OgreSprites A More In-Depth Version of the 2D Sprite Manager

* Add screenshot (maybe in forum?)
OgreSprites!

I've written this code, based primarily off the work done by H. Hernán Moraldo (Moraldo Games). I've adapted it for my use in a game I'm working on, and I thought it might be useful to someone else.

This code is obviously more complicated than the the SpriteManager2d class, and is intended to extend the original functionality should you desire it. If you're just after simplicity, the original class is much better. I tend to overdesign things ~_^

Because of the way this class draws (in a non-persistent fashion), you will likely not create the same amazing frame rates full 3d Ogre applications are used to. In other words, this is quite likely going to be relatively slow. If you really want to get good performance, implementing some sort of "dirty rectangle" system (or some such system) would greatly improve speed. Hopefully it runs fast enough to do what you're hoping for.

Lastly, I'm a novice, hobbyist coder. Please excuse any terrible violations of conventions, coding practices, unsightly gnomes, etc. Please let me know if I've done something terrible! I'm very open to constructive criticism. Sarcasm can be deposited in the bin at the back. Thanks! 🙂

(Very lastly, thanks to absolutely everyone involved with Ogre in any way. You've not only created an awesome graphics engine, you've created a welcoming and helpful community!)

Andrew C Lytle (alytle@___gmail.com)

See also

- -Billboard
- Ogre Magic website
- forum thread

OgreSprites.h:

```
/*
    OgreSprites

    OgreSprites is made available under the MIT License.

    Permission is hereby granted, free of charge, to any person obtaining a copy
    of this software and associated documentation files (the "Software"), to deal
    in the Software without restriction, including without limitation the rights
    to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
    copies of the Software, and to permit persons to whom the Software is
    furnished to do so, subject to the following conditions:

    The above copyright notice and this permission notice shall be included in
    all copies or substantial portions of the Software.

    THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
    IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
    FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
    AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
    LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
    OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
    THE SOFTWARE.

    Written by Andrew C Lytle, June 2007.

    Developed based on code by H. Hernán Moraldo from Moraldo Games
    www.hernan.moraldo.com.ar/pmenglish/field.php
*/

#ifndef __OGRE_SPRITES_H__
#define __OGRE_SPRITES_H__

#include <Ogre.h>
#include <OgreRenderQueueListener.h>
#include <string>
#include <list>

namespace OgreSprites {

    /** Holds information about a single sprite each frame.
    @remarks
        Used internally.
    */
    struct SpriteElement
    {
        float x1, y1, x2, y2;// sprite coordinates
        float tx1, ty1, tx2, ty2;// texture coordinates
        Ogre::ResourceHandle texHandle;// texture handle

        float alpha;
    };

    /** Holds vertex information.
    @remarks
        Used internally.
    */
```

```
struct VertexChunk {
    Ogre::ResourceHandle texHandle;
    unsigned int vertexCount;

    float alpha;
};

/** Available metrics.
@remarks
    See SpriteManager notes for details.
*/
enum OSMetrics {
    OSPRITE_METRIC_PIXELS,
    OSPRITE_METRIC_RELATIVE,
    OSPRITE_METRIC_OGRE
};

/** Rectangle class to represent either screen or texture space.
*/
struct Rect
{
    /// Default contstructor
    Rect()
    {
        x1 = y1 = x2 = y2 = 0.0f;
    }

    /// Copy constructor
    Rect(Rect &r)
    {
        x1 = r.x1;
        y1 = r.y1;
        x2 = r.x2;
        y2 = r.y2;
    }

    /// Parameter constructor
    Rect(Ogre::Real px1, Ogre::Real py1, Ogre::Real px2, Ogre::Real py2)
    {
        x1 = px1;
        y1 = py1;
        x2 = px2;
        y2 = py2;
    }

    /// Equality operator
    bool operator==(Rect &r) {
        if( (x1==r.x1) && (x2==r.x2) && (y1==r.y1) && (y2==r.y2) )
            return true;
        else
            return false;
    }

    /// Inequality operator
    bool operator!=(Rect &r) {
        return !(operator==(r));
    }
```

```
        /// Left coordinate
        Ogre::Real x1;

        /// Top coordinate
        Ogre::Real y1;

        /// Right coordinate
        Ogre::Real x2;

        /// Bottom coordinate
        Ogre::Real y2;   // bottom
    };

    /// Macro typedef of a Rect representing the entire texture space of a sprite.
    extern OgreSprites::Rect FULL_SPRITE;

    /// Macro typedef of a Rect representing the entire screen space.
    extern OgreSprites::Rect FULL_SCREEN;

    /** Controls all sprite rendering operations.
    @remarks
        Usage:

        SETUP:

            1) Instantiate the OgreSprites::SpriteHandler object as usual with a standard ne
w,
            and leave it in a place that your program will be able to access the pointer late
r.

                OgreSprites::SpriteHandler* spriteHandler = new OgreSprites::SpriteHandler();

            2) Give the SpriteHandler a resource path, this *must* be done or you won't be ab
le to load
              any sprites. Naturally, change the path to match where you save your sprites.

                spriteHandler->SetSpriteLocation("../sprites");

            3) Load your sprites, manually. Notice we aren't using the standard Ogre::Materia
l concept,
              we're just loading regular old textures.
                spriteHandler->LoadSprite("sprite.png");
                spriteHandler->LoadSprite("sprite2.png");
                spriteHandler->LoadSprite("folder/sprite3.png");

            4) Initialize the library, giving it your current Ogre scene manager and viewpor
t.

                spriteHandler->Init(mSceneMgr, mWindow->getViewport(0));

        METRICS:

        - Decide on your metrics. The SpriteManager can interpret both screen values and spri
te locations
          in three different ways. If I've  made this unreasonably complicated, just stick to
the defaults
```

```
          and you'll be fine.

        - OSPRITE_METRIC_RELATIVE: This is the default screen metric. It considers the screen location from
          (0.0, 0.0) -> (1.0, 1.0) going from top left to bottom right of the screen. For sprites, the same
          locations represent the top left to bottom right of the sprite texture. Using this metric for your
          screen values ensures that your sprites will always been drawn in the same relative screen
          position, regardless of the size (or resolution) of your viewport. Using it for your sprites is not
          recommended, since your sprite data is usually static, it is often better to use
          OSPRITE_METRIC_PIXEL instead for sprites.

        - OSPRITE_METRIC_OGRE: This metric is the default for Ogre's internal drawing system, and although
          similar to OSPRITE_METRIC_RELATIVE, it goes from (-1, 1) to (1, -1) from top left to bottom right.
          You can not use this metric for sprites.

        - OSPRITE_METRIC_PIXELS: This is the default sprite metric. This metric works in actual pixel
          locations. If you use this for your screen metric, you will always be drawing in the same pixel
          locations, regardless of screen resolution. Since this is often undesireable, it is recommended you
          use OSPRITE_METRIC_RELATIVE for screen metrics.

        DRAWING:

        - You can draw your sprites using whichever of the various DrawSprite methods suits your needs.
          I've forced myself to contain my code bloat and keep them to only three. There's lots of room
          to expand on them if you want specialized drawing concepts.

        - Method 1: DrawSprite using fixed location and the entire sprite.

            DrawSprite("spriteName", xLocation, yLocation, alphaValue);

        - Method 2: DrawSprite using fixed destination size and partial sprite.

            DrawSprite("spriteName", xLocation, yLocation, OgreSprites::Rect(spriteX1, spriteY1, spriteX2, spriteY2), alphaValue);

        - Method 2: DrawSprite using arbitrary destination size and partial sprite.

            DrawSprite("spriteName", OgreSprites::Rect(destX1, destY1, destX2, destY2), OgreSprites::Rect(spriteX1, spriteY1, spriteX2, spriteY2), alphaValue);

        SHUTDOWN:

        - Call spriteHandler->Shutdown() during program shutdown, and delete the pointer normally.
          That's it!
    */
    class SpriteHandler : public Ogre::RenderQueueListener
    {

    public:
        /// Default constructor
        SpriteHandler();

        /// Destructor
        virtual ~SpriteHandler();

        /// Used internally by Ogre
        virtual void renderQueueStarted(Ogre::uint8 queueGroupId, const Ogre::String &invocat
ion, bool &skipThisInvocation);

        /// Used internally by Ogre
        virtual void renderQueueEnded(Ogre::uint8 queueGroupId, const Ogre::String &invocatio
n, bool &repeatThisInvocation);

        /** Initialize the sprite system, and register it with Ogre.
        @remarks
            This method setups up the sprite system, and must be called only after the Ogre
            Scene Manager has been created.
            @param
                sceneMan A pointer to the current scene manager.
            @param
                viewPort The Ogre viewport we should be rendering to.
            @param
                targetQueue The render queue that we are inserting this render operation int
o.
            @param
                afterQueue Should we render after this Render Queue? If not, we'll do it befo
re.
        */
        void Init(Ogre::SceneManager* sceneMan, Ogre::Viewport* viewPort, Ogre::uint8 targetQ
ueue = Ogre::RENDER_QUEUE_OVERLAY, bool afterQueue = true);

        /** Shutdown the sprite system.
        @remarks
            This will be done automatically by deleting the object, but can be done manually
            should you desire it.
        */
        void Shutdown(void);

        /** Control screen metrics.
        @remarks
            The three available metrics for the screen are OSPRITE_METRIC_RELATIVE, OSPRITE_M
ETRIC_PIXELS,
            and OSPRITE_METRIC_OGRE. For the screen, RELATIVE means top-left origin, with bot
tom right as
            (1,1). OGRE means (-1,1) to (1,-1).  And PIXELS means (0,0) to (screenWidth, scre
enHeight).
            The default value is OSPRITE_METRIC_RELATIVE.
            @param
                metric Which metric to use
        */
        void SetScreenMetric(OSMetrics metric);
        /** Control sprite metrics.
        @remarks
            The available metrics for the screen are OSPRITE_METRIC_RELATIVE, and OSPRITE_MET
RIC_PIXELS.
            The function similar to screen metrics, with (0,0) always as the top-left of the
texture.
            The default value is OSPRITE_METRIC_PIXELS.
            @param
                metric Which metric to use
        */
        void SetSpriteMetric(OSMetrics metric);

        /** Set the folder location of the sprite data.
        @remarks
            This MUST be done before any sprites can be loaded. Failure to do so will cause
            LoadSprite to fail with an exception.
            @param
                pathName The relative or absolute path to the sprite texture files.
        */
        void SetSpriteLocation(const std::string& pathName);

        /** Load a sprite into memory.
        @remarks
            This method will load a sprite from a texture file. You must do this with
            each sprite you intend to draw later. This texture file must be found in the
            path given by SetSpriteLocation previously.
            @param
                pathName The texture file name (with extension)
        */
        void LoadSprite(const std::string& spriteName);

        /** Render a sprite.
        @remarks
            This method will draw a sprite at a given location, with a given alpha. The entir
e
            texture will be used to render the sprite.
            @param
                spriteName The file name that was loaded with LoadSprite (with extension).
            @param
                x The x coordinate to begin drawing the sprite at (relative or pixel, based o
n metrics)
            @param
                y The y coordinate to begin drawing the sprite at (relative or pixel, based o
n metrics)
            @param
                alpha The alpha value to used when drawing. 0.0 is totally transparent, 1.0 i
s completely solid.
        */
        void DrawSprite(const std::string& spriteName, float x, float y, float alpha);

        /** Render a sprite.
        @remarks
            This method will draw a portion of the sprite at a given location, with a given a
lpha.
            The area specified in spriteRect will be be used to render the sprite.
            @param
                spriteName The file name that was loaded with LoadSprite (with extension).
            @param
                x The x coordinate to begin drawing the sprite at (relative or pixel, based o
n metrics)
            @param
                y The y coordinate to begin drawing the sprite at (relative or pixel, based o
n metrics)
            @param
                spriteRect A rectangle representing a 2D location in the texture (relative or
pixel, based on metrics)
            @param
                alpha The alpha value to used when drawing. 0.0 is totally transparent, 1.0 i
s completely solid.
        */
        void DrawSprite(const std::string& spriteName, float x, float y, OgreSprites::Rect& s
priteRect, float alpha);

        /** Render a sprite.
        @remarks
            This method will draw a portion of the sprite at a given location and given endin
g point,
            with a given alpha. The area specified in spriteRect will be be used to render th
e sprite.
            The area specified in destRect will be used to determine location and size of the
final
            drawing operation.
            @param
                spriteName The file name that was loaded with LoadSprite (with extension).
            @param
                x The x coordinate to begin drawing the sprite at (relative or pixel, based o
n metrics)
            @param
                y The y coordinate to begin drawing the sprite at (relative or pixel, based o
n metrics)
            @param
                spriteRect A rectangle representing a 2D location in the texture (relative or
pixel, based on metrics)
            @param
                spriteRect A rectangle representing a 2D location in the texture (relative or
pixel, based on metrics)
            @param
                alpha The alpha value to used when drawing. 0.0 is totally transparent, 1.0 i
s completely solid.
        */
        void DrawSprite(const std::string& spriteName, OgreSprites::Rect& destRect, OgreSprit
es::Rect& spriteRect, float alpha);

    private:

        /// Render all the 2d data stored in the hardware buffers.
        void renderBuffer();

        /// Create a new hardware buffer
        void createHardwareBuffer(unsigned int size);

        /// Destroy the hardware buffer
        void destroyHardwareBuffer();
        /// Set Ogre for rendering
        void prepareForRender();

        /// Convert metrics
        void convertScreenMetrics(OSMetrics metricFrom, const float sx, const float sy, OSMet
rics metricTo, float& dx, float& dy);

        /// Ogre Specific: render operation handler
        Ogre::RenderOperation renderOp;

        /// Ogre Specific: hardware buffer
        Ogre::HardwareVertexBufferSharedPtr hardwareBuffer;

        /// Sprite Buffer
        std::list<SpriteElement> sprites;

        /// Save our sprite texture path
        std::string    spriteLocation;

        /// Scene manager reference pointer
        Ogre::SceneManager* sceneMan;

        /// Which queue we're rendering on
        Ogre::uint8 targetQueue;

        /// Render after or before this queue
        bool afterQueue;

        /// Viewport width
        int _vpWidth;

        /// Viewport height
        int _vpHalfWidth;

        /// Half viewport width, save time calculating later
        int _vpHeight;

        /// Half viewport height, save time calculating later
        int _vpHalfHeight;

        /// Current screen metrics
        OSMetrics _metricScreen;

        /// Current sprite metrics
        OSMetrics _metricSprite;

    };

}

#endif // __OGRE_SPRITES_H__
```

OgreSprites.cpp:

```
/*
    OgreSprites

    OgreSprites is made available under the MIT License.

    Permission is hereby granted, free of charge, to any person obtaining a copy
    of this software and associated documentation files (the "Software"), to deal
    in the Software without restriction, including without limitation the rights
    to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
    copies of the Software, and to permit persons to whom the Software is
    furnished to do so, subject to the following conditions:

    The above copyright notice and this permission notice shall be included in
    all copies or substantial portions of the Software.

    THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
    IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
    FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
    AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
    LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
    OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
    THE SOFTWARE.

    Written by Andrew C Lytle, June 2007.

    Developed based on code by H. Hernán Moraldo from Moraldo Games
    www.hernan.moraldo.com.ar/pmenglish/field.php
*/

#include "OgreSprites.h"
#include <Ogre.h>
#include <OgreMesh.h>
#include <OgreHardwareBuffer.h>

#define OGRE2D_MINIMAL_HARDWARE_BUFFER_SIZE 120

namespace OgreSprites {

    OgreSprites::Rect FULL_SPRITE(-1000.0f, -1000.0f, -1000.0f, -1000.0f);
    OgreSprites::Rect FULL_SCREEN(-1000.0f, -1000.0f, -1000.0f, -1000.0f);

    //----------------------------------------------------------------------
    SpriteHandler::SpriteHandler() :
        _metricScreen(OSPRITE_METRIC_RELATIVE),
        _metricSprite(OSPRITE_METRIC_PIXELS)
    {
    }
    //----------------------------------------------------------------------
    SpriteHandler::~SpriteHandler()
    {
    }
    //----------------------------------------------------------------------
    void SpriteHandler::renderQueueStarted(Ogre::uint8 queueGroupId, const Ogre::String &invo
cation, bool &skipThisInvocation)
    {
       if (!afterQueue && queueGroupId==targetQueue)
```

```
          renderBuffer();
    }
    //-----------------------------------------------------------------------
    void SpriteHandler::renderQueueEnded(Ogre::uint8 queueGroupId, const Ogre::String &invocation, bool &repeatThisInvocation)
    {
       if (afterQueue && queueGroupId==targetQueue)
          renderBuffer();
    }
    //-----------------------------------------------------------------------
    void SpriteHandler::Init(Ogre::SceneManager* sceneMan, Ogre::Viewport* viewPort, Ogre::uint8 targetQueue, bool afterQueue)
    {
        // Save scene manager data
        SpriteHandler::sceneMan = sceneMan;
        SpriteHandler::afterQueue = afterQueue;
        SpriteHandler::targetQueue = targetQueue;

        // Ensure our hardware buffer is set to zero
        hardwareBuffer.setNull();

        // Set this object as a render queue listener with Ogre
        sceneMan->addRenderQueueListener(this);

        // Gather viewport info
        _vpWidth = viewPort->getActualWidth();
        _vpHeight = viewPort->getActualHeight();
        _vpHalfWidth = (int)(_vpWidth / 2);
        _vpHalfHeight = (int)(_vpHeight / 2);
    }
    //-----------------------------------------------------------------------
    void SpriteHandler::Shutdown()
    {
        // Destroy the hardware buffer
        if (!hardwareBuffer.isNull())
            destroyHardwareBuffer();

        // Delist our renderqueuelistener
        sceneMan->removeRenderQueueListener(this);
    }
    //-----------------------------------------------------------------------
    void SpriteHandler::SetScreenMetric(OSMetrics metric)
    {
        _metricScreen = metric;
    }
    //-----------------------------------------------------------------------
    void SpriteHandler::SetSpriteMetric(OSMetrics metric)
    {
        if(metric == OSPRITE_METRIC_RELATIVE || metric == OSPRITE_METRIC_PIXELS)
            _metricSprite = metric;
        else
            _metricSprite = OSPRITE_METRIC_RELATIVE;
    }
    //-----------------------------------------------------------------------
    void SpriteHandler::SetSpriteLocation(const std::string& pathName)
    {
        spriteLocation = pathName;
        Ogre::ResourceGroupManager::getSingleton().addResourceLocation(pathName, "FileSystem", "OgreSprites");
        Ogre::ResourceGroupManager::getSingleton().initialiseResourceGroup("OgreSprites");
    }
    //-----------------------------------------------------------------------
    void SpriteHandler::LoadSprite(const std::string& spriteName)
    {
        Ogre::TextureManager::getSingleton().load(spriteName, "OgreSprites");
    }
    //-----------------------------------------------------------------------
    void SpriteHandler::DrawSprite(const std::string& spriteName, float x, float y, float alpha)
    {
        // Retrieve pointer to texture resource
        Ogre::TexturePtr texturePtr = Ogre::TextureManager::getSingleton().getByName(spriteName);

        // This is the size of the original image data (pixels)
        int iSpriteWidth = (int)texturePtr->getWidth();
        int iSpriteHeight = (int)texturePtr->getHeight();

        // Get texture handle from texture resource
        SpriteElement spriteElement;
        spriteElement.texHandle = texturePtr->getHandle();

        // Convert destination start to Pixels
        float fPixelStartX = 0;
        float fPixelStartY = 0;
        convertScreenMetrics(_metricScreen, x, y, OSPRITE_METRIC_PIXELS, fPixelStartX, fPixelStartY);
        int iPixelEndX = (int)fPixelStartX + (int)iSpriteWidth;
        int iPixelEndY = (int)fPixelStartY + (int)iSpriteHeight;

        // Convert from pixels to Ogre
        convertScreenMetrics(OSPRITE_METRIC_PIXELS, fPixelStartX, fPixelStartY, OSPRITE_METRIC_OGRE, spriteElement.x1, spriteElement.y1);
        convertScreenMetrics(OSPRITE_METRIC_PIXELS, iPixelEndX, iPixelEndY, OSPRITE_METRIC_OGRE, spriteElement.x2, spriteElement.y2);

        // We want to draw the entire sprite
        spriteElement.tx1 = 0.0f;
        spriteElement.ty1 = 0.0f;
        spriteElement.tx2 = 1.0f;
        spriteElement.ty2 = 1.0f;

        // save alpha value
        spriteElement.alpha = alpha;

        // Add this sprite to our render list
        sprites.push_back(spriteElement);
    }
    //-----------------------------------------------------------------------
    void SpriteHandler::DrawSprite(const std::string& spriteName, float x, float y, OgreSprites::Rect& spriteRect, float alpha)
    {
        // Retrieve pointer to texture resource
        Ogre::TexturePtr texturePtr = Ogre::TextureManager::getSingleton().getByName(spriteNa
me);

        // This is the size of the original image data (pixels)
        int iSpriteWidth = (int)texturePtr->getWidth();
        int iSpriteHeight = (int)texturePtr->getHeight();

        // Get texture handle from texture resource
        SpriteElement spriteElement;
        spriteElement.texHandle = texturePtr->getHandle();

        // Drawing size
        int iDrawingWidth = (spriteRect.x2 - spriteRect.x1);
        int iDrawingHeight = (spriteRect.y2 - spriteRect.y1);

        // Convert destination start to Pixels
        float fPixelStartX = 0;
        float fPixelStartY = 0;
        convertScreenMetrics(_metricScreen, x, y, OSPRITE_METRIC_PIXELS, fPixelStartX, fPixel
StartY);
        int iPixelEndX = (int)fPixelStartX + (int)iDrawingWidth;
        int iPixelEndY = (int)fPixelStartY + (int)iDrawingHeight;

        // Convert from pixels to Ogre
        convertScreenMetrics(OSPRITE_METRIC_PIXELS, fPixelStartX, fPixelStartY, OSPRITE_METRI
C_OGRE, spriteElement.x1, spriteElement.y1);
        convertScreenMetrics(OSPRITE_METRIC_PIXELS, iPixelEndX, iPixelEndY, OSPRITE_METRIC_OG
RE, spriteElement.x2, spriteElement.y2);

        // We want to draw only a portion of the sprite
        spriteElement.tx1 = spriteElement.ty1 = 0.0f;
        spriteElement.tx2 = spriteElement.ty2 = 1.0f;

        if(spriteRect != FULL_SPRITE) {
            if(_metricSprite == OSPRITE_METRIC_RELATIVE) {
                spriteElement.tx1 = spriteRect.x1;
                spriteElement.ty1 = spriteRect.y1;
                spriteElement.tx2 = spriteRect.x2;
                spriteElement.ty2 = spriteRect.y2;
            }
            else if(_metricSprite == OSPRITE_METRIC_PIXELS) {

                spriteElement.tx1 = (float)(spriteRect.x1 / iSpriteWidth);
                spriteElement.ty1 = (float)(spriteRect.y1 / iSpriteHeight);
                spriteElement.tx2 = (float)(spriteRect.x2 / iSpriteWidth);
                spriteElement.ty2 = (float)(spriteRect.y2 / iSpriteHeight);
            }
        }

        // save alpha value
        spriteElement.alpha = alpha;

        // Add this sprite to our render list
        sprites.push_back(spriteElement);
    }
    //-----------------------------------------------------------------------
    void SpriteHandler::DrawSprite(const std::string& spriteName, OgreSprites::Rect& destRec
t, OgreSprites::Rect& spriteRect, float alpha)
    {
        // Retrieve pointer to texture resource
        Ogre::TexturePtr texturePtr = Ogre::TextureManager::getSingleton().getByName(spriteNa
me);

        // Get texture handle from texture resource
        SpriteElement spriteElement;
        spriteElement.texHandle = texturePtr->getHandle();

        // This is the size of the original image data (pixels)
        int iSpriteWidth = (int)texturePtr->getWidth();
        int iSpriteHeight = (int)texturePtr->getHeight();

        if(destRect != FULL_SCREEN) {
            // Convert destination start to Pixels
            float fPixelStartX = 0;
            float fPixelStartY = 0;
            convertScreenMetrics(_metricScreen, destRect.x1, destRect.y1, OSPRITE_METRIC_PIXE
LS, fPixelStartX, fPixelStartY);

            // Convert size to pixels
            float fPixelEndX = 0;
            float fPixelEndY = 0;
            convertScreenMetrics(_metricScreen, destRect.x2, destRect.y2, OSPRITE_METRIC_PIXE
LS, fPixelEndX, fPixelEndY);

            // Convert from pixels to Ogre
            convertScreenMetrics(OSPRITE_METRIC_PIXELS, fPixelStartX, fPixelStartY, OSPRITE_M
ETRIC_OGRE, spriteElement.x1, spriteElement.y1);
            convertScreenMetrics(OSPRITE_METRIC_PIXELS, fPixelEndX, fPixelEndY, OSPRITE_METRI
C_OGRE, spriteElement.x2, spriteElement.y2);
        }
        else {
            spriteElement.x1 = -1;
            spriteElement.x2 = 1;
            spriteElement.y1 = 1;
            spriteElement.y2 = -1;
        }

        // We want to draw only a portion of the sprite
        spriteElement.tx1 = spriteElement.ty1 = 0.0f;
        spriteElement.tx2 = spriteElement.ty2 = 1.0f;

        if(spriteRect != FULL_SPRITE) {
            if(_metricSprite == OSPRITE_METRIC_RELATIVE) {
                spriteElement.tx1 = spriteRect.x1;
                spriteElement.ty1 = spriteRect.y1;
                spriteElement.tx2 = spriteRect.x2;
                spriteElement.ty2 = spriteRect.y2;
            }
            else if(_metricSprite == OSPRITE_METRIC_PIXELS) {
                spriteElement.tx1 = (float)(spriteRect.x1 / iSpriteWidth);
                spriteElement.ty1 = (float)(spriteRect.y1 / iSpriteHeight);
                spriteElement.tx2 = (float)(spriteRect.x2 / iSpriteWidth);
                spriteElement.ty2 = (float)(spriteRect.y2 / iSpriteHeight);
            }
        }

        // save alpha value
        spriteElement.alpha = alpha;

        // Add this sprite to our render list
        sprites.push_back(spriteElement);
    }
    //-----------------------------------------------------------------------
    void SpriteHandler::renderBuffer()
    {
       Ogre::RenderSystem* rs=Ogre::Root::getSingleton().getRenderSystem();
       std::list<SpriteElement>::iterator currSpr, endSpr;

       VertexChunk thisChunk;
       std::list<VertexChunk> chunks;

       unsigned int newSize;

       newSize = (int)(sprites.size())*6;
       if (newSize<OGRE2D_MINIMAL_HARDWARE_BUFFER_SIZE)
          newSize=OGRE2D_MINIMAL_HARDWARE_BUFFER_SIZE;

       // grow hardware buffer if needed
       if (hardwareBuffer.isNull() || hardwareBuffer->getNumVertices()<newSize)
       {
          if (!hardwareBuffer.isNull())
             destroyHardwareBuffer();

          createHardwareBuffer(newSize);
       }

       // If we have no sprites this frame, bail here
       if (sprites.empty()) return;

       // write quads to the hardware buffer, and remember chunks
       float* buffer;
       float z=-1;

       buffer=(float*)hardwareBuffer->lock(Ogre::HardwareBuffer::HBL_DISCARD);

       endSpr=sprites.end();
       currSpr=sprites.begin();
       thisChunk.texHandle=currSpr->texHandle;
       thisChunk.vertexCount=0;
       while (currSpr!=endSpr)
       {
           thisChunk.alpha = currSpr->alpha;

          // 1st point (left bottom)
          *buffer=currSpr->x1; buffer++;
          *buffer=currSpr->y2; buffer++;
          *buffer=z; buffer++;
          *buffer=currSpr->tx1; buffer++;
          *buffer=currSpr->ty2; buffer++;
          // 2st point (right top)
          *buffer=currSpr->x2; buffer++;
```

```
          *buffer=currSpr->y1; buffer++;
          *buffer=z; buffer++;
          *buffer=currSpr->tx2; buffer++;
          *buffer=currSpr->ty1; buffer++;
          // 3rd point (left top)
          *buffer=currSpr->x1; buffer++;
          *buffer=currSpr->y1; buffer++;
          *buffer=z; buffer++;
          *buffer=currSpr->tx1; buffer++;
          *buffer=currSpr->ty1; buffer++;

          // 4th point (left bottom)
          *buffer=currSpr->x1; buffer++;
          *buffer=currSpr->y2; buffer++;
          *buffer=z; buffer++;
          *buffer=currSpr->tx1; buffer++;
          *buffer=currSpr->ty2; buffer++;
          // 5th point (right bottom)
          *buffer=currSpr->x2; buffer++;
          *buffer=currSpr->y1; buffer++;
          *buffer=z; buffer++;
          *buffer=currSpr->tx2; buffer++;
          *buffer=currSpr->ty1; buffer++;
          // 6th point (right top)
          *buffer=currSpr->x2; buffer++;
          *buffer=currSpr->y2; buffer++;
          *buffer=z; buffer++;
          *buffer=currSpr->tx2; buffer++;
          *buffer=currSpr->ty2; buffer++;

          // remember this chunk
          thisChunk.vertexCount+=6;
          currSpr++;
          if (currSpr==endSpr || thisChunk.texHandle!=currSpr->texHandle || thisChunk.alpha !
= currSpr->alpha)
          {
             chunks.push_back(thisChunk);
             if (currSpr!=endSpr)
             {
                thisChunk.texHandle=currSpr->texHandle;
                thisChunk.vertexCount=0;
             }
          }
       }

       hardwareBuffer->unlock();

       // set up...
       prepareForRender();

       // do the real render!
       Ogre::TexturePtr tp;
       std::list<VertexChunk>::iterator currChunk, endChunk;

       endChunk=chunks.end();
       renderOp.vertexData->vertexStart=0;
       for (currChunk=chunks.begin(); currChunk!=endChunk; currChunk++)
```

```
   {
      renderOp.vertexData->vertexCount=currChunk->vertexCount;
      tp=Ogre::TextureManager::getSingleton().getByHandle(currChunk->texHandle);
      rs->_setTexture(0, true, tp->getName());

      Ogre::LayerBlendModeEx alphaBlendMode;

      alphaBlendMode.blendType=Ogre::LBT_ALPHA;
      alphaBlendMode.source1=Ogre::LBS_TEXTURE;
      alphaBlendMode.operation=Ogre::LBX_BLEND_MANUAL;
      alphaBlendMode.factor = currChunk->alpha;
      rs->_setTextureBlendMode(0, alphaBlendMode);

      rs->_render(renderOp);

      renderOp.vertexData->vertexStart+=currChunk->vertexCount;
   }

   // sprites go home!
   sprites.clear();
}
//-----------------------------------------------------------------------
void SpriteHandler::prepareForRender()
{
   Ogre::LayerBlendModeEx colorBlendMode;
   Ogre::LayerBlendModeEx alphaBlendMode;
   Ogre::TextureUnitState::UVWAddressingMode uvwAddressMode;

   Ogre::RenderSystem* rs=Ogre::Root::getSingleton().getRenderSystem();

   colorBlendMode.blendType=Ogre::LBT_COLOUR;
   colorBlendMode.source1=Ogre::LBS_TEXTURE;
   colorBlendMode.operation=Ogre::LBX_SOURCE1;

   alphaBlendMode.blendType=Ogre::LBT_ALPHA;
   alphaBlendMode.source1=Ogre::LBS_TEXTURE;
   alphaBlendMode.operation=Ogre::LBX_SOURCE1;

   uvwAddressMode.u=Ogre::TextureUnitState::TAM_CLAMP;
   uvwAddressMode.v=Ogre::TextureUnitState::TAM_CLAMP;
   uvwAddressMode.w=Ogre::TextureUnitState::TAM_CLAMP;

   rs->_setWorldMatrix(Ogre::Matrix4::IDENTITY);
   rs->_setViewMatrix(Ogre::Matrix4::IDENTITY);
   rs->_setProjectionMatrix(Ogre::Matrix4::IDENTITY);
   rs->_setTextureMatrix(0, Ogre::Matrix4::IDENTITY);
   rs->_setTextureCoordSet(0, 0);
   rs->_setTextureCoordCalculation(0, Ogre::TEXCALC_NONE);
   rs->_setTextureUnitFiltering(0, Ogre::FO_LINEAR, Ogre::FO_LINEAR, Ogre::FO_POINT);
   rs->_setTextureBlendMode(0, colorBlendMode);
   rs->_setTextureBlendMode(0, alphaBlendMode);
   rs->_setTextureAddressingMode(0, uvwAddressMode);
   rs->_disableTextureUnitsFrom(1);
   rs->setLightingEnabled(false);
   rs->_setFog(Ogre::FOG_NONE);
   rs->_setCullingMode(Ogre::CULL_NONE);
   rs->_setDepthBufferParams(false, false);
```

```
       rs->_setColourBufferWriteEnabled(true, true, true, false);
       rs->setShadingType(Ogre::SO_GOURAUD);
       rs->_setPolygonMode(Ogre::PM_SOLID);
       rs->unbindGpuProgram(Ogre::GPT_FRAGMENT_PROGRAM);
       rs->unbindGpuProgram(Ogre::GPT_VERTEX_PROGRAM);
       rs->_setSceneBlending(Ogre::SBF_SOURCE_ALPHA, Ogre::SBF_ONE_MINUS_SOURCE_ALPHA);
       rs->_setAlphaRejectSettings(Ogre::CMPF_ALWAYS_PASS, 0);
    }
    //-----------------------------------------------------------------------
    void SpriteHandler::createHardwareBuffer(unsigned int size)
    {
       Ogre::VertexDeclaration* vd;

       renderOp.vertexData=new Ogre::VertexData;
       renderOp.vertexData->vertexStart=0;

       vd=renderOp.vertexData->vertexDeclaration;
       vd->addElement(0, 0, Ogre::VET_FLOAT3, Ogre::VES_POSITION);
       vd->addElement(0, Ogre::VertexElement::getTypeSize(Ogre::VET_FLOAT3),
          Ogre::VET_FLOAT2, Ogre::VES_TEXTURE_COORDINATES);

       hardwareBuffer=Ogre::HardwareBufferManager::getSingleton().createVertexBuffer(
          vd->getVertexSize(0),
          size,// buffer size
          Ogre::HardwareBuffer::HBU_DYNAMIC_WRITE_ONLY_DISCARDABLE,
          false);// use shadow buffer? no

       renderOp.vertexData->vertexBufferBinding->setBinding(0, hardwareBuffer);

       renderOp.operationType=Ogre::RenderOperation::OT_TRIANGLE_LIST;
       renderOp.useIndexes=false;

    }
    //-----------------------------------------------------------------------
    void SpriteHandler::destroyHardwareBuffer()
    {
       delete renderOp.vertexData;
       renderOp.vertexData=0;
       hardwareBuffer.setNull();
    }
    //-----------------------------------------------------------------------
    void SpriteHandler::convertScreenMetrics(OSMetrics metricFrom, const float sx, const floa
t sy, OSMetrics metricTo, float& dx, float& dy)
    {
        // trivial case
        if(metricFrom == metricTo) {
            dx = sx;
            dy = sy;
            return;
        }

        // Convert from pixels ..
        if(metricFrom == OSPRITE_METRIC_PIXELS) {
            // .. to Ogre.
            if(metricTo == OSPRITE_METRIC_OGRE) {
                dx = (sx / _vpHalfWidth) - 1;
                dy = 1 - (sy / _vpHalfHeight);
            }
            // .. to relative.
            else if(metricTo == OSPRITE_METRIC_RELATIVE) {
                dx = (sx / (float)_vpWidth);
                dy = (sy / (float)_vpHeight);
            }
        }
        // Convert from relative ..
        else if(metricFrom == OSPRITE_METRIC_RELATIVE) {
            // .. to Ogre.
            if(metricTo == OSPRITE_METRIC_OGRE) {
                dx = (sx * 2) - 1;
                dy = (sy * -2) + 1;
                return;
            }
            // .. to pixels.
            else if(metricTo == OSPRITE_METRIC_PIXELS)  {
                dx = (sx * _vpWidth);
                dy = (sy * _vpHeight);
                return;
            }
        }
        // Convert from ogre ..
        else if(metricFrom == OSPRITE_METRIC_OGRE) {
            // .. to pixels.
            if(metricTo == OSPRITE_METRIC_PIXELS) {
                float relx = (sx + 1) / 2;
                float rely = (sy - 1) / (-2);

                dx = (relx * _vpWidth);
                dy = (rely * _vpHeight);
                return;
            }
            // .. to relative.
            else if(metricTo == OSPRITE_METRIC_RELATIVE) {
                dx = (sx + 1) / 2;
                dy = (sy - 1) / (-2);
                return;
            }
        }

    }
    //-----------------------------------------------------------------------
}
```

Contributors to this page: jacmoe and Beauty .

Page last modified on Sunday 31 of July, 2011 14:40:15 -08 by jacmoe.

Source History

Search by Tags

Search Wiki by Freetags

Latest Changes

1. AMOFGameEngine
2. MOGRE
3. Roadmap
4. Home
5. Easy Ogre Exporter
6. OGRE Exporters
7. Light mapping
8. Ogre 2.1 FAQ
9. ManualObject
10. Advanced Mogre Framework

...more

Search

Find

Go

☑ Advanced ? Search Help

Online Users

19 online users

-