actualiza la plantilla, permitiendo a las siguientes operaciones diferenciar entre luz y sombra. Cómo se utiliza esto para renderizar las sombras depende de si se usan 7.3 Sombras modulativas o 7.4 Máscara de luz aditiva. Los objetos pueden arrojar y recibir sombras de plantilla, de forma que el semi sombreado es natural.

La ventaja de las sombras de plantilla es que pueden hacer el semi sombreado simple en hardware de gama baja, haciendo que se tenga que llevar la cuenta de polígonos bajo control. En contraste hacer semi sombreado con sombras de textura requiere una máquina más moderna (Ver sección 7.2 Sombras basadas en texturas). Por esta razón, se eligirá utilizar una sombra de plantilla si se necesita una solución de sombra precisa para una aplicación destinada a maquínas viejas con menos recursos.

Las desventajas de las sombras de plantilla son numerosas, aunque especialmente en hardware moderno. Como la sombra de plantilla es una técnica geométrica, hay inherentemente un aumento del coste según aumenta el número de polígonos que se usan, indicando que se penaliza el aumento del nivel de detalle en las mallas. El coste de fillrate (número de pixeles que una tarjeta de vídeo puede renderizar en un segundo), que viene de tener que renderizar volúmenes de sombra, también se escalará de la misma forma. Mientras algunas aplicaciones modernas pueden utilizar grandes cuentas de polígonos, las sombras de plantilla pueden comenzar a ser un cuello de botella. Además, el aspecto visual de las sombras de plantillas es un poco primitivo – las sombras siempre tienen bordes duros, y no hay posibilidad de hacer cosas con los shaders ya que la plantilla no está disponible para la manipulación. Sin embargo, si una aplicación está destinada a máquinas potentes, se puede preferir considerablemente cambiar a sombras de texturas (Ver 7.2 Sombras basadas en texturas).

Hay un número de asuntos que son específicos para sombras de plantilla que hay que considerar:

- Sobrecarga de CPU
- Distancia de extrusion
- Posicionamiento plano de cámara lejana
- Lista de bordes de malla
- El borde de silueta
- Ser realista
- Optimizaciones de plantilla realizadas por Ogre

Sobrecarga de CPU

Calcular el volumen de la sombra de una malla puede ser costoso, y tiene que hacerse sobre la CPU, no es una característica del hardware de aceleración. Por lo tanto, se puede encontrar que si hay un uso excesivo de esta característica, se puede crear un cuello de botella de la CPU para la aplicación. Ogre de forma agresiva elimina los objetos que no pueden tener sombras proyectadas sobre el tronco, pero hay límites en lo que se puede hacer, y grandes, las sombras alargadas (por ejemplo, representar un sol muy bajo) son muy difíciles de desechar de manera eficiente. Hay que tratar de evitar tener demasiados proyectores de sombra alrededor de un objeto, y evitar las largas sombras si se puede. Asimismo, hacer uso del parámetro de “sombra lejana” en la SceneManager, puede eliminar proyectores de sombra distantes de la construcción de volumen de sombras y ahorrar tiempo, a expensas de tener sombras sólo para objetos cercanos. Por último, hacer uso de las características Nivel-De-Detalle (LOD) de Ogre; se pueden generar automáticamente cálculos de LOD para las mallas en el código (ver la documentación de la API de malla), o al utilizar las herramientas de malla como OgreXmlConverter y OgreMeshUpgrader. Como alternativa, se pueden asignar propios LODs manuales por el envío de archivos de malla alternativos a menores niveles de detalle. Ambos métodos causan que la complejidad de volumen de sombra disminuya a medida que los objetos se alejan, lo que ahorra un valioso tiempo de cálculo de volumen.

Distancia de Extrusión

Cuando los programas de vértice no están disponibles, Ogre sólo puede extrudar volúmenes de sombra en una distancia finita desde el objeto. Si un objeto está demasiado cerca de una luz, cualquier distancia de extrusión finita será inadecuada para garantizar que todos los objetos sean sombreados apropiadamente por este objeto. Sin embargo, se avisa de no permitir a proyectores de sombra pasar demasiado cerca de fuentes de luz si se puede evitar, a no ser que se pueda garantizar que el objetivo tenga un hardware de programa de vértice capaz (en este caso, Ogre extruda el volumen hasta el infinito usando un programa de vértice de forma que el problema no aparece).

Cuando no es posible una extrusión infinita, Ogre usa extrusión finita, ya sea derivada de un rango de atenuación de una luz (en el caso de un punto de luz o foco), o una distancia de extrusión fijada en la aplicación en el caso de luces direccionales. Para cambiar la distancia de extrusión de luces direccionales, hay que usar SceneManager::setShadowDirectionalLightExtrusionDistance.

Posicionamiento lejano de cámara plana

Los volúmenes de sombras de plantilla dependen mucho de no chocar con el plano lejano. Cuando se habilitan sombras de plantilla, Ogre inmediatamente cambia las propiedades del plano lejano de las cámaras de forma que no haya plano lejano – es decir, se establece en el infinito (Camera::setFarClipDistance(0)). Esto evita artefactos causados por la colisión de capas oscuras sobre volúmenes de sombra, a expensas de una (muy) pequeña cantidad de precisión de profundidad.

Listas de bordes de malla

Las sombras de plantilla sólo pueden calcularse cuando se ha construido una 'lista de bordes' para toda la geometría en una malla. Los exportadores oficiales y herramientas automáticamente hacen estas listas (o tienen una opción para ello), pero si se crean mallas propias, hay que recordar que hay que construir listas de bordes para las mallas ántes de usarlas con sombras de plantilla – se puede hacer esto usando OgreMeshUpgrade o OgreXmlConverter, o llamando a Mesh::buildEdgeList antes de exportar o utilizar la malla. Si una malla no tiene la lista de bordes, Ogre asume que la malla no admite la proyección de sombras.

El borde de silueta

El sombreado de plantilla se hace buscando la silueta de una malla, y proyectándola formando un volumen. Esto significa que hay un límite definido en el proyector de sombra entre la luz y la sombra; un juego de bordes donde hay un triángulo cuyo lado está de cara hacia la luz, y otro está en la cara opuesta. Esto produce un borde afilado alrededor de la malla como transcurre en la transicción. Siempre que haya una pequeña o ninguna luz en la escena, y la malla tenga normales suaves para producir un cambio gradual de luz en su sombra subyacente, el borde de la silueta puede ocultarse – esto funciona mejor cuanto mayor sea el mosaico de la malla. Sin embargo si la escena incluye luz ambiente, entonces la diferencia está más marcada. Esto es especialmente cierto cuando se utilizan 7.3 Sombras modulativas, porque la contibución de la luz de cada área sombreada no se tiene en cuenta por esta aproximación simplificada, y usando 2 o más luces en una escena usando también sombras de plantilla modulativas no es aconsejable; los bordes de la silueta siempre se marcarán. Las luces aditivas no sufren esto, ya que malamente cada luz se enmascara individualmente, por lo que sólo la luz ambiente es la que puede mostrar los bordes de las siluetas.

Ser realista

No hay que esperar poder lanzar cualquier escena usando cualquier hardware con el algoritmo de sombras de plantilla y que salga perfecto, con resultados de velocidad óptimos. Las sombras son una técnica cara y compleja, por lo que se suelen imponer algunas limitaciones razonables a la hora de establecer luces y objetos; estos no son realmente una restricción, pero hay que ser conscientes de que no hay libertad para todo.

- Hay que intentar evitar dejar pasar objetos muy cerca (o incluso a través) de luces – puede parecer correcto, pero es uno de los casos en los que pueden aparecer artefactos en máquinas que no son capaces de soportar programas de vértice.

- Hay que tener en cuenta que los volúmenes de sombra no respetan la 'solidez' de los objetos a través de los que pasan, y si estos objetos por si mismos no son proyectores de sombras (que ocultan el efecto), entonces el resultado será que se pueden ver sombras en el otro lado que deberían ser de un objeto oculto.

- Se puede hacer uso de SceneManager::setShadowFarDistance para limitar el número de volúmenes de sombras construidos.

- Se puede hacer uso de LOD para reducir la complejidad de los volúmenes de sombra a distancia.

- Evitar sombras muy largas (anochecer y amanecer) – exacerban otras cuestiones como el volumen de recorte, fillrate, y causan que muchos objetos a gran distancia requieran de construcción de volumen.

Optimizaciones de Plantilla realizadas por Ogre

A pesar de todo, las sombras de plantilla pueden verse muy bien (especialmente con 7.4 Máscaras de luz aditivas) y puede ser rápido si se respetan las reglas de abajo. Además, Ogre viene pre-empaquetado con un montón de optimizaciones que pueden ayudar a hacer esto tan rápido como sea posible. Esta sección es buena para desarrolladores o gente interesada en saber algo sobre el funcionamiento de Ogre.

Extrusión de programas de vértice

Como se ha mencionado anteriormente, Ogre realiza la extrusión de volúmenes de sombras en el hardware, en hardware capaz de usar programas de vértice (ejemplos: GeForce3, Raedon 8500 o mejores). Esto tiene 2 beneficios mayores; uno obvio es la velocidad, pero el segundo es que los programas de vérice pueden extrudar puntos hasta el infinito, mientras que un pipeline de función fija no puede, por lo menos no sin realizar todos los cálculos en software. Esto lleva a volúmenes más robustos, y también elimina más de la mitad de los triángulos de volumen en luces direccionales ya que todos los puntos se proyectan a un punto simple en el infinito.

Optimización del test de recorte

Ogre usa un rectángulo de recorte para limitar el efecto de un punto/foco de luz cuando su rango no cubre la vista al completo; esto significa que se guarda fillrate cuando se renderizan volúmenes de plantilla, especialmente con luces distantes.

Algoritmos de pasada-Z y fallo-Z

El algoritmo de fallo-Z, atribuido a John Carmack, se utiliza en Ogre para estar seguro de que las sombras son robustas cuando la cámara pasa a través de un volumen de sombra. Sin embargo, el algoritmo de fallo-Z es más caro que el tradicional pasada-Z; por ello Ogre detecta cuando el fallo-Z es requerido y lo usa, pasada-Z se utiliza el resto de las veces.

Plantilla de 2-caras y embalaje de plantilla

Ogre soporta estas dos extensiones (2-Siced stencilling y stencil wrapping), cuando se admiten permiten que los volúmenes sean renderizados en una simple pasada en lugar de tener que hacer una pasada para la cara trasera y otra para la cara delantera. Esto no salva el fillrate, ya que se realiza el mismo número de actualizaciones de plantilla, pero salva la configuración inicial y la sobrecarga que ocurre en el driver cada vez que se hace una llamada de renderizado.

Sacrificio de volúmenes de sombra agresivo

Ogre es muy bueno detectando qué luces pudieran estar afectando los troncos, y desde cuales, qué objetos pudieran proyectar sombra en el tronco. Esto significa que no se invierte tiempo construyendo la geometría de la sombra ya que no se necesita. Establecer la distancia lejana de sombra es otra forma importante de reducir la sobrecarga por las sombras de plantilla ya que se sacrifican volúmenes de sombras lejanas incluso si son visibles, que es beneficioso en la práctica, ya que el interés se centra en las sombras de objetos cercanos.

7.2 SOMBRAS BASADAS EN TEXTURAS

Las sombras de textura involucran a renderizar proyectores de sombras que van desde el punto de vista de la luz hasta una textura, que es proyectada en los receptores de sombra. La ventaja principal de las sombras de textura en contraposición a 7.1 Sombras de plantilla es que la sobrecarga por aumentar el detalle geométrico es bastante menor, ya que no se necesitan realizar cálculos por cada triángulo. La mayor parte del trabajo de renderizar una sombra de textura se realiza en la tarjeta gráfica, que en el presente superan a las CPUs en términos de velocidad de desarrollo. Además las sombras de textura son **mucho** más personalizables – se pueden poner en shaders para aplicarlas como se quiera (particularmente con Sombras de Textura Integradas), se puede realizar un filtro para crear sombras menos fuertes o realizar otros efectos especiales sobre ellas. Básicamente, las máquinas más modernas usan sombras de textura como su primera técnica de sombreado simplemente porque son más potentes, y el incremento de la velocidad de las GPUs rápidamente amortiza los costes de fillrate o costes de acceso a las texturas.

La principal desventaja de las sombras de texturas es que, como son simplemente una textura, tienen una resolución fijada, por lo que si se extiende, la pixelación de la textura puede ser obvia. Hay varias formas de combatir esto:

Escogiendo una proyección base

La proyección más simple justamente es renderizar los proyectores de sombras desde la perspectiva de la luz usando una configuración de cámara normal. Esto puede parecer mal, aunque hay muchas otras proyecciones que pueden ayudar a mejorar la calidad desde la perspectiva de la cámara principal. Ogre admite plugins de bases de proyección a través de la clase ShadowCameraSetup, y viene con varias opciones existentes – Uniform (el más simple), Uniform Focussed (que es una proyección de cámara normal, excepto que la cámara está enfocada al área en el que la cámara principal está mirando), LiSPSM (Light Space Perspective Shadow Mapping – con ambos focos y distorsiona el tronco de sombra basándose en la cámara principal) y Plan Optimal (que tata de optimizar la fidelidad de sombra para un receptor plano simple).

Filtrado

Se pueden probar texturas de sombra varias veces en lugar de una para suavizar los bordes y mejorar la apariencia. El porcentaje de filtrado más cercano (PCF) es el método más popular, aunque existen múltiples variantes dependiendo de la cantidad y el patrón de las muestras que se tomen. Las sombras demo incluyen un 5-TAP PCF de ejemplo combinado con el mapeo de profundidad de sombra.

Usando una textura mayor

De nuevo como las GPUs son rápidas y ganan más memoria, se puede escalar la textura para utilizar esta ventaja.

Si se convinan estas 3 técnicas, se puede obtener una solución con una sombra de calidad muy alta.

Otra cuestión son los puntos de luz. Como las sombras de textura requieren una renderización en la dirección de la luz, las luces omnidireccionales (puntos de luz) requerirían 6 renderizadores para cubrir totalmente todas las direcciones en que las sombras pueden proyectarse. Por esta razón, Ogre principalmente soporta luces direccionales y focos para generar sombras de textura; se pueden utilizar puntos de luz pero sólo funcionarán si están fuera de cámara, ya que esencialmente se convierte en un foco brillante en el tronco de la cámara para los propósitos de sombras de textura.

Luces direccionales

Las luces direccionales en teoría sombrean la escena por completo desde una luz infinitamente distante. Ahora, como sólo se tienen texturas finitas se verán con peor calidad si se estiran a lo largo de toda la escena, claramente se requiere una simplificación. Ogre establece una textura de sombra por encima del área inmediatamente delante de la cámara, y lo mueve como se mueve la cámara (aunque gire en su movimiento de forma que el pequeño efecto de 'sombra nadadora' causado por el movimiento de la textura es mínimizado). El rango en el que la sombra se extiende, y el recorrido usado para moverla delante de la cámara son configurables (Ver sección Configurando sombras de texturas). En el borde de la sombra, Ogre desvanece la sombra basándose en otros parámetros configurables de forma que el final de la sombra es más blando.

Focos

Los focos son mucho más fáciles de representar como texturas de sombras renderizables que las luces direccionales, ya que naturalmente son un tronco. Ogre representa los focos renderizando directamente la sombra desde la posición de la luz, en la dirección del cono de luz; el campo de visión de la cámara de textura se ajusta basándose en los ángulos de caida del foco. Además, para ocultar el hecho de que la textura de sombra es cuadrada y tiene bordes definidos que pueden verse fuera del foco, Ogre utiliza una segunda unidad de textura cuando proyecta la sombra en la escena que desvanece gradualmente la sombra en un círculo proyectado alrededor del foco.

Puntos de Luz

Como se ha mencionado anteriormente, para soportar la propiedad de puntos de luz se requerirían múltiples renderizadores (6 para un renderizado cúbico o quizás 2 para un mapeo parabólico menos preciso), de forma que se pueden aproximar los puntos de luz como focos, donde se cambia la configuración al vuelo para hacer el brillo de luz desde su posición por todo el tronco de visión. Esto no es una configuración ideal ya que indica que realmente sólo se puede trabajar si las posiciones de los puntos de luz están fuera de vista, y además los cambios de parametrización pueden causar algo de 'nado' de la textura. Generalmente se recomienda evitar hacer puntos de luz proyectando sombras de textura.

Proyectores de sombra y receptores de sombra

Para habilitar las sombras de textura, se usan las técnicas SHADOWTYPE_TEXTURE_MODULATIVE o SHADOWTYPE_TEXTURE_ADDITIVE; tal y como sugieren los nombres, estas producen 7.3 Sombras modulativas o 7.4 Máscara de luz aditiva respectivamente. Lo más económico y simple de las técnicas de sombras de textura es no utilizar la información de profundidad, simplemente renderizan proyectores en una textura y la renderizan en los receptores como color plano – esto indica que el semi sombreado no es posible utilizando estos métodos. Este es el comportamiento por defecto si se usa de modo automático, compatible con técnicas de sombreado de textura de función fija (y por ello válido en

hardware final poco potente). Sin embargo se pueden utilizar técnicas basadas en shaders a través de materiales con sombra personalizada para proyectores y receptores para realizar algoritmos de sombreado más complejos, como el mapeo de sombreado de profundidad que realiza el semi sombreado. Ogre trae un ejemplo de esto en su demo de sombreado, aunque sólo es utilizable con tarjetas de Shader Model 2 o mejores. Si bien la función fija de mapeo de sombra de profundidad está disponible en OpenGL, no se ha estandarizado nunca en Direct3D de forma que utilizar shaders en materiales receptores y proyectores personalizados es la única forma portable de hacerlo. Si se utiliza este enfoque, hay que llamar a SceneManager::setShadowTextureSelfShadow con el parámetro a 'true' para permitir que los proyectores de sombra de textura sean también receptores.

Si no se utiliza el mapeo de sombra de profundidad, Ogre divide los proyectores y receptores de sombra en dos grupos separados. Simplemente apagando la proyección de sombra en un objeto, automáticamente se convierte en un receptor de sombra (aunque esto se puede deshabilitar estableciendo la opción 'receive_shadows' a 'false' en el script de material). Similarmente, si un objeto se establece como proyector de sombra, no puede recibir sombras.

CONFIGURANDO SOMBRAS DE TEXTURA

Hay un número de configuraciones que pueden ayudar a configurar sombras basadas en textura.

- Número máximo de texturas de sombra
- Tamaño de textura de sombra
- Distancia de sombras
- Offset de textura de sombra (Luces direccionales)
- Opciones de desvanecimiento de sombra
- Configuración de cámara de sombra personalizada
- Sombras de textura integradas

Número máximo de texturas de sombra

Las sombras de textura ocupan memoria de textura, y para evitar colapsar el pipeline de renderizado, Ogre no reutiliza la misma textura de sombra para múltiples luces en el mismo frame. Esto significa que cada luz que va a proyectar sombras debe tener su propia textura de sombra. En la práctica si se tienen muchas luces en la escena, no se debería de desear incurrir en este tipo de sombrecarga de textura.

Se puede ajustar esto de forma manual simplemente apagando la proyección de sombras para luces que no se quiera que proyecten sombras. Además, se puede establecer un límite máximo en el número de texturas de sombra que Ogre está permitido a utilizar llamando a SceneManager::setShadowTextureCount. En cada frame, Ogre determina la luz que puede estar afectando al tronco, y entonces asigna el número de texturas de sombra que está permitido usar para las luces usando la base primero en llegar, primero en servir. Cualquier luz adicional no proyectará sombras en este frame.

Hay que tener en cuenta que se puede establecer el número de texturas de sombras y su tamaño al mismo tiempo usando el método SceneManager::setShadowTextureSettings; esto es útil porque cada llamada individual requiere recursos potenciales para la creación/destrucción de la textura.

Tamaño de textura de sombra

El tamaño de las texturas utilizado para renderizar en los proyectores de sombras puede ser alterado; usando texturas grandes, claramente se obtendrán sombras de calidad, pero a un coste mayor de uso de memoria. Para cambiar el tamaño de una textura se hace llamando a SceneManager::setShadowTextureSize – Se asume que las texturas son cuadradas y se debe especificar el tamaño de textura como potencia de 2. Hay que tener cuidado de que cada textura de sombra modulativa tendrá un tamaño de size*size*3 bytes de memoria de textura.

Importante: Si se utiliza el sistema de renderizado GL, el tamaño de textura de sombra sólo puede ser mayor (en cualquier dimensión) que el tamaño de la superficie de ventana principal si el hardware soporta las extensiones Frame Buffer Object (FBO) o Pixel Buffer Object (PBO). La mayor parte de las tarjetas modernas soportan esto, pero hay que tener cuidado con las tarjetas antiguas – se puede comprobar la habilidad del hardware para controlar esto a través de ogreRoot->getRenderSystem()->getCapabilities()->hasCapability(RSC_HWRENDER_TO_TEXTURE). Si esto retorna falso, si se crea una textura de sombra mayor en cualquier diemensión que la superficie principal, el resto de la textura de sombra estará en blanco.

Distancia de sombra

Determina la distancia en que se terminan las sombras; también determina lo lejos que se estiran las sombras de textura en la distancia para luces direccionales – reduciendo este valor, o incrementando el tamaño de la textura, se puede mejorar la calidad de las sombras para luces direccionales con el coste de una terminación de sombra cercana o el incremento del uso de memoria, respectivamente.

Offset de textura de sombra (Luces direccionales)

Como se ha comentado anteriormente en la sección de luces direccionales, el renderizado de sombras para luces direccionales es una aproximación que permite utilizar un simple renderizador para cubrir una gran área con sombras. Este parámetro de Offset afecta cómo de lejos está la posición de la cámara del centro, como una proporción de la distancia de la sombra. Cuanto mayor sea este valor, más 'útil' será la textura de sombra, ya que estará por delante de la cámara, pero también será mayor el Offset, hay más posibilidades de ver accidentalmente el borde de la textura de sombra en ángulos más extremos. Se cambia este valor llamando a SceneManager::setShadowDirLightTextureOffset, por defecto es 0.6.

Opciones de desvanecimiento de sombra

El desvanecimiento de sombra antes de la distancia de sombra hace que la terminación de la sombra no sea brusca. Se pueden configurar los puntos de inicio y fin de este desvanecimiento llamando a los métodos SceneManager::setShadowTextureFadeStart y SceneManager::setShadowTextureFadeEnd, ambos toman las distancia como proporción de la distancia de sombra. Debido a la inexactitud causada por usar una textura cuadrada y una distancia de desvanecimiento radial, hace que no se pueda utilizar el valor 1.0 como desvanecimiento final. Si se establece así, se verán artefactos en el extremo opuesto de los bordes. Los valores por defecto son 0.7 y 0.9, que se utilizan para la mayor parte de los propósitos, aunque se pueden cambiar si se desea.

SOMBRAS DE TEXTURA Y PROGRAMAS DE VÉRTICES/FRAGMENTOS

Cuando se renderizan proyectores de sombra en una textura de sombra modulativa, Ogre apaga todas las texturas, y todos los contribuidores de luz excepto la luz ambiente, que se establece como el color de la sombra (Color de sombra). Para sombras aditivas, se renderizan los proyectores en una textura en blanco y negro. Esto es suficiente para renderizar proyectores de sombra para técnicas de materiales de función fija, sin embargo donde se utilizan programas de vértice, Ogre no tiene mucho control. Si se utiliza un programa de vértice en la **primera pasada** de una técnica, entonces se debe de comunicar a Ogre qué programa de vértice se desea utilizar cuando se rendericen los proyectores de sombra; ver Sombras y programa de vértice para detalles completos.

Configuraciones de cámaras de sombra personalizadas

Como se ha comentado anteriormente, uno de los inconvenientes de las sombras de texturas es que la resolución de la textura es finita, y es posible obtener aliasing cuando el tamaño del texel de sombra es mayor que un pixel de la pantalla, debido a la proyección de la textura. Para direccionar esto, se pueden especificar unas bases de proyección alternativas usando o creando subclases de la clase ShadowCameraSetup. La versión por defecto se llama DefaultShadowCameraSetup y establece un tronco simple para puntos y focos y un tronco ortográfico para luces direccionales. También hay una clase PlaneOptimalShadowCameraSetup especializada en la proyección en un plano, aportando una mejor definición siempre y cuando los receptores de sombra existan principalmente en un solo plano. Otras clases de configuración (por ejemplo, se puede crear una versión de mapeo de sombra de perspectiva o trapezoidal) se pueden crear y conectar en tiempo de ejecución ya sea para luces individuales como para el SceneManager en su conjunto.

Sombras de textura integradas

Las sombras de textura tienen una ventaja principal sobre las sombras de plantilla – los datos utilizados para representarlas pueden referenciarse en shaders habituales. Aunque los modos de sombras de textura por defecto (SHADOWTYPE_TEXTURE_MODULATIVE y SHADOWTYPE_TEXTURE_ADDITIVE) automáticamente renderizan sombras, las desventajas son que estos son generalizados como complementos de los materiales, que tienden a utilizar más pasadas de la escena para usarlos. Además, no se tiene todo el control sobre la composición de las sombras.

Aquí es donde las sombras de texturas 'integradas' avanzan. Ambos tipos de sombras de textura de arriba tienen unas versiones alternativas llamadas SHADOWTYPE_TEXTURE_MODULATIVE_INTEGRATED y SHADOWTYPE_TEXTURE_ADDITIVE_INTEGRATED, donde en lugar de renderizarse las sombras, se crea una sombra de textura y se espera que el usuario utilice esa sombra de textura como se desee cuando se renderizan objetos receptores en la escena. La desventaja es que se tiene que tener en cuenta la recepción de sombra en cada uno de los materiales si se utiliza esta opción – La ventaja es que se tiene control total sobre cómo se utilizan las texturas de sombra. La gran ventaja aquí es que se puede realizar un sombreado más complejo, teniendo en cuenta las sombras, lo que es posible utilizando los enfoques de tuerca generalizados, y probablemente se puedan escribir un pequeño número de pasadas, ya que se sabe exactamente lo que se necesita y se pueden combinar pasadas donde sea posible. Cuando se utiliza alguno de estos enfoques de sombreado, la única diferencia entre aditivo y modulativo es el color de los proyectores en la textura de sombra (el color de sombra para el modulativo, y en negro para aditivo) – el cálculo real de cómo la textura afecta a los receptores depende del usuario. No se realizarán pasadas modulativas separadas, no se separarán de los materiales cuando ocurran luz ambiente/por

luz/etc – todo es absolutamente determinado por el material original (que puede tener pasadas modulativas o iteraciones por luz si se desea, aunque no es requerido).

Se hace referencia a una textura de sombra en un material que implemente este enfoque usando la directiva 'content_type shadow' en la {texture_unit}. Implícitamente se referencia a una textura de sombra basada en el número de veces de se utiliza esta directiva en la misma pasada, y la opción light_start o la iteración de pasada basada en luz, que puede comenzar con un índice de luz mayor que 0.

7.3 SOMBRAS MODULATIVAS

Las sombras modulativas trabajan ensombreciendo la escena ya renderizada con un color fijado. Primero, se renderiza la escena normalmente conteniendo todos los objetos que serán sombreados, entonces se realiza una pasada modulativa por luz que sombrea áreas con sombra. Finalmente, se renderizan los objetos que no reciben sombras.

Hay dos técnicas de sombras modulativas; basadas en plantillas (ver sección 7.1 Sombras de plantilla: SHADOWTYPE_STENCIL_MODULATIVE) y basadas en textura (ver sección 7.2 Sombras basadas en texturas: SHADOWTYPE_TEXTURE_MODULATIVE). Las sombras modulativas están en un modelo impreciso de iluminación, ya que oscurecen las áreas de sombra de forma uniforme, independientemente de la cantidad de luz que tendría que caer en el área. Sin embargo, pueden dar unos resultados bastante atractivos con mucha menos sobrecarga que otros métodos más 'correctos' como 7.4 Máscara de luz aditiva, y también combinan bien con la iluminación estática pre-horneada (como los mapas de luces pre-calculados), mientras que la iluminación aditiva no. La cosa principal a considerar es que usando múltiples fuentes de luz pueden resultar sombras oscuras en exceso (donde las sombras se superponen, que intuitivamente parece correcto, pero no lo es físicamente) y artefactos cuando se utilizan sombras de plantilla (Ver la sección El borde de silueta).

Color de sombra

El color que se utiliza para oscurecer las áreas con sombra se establece mediante SceneManager::setShadowColour; por defecto se establece en gris oscuro (de forma que el color subyacente se sigue mostrando a través de un bit).

Hay que tener en cuenta que si se están utilizando sombras de textura, se tiene la opción adicional de utilizar Sombras de textura integradas en lugar de verse forzado a utilizar pasadas separadas de la escena para renderizar las sombras. En este caso, el aspecto 'modulativo' de la técnica de sombreado afecta al color de la textura de sombra.

7.4 MÁSCARA DE LUZ ADITIVA

En enmascarado de luz aditiva hace el renderizado de la escena varias veces, cada vez representando la contribución de una única luz cuya influencia es suprimida en áreas de sombra. Cada pasada se combina (añade) a la previa de forma que cuando se han completado todas las pasadas, toda la contribución de luz está correctamente acumulada en la escena, y cada luz se ha prevenido de afectar áreas que no se deberían debido a los proyectores de sombra. Esto es una técnica efectiva que da como resultado un aspecto de iluminación muy realista, pero que tiene un precio: más pasadas de renderizado.

Como muchos documentos técnicos (y marketing de juegos) dicen, renderizar una iluminación realista requiere muchas pasadas. Como es amigo de los motores pequeños, Ogre libera de gran parte del trabajo duro, permitiendo utilizar exactamente las mismas definiciones de material se use o no esta

técnica de iluminación (para la mayor parte, ver Clasificación de pasadas y programas de vértice). Para realizar esta técnica, Ogre automáticamente categoriza las 3.1.2 Pasadas que se definen en los materiales en tres tipos:

- Pasadas ambiente categorizadas como 'ambient' que incluyen cualquier pasada base que no esté iluminada por una luz particular, es decir, ocurren incluso si no hay luz ambiente en la escena. La pasada ambiente siempre ocurre primero, y establece el valor inicial de la profundidad de los fragmentos, y el color ambiente si es aplicable. También incluye cualquier contribución de iluminación emisiva o de semi-iluminación. Sólo las texturas que afectan a la luz ambiente (por ejemplo mapas de oclusión ambiente) deben de renderizarse en esta pasada.

- Pasadas difusas/especulares categorizadas como 'diffuse/specular' (o 'per-light') se renderizan una vez por luz, y cada pasada contribuye al color difuso y especular desde el que una única luz se refleja por las condiciones en la pasada. Las áreas sombreadas por esta luz son enmascaradas y no se actualizan. El color enmascarado resultante se añade al color existente en la escena. De nuevo, no se utilizan texturas en esta pasada (excepto para texturas utilizadas en cálculos de luz como los mapas normales).

- Pasadas de pegatina categorizados como 'decal' añaden el color de textura final a la escena, que es modulada a partir de la acumulación de luz construida por todas las pasadas ambientes y difusas/especulares.

En la práctica, 3.2.1 Pasadas raramente caen bien justo en una de estas categorías. Para cada técnica, Ogre compila una lista de 'pasadas de iluminación', que derivan de las pasadas definidas por el usuario, pero que se pueden separar, para garantizar que las divisiones entre las categorías de pasadas de iluminación puedan mantenerse. Por ejemplo, si se utiliza una definición simple de material:

```
material TestIllumination
{
        technique
        {
                pass
                {
                        ambient 0.5 0.2 0.2
                        diffuse 1 0 0
                        specular 1 0.8 0.8 15
                        texture_unit
                        {
                                texture grass.png
                        }
                }
        }
}
```

Código 77: Definición simple de material, sin pasadas de iluminación.

Ogre lo separará en 3 pasadas de iluminación, que podrían ser equivalentes a esto:

```
material TestIlluminationSplitIllumination
{
        technique
        {
                // Ambient pass
                pass
                {
                        ambient 0.5 0.2 0.2
```

```
                        diffuse 0 0 0
                        specular 0 0 0
                }

                // Diffuse / specular pass
                pass
                {
                        scene_blend add
                        iteration once_per_light
                        diffuse 1 0 0
                        specular 1 0.8 0.8 15
                }

                // Decal pass
                pass
                {
                        scene_blend modulate
                        lighting off
                        texture_unit
                        {
                                texture grass.png
                        }
                }
        }
}
```

Código 78: Separación de pasadas de iluminación por Ogre.

Como se puede observar, incluso un material simple requiere un mínimo de 3 pasadas cuando se utiliza esta técnica de sombreado, y de hecho se requieren (num_luces + 2) pasadas en el caso general. Se pueden utilizar más pasadas en el material original y Ogre se las arreglará también, pero hay que tener en cuenta de que cada pasada se convertirá en varias si se utiliza más de un tipo de contribución de luz (ambiente contra difusa/especular) y/o tiene unidades de textura. Lo mejor es que se obtenga un comportamiento de iluminación multipasada completo incluso si no se definen los materiales en términos de ello, significando que las definiciones de material pueden seguir siendo las mismas sin importar el enfoque de luz que se decide utilizar.

Pasadas de iluminación categorizadas manualmente

Alternativamente, si se desea tener un control más directo sobre la categorización de las pasadas, se puede utilizar la opción illumination_stage en la pasada para asignar explícitamente una pasada sin cambios a un estado de iluminación. De esta forma se puede estar seguro de conocer con precisión cómo se renderizará el material bajo las condiciones de iluminación aditiva.

Clasificación de pasadas y programas de vértice

Ogre es muy bueno clasificando y separando las pasadas para garantizar el enfoque de renderizado multipasada requerido para que la iluminación aditiva funcione correctamente sin tener que cambiar la definición del material. Sin embargo, hay una excepción; cuando se utilizan programas de vértice, los atributos normales de luz ambiente, difusa, especular, etc no se utilizan, porque todos ellos se determinan por el programa de vértice. Ogre no tiene manera de conocer qué se está haciendo dentro de un programa de vértice, por lo que se le tiene que decir.

En la práctica esto es muy fácil. Incluso aunque el programa de vértice pudiera hacer un montón de procesado complejo y altamente personalizable, puede ser clasificable en uno de los tres tipos listados

arriba. Todo lo que se necesita decir a Ogre es lo que se está haciendo para usar los atributos de pasadas ambiente, difusa, especular y self_ilumination, igual que si no se estuviera utilizando un programa de vértice. Estos atributos no hacen nada (en relación con el renderizado) cuando se utilizan programas de vértice, pero es la forma más fácil de indicar a Ogre los componentes de luz que se están utilizando en el programa de vértice. Ogre clasificará y dividirá potencialmente la pasada programable basándose en esta información – dejando el programa de vértice como está (de forma que cualquier división de de pasada respetará cualquier modificación de vértice que se haga).

Hay que tener en cuenta que cuando se clasifica una pasada programable difusa/especular, Ogre chequea para comprobar si se ha indicado que la pasada puede ejecutarse una vez por luz (iteration once_per_light). Si es así, la pasada se mantiene intacta, incluyendo sus programas de vértice y fragmento. Sin embargo, si no se incluye este atributo en la pasada, Ogre intenta dividir la parte 'por luz', y al hacerlo desactivará el programa de fragmento, ya que en la ausencia del atributo 'iteration once_per_light' sólo puede asumir que el programa de fragmento está realizando un trabajo de pegatina y por lo tanto no debe de usarse por luz.

Así de claro, cuando se utilizan máscaras de luz aditiva como técnica de sombra, se necesita estar seguro de que las pasadas programables que se utilizan están propiamente establecidas de forma que pueden clasificarse correctamente. Sin embargo, también se tiene que tener en cuenta que los cambios que se tienen que realizar para asegurar que la clasificación sea correcta no afectan a la forma del renderizado de material cuando se elige no utilizar iluminación aditiva, de forma que el principio que se debe de seguir es usar las mismas definiciones de material para todos los escenarios de iluminación que aún se mantienen. Aquí hay un ejemplo de un material programable que será clasificado correctamente por el clasificador de pasadas de iluminación:

```
// Per-pixel normal mapping Any number of lights, diffuse and specular
material Examples/BumpMapping/MultiLightSpecular
{

        technique
        {
                // Base ambient pass
                pass
                {
                        // ambient only, not needed for rendering, but
as information
                        // to lighting pass categorisation routine
                        ambient 1 1 1
                        diffuse 0 0 0
                        specular 0 0 0 0
                        // Really basic vertex program
                        vertex_program_ref
Ogre/BasicVertexPrograms/AmbientOneTexture
                        {
                                param_named_auto worldViewProj
worldviewproj_matrix
                                param_named_auto ambient
ambient_light_colour
                        }

                }
                // Now do the lighting pass
                // NB we don't do decal texture here because this is
repeated per light
                pass
                {
```

```
                        // set ambient off, not needed for rendering,
but as information
                        // to lighting pass categorisation routine
                        ambient 0 0 0
                        // do this for each light
                        iteration once_per_light


                        scene_blend add

                        // Vertex program reference
                        vertex_program_ref Examples/BumpMapVPSpecular
                        {
                                param_named_auto lightPosition
light_position_object_space 0
                                param_named_auto eyePosition
camera_position_object_space
                                param_named_auto worldViewProj
worldviewproj_matrix
                        }

                        // Fragment program
                        fragment_program_ref Examples/BumpMapFPSpecular
                        {
                                param_named_auto lightDiffuse
light_diffuse_colour 0
                                param_named_auto lightSpecular
light_specular_colour 0
                        }

                        // Base bump map
                        texture_unit
                        {
                                texture NMBumpsOut.png
                                colour_op replace
                        }
                        // Normalisation cube map
                        texture_unit
                        {
                                cubic_texture nm.png combinedUVW
                                tex_coord_set 1
                                tex_address_mode clamp
                        }
                        // Normalisation cube map #2
                        texture_unit
                        {
                                cubic_texture nm.png combinedUVW
                                tex_coord_set 1
                                tex_address_mode clamp
                        }
                }

                // Decal pass
                pass
                {
                        lighting off
                        // Really basic vertex program
                        vertex_program_ref
Ogre/BasicVertexPrograms/AmbientOneTexture
                        {
```

```
                                param_named_auto worldViewProj
worldviewproj_matrix
                                param_named ambient float4 1 1 1 1
                        }
                        scene_blend dest_colour zero
                        texture_unit
                        {
                                texture RustedMetal.jpg
                        }

                }
        }
}
```

Código 79: Ejemplo de material programable que será correctamente clasificado por el clasificador de pasadas de iluminación.

Hay que tener en cuenta que si se están utilizando sombras de textura se tiene la opción adicional de usar sombras de texturas integradas en lugar de forzarlo a usar esta secuencia específica – permitiendo comprimir el número de pasadas en un número menor a costa de definir un número mayor de luces proyectoras de sombra. En este caso el aspecto 'aditivo' de la técnica de sombreado afecta al color de la textura de sombra y le toca al usuario combinar las texturas de sombra en los receptores como se quiera.

Iluminación estática

A pesar de su potencia, las técnicas de iluminación aditiva tienen una limitación adicional; no se combinan bien con la iluminación estática pre-calculada en la escena. Esto se debe a que están basadas en el principio de que la sombra es una ausencia de luz, pero como la iluminación estática en la escena incluye también áreas de luz y sombra, la iluminación aditiva no puede eliminar luz para crear nuevas sombras. Sin embargo, si se utilizan técnicas de iluminación aditiva, se debe o usar estas técnicas de forma exclusiva como la solución de iluminación (y se pueden combinar con iluminación por pixel para crea una solución de luz dinámica impresionante), o se debe de usar sombras de textura integradas para combinar con la iluminación estática de acuerdo con el enfoque elegido.

8. ANIMACIÓN

Ogre soporta un sistema de animación muy flexible que permite la animación para diferentes propósitos:

8.1 Animación esquelética
Animación de malla utilizando una estructura para determinar cómo se deforma la malla.

8.3 Animación de vértice
Animación de malla utilizando instantáneas de datos de vértice para determinar cómo cambia la figura de la malla.

8.4 Animación SceneNode
Animar automáticamente SceneNodes para crear efectos como los barridos de cámara, objetos siguiendo caminos predefinidos, etc.

8.5 Animación de valor numérico
Usando la estructura de clase extensible de Ogre para animar cualquier valor.

8.1 ANIMACIÓN ESQUELÉTICA

La animación esquelética es un proceso de animar una malla moviendo una serie de jerarquía de huesos dentro de la malla, que a su vez mueven los vértices del modelo según las asignaciones de huesos almacenados en cada vértice. Un término alternativo para este enfoque es el de 'pelaje'. La forma habitual para crear estas animaciones es una herramienta de modelado como Softimage XSI, Milkshape 3D, Blender, 3D Studio o Maya entre otros. Ogre proporciona exportadores para permitir obtener los datos de esos modeladores e introducirlo en el motor gráfico. Ver sección 4.1 Exportadores.

Hay muchos grados de animación esquelética, y no todos los motores (o modeladores de esta materia) soportan todos. Ogre admite las siguientes características:

- Cada malla puede enlazarse a un único esqueleto.
- Hay ilimitados huesos por esqueleto.
- Jerarquía con interés en la cinemática en los huesos.
- Múltiples animaciones nombradas por esqueleto (ejemplo 'Walk', 'Run', 'Jump', 'Shoot', etc).
- Fotogramas clave ilimitados por animación.
- Interpolación linear o basada en splines entre fotogramas clave.
- Se puede asignar un vértice a múltiples huesos y asignarle coeficientes de mezcla suave.
- Se pueden aplicar múltiples animaciones a una malla al mismo tiempo, de nuevo con coeficientes de mezclado.

Los esqueletos y las animaciones que van con ellos, se mantienen en ficheros .skeleton, que se producen con los exportadores de Ogre. Estos ficheros se cargan automáticamente cuando se crea una entidad (Entity) basada en la malla que está vinculada al esqueleto en cuestión. Después, se utiliza 8.2 Estado de la animación para establecer el uso de la animación en la entidad en cuestión.

La animación esquelética puede realizarse en software, o implementarse en shaders (hardware de pelaje). Claramente el último es preferible, ya que saca algo de trabajo de la CPU y se lo pasa a la tarjeta gráfica, y también significa que los datos de vértice no necesitarán ser recargados en cada fotograma. Esto es especialmente importante para modelos grandes y detallados. Se debe intentar usar recubrimiento de hardware siempre que sea posible; esto significa básicamente asignar un material que tiene un programa de vértice creado por una técnica. Ver Animación Esquelética en Programas de

Vértice para más detalles. La animación esquelética puede combinarse con animación de vértice, ver sección 8.3.3 Combinando animación esquelética y de vértice.

8.2 ESTADOS DE ANIMACIÓN

Cuando se crea una entidad que contiene una animación de cualquier tipo, se da un objeto 'estado de animación' por animación que permite especificar el estado de la animación de una única entidad (se pueden animar múltiples entidades usando las mismas definiciones de animación, Ogre ordena la reutilización internamente).

Se puede recuperar un puntero al objeto AnimationState llamado a Entity::getAnimationState. Se puede entonces llamar a métodos de este objeto retornado para actualizar la animación, probablemente en el evento de frameStarted. Cada AnimationState necesita estar habilitado a través del método setEnabled antes de que la animación tenga efecto, y se pueden establecer tanto el peso como la posición del tiempo (donde sea apropiado) para afectar a la aplicación en la animación usando métodos correlativos. AnimationState también tiene un método muy simple 'addTime' que permite alterar la posición de la animación incrementalmente, y hará un bucle automáticamente. addTime puede tener valores positivos o negativos (de forma que se puede invertir la animación si se desea).

8.3 ANIMACIÓN DE VÉRTICE

La animación de vértice trata de usar la información del movimiento de los vértices directamente para animar la malla. Cada pista en los objetivos de una animación de vértice es una única instancia a VertexData. La animación de vértice se almacena dentro de un fichero .mesh ya que está estrechamente vinculada a la estructura de vértice de la malla.

Realmente hay dos subtipos de animación de vértice, por razones que se discutirán en su momento.

8.3.1 Animación morfológica

La animación morfológica es una técnica muy simple que interpola instantáneas de malla por el tiempo en los fotograma clave. La animación morfológica tiene correlación directa con las técnicas de animación de carácter de la vieja escuela usados ampliamente antes de la animación esquelética.

8.3.2 Animación de postura

La animación de postura trata de mezclar posturas discretas múltiples, expresadas como offset los datos de vértice base, con pesos diferentes para dar un resultado final. Las animaciones de postura son las más obvias para animación facial.

¿POR QUÉ DOS SUBTIPOS?

¿Por qué hay dos subtipos de animación de vértices? ¿No se podrían haber implementado usando el mismo sistema? La respuesta correcta es si, de hecho se pueden implementar ambos tipos usando animación de postura. Pero por algunas buenas razones se decidió permitir especificar la animación morfológica separada ya que el sub-juego de características es fácil de definir y tiene menos requerimientos en los shaders hardware, si se implementa la animación a través de ellos. Si no hay preocupación sobre las razones de por qué se han implementado de forma diference, se puede omitir la siguiente parte.

La animación morfológica es un enfoque simple donde se tienen una serie de instantáneas de datos de vértice que deben de ser interpolados, como por ejemplo una animación corriendo implementada como

objetivos morfológicos. Como está basada en simples instantáneas, es mucho más rápido usarlo cuando se está animando una malla al completo porque es un único cambio linear entre los fotogramas. Sin embargo, este enfoque simplista no admite mezcla entre múltiples animaciones morfológicas. Si se necesita mezcla de animaciones, se aconseja usar animación esquelética para animaciones de malla completa, y animación de postura para animar subpartes de la malla o donde la animación esquelética no funcione correctamente – por ejemplo en la animación facial. Para animar en un shader de vértice, la animación morfológica es bastante más simple y sólo requiere 2 búferes de vértices (uno es el búfer de la posición original) de datos de posición absoluta, y un factor de interpolación. Cada pista en una animación morfológica referencia un único juego de datos de vértice.

Las animaciones de postura son más complejas. Como en las animaciones morfológicas, cada pista referencia a un único ser de datos de vértice, pero al contrario que las animaciones morfológicas, cada fotograma referencia 1 o más 'posturas', cada uno con un nivel de influencia. Una postura es una serie de offset de los datos de vértice base, y pueden escasear – es decir, pueden no referenciar todos los vértices. Como son offsets, pueden mezclarse –ambos dentro de una pista y entre animaciones. Este juego de características es idóneo para animación facial.

Por ejemplo, si hay una cara modelada (un juego de datos de vértice), y se define un juego de posturas que representan varias posiciones fonéticas de la cara. Se puede definir una animación llamada 'SayHello', conteniendo una única pista que referencia los datos de vértice de la cara, y que incluye una serie de fotogramas clave, que cada uno de ellos referenciados por uno o más posiciones de la cara como diferentes niveles de influencia –la combinación de estos en el tiempo hacen que la cara tome las formas necesarias para decir la palabra 'hola'. Como las posiciones se almacenan sólo una vez, pero pueden ser referenciadas muchas veces en muchas animaciones, esta es una forma muy potente de crear un sistema de habla.

El inconveniente de la animación de postura es que puede ser más difícil de configurar, ya que requiere que las poses se definan de forma separada y luego se referencien en los fotogramas. También, como utiliza más búferes (uno para los datos de base, y otro para cada posición activa), si se está animando en hardware usando shaders de vértice, se necesita mantener un ojo en cuantas posturas se están mezclando al mismo tiempo. Se define un número máximo de admitidas en la definición del programa de vértice, a través de la entrada en el script de material includes_pose_animation, Ver sección Animación de postura en programas de vértice.

Así, con la partición de los enfoques de animaciones de vértices en 2, se mantiene la simple técnica morfológica de fácil uso, mientras que se siguen permitiendo las técnicas más potentes. Hay que tener en cuenta que la animación morfológica no puede mezclarse con otros tipos de animación de vértice en los mismos datos de vértice (animación de postura u otra animación morfológica); la animación de postura puede mezclarse con otras animaciones de postura, y ambos tipos pueden combinarse con animación esquelética. Esta limitación a la combinación se aplica por juegos de datos de vértice, aunque no globalmente a través de la malla (ver abajo). También hay que tener en cuenta que todas las animaciones morfológicas pueden expresarse (de un modo más complejo) como animación de postura, pero no al contrario.

SUBTIPOS APLICADOS POR PISTA

Es importante tener en cuenta que el subtipo en cuestión se mantiene a un nivel de la pista, y no a nivel de animación o de malla. Como el mapa de pistas está en instancias VertexData, esto significa que si la malla se divide en SubMeshes, cada una de ellas con su geometría dedicada, se puede tener una SubMesh animada utilizando animación de postura, y otras animadas con animación morfológica (o ninguna animación de vértice).

Por ejemplo, una configuración común para un carácter complejo que necesita animación esquelética y animación facial; puede dividirse la cabeza en una SubMesh con su propia geometría, entonces aplicar animación esquelética a las dos sub-mallas, y animación de postura a la cabeza.

Para ver cómo aplicar la animación de vértice, ver la sección 8.2 Estados de animación.

COLOCACIÓN DE BÚFER DE VÉRTICES

Cuando se utiliza animación de vértice en software, los búferes de vértice necesitan estar colocados como residen las posiciones de vértices en su propio búfer hardware. Esto sirve para evitar tener que cargar todos los datos de vértice cuando se actualiza, que podría saturar rápidamente el bus de la GPU. Cuando se utiliza el formato .mesh de Ogre y las herramientas/exportadores que van con él, Ogre organiza esto de forma automática. Pero si se crean los búferes de forma manual, hay que ser consciente de los acuerdos de distribución.

Para hacer esto, hay unas funciones de ayuda en Ogre::Mesh. Ver las entradas en el API para Ogre::VertexData::reorganiseBuffers() y Ogre::VertexDeclaration::getAutoOrganisedDeclaration(). El último cambia una declaración de vértice en una que es recomendada para el uso que se ha indicado, y el primero reorganizará el contenido de un set de búferes para conformar este acuerdo.

8.3.1 ANIMACIÓN MORFOLÓGICA

La animación morfológica trabaja guardando instantáneas de las posiciones de vértice absolutas en cada fotograma, e interpolando entre ellos. Las animaciones morfológicas son principalmente útiles para animar objetos que no pueden manejarse adecuadamente usando animación esquelética; esta es la mayoría de objetos que tienen que cambiar su estructura y forma radicalmente de tal forma que una estructura esquelética no es apropiada.

Como se utilizan posiciones absolutas, no es posible mezclar más de una animación morfológica en los mismos datos de vértice; se debería utilizar animación esquelética si se quiere utilizar mezcla de animaciones ya que es mucho más eficiente. Si se activa más de una animación que incluye pistas morfológicas para los mismos datos de vértice, sólo tendrá efecto el último. Esto también significa que la opción 'weight' (peso) en el estado de la animación no se usa en animaciones morfológicas.

Las animaciones morfológicas pueden combinarse con animación esquelética si se requiere. Ver sección 8.3.3 Combinando animaciones esqueléticas y de vértice. Las animaciones morfológicas también pueden implementarse en hardware usando shaders vectoriales, ver sección Animación morfológica en programas de vértice.

8.3.2 ANIMACIÓN DE POSTURA

Las animaciones de postura permiten mezclar potencialmente posiciones de vértice múltiples en niveles de influencia diferentes en un estado de vértice final. Un uso típico de esta animación es para la animación facial, donde se pone cada expresión facial en una animación separada, y las influencias son utilizadas para mezclar cada expresión con otra, o para combinar expresiones completas si cada pose sólo afecta a partes de la cara.

Para hacer esto, la animación de postura utiliza un juego de poses de referencia definidas en la malla, expresadas como offset de los datos de vértice originales. No requiere que cada vértice tenga un offset - aquellos que no se dejan solos. Cuando se mezcla en software, estos vértices son totalmente omitidos -

cuando se mezclan en hardware (que requiere una entrada de vértice por cada vértice), los offset de cero para vértices que no se mencionan se crean automáticamente.

Una vez que se definen las posturas, se pueden referenciar en animaciones. Cada pista de animación de postura hace referencia a un único juego de geometría (ya sea una forma geométrica de la malla, o una geometría dedicada en una sub-malla), y cada fotograma en la pista puede hacer referencia a una o más posturas, cada una con su propio nivel de influencia. El peso aplicado a la animación al completo escala también los niveles de influencia. Se pueden definir tantos fotogramas que causen la mezcla de posturas para cambiar en el tiempo como se desee. La ausencia de referencias de posturas en un fotograma clave cuando está presente en un vecino provoca el ser tratado como una influencia de 0 para la interpolación.

Hay que tener cuidado en cuantas posturas se aplican a la vez. Cuando se realizan animación de postura en hardware (ver animación de postura en programas de vértices), todas las posturas activas requieren otro búfer de vértices para ser añadido al shader, y cuando se anima en software también cogerá las posturas más activas que haya. Hay que tener en cuenta que si se tienen 2 posturas en un fotograma, y 2 diferentes en la siguiente, que realmente significa que hay 4 fotogramas activos cuando están interpolando entre ellos.

Se pueden combinar animaciones de postura con animaciones esqueléticas, ver sección 8.3.3 Combinando animaciones esqueléticas y de vértices, y se puede acelerar por hardware la aplicación de la mezcla con un shader de vértice, ver sección Animación de postura en programas de vértice.

8.3.3 COMBINANDO ANIMACIONES ESQUELÉTICAS Y DE VÉRTICE

La animación esquelética y la animación de vértice (cualquier subtipo) pueden estar habilitadas en la misma entidad al mismo tiempo (ver 8.2 Estados de animación). El efecto de esto es que la animación de vértice se aplica primero a la malla base, luego se aplica la animación esquelética al resultado. Esto permite, por ejemplo, animar facialmente un personaje usando animación de vértices de postura, mientras se realiza el movimiento principal usando animación esquelética.

Combinar los dos es, desde la perspectiva del usuario, tan simple como habilitar las dos animaciones al mismo tiempo. Para usar estar característica eficientemente hay algunos puntos a tener en cuenta:

- Pelado de hardware combinado
- División de sub-mallas

PELADO DE HARDWARE COMBINADO

Para personajes complejos es una buena idea implementar el pelado hardware (hardware skinning) incluyendo una técnica en los materiales que tiene un programa de vértices que puede realizar los tipos de animación que se realizan en hardware. Ver Animación esquelética en programas de vértices, Animación morfológica en programas de vértices, Animación de postura en programas de vértices.

Cuando se combinan tipos de animación, los programas de vértice deben de soportar ambos tipos de animación que las mallas combinadas necesitan, por otra parte, hardware skinning debe deshabilitarse. Se debe implementar la animación de la misma forma que Ogre lo hace, es decir, realizando primero la animación de vértice, y luego aplicar la animación esquelética al resultado de esta. Hay que recordar que la implementación de la animación morfológica pasa 2 búferes de instantáneas absolutas para y desde los fotogramas, a través de un único paramétrico, que hay que interpolar linearmente, mientras

que la animación de postura pasa los datos de vértices base más 'n' búferes de offset de postura, y 'n' valores de peso paramétricos.

SEPARACIÓN DE SUB-MALLAS

Si sólo se necesita combinar animaciones de vértice y esquelética para una pequeña parte de la malla, por ejemplo, la cara, se puede dividir la malla en 2 partes, una que necesita combinación y otra que no, para reducir la sobrecarga de cálculos. Hay que tener en cuenta que también se reducirá la utilización de búfer de vértices ya que los búferes de postura/fotograma de vértices serán también más pequeños. Hay que tener en cuenta que si se utiliza pelado hardware, se deberá entonces implementar 2 programas de vértice por separado, uno que sólo hace la animación esquelética, y otro que hace las animaciones esquelética y de vértice.

8.4 ANIMACIÓN SCENENODE

La animación del SceneNode se crea desde el SceneManager para animar el movimiento de los ScenesNodes, para hacer que cualquier objeto adjunto se mueva automáticamente. Esto se puede ver realizando un abatimiento de cámara en Demo_CameraTrack, o controlando cómo se mueven los peces en el estanque en Demo_Fresnel.

La animación de nodos de escena es casi el mismo código con el que se anima el esqueleto subyacente en animación esquelética. Después de crear la Animation principal usando SceneManager::createAnimation, se puede crear un NodeAnimationTrack por SceneNode que se quiera animar, y crear fotogramas claves que controlen su posición, orientación y escala que pueden interpolarse linealmente o a través de splines. Se usa 8.2 Estado de Animación de la misma forma que se hace para animaciones esqueléticas/de vértices, excepto que se obtiene el estado desde el SceneManager en lugar de una Entity individual. Las animaciones se aplican automáticamente en cada fotograma, o estado, pueden aplicarse manualmente en un uso avanzado utilizando el método _applySceneAnimations() en SceneManager. Ver la referencia API para los detalles completos del interfaz para configurar las animaciones de escena.

8.5 ANIMACIÓN DE VALOR NUMÉRICO

A parte de los tipos específicos de animación de los que pueden estar compuestos los usos más comunes de un framework de animación, se pueden utilizar animaciones para alterar cualquier valor que se expone a través del interfaz AnimableObject.

ANIMABLEOBJECT

El AnimableObject es un interfaz del que puede extender cualquier clase para permitir el acceso a un número de AnimableValues. Tiene un 'diccionario' de las propiedades de animación disponibles que pueden enumerarse a través del método getAnimableValueNames, y cuando se llama a su método createAnimableValue, retorna una referencia a un objeto valor que forma un puente entre los interfaces de animación genéricos, y la propiedad del objeto específico subyacente.

Un ejemplo de esto es la clase Light (luz). Extiende de AnimableObject y proporciona AnimableValues para propiedades como "diffuseColour" y "attenuatio". Las pistas de la animación pueden crearse para estos valores y también propiedades de la luz se pueden hacer en script para cambiarlo. Otros objetos, incluyendo los objetos personalizados, pueden extender de este interfaz de la misma forma que se proporciona soporte para la animación a sus propiedades.

ANIMABLEVALUE

Cuando se implementan propiedades para animar personalizadas, se tiene también que implementar un número de métodos en el interfaz AnimableValue – básicamente cualquier cosa que se haya marcado como no-implementable. No hay métodos virtuales puros simplemente porque sólo hay que implementar los métodos requeridos para el tipo de valor que se está animando. De nuevo, ver los ejemplos en la clase Light para ver cómo se hace.