
TENGINE

Table of Content

Project settings

Include files are located in *c:\tg\2.0\win32_d3d9\include*
Library files are located in *c:\tg\2.0\win32_d3d9\lib\debug* and
c:\tg\2.0\win32_d3d9\lib\release

Here is an example of how you can use the pragma messages in a global file:

```
// Generic libraries
#pragma	comment( lib, "dbghelp.lib" )
#pragma	comment( lib, "winmm.lib" )

// DirectX libraries
#pragma	comment( lib, "d3d9.lib" )
#ifdef DEBUG
#pragma	comment( lib, "d3dx9d.lib" )
#else // DEBUG
#pragma	comment( lib, "d3dx9.lib" )
#endif // DEBUG

// TenGine libraries
#pragma	comment( lib, "tgCCore.lib" )
#pragma	comment( lib, "tgCPluginConsole.lib" )
#pragma	comment( lib, "tgCPluginInput.lib" )
#pragma	comment( lib, "tgCPluginSkydome.lib" )
#pragma	comment( lib, "tgCPluginWindow.lib" )

// TenGine includes
#include	<tgCCore.h>
#include	<tgCPluginConsole.h>
#include	<tgCPluginInput.h>
#include	<tgCPluginSkydome.h>
#include	<tgCPluginWindow.h>
```

Types

TenGine	typedef	size
tgBool	bool	8 bits
tgChar	char	8 bits
tgInt8	char	8 bits
tgUInt8	unsigned char	8 bits
tgSInt8	signed char	8 bits
tgWChar	unsigned short	16 bits
tgInt16	short	16 bits
tgUInt16	unsigned short	16 bits
tgSInt16	signed short	16 bits
tgInt32	int	32 bits
tgUInt32	unsigned int	32 bits
tgSInt32	signed int	32 bits
tgInt64	long long	64 bits
tgUInt64	unsigned long long	64 bits
tgSInt64	signed long long	64 bits
tgFloat	float	32 bits
tgDouble	double	64 bits

Memory functions

TenGine has an integrated memory manager. By overloading the functions *new* and *delete*, TenGine will keep track of memory leaks and print them out into a file called *tgMemory.txt*. You can also call *tgMemoryCheckCorruption()* once a frame to catch potential buffer under/over runs. Only used in debug builds.

FUNCTION	OPERATION
tgMemoryGetUsage()	Current memory usage
tgMemoryGetUsagePeak()	Highest memory usage
tgMemoryRealUsage()	Current real memory usage
tgMemoryGetRealUsagePeak()	Highest real memory usage
tgMemoryBreakOnAddress()	Set a memory address to break on
tgMemoryBreakOnIndex()	Set a memory index to break on
tgMemorySetBreakOnWarnings()	Enables / disables breaking on warnings
tgMemorySetLogAllocs()	Enables / disables logging of new / delete calls
tgMemoryCheckCorruption()	Check all memory chunks for buffer over / under run

Assert functions

The *tgAssert* is not a class, but there are some functions that handles assertions.

FUNCTION	OPERATION
tgAssert()	The assert function
tgAssertSetMode()	Changes the assertion mode

Example code

```
// Make assertions break like a normal assert
tgAssertSetMode( tgASSERT_MODE_BREAK );

// Make assertions log the message to a file and no breaking
tgAssertSetMode( tgASSERT_MODE_LOG );

tgAssert( this != &rIn, "this matrix and rIn are not allowed to be the same matrix" );
```

Endian functions

The *tgEndian* is not a class, but there are some functions that handles assertions.

FUNCTION	OPERATION
tgEndianLittle()	Swap a little endian value
tgEndianBig()	Swap a big endian value

The endian functions either performs an endian swap, or just copies the value depending on if ENDIAN_BIG is defined or not.

Windows is little endian so a *tgEndianLittle* would do a copy while a *tgEndianBig* would do a swap.

PS3 is big endian so there a *tgEndianLittle* would do a swap while a *tgEndianBig* would do a copy.

The possible inputs to the swap functions are *tgSInt16*, *tgUInt16*, *tgSInt32*, *tgUInt32* and *tgFloat*

Example code

```
// Always perform endian swap when loading files from disc
ZipFile.Size                = tgEndianLittle( Header.SizeCompressed );
ZipFile.SizeDecompressed= tgEndianLittle( Header.SizeDecompressed );
```

Frustum functions

The *tgFrustum* is not a class, but there are some functions that handles point and sphere testing against a frustum.

FUNCTION	OPERATION
tgFrustumTestPoint()	Tests if a point is inside the frustum
tgFrustumTestSphere()	Tests if a sphere is inside the frustum
tgFrustumTestAABox()	Tests if a sphere is inside the axis aligned box

Example code

```
// Test if a light is inside the frustum
if( tgFrustumTestSphere( pCamera->GetFrustum(), 5, LightBSphere ) )
        LightInFrustum = TRUE;
```

Frustum planes

A frustum is an area limited by a set amount of planes.
A point or a sphere is considered to be outside a frustum when it's behind any of the planes.

Logging functions

TenGine has several logging functions for logging messages to text files. Following functions are available:

FUNCTION	OPERATION
tgLogPrint()	Prints a message to tgLog.txt
tgLogWarning()	Prints a warning message to tgWarning.txt
tgLogError()	Prints an error message to tgError.txt
tgLogAssert()	Prints an assert message to tgAssert.txt
tgLogMemory()	Prints a memory message to tgMemory.txt
tgLogProfile()	Prints a profile message to tgProfile.txt
tgLogEnable()	Enables logging
tgLogBreakOnWarnings()	Enables / disables breaking on warnings
tgLogBreakOnErrors()	Enables / disables breaking on errors
tgLogSetPath()	Sets the path for the log files

The logging functions are defines which makes logging easier and automatically adds the name of the function that the logging function was called from, to the message.

Example code

```
tgLogEnable( tgLOG_ENABLE_EVERYTHING );
tgLogPrint( "Initializing tgCLogger\n" );
tgLogError( "Found not destroyed texture: %s, reference count: %d\n", pTexture->m_Name,
pTexture->m_ReferenceCount );
```

Math functions

TenGine has several inline math functions and math defines for doing miscellaneous maths calculations. Following functions are available:

FUNCTION	OPERATION
tgMathIsNaN()	Checks if a number is not a number
tgMathLog()	Calculates the log of a value
tgMathLog2()	Calculates the log2 of a value
tgMathLog10()	Calculates the log10 of a value
tgMathAbs()	Removes the – on a negative value making it positive
tgMathCos()	Calculates the cos of a value
tgMathSin()	Calculates the sin of a value
tgMathTan()	Calculates the tan of a value
tgMathACos()	Calculates the acos of a value
tgMathASin()	Calculates the asin of a value
tgMathATan()	Calculates the atan of a value
tgMathATan2()	Calculates the atan2 of a value
tgMathCeil()	Rounds a float value down to the closest integer
tgMathFloor()	Rounds a float value up to the closest integer
tgMathMin()	Compares two numbers and returns the smallest
tgMathMax()	Compares two numbers and returns the biggest
tgMathClamp()	Clamps a number to be between min and max
tgMathBetween()	Checks if a number is between min and max
tgMathAlign()	Aligns a number
tgMathRandom()	Returns a random float between min and max
tgMathInterpolateLinear()	Interpolates a number between start and end
tgMathInterpolateCosine()	Interpolates a number between start and end
tgMathInterpolateCubic()	Interpolates a number between start and end
tgMathInterpolateHermite()	Interpolates a number between start and end
tgMathFastCosAndSin()	Calculates the cos and sin of a value (faster than using cos and sin individually)
tgMathFastSquareRoot()	Calculates the square root of a number (faster than using sqrt)
tgMathFastNextPow2()	Calculates the next power of 2 value
tgMathFastSwap()	Swaps 2 varaibles

Defines

```
TG_E            // e value
TG_PI           // pi value
TG_PI_2         // TG_PI * 2.0
TG_PI_HALF      // TG_PI * 0.5
TG_RAD_TO_DEG   // convert from radians to degrees
TG_DEG_TO_RAD   // convert from degrees to radians
TG_RAND_MAX     // biggest possible integer returned by rand
TG_SINT8_MIN    // min value of a signed char
TG_SINT8_MAX    // max value of a signed char
```

TG_UINT8_MAX *// max value of an unsigned char*
TG_SINT16_MIN *// min value of a signed short*
TG_SINT16_MAX *// max value of a signed short*
TG_UINT16_MAX *// max value of an unsigned short*
TG_SINT32_MIN *// min value of a signed int*
TG_SINT32_MAX *// max value of a signed int*
TG_UINT32_MAX *// max value of an unsigned int*
TG_SINT64_MIN *// min value of a signed long long*
TG_SINT64_MAX *// max value of a signed long long*
TG_UINT64_MAX *// max value of an unsigned long long*
TG_FLOAT_MIN *// min value of a float*
TG_FLOAT_MAX *// max value of a float*

Example code

```
// Enlarge the box if its either smaller than min
m_Min.x     = tgMathMin( m_Min.x, rPoint.x );
m_Min.y     = tgMathMin( m_Min.y, rPoint.y );
m_Min.z     = tgMathMin( m_Min.z, rPoint.z );

// or bigger than max
m_Max.x     = tgMathMax( m_Max.x, rPoint.x );
m_Max.y     = tgMathMax( m_Max.y, rPoint.y );
m_Max.z     = tgMathMax( m_Max.z, rPoint.z );
```

String functions

TenGine has several inline string functions and math defines for doing miscellaneous maths calculations. Following functions are available:

FUNCTION	OPERATION
tgStringClear()	Clears the string
tgStringSet()	Sets the text in the string
tgStringCreate()	Creates the text in the string
tgStringClearAndSet()	Clears and sets the text in the string
tgStringClearAndCreate()	Clears and creates the text in the string
tgStringStripPath()	Strips away the path from a filename
tgStringStripExtension()	Strips away the extension from a filename
tgStringSpaceToUnderscore()	Convert all spaces to underscore

Example code

```
// Clear the string
tgStringClear( m_Name, sizeof( m_Name ) );

// Set the text in the string
tgStringClearAndSet( Technique, sizeof( Technique ), "Skin" );

// Create the text in the string
tgStringCreate( NameExt, sizeof( NameExt ), "%s%s.tga", m_Paths[ PathIndex ], Name );
```

Plugins

tgCPluginConsole		Tweakable Ingame Console Window in application
tgCPluginFMOD	*	Manages audio from FMOD Designer
tgCPluginInput		Manages input
tgCPluginMorpheme	*	Manages advanced animations from Morpheme
tgCPluginMusicManager		Music plugin (Direct Show) for playing mp3
tgCPluginSkydome		Creates a skydome
tgCPluginSoundManager		Sound plugin (Direct Sound) for playing wav
tgCPluginSun		Calculates the sun's position based on long and lat
tgCPluginUVAnimation		Animates UV's on meshes
tgCPluginVideo		Used to display a video (Video for Windows)
tgCPluginWater		Creates a water effect
tgCPluginWindow		Functionality for creating a window (Win32 API)

* Note!
These plugins requires licenses from respective company. Contact us if you are interested in using the plugins.

FMOD:
http://www.fmod.org/

Morpheme:
http://www.naturalmotion.com/morpheme.htm

Information about class constructors in TenGine

The default constructor will be called if you write in any of the following ways:

tgCV3D MyVector;
*tgCV3D** pMyVector = tgNew *tgCV3D();*

The default constructor for *tgCV2D*, *tgCV3D*, *tgCV4D*, *tgCMatrix* and *tgCTransform* does nothing at all when executed. And with nothing we mean NOTHING. The member pointers are NOT set to NULL, no counter variables are reset to 0 etc.

The reason why to this is that we want to give the possibility for the end user (you) to be able to optimize as much as possible. For example, if you know that you manually are going to set all member variables yourself there is no reason to first reset all data and then copy your own data into the variables.

tgCMatrix InvMatrix;
tgCV3D LocalLightDir;

```
InvMatrix.Invert( *pMesh->GetTransform().GetMatrixWorld() );
LocalLightDir.TransformVector( InvMatrix, MyLight.At );
```

In this example we inverted the world matrix from the mesh and stored it directly into InvMatrix. Here it's totally unnecessary to set the matrix to identity before inverting, because the matrix will anyway be overwritten.

The same thing follows with the *tgCV3D* vector class. There is no point in setting the x, y and z coordinates to 0.0f (in a constructor) before calling the TransformVector function, they will anyhow be overwritten.

It is recommended to use default constructors in loops.

Beware that all pointers, if the class has any, will have the default value *0xcdcdcdcd* or similar (depending on your system) if the default constructor is used. This may cause a crash even if you have an if-check, for example:

```
tgCTransform* pMyTransform = tgNew tgCTransform();
tgCTransform* pParent = pMyTransform->GetParent();
if ( pParent )                  // will pass the test because the pParent pointer is not NULL
    pParent->Update();          // will crash because the pParent pointer is 0xcdcdcdcd
```

Available macros in TenGine

Use following macro if you want to display a message to yourself that appears in the output window (Visual Studio) when compiling:
#pragma tgFixMe("Remember to fix this before next release!")

Use *tgBreak* to break the code in runtime.
tgWarning will give a warning message during your compile.

The D3D version of TenGine uses a *tgUInt32* color value. There are a bunch of macros available to convert between RGBA (UInt8) and the D3D color *tgUInt32*.
TG_RGBA_TO_UINT32
TG_UINT32_TO_RGBA
TG_UINT32_TO_R
TG_UINT32_TO_G
TG_UINT32_TO_B
TG_UINT32_TO_A

There are also some predefined color values (in D3D format) available:
TG_RGBA_BLACK
TG_RGBA_GREY
TG_RGBA_WHITE
TG_RGBA_RED
TG_RGBA_GREEN
TG_RGBA_BLUE
TG_RGBA_YELLOW
TG_RGBA_PURPLE
TG_RGBA_CYAN

Examples:

```
tgUInt32 Color = TG_RGBA_TO_UINT32( 255, 0, 0, 255 );

tgCV4D MyCol;
TG_UINT32_TO_RGBA( Color, MyCol.x, MyCol.y, MyCol.z, MyCol.w );
```

tgCAABox2D

The box class contains a min and max *tgCV2D* variable.

FUNCTION	OPERATION
Set()	Set the min and max variables
Copy()	Copies an other box to this box
Transform()	Converts a box between spaces
AddBox()	Expands the box to fit the given *tgCAABox*
AddPoint()	Adds a point to the box, calculates new min and max
PointInside()	Checks if a *tgCV3D* point is inside the box
Intersect()	Checks collision between two boxes
GetMin()	Get the min value
GetMax()	Get the max value

Constructors

tgCAABox2D box0; // Default constructor does nothing = fast
tgCAABox2D box1(*tgCV2D(0.0f)*); // Sets the min and max points to (0.0f, 0.0f)
tgCAABox2D box2(*tgCV2D*(2.0f, 3.0f), *tgCV2D*(5.0f, 8.0f));
// Sets min (2.0f, 3.0f) and max (5.0f, 8.0f)
tgCAABox2D box3(box2); // Copies box2 into box3

When using the *tgCAABox2D* class you have to run the *Set* function first before starting to add points to the box. For example:

tgCAABox2D MyBox;

// Add points to the box
MyBox.Set(FirstPoint, FirstPoint);
MyBox.AddPoint(SecondPoint);
MyBox.AddPoint(ThirdPoint);
etc…

If you don't run the *Set* function the default value for min and max will be calculated into the box. For example if you have cleared the box, the point (0.0f, 0.0f) will be a point belonging to the box. Or even worse if you have used the default constructor, the min and max have a garbage value from the beginning.

tgCAABox3D

The box class contains a min and max *tgCV3D* variable. All objects (meshes, world sectors and worlds) have their own bounding boxes of type *tgCAABox3D*.

FUNCTION	OPERATION
Set()	Set the min and max variables
Copy()	Copies an other box to this box
Transform()	Converts a box between spaces
AddBox()	Expands the box to fit the given *tgCAABox*
AddPoint()	Adds a point to the box, calculates new min and max
PointInside()	Checks if a *tgCV3D* point is inside the box
Intersect()	Checks collision between two boxes
GetMin()	Get the min value
GetMax()	Get the max value

Constructors

```
tgCAABox3D box0;                    // Default constructor does nothing = fast
tgCAABox3D box1( tgCV3D(0.0f) );    // Sets the min and max points to (0.0f, 0.0f, 0.0f)
tgCAABox3D box2( tgCV3D(2.0f, 0.0f, 3.0f), tgCV3D(5.0f, 9.0f, 8.0f) );
                                    // Sets min (2.0f, 0.0f, 3.0f) and max (5.0f, 9.0f, 8.0f)
tgCAABox3D box3( box2 );            // Copies box2 into box3
```

When using the *tgCAABox3D* class you have to run the *Set* function first before starting to add points to the box. For example:

```
tgCAABox3D MyBox;

// Add points to the box
MyBox.Set( FirstPoint, FirstPoint );
MyBox.AddPoint( SecondPoint );
MyBox.AddPoint( ThirdPoint );
etc…
```

If you don't run the *Set* function the default value for min and max will be calculated into the box. For example if you have cleared the box, the point (0.0f, 0.0f, 0.0f) will be a point belonging to the box. Or even worse if you have used the default constructor, the min and max have a garbage value from the beginning.

tgCAnimation

The *tgCAnimation* class contains functions for using an animation in TenGine.
Following functions are available:

FUNCTION	OPERATION
Create()	Creates an animation
Save()	Saves the animation to a file
GetHierarchy()	Retrieves the animation's hierarchy
GetUserData()	Get a pointer to the user data
SetUserData()	Set new user data
GetName()	Retrieves the animation's name
GetDuration()	Retrieves the animation's duration
GetNumObjects()	Retrieves the animation's number of objects
GetNumFrames()	Retrieves the animation's number of frames

```
// Save an animation
// We can use 0 as start and 9999999 as end frames because the Save function
// clamps start to 0 and end to ( pAnimation->NumFrames – 1 )
pAnimation->Save( "walk.tfa", *pAnimation, 0, 9999999 );

// We could also just save a small piece of an existing animation as a new animation
pAnimation->Save( "walk_short.tfa", *pAnimation, 48, 63 );
```

tgCAnimationManager

The *tgCAnimationManager* class contains functions for creating and destroying *tgCAnimation*'s in TenGine. Following functions are available:

FUNCTION	OPERATION
Create()	Creates a *tgCAnimation*
Destroy()	Destroys a *tgCAnimation*

Constructors

tgCAnimationManager AnimationManager(); // Default constructor clears the member variables (called when initializing the singleton)

```
// Create an animation
tgCAnimation* pAnimation;
pAnimation = tgCAnimationManager::GetInstance().Create( "walk.tfa" );

// Destroy an animation
tgCAnimationManager::GetInstance().Destroy( &pAnimation );
```

tgCCamera

The *tgCCamera* class contains functions for using a camera in TenGine.
Following functions are available:

FUNCTION	OPERATION
Create()	Creates a camera
Update()	Updates the camera's transform, view matrix and view projection matrix
CalcFrustum()	Calculates the camera's view frustum
ClearBuffers()	Clears the specified buffers
BeginRender()	Begins the rendering
EndRender()	Ends the rendering
ScreenToWorld()	Converts a 2D screenspace position to a 3D worldspace position
WorldToScreen()	Converts a 3D worldspace position to a 2D screenspace position
GetRenderTarget()	Retrieves one of the camera's render targets
SetRenderTarget()	Sets one of the camera's render targets
GetDepthStencil()	Retrieves the camera's depth stencil
SetDepthStencil()	Sets the camera's depth stencil
GetUserData()	Get a pointer to the user data
SetUserData()	Set new user data
GetFrustum()	Retrieves the camera's view frustum
GetName()	Retrieves the camera's name
SetName()	Sets the camera's name
GetViewPort()	Retrieves the camera's viewport
SetViewPort()	Sets the camera's viewport
GetScissor()	Retrieves the camera's scissor
SetScissor()	Sets the camera's scissor
GetProjection()	Retrieves the camera's projection type
SetProjection()	Sets the camera's projection type
GetTransform()	Retrieves the camera's transform
SetTransform()	Sets the camera's transform
GetPerspectiveMatrix()	Retrieves the camera's perspective projection matrix
Get3DOrthoMatrix()	Retrieves the camera's 3D ortho projection matrix
Get2DOrthoMatrix()	Retrieves the camera's 2D ortho projection matrix
GetViewMatrix()	Retrieves the camera's view matrix
GetViewProjectionMatrix()	Retrieves the camera's view projection matrix
GetAspect()	Retireves the camera's aspect ratio
SetAspect()	Sets the camera's aspect ratio and calculates a new perspective projection matrix
GetNearClip()	Retireves the camera's near clip
SetNearClip()	Sets the camera's near clip and calculates new perspective and 3d ortho projection matrises
GetFarClip()	Retrieves the camera's far clip
SetFarClip()	Sets the camera's far clip and calculates new perspective

	and 3d ortho projection matrises
GetFov()	Retrieves the camera's field of view
SetFov()	Sets the camera's filed of view and calculates a new perspective projection matrix
Get3DOrthoSize()	Retrieves the camera's 3d ortho size
Set3DOrthoSize()	Sets the camera's 3d ortho size
Get2DOrthoSize()	Retrieves the camera's 2d ortho size
Set2DOrthoSize()	Sets the camera's 2d ortho size
GetClearColor()	Retrieves the camera's clear color
SetClearColor()	Sets the camera's clear color

There are three kinds of *projection types* when using cameras:

tgCAMERA_PROJECTION_PERSPECTIVE – Perspective projection.
tgCAMERA_PROJECTION_3D_ORTHO – 3d ortho projection, 0,0 is in the middle of the screen, x points left and y points up.
tgCAMERA_PROJECTION_2D_ORTHO - 2d ortho projection, 0,0 is in the top left corner of the screen, x points right and y points down.

Constructors

tgCCamera Camera(); // Default constructor clears the member variables

Example code

```
// Create a camera
tgCCamera* pCamera;
pCamera = tgCCameraManager::GetInstance().Create( pCameraName, X, Y, Width, Height, Fov, Aspect, NearClip, FarClip, ProjectionType );
tgCCameraManager::GetInstance().SetCurrentCamera( *pCamera );

...

// Render a frame
if( pCamera->BeginRender() )
{
		// RenderCalls

...

		pCamera->EndRender();
}

// Show the rendered buffer on screen
tgCD3D9::GetInstance().GetDevice()->Present( NULL, NULL, NULL, NULL );
```

tgCCameraManager

The *tgCCameraManager* class contains functions for creating and destroying *tgCCamera*'s in TenGine. Following functions are available:

FUNCTION	OPERATION
Create()	Creates a *tgCCamera*
Destroy()	Destroys a *tgCCamera*
GetCurrentCamera()	Retrieves the current camera
SetCurrentCamera()	Sets the current camera

Constructors

tgCCameraManager CameraManager(); // Default constructor clears the member variables (called when initializing the singleton)

Example code

```
// Create a camera
tgCCamera* pCamera;
pCamera = tgCCameraManager::GetInstance().Create( pCameraName, X, Y, Width, Height, Fov, Aspect, NearClip, FarClip, ProjectionType );
tgCCameraManager::GetInstance().SetCurrentCamera( *pCamera );

// Destroy a camera
tgCCameraManager::GetInstance().Destroy( &pCamera );
```

tgCCollision

The *tgCCollision* class contains functions for collision handling in TenGine.
Following functions are available:

FUNCTION	OPERATION
Clear()	Clear all variables and sets Fraction to 1.0f
Copy()	Copies collision data into this collision
LineMesh()	
LineAllMeshesInModel()	Line collision with all meshes in a model
LineAllMeshesInWorldSector()	Line collision with all meshes in a world sector
LineAllMeshesInWorld()	Line collision with all meshes in a world
SphereMesh()	
SphereAllMeshesInModel()	Sphere collision with all meshes in a model
SphereAllMeshesInWorldSector()	Sphere collision with all meshes in a world sector
SphereAllMeshesInWorld()	Sphere collision with all meshes in a world
SetTriangleCollisionCallback()	Set the triangle collision callback
GetMesh()	Retrieves the mesh that had collision
GetLocalIntersection()	Retrieves the intersection point of the collision
GetLocalNormal()	Retrieves the normal of the collision
GetFraction()	Retrieves the fraction of the collision in as a value between 0.0 and 1.0 For a line this the distance from start of line to the intersection position, normalized to the length of the line For a sphere this is the distance from the sphere position to the intersection position, normalized to the radius of the sphere
SetFraction()	Sets the fraction of the collision (mostly used to reset it to 1.0 for a new collision check)
GetTriangleIndex()	Retrieves the index to the triangle in the mesh that had collision
SetType()	Sets which mesh types to collide against
GetNumTrianglesTested()	Retrieves the number of triangles tested
SetIgnoreAlpha()	Sets if collision against alpha meshes should be ignored or not

Constructors

```
tgCCollision Collision0;                // Default constructor does nothing = fast
tgCCollision Collision1( true );        // Clears the collision variables, Fraction = 1.0f
tgCCollision Collision2( Collision1 );  // Copies collision1 into collision2
```

Note!
It is recommended that you use the constructor that clears the data in the class, or call the *Clear*() function after creation. Else the variables are not initialized and the collision test will probably not behave as you expect.

Collision types

There are two types of collisions: *tgMESH_TYPE_MODEL* and *tgMESH_TYPE_WORLD*. The collision type you set with *SetType*() specifies if you want to check collision against meshes in a model or in the world. You can also check collision against meshes in both models and world by typing *SetType*(*tgMESH_TYPE_MODEL* | *tgMESH_TYPE_WORLD*).

Collision flags

Be aware that you are able to remove or set the flag *tgMESH_FLAGS_COLLISION* on any mesh with the function *tgCMesh*::*SetFlags*(). The collision test will be ignored for the specified mesh if the flag is not set. The *tgMESH_FLAGS_COLLISION* flag is set by default when a model is created.

Fraction

The fraction is a floating point value between 0.0f and 1.0f that will be calculated when collisions occur. The fraction represents a scaled value between a lines start- and endpoint. For example if the fraction is 0.5f, the collision has occurred in the middle of the line. If the fraction is 0.9f, the collision has occurred at a point 9/10 part towards the end point.
The fraction value for a sphere is calculated with the spheres middle position as the start point in a line with length of the spheres radius.

If you have set the collision callback, the callback will be called each time a collision occurs. Else if the callback is NULL, you will only get the collision intersection nearest to the start point (the lowest fraction value).

Between different collision testes (if you use the same collision class variable) you have to reset the fraction value to 1.0f. You can do this by calling *SetFraction*() or *Clear*() function.

Sphere Collision

When using sphere collision against meshes you will get the closest point on the nearest triangle that the sphere intersects (the closest point from sphere centre to triangle) by calling *GetLocalIntersection()*. If the centre of the sphere is outside the triangle you will get a point on the triangle edge.

To understand the fraction value, imagine a line from sphere centre through the collision point (sphere radius line). The fraction value is 0.0f at the sphere centre and 1.0f at the sphere hull-boundary.

Alpha

You can use the function *SetIgnoreAlpha*(TRUE) if you want the collision test to ignore meshes with alpha. When the *Clear*() function is called, the ignore alpha variable will be reset to FALSE (default).

Note!
If you manually change the texture, material or vertex color for an object, make sure that you also update the *HasAlpha* boolean variable in the affected mesh.

Collision callback

You can specify a callback function that will be executed each time a collision occurs against a triangle. The function *SetTriangleCollisionCallback*() does this for you. The callback shall return a *tgBool* value, if you return FALSE the collision check will be terminated. If you return TRUE TenGine will continue checking all other triangle collisions that may occur with the object.

This is how you write the callback in the class .h file:
static *tgBool* TriangleFastCollisionCB(*tgCCollision* &rCollision, void* pUserData);

And here is an example of a fast triangle callback. This is the way to do it if you only want to know if collision occurred or not.

```
tgBool CMyClass::TriangleLineCollisionCB( tgCCollision &rCollision, void* pUserData )
{
    return FALSE;
}
```

Note!

The variables in the collision class are not reset by default between collision tests.

When a collision occur the class variables will be set with proper data. For example the pointer to the mesh will be set and can be accessed by calling *GetMesh()*. If you continue with collision checks and the check fails, the pointer will still point to the last mesh that you had collision with.
That's why it's very important to call the *Clear()* function between collision tests.

Example code

```
// Create the collision object and clear it
tgCCollision CollisionObject( TRUE );

// Create a 3D line
tgCLine3D CollisionLine( StartPosition, EndPosition );

// Set the desired type
CollisionObject.SetType( tgMESH_TYPE_MODEL );

// Set a callback
CollisionObject.SetTriangleCollisionCallback( TriangleLineCollisionCB, NULL );

// Do the collision check
if ( CollisionObject.LineAllMeshesInModel( CollisionLine, *m_pModel ) )
{
    // Collision between line and model occurred!
    tgCV3D* pCollisionPoint = CollisionObject.GetIntersection();
    ...
}
else
{
    // Collision between line and model failed!
}
```

tgCCircle

The circle class contains a radius and position variable.

FUNCTION	OPERATION
Set()	Sets the member variables
Copy()	Copies data from an other circle into this circle
Intersect()	Collision testing against the circle
GetPos()	Get the circle position
SetPos()	Set the circle position
GetRadius()	Get the circle radius
SetRadius()	Set the circle radius

Constructors

```
tgCCircle circle0;                          // Default constructor does nothing = fast
tgCCircle circle1( tgCV2D( 2.0, 3.0f ), 5.0f );
                                            // Sets the pos to (2.0f, 3.0f ) with radius 5.0f
tgCCircle circle2( circle1 );               // Copies data from circle1 into circle2
```

tgCCore

The *tgCCore* class is the class that is the central part of TenGine and ties all other parts together.

FUNCTION	OPERATION
HandleMessages()	Loops through and handles all OS messages
SplashScreen()	Loads a BMP texture and shows it as a splash screen
GetAnisotropy()	Retrieves the anisotropy of the main window
SetAnisotropy()	Sets the anisotropy of the main window
GetHasFocus()	Retrieves if the main window has focus or not
SetHasFocus()	Sets if the main window has focus or not
GetHadFocus()	Retrieves if the main window had focus or not
SetHadFocus()	Sets if the main window had focus or not
GetQuit()	Retrieves if the application should quit or not
SetQuit()	Sets if the application should quit or not
GetWinCursorVisibility()	Retrieves the applications wanted visibility of the windows cursor
SetWinCursorVisibility()	Sets the applications wanted visibility of the windows cursor
GetWindowFullscreen()	Retrieves the fullscreen status of the main window
SetWindowFullscreen()	Sets the fullscreen status of the main window
GetWindowHandle()	Retrieves the handle of the main window
SetWindowHandle()	Sets the handle of the main window
GetWindowHeight()	Retrieves the height of the main window
SetWindowHeight()	Sets the height of the main window
GetWIndowName()	Retrieves the name of the main window
SetWindowName()	Sets the name of the main window
GetWindowPos()	Retrieves the position of the main window
GetWindowPosX()	Retrieves the X position of the main window
GetWindowPosY()	Retrieves the Y position of the main window
SetWindowPos()	Sets the position of the main window
GetWindowRefreshRate()	Retrieves the refresh rate of the main window
SetWindowRefreshRate()	Sets the refresh rate of the main window
GetWindowVSync()	Retrieves the vsync status of the main window
SetWindowVSync()	Sets the vsync status of the main window
GetWindowWidth()	Retrieves the width of the main window
SetWindowWidth()	Sets the width of the main window

Constructors

tgCCore Core(); // Default constructor clears the member variables (called when initializing the singleton)

When the core instance is initialized it sets a few default values such as:
Window name is set to "TenGine"
Anisotropic filtering is set to 1
Window width is set to 640
Window height is set to 480

Window refresh rate is set to 60Hz
Window fullscreen is set to FALSE
Window vertical sync is set to TRUE
CursorVisible is set to FALSE
HasFocus is set to TRUE
HadFocus is set to TRUE
Quit is set to FALSE

When all those are set it initializes the following instances in this order:
tgCCPUInfo
tgCD3D
tgCFileSystem
tgCCameraManager
tgCShaderManager
tgCTextureManager
tgCLightManager
tgCWorldManager
tgCModelManager
tgCAnimationManager
tgCQuadManager
tgCSpriteManager
tgCSplineGroupManager
tgCFontManager
tgCDebugManager

Example code

```
// Create the core
tgCCore::Initialize();

// Destroy the core
if( tgCCore::GetInstancePtr() )
                tgCCore::Deinitialize();
```

TODO: write more about tgCCore

tgCCPUInfo

The *tgCCPUInfo* class is the class that fetches all the cpu information such as cpu speed, model name, manufacturer, assembly capabilities etc.

FUNCTION	OPERATION
GetName()	Retrieves the cpu name
GetManufacturer()	Retrieves the cpu manufacturer name
GetSpeed()	Retrieves the cpu speed in mhz
GetL1CacheSize()	Retrieves the cpu L1 cache size in kb
GetL2CacheSize()	Retrieves the cpu L2 cache size in kb
GetL3CacheSize()	Retrieves the cpu L3 cache size in 512 kb chunks
GetNumLogicalCores()	Retrieves the cpu's number of logical cores
GetNumRealCores()	Retrieves the cpu's number of real cores
GetHasFPU()	Retrieves if the cpu has a floating point unit or not
GetHasHyperThreading()	Retrieves if the cpu has hyper threading or not
GetHasMMX()	Retrieves if the cpu can use MMX assembly or not
GetHasAmdMMX()	Retrieves if the cpu can use AmdMMX assembly or not
GetHasSSE()	Retrieves if the cpu can use SSE assembly or not
GetHasSSE2()	Retrieves if the cpu can use SSE2 assembly or not
GetHasSSE3()	Retrieves if the cpu can use SSE3 assembly or not
GetHasSSSE3()	Retrieves if the cpu can use SSSE3 assembly or not
GetHasSSE4Rev1()	Retrieves if the cpu can use SSE4Rev1 assembly or not
GetHasSSE4Rev2()	Retrieves if the cpu can use SSE4Rev2 assembly or not
GetHasSSERevA()	Retrieves if the cpu can use SSE4RevA assembly or not
GetHasSSE5()	Retrieves if the cpu can use SSE5 assembly or not
GetHas3DNow()	Retrieves if the cpu can use 3DNow assembly or not
GetHas3DNow2()	Retrieves if the cpu can use 3DNow2 assembly or not

Constructors

tgCCPUInfo CPUInfo(); // Default constructor clears the member variables (called when initializing the singleton)

tgCD3D9

The *tgCD3D9* class is the class that handles all the Direct3D specific things such as creating a device.

FUNCTION	OPERATION
CreateDevice()	Creates the Direct3D device using information from tgCCore
DestroyDevice()	Destroys the Direct3D device
GetDefaultRenderTarget()	Retrieves a pointer to the default render target
GetDefaultDepthStencil()	Retrieves a pointer to the default depth and stencil
GetD3D()	Retrieves a pointer to the actual D3D object
GetDevice()	Retrieves a pointer to the Direct3D device
GetPresentParameters()	Retrieves a pointer to the present parameters
GetCaps()	Retrieves the caps for the created Direct3D device
GetVertexShaderVersion()	Retrieves the highest vertex shader version the Direct3D device supports
GetPixelShaderVersion()	Retrieves the highest pixel shader version the Direct3D device supports
GetAdapterIndex()	Retrieves the adapter index that is used
GetMaxAnisotropy()	Retrieves the maximum anisotropy the device can use
GetMaxMultiSampleQuality()	Retrieves the maximum multisample quality the device can use
ResetDevice()	Resets a lost device
OnLostDevice()	Release all needed Direct3D objects whenever the device is lost
OnResetDevice()	Restores all needed Direct3D objects whenever the device has been reset
ScreenShot()	Takes a screenshot
ToggleFullscreen()	Toggles between window mode and fullscreen
SetClipPlane()	Sets a user defined clipping plane

Constructors

tgCD3D9 D3D9(); // Default constructor clears the member variables (called when initializing the singleton)

Example code

```
// Create the device
tgCD3D9::GetInstance().CreateDevice( MultiSampleQuality,
tgPERFORMANCEDEVICE_NONE );

// Destroy the device
tgCD3D9::GetInstance().DestroyDevice();
```

Whenever the OS switches between window and fullscreen mode the D3D device is lost. In order to keep the application running the device has to be **Reset**.

Calling **Reset** causes all texture memory surfaces to be lost, managed textures to be flushed from video memory, and all state information to be lost. Before calling the **Reset** method for a device, an application should release any explicit render targets, depth stencil surfaces, additional swap chains, state blocks, and D3DPOOL_DEFAULT resources associated with the device.

When the application is wanted to switch to fullscreen or window mode one should use the function **ToggleFullscreen**. It alters the **D3DPRESENT_PARAMETERS Windowed** parameter and switches the win32 window to fullscreen or window mode to match the new parameter. This triggers a lost device which builds a new device with the new settings.

tgCFileRead

The *tgCFileRead* class is the class that allows the user to read and parse files into memory by stream reading them while parsing, rather than loading the entire file into memory before parsing

FUNCTION	OPERATION
SkipForward()	Skips forward in the stream buffer
SeekSet()	Sets an absolute position in the stream buffer
ReadString()	Reads a string from the stream buffer
ReadSInt8()	Reads signed chars from the stream buffer
ReadUInt8()	Reads unsigned chars from the stream buffer
ReadSInt16()	Reads signed shorts from the stream buffer
ReadUInt16()	Reads unsigned shorts from the stream buffer
ReadSInt32()	Reads signed ints from the stream buffer
ReadUInt32()	Reads unsigned ints from the stream buffer
ReadFloat()	Reads floats from the stream buffer
GetByte()	Retrieves a specific byte in the stream buffer
GetEOF()	Checks if all data has been read

Constructors

tgCFileRead FileReader(FileIndex, 128 * 1024); // Default constructor initializes the reader

Example code

```
// Create file reader with a 128kb stream buffer
tgCFileRead FileRead( FileIndex, 128 * 1024 );

// Read the identifier
FileRead.ReadString( Identifier, 4 );

// Parse header
FileRead.ReadFloat( ( tgFloat* )&rWorld.m_AABBox,        6 );
FileRead.ReadFloat( ( tgFloat* )&rWorld.m_BSphere,       4 );
FileRead.ReadUInt32( &rWorld.m_NumPortals );
FileRead.ReadUInt32( &NumTotalMeshes );
FileRead.ReadUInt32( &NumTotalSubMeshes );
```

tgCFileSystem

The *tgCFileSystem* class is the class that handles all file related tasks, such as reading, moving and deleting files / directories

FUNCTION	OPERATION
SetFilePath()	Sets the path to the data directories the filesystem will search for files inside
FileExists()	Checks if a file exists with the given name
FileLoad()	Loads a file into a memory buffer
FileOpen()	Opens a file
FileClose()	Closes a file
FileRead()	Reads data from a file
FileWrite()	Writes data to a file
FileSeek()	Seeks within a file
FileCopy()	Copies a file and creates any needed directories
FileMove()	Moves a file and creates any needed directories
DirectoryCopy()	Copies a directory with all sub-directories and all files inside them
DirectoryCreate()	Creates a directory and all needed parent directories
DirectoryDelete()	Deletes a directory with all sub-directories and all files inside them

Constructors

tgCFileSystem FileSystem(); // Default constructor clears the member variables (called when initializing the singleton)

Example code

```
// Copy a file
tgStringCreate( TextureName, sizeof( TextureName ), "textures/temp/%s",
tgStringStripPath( pTextureName ) );
tgCFileSystem::GetInstance().FileCopy( pTextureName, TextureName  );

// Delete a directory
tgCFileSystem::GetInstance().DirectoryDelete( "textures/temp/" );
```

Zip files

The filesystem supports reading most file types from inside zip files.

tgCFont

The *tgCFont* class contains functions for using bitmap fonts in TenGine.
Following functions are available:

FUNCTION	OPERATION
Creates()	Creates the font
RenderText()	Queues the given text for rendering using this font
GetTextWidth()	Retrieves the width of the given text
GetTextHeight()	Retrieves the height of the given text
GetName()	Retrieves a pointer to the the font's name

Constructors

tgCFont Font(); // Default constructor clears the member variables

Use the tgCFontManager to create a font.

tgCFontManager

The *tgCFontManager* class contains functions for creating and destroying *tgCFonts* in TenGine. Following functions are available:

FUNCTION	OPERATION
Create()	Creates a *tgCFont*
Destroy()	Destroys a *tgCFont*

Constructors

tgCFontManager FontManager(); // Default constructor clears the member variables (called when initializing the singleton)

Example code

```
// Create a bitmap font
tgCFont*    pFont;
pFont       = tgCFontManager::GetInstance().Create( "arial_9_normal" );

// Destroy the font
tgCFontManager::GetInstance().Destroy( &pFont );
```

tgCInterpolator

The *tgCInterpolator* class contains functions for handling animations in TenGine.
Following functions are available:

FUNCTION	OPERATION
Create()	Creates an interpolator
Destroy()	Destroys an interpolator
Copy()	Copies an interpolator into another interpolator
AddTime()	Advance the interpolator
Blend()	Blends two interpolator together
GetAnimation()	Retrieves the interpolator's animation
SetAnimation()	Sets the interpolator's animation
GetInterpoaltedArray()	Retrieves the interpolator's interpolated array
GetTime()	Retrieves the interpolator's current time
SetTime()	Sets the interpolator's current time
GetFirstObject()	Retrieves the interpolator's first object
GetLastObject()	Retrieves the interpolator's last object
GetCurrentFrame()	Retrieves the interpolator's current frame
SetEndofAnimationCallback()	Sets the callback to be run when animation ends

Constructors

tgCInterpolator Interpolator(); // Default constructor clears the member variables

Animation types

There are two types of interpolators: regular interpolators and sub-interpolators.
When creating an interpolator you have to specify the first and last object. These tell which parts of the model hierarchy that will be animated. These defaults to 0 and 65535 which means it will be a regular interpolator.

Blending

Blending takes 2 interpolators and blends them together into a third interpolator. All those 3 interpolators have to have been created from animations for the same model but they can be both regular and sub-interpolators. It will store the animation data between the third interpolators first and last object, using interpolator1, interpolator2, a blend of those, or keep the third animations data all depending on how the 3 interpolators look. The flag OrthoNormalize can be set to correct errors in the blending but this will also remove all scaling done to any objects in the animation.

End of animation callback

You can specify a callback function that will be executed each time an interpolator reaches its end. This can for example be used to automatically switch to another interpolator when the first one is done, rather than just looping it.

This is how you write the callback in the class .h file:
static *tgBool* EndOfAnimationCB(*tgCAnimation*& rAnimation, *void** pUserData);

Example code

```
// Create an animation
tgCAnimation* pAnimation;
pAnimation = tgCAnimationManager::GetInstance().Create( "walk.tfa" );

// Create an interpolator
tgCInterpolator* pInterpolator = tgNew tgCInterpolator();
pInterpolator->Create( *pAnimation );

…

// Advance the interpolator in time and update the model
pInterpolator->AddTime( DeltaTime );
pModel->SetAnimationMatrices( *pInterpolator );
pModel->Update( TRUE, TRUE );

...

// Set a callback
pInterpolator->SetEndofAnimationCallback( EndOfAnimationCB, NULL );
```

tgCLight

The *tgCLight* class contains functions for using lights in TenGine.
Following functions are available:

FUNCTION	OPERATION
Create()	Creates the light
Update()	Updates the light's world space transform matrix by either copying its local space matrix or multiplying by a parent's world space matrix.
GetUserData()	Get a pointer to the user data
SetUserData()	Set new user data
GetName()	Retrieves the light's name
SetName()	Sets the light's name
GetTransform()	Retrieves a reference to the light's transform
SetTransform()	Sets the light's transform
GetColor()	Retrieves the light's color
SetColor()	Sets the light's color
GetIntensity()	Get the intensity value
SetIntensity()	Set the intensity value
GetConeAngle()	Retrieves the light's cone angle
SetConeAngle()	Sets the light's cone angle
GetRadius()	Retrieves the light's radius
SetRadius()	Sets the light's radius
GetType()	Retrieves the light's type
SetType()	Sets the light's type
GetEnabled()	Retrieves if the light is enabled or not
SetEnabled()	Sets if the light is enabled or not

Constructors

tgCLight Light(); // Default constructor clears the member variables

There are four types of lights available:
tgLIGHT_TYPE_AMBIENT
tgLIGHT_TYPE_DIRECTIONAL
tgLIGHT_TYPE_POINT
tgLIGHT_TYPE_SPOT

Use the tgCLightManager to create a light.

A scene always need an ambient light for the base render since the other 3 kinds of light just brighten up what was rendered with the ambient light.

tgCLightManager

The *tgCLightManager* class contains functions for creating and destroying *tgCLight*'s in TenGine. Following functions are available:

FUNCTION	OPERATION
Create()	Creates a *tgCLight*
Destroy()	Destroys a *tgCLight*
GetCurrentLight()	Retrieves the current light
SetCurrentLight()	Sets the current light

Constructors

tgCLightManager LightManager(); // Default constructor clears the member variables (called when initializing the singleton)

Example code

```
// Create a red light
tgCLight*               pLight;
tgELightType            Type;
tgUInt32                Color;
Type        = tgLIGHT_TYPE_DIRECTIONAL
Color       = RGBA_TO_UINT32( 255, 0, 0, 255 );
pLight      = tgCLightManager::GetInstance().Create( ”Light2”, Type, Color, 1.0f, 1.0f,
90.0f );


// Render using a specific light
tgCLightManager::GetInstance().SetCurrentLight( *pLight );
pMesh->Render();


// Destroy the light
tgCLightManager::GetInstance().Destroy( &pLight );
```

tgCLine2D

The *tgCLine2D* class contains two *tgCV2D* member vectors (start and end).

FUNCTION	OPERATION
Clear()	Clears the member variables
Set()	Set the start and end variables
Copy()	Copies a line into this line
Length()	Calculates the length of the line
LengthSquared()	Calculates the squared length of the line
ClampToAABox()	Clamps the line to be fully inside the box
ClosestPoint()	Calculate closest point on line
Intersect()	Intersection test between two lines
GetStart()	Get the start position
SetStart()	Set the start position
GetEnd()	Get the end position
SetEnd()	Set the end position

Constructors

tgCLine2D line0; // Default constructor does nothing = fast
tgCLine2D line1(true); // Sets the start and end points to (0.0f, 0.0f)
tgCLine2D line2(*tgCV2D*(2.0f, 3.0f), *tgCV2D*(5.0f, 6.0f));
// Sets start (2.0f, 3.0f) and end (5.0f, 6.0f)
tgCLine2D line3(line2); // Copies line2 into line3

tgCLine3D

The *tgCLine3D* class contains two *tgCV3D* member vectors (start and end).

FUNCTION	OPERATION
Clear()	Clears the member variables
Set()	Set the start and end variables
Copy()	Copies a line into this line
Length()	Calculates the length of this line
LengthSquared	Calculates the squared length of this line
ClampToAABox()	Clamps the line to be fully inside the box
ClosestPoint()	Calculates closest point on line from given point
Intersect()	Intersection tests between objects and the line
GetStart()	Get the start position
SetStart()	Set the start position
GetEnd()	Get the end position
SetEnd()	Set the end position

Constructors

tgCLine3D line0; // Default constructor does nothing = fast
tgCLine3D line1(true); // Sets the start and end points to (0.0f, 0.0f)
tgCLine3D line2(*tgCV3D*(2.0f, 0.0f, 3.0f), *tgCV3D*(5.0f, 9.0f, 8.0f));
// Sets start (2.0f, 0.0f, 3.0f) and end (5.0f, 9.0f, 8.0f)
tgCLine3D line3(line2); // Copies line2 into line3

tgCMaterial

The *tgCMaterial* class contains functions defining which textures and shaders a mesh should use. The following functions are available:

FUNCTION	OPERATION
Clear()	Clears all the member varaibles
Copy()	Copies a material
GetColormap()	Retrieves a pointer to the material's colormap
SetColormap()	Sets the material's colormap
GetLayermap1()	Retrieves a pointer to the material's first layermap
SetLayermap1()	Sets the material's first layermap
GetLayermap2()	Retrieves a pointer to the material's second layermap
SetLayermap2()	Sets the material's second layermap
GetLayermap3()	Retrieves a pointer to the material's third layermap
SetLayermap3()	Sets the material's third layermap
GetNormalmap()	Retrieves a pointer to the material's normalmap
SetNormalmap()	Sets the material's normalmap
GetAmbientmap()	Retrieves a pointer to the material's ambientmap
SetAmbientmap()	Sets the material's ambientmap
GetSpecualrmap()	Retrieves a pointer to the material's specularmap
SetSpecularmap()	Sets the material's specularmap
GetReflectionmap()	Retrieves a pointer to the material's reflectionmap
SetReflectionmap()	Sets the material's reflectionmap
GetShader()	Retrieves a pointer to the material's shader
SetShader()	Sets the material's shader
GetUserData()	Get a pointer to the user data
SetUserData()	Set new user data
GetName()	Retrieves the material's name
GetColor()	Retrieves the material's color
SetColor()	Sets the material's color
GetReflectionFactor()	Retrieves the material's reflection factor
SetReflectionFactor()	Sets the material's reflection factor

Constructors

```
tgCMaterial Material1;                  // Default constructor does nothing = fast
tgCMaterial Material2( true );          // Clears all tgCMaterial variables
tgCMaterial Material3( Material2 );     // Copies the Material2 values into Material3
```

Example code

```
// Setting up the textures for rendering with a shader
pEffect->SetTexture( "g_Colormap",      pMaterial->GetColormap()->GetD3DTexture() );
pEffect->SetTexture( "g_Normalmap",   pMaterial->GetNormalmap()->GetD3DTexture() );
pEffect->SetTexture( "g_Specularmap", pMaterial->GetSpecularmap()->GetD3DTexture() );
```

tgCMatrix

The *tgCMatrix* class the member varaibles *Left, Up,* At and Pos of the type tgCV3D and also 4 *Pads* of the type *tgFloat.*

$$\begin{pmatrix} L_x & L_y & L_z & 0 \\ U_x & U_y & U_z & 0 \\ A_x & A_y & A_z & 0 \\ P_x & P_y & P_z & 1 \end{pmatrix}$$

FUNCTION	OPERATION
Identity ()	Set the matrix to identity
Transpose()	Transpose the matrix
Rotate()	Creates a rotation matrix around the given axis
Rotate X / Y / Z ()	Creates a rotation matrix around x, y or z axis
RotateXYZ()	Creates a combined rotation matrix
RotateXYZTranslate()	Creates a combined rotation matrix and translates it
Scale()	Matrix scaling
Translate()	Matrix translation
Transform()	Matrix transformation
OrthoNormalize()	Sets the matrix vectors orthogonal and normalized
InterpolateLinear()	Linear interpolation between two matrices
InterpolateCosine()	Cosine interpolation between two matrices
LargestAxisLength()	Retrieves the length of the largest axis of this matrix
Invert()	Inverts the matrix
InvertFast()	Inverts the matrix(only works on orthogonal matrices)
Reflect()	Matrix reflection across a plane
Determinant()	Calculates the determinant
ToEuler()	Calculates Euler angles

There are three kinds of *combine types* when using matrices:

tgMATRIX_COMBINE_REPLACE – the new transform replaces the existing one.
tgMATRIX_COMBINE_PRE_MULTIPLY – the new transform will take effect before the existing one (object space).
tgMATRIX_COMBINE_POST_MULTIPLY - the new transform will take effect after the existing one (world space).

Constructors:
tgCMatrix matrix0; // Default constructor does nothing = fast

tgCMatrix matrix1(true); // Sets the matrix1 to identity matrix
tgCMatrix matrix2(matrix1); // Copies all values from matrix1 into matrix2

Note that there are two sets of the following functions:
Transpose, OrthoNormalize, Invert, InvertFast and *Reflect*.

The matrix class also have a wide selection of operator overloads

Example code

```
// rotating and translating a matrix
pMatrix->RotateX( m_Rot.x, tgMATRIX_COMBINE_REPLACE );
pMatrix->RotateY( m_Rot.y, tgMATRIX_COMBINE_PRE_MULTIPLY );
pMatrix->RotateZ( m_Rot.z, tgMATRIX_COMBINE_POST_MULTIPLY );
pMatrix->Translate( m_Pos, tgMATRIX_COMBINE_ POST_MULTIPLY );

// or the more optimized way
pMatrix->RotateXYZTranslate( m_Rot, m_Pos, tgMATRIX_COMBINE_REPLACE );
```

tgCMesh

The *tgCMesh* class contains functions for manipulating meshes.
Following functions are available:

FUNCTION	OPERATION
Render()	Renders the mesh using either the default or user specified render callback
GetTransform()	Retrieves a reference to the mesh's transform
GetRenderCallback()	Retrieves the mesh's custom render callback
SetRenderCallback()	Sets the mesh's render callback
GetInWorldSectorArray()	Retrieves a pointer to the array that holds the information on which world sectors the mesh is in
GetInWorldSector()	Retrieves a pointer to the world sector the mesh is in from the array at index
SetInWorldSector()	Set the world sector the mesh is in into the array at index
GetVertexBuffer()	Retrieves a pointer to the mesh's vertex buffer
SetVertexBuffer()	Sets the mesh's vertex buffer
GetIndexBuffer()	Retrieves a pointer to the mesh's index buffer
SetIndexBuffer()	Sets the mesh's index buffer
GetTriStripIndexBuffer()	Retrieves a pointer to the tristrip index buffer
SetTriStripIndexBuffer()	Sets the tristrip index buffer
GetParentModel()	Retrieves the model the mesh is in if its is a modelmesh
GetSubMeshArray()	Retrieves a pointer to the mesh's submesh array
GetSubMesh()	Retrieves a pointer to the mesh's submesh at index
GetVertexArray()	Retrieves a pointer to the mesh's vertex array
GetVertex()	Retrieves a pointer to the mesh's vertex at index
GetIndexArray()	Retrieves a pointer to the mesh's index array
GetIndex()	Retrieves an index to the mesh's index
GetTristripIndexArray()	Retrieves a pointer to the mesh’s tristrip index array
GetTriStripIndex()	Retrieves an index to the array index
GetTriangleNeighbourArray()	Retrieves a pointer to the mesh's triangle neighbour array
GetTriangleNeighbour()	Retrieves an index to the mesh's triangle neighbour at index
GetUserData()	Retrieves a pointer to the user data
SetUserData()	Sets the user data
GetName()	Retrieves a pointer to the mesh's name
GetAABBox()	Retrieves a reference to the mesh's bounding box
GetBSphere()	Retrieves a reference to the mesh's bounding sphere
GetNumTotalVertices()	Retrieves the mesh's total amount of vertices
GetNumTotalIndices()	Retrieves the total number of indices
GetNumTotalTriStripIndices()	Retrieves the total number of tristrip indices
GetNumTotalTriangles()	Retrieves the mesh's total amount of triangles
GetNumTotalTriStripTriangles()	Retrieves the total number of tristrip triangles

GetHierarchyID()	Retrieves the mesh's hierarchy id
GetNumSubMeshes()	Retrieves the mesh's amount of sub meshes
GetNumInWorldSectors()	Retrieves the mesh's amount of world sectors it's in
GetFlags()	Retrieves the mesh's flags
SetFlags()	Sets the mesh's flags
GetType()	Retrieves the mesh's type
GetHasAlpha()	Retrieves if the mesh has alpha or not
SetHasAlpha()	Sets if the mesh has alpha or not

Constructors

tgCMesh Mesh(); // Default constructor clears the member variables

Example code

TODO: Write an example here

tgCModel

The *tgCModel* class contains functions for manipulating models.
Following functions are available:

FUNCTION	OPERATION
Update()	Updates the models transforms and skins it
Render()	Renders the model
ClearAnimationMatrices()	Clears the model's animation matrices
SetAnimationMatrices()	Copies the matrices from the animation into the model's animation matrices
AddAnimationMatrices()	Multiplies the matrices from the animation into the model's animation matrices
GetHierarchyIDFromName()	Retrieves the object's hierarchy id using it's name
GetGroupHierarchyIDFromName()	Retrieves the group's hierarchy id using it's name
GetJointHierarchyIDFromName()	Retrieves the joint's hierarchy id using it's name
GetMeshHierarchyIDFromName()	Retrieves the mesh's hierarchy id using it's name
GetGroupIDFromName()	Retrieves the group's id using it's name
GetJointIDFromName()	Retrieves the joint's id using it's name
GetMeshIDFromName()	Retrieves the mesh's id using it's name
GetHierarchy()	Retrieves the model's hierarchy index
GetGroup()	Retrieves the model's group at index
GetJoint()	Retrieves the model's joint at index
GetMesh()	Retrieves the model's mesh at index
GetDeformationArray()	Retrieves the deformation array
GetGPUSkinArray()	Retrieves the model's deformation matrix array for GPU skinning
GetUserData()	Retrieves a pointer to the user data
SetUserData()	Sets the user data
GetName()	Retrieves the model's name
GetTransform()	Retrieves the model's transform
GetNumObjects()	Retrieves the model's amount of objects
GetNumGroups()	Retrieves the model's amount of groups
GetNumJoints()	Retrieves the model's amount of joints
GetNumMeshes()	Retrieves the model's amount of meshes
GetIsAnimated()	Retrieves if the model is animated or not
SetIsAnimated()	Sets if the model is animated or not

Constructors

tgCModel Model(); // Default constructor clears the member variables

tgCModelManager

The *tgCModelManager* class contains functions for creating and destroying *tgCModel*'s in TenGine. Following functions are available:

FUNCTION	OPERATION
Create()	Creates a *tgCModel*
Destroy()	Destroys a *tgCModel*
Save()	Saves a model to disc
GetVertexDeclaration()	Retrieves the model specific vertex declaration
GetForceMeshRenderCallback()	Retrieves the force mesh render callback
SetForceMeshRenderCallback()	Sets the force mesh render callback
GetDefaultMeshRenderCallback()	Retrieves the default mesh render callback
SetDefaultMeshRenderCallback()	Sets the default mesh render callback

Constructors

tgCModelManager ModelManager(); // Default constructor clears the member variables (called when initializing the singleton)

Render Callbacks

The default render callback in *tgCModelManager* is used if a mesh doesn't have the render callback set. Usually you will set up render callbacks for all your meshes in load-time. This could be done with some naming convention or other system created.

If you need to override all the meshes's callbacks you can set the "force mesh render callback". By doing this TenGine will skip the mesh render callback and use the force mesh render callback for all meshes. This is very handy when doing some special rendering techniques, for example rendering shadows.

Note! The force mesh render callback will not overwrite the mesh callback, when you set the force callback back to NULL all the original mesh callbacks will be used.

tgCMutex

The mutex class is for creating and using mutexes.

FUNCTION	OPERATION
Lock()	Locks the mutex
Unlock()	Unlocks the mutex

Constructors

tgCMutex Mutex; // Default constructor creates the mutex

The destructor destroys the mutex.

Example code

```
// Create the mutex
tgCMutex      Mutex;

// Lock mutex
Mutex.Lock( Timeout );

// Do things
…

// Unlock Mutex
Mutex.Unlock();
```

tgCMutexScopeLock

The *tgCMutexScopeLock* class is for automatically lock a mutex when created and automatically unlock when it goes out of scope.

Constructors

tgCMutexScopeLock MutexScopeLock(Mutex); // Default constructor locks the mutex

The destructor unlocks the mutex.

Example code

```
// Create the mutex
tgCMutex	Mutex;

// Enter new scope
if( Something )
{
		// Automatically lock mutex
		tgCMutexScopeLock	MutexScopeLock( Mutex );

		// Do things
		….

		// MutexScopeLock will go out of scope which will
		// call its destructor and unlock the mutex
}
```

tgCPlane2D

The *tgCPlane2D* class contains a normal and a distance variable.

FUNCTION	OPERATION
Clear()	Clears the member variables
Set()	Set the normal and distance to zero
Copy()	Copies a plane into this plane
Intersect()	Intersection test between plane and line
GetNormal()	Get the normal
SetNormal()	Set the normal
GetDistance ()	Get the distance
SetDistance()	Set the distance

Constructors

```
tgCPlane2D plane0;                  // Default constructor does nothing = fast
tgCPlane2D plane1( true );          // Clears the normal and distance to zero
tgCPlane2D plane2( tgCV3D(0.0f, 1.0f,), 3.0f );
                                    // Sets the normal (0.0f, 1.0f) with distance 3.0f
tgCPlane2D plane3( plane2 );        // Copies plane2 into plane3
```

The distance variable holds the value (distance) from Origin to nearest point on the plane.

tgCPlane3D

The *tgCPlane3D* class contains a normal and a distance variable.

FUNCTION	OPERATION
Clear()	Clears the member variables
Set()	Set the normal and distance to zero
Copy()	Copies a plane into this plane
Intersect()	Intersection test between plane and line
GetNormal()	Get the normal
SetNormal()	Set the normal
GetDistance ()	Get the distance
SetDistance()	Set the distance

Constructors

```
tgCPlane3D plane0;                  // Default constructor does nothing = fast
tgCPlane3D plane1( true );          // Clears the normal and distance to zero
tgCPlane3D plane2( tgCV3D(0.0f, 1.0f, 0.0f), 3.0f );
                                    // Sets the normal (0.0f, 1.0f, 0.0f) with distance 3.0f
tgCPlane3D plane3( plane2 );        // Copies plane2 into plane3
```

The distance variable holds the value (distance) from Origin to nearest point on the plane.

tgCProfiling

The *tgCProfiling* class contains functions for profiling code.
Following macros are available:

tgProfilingBegin
tgProfilingEnd
tgProfilingScope

Constructors

tgCProfiling Profiling(); // Default constructor clears the member variables (called when initializing the singleton)

Example code

```
tgProflingBegin(  )

tgProflingBegin( "LoadWorld .tfw" );

pWorld        = tgCWorldManager::GetInstance().Create( FileName );

tgProflingEnd();

// Enter new scope
if( Something )
{
                // Automatically start profile
                tgProfilingScope(  “DoThings” );

                // Do things
                ….

                // tgProfilingScope will go out of scope which will
                // call its destructor and stop the profile
}
```

tgCQuad

The *tgCQuad* class contains functions for using quads in TenGine.
Following functions are available:

FUNCTION	OPERATION
Create()	Creates the quad
Render()	Renders the quad
GetTexture()	Retrieves a pointer to the quad's texture
SetTexture()	Sets a new texture on the quad
GetVertices()	Retrieves a pointer to the quad's vertices
SetPos()	Sets the quad's transform's local matrix position
SetSize()	Sets the quad's size
SetColor()	Sets the quad's color
SetTexCoords()	Sets the quad's texture coordinates
GetName()	Retrieves a pointer to the the quad's name
GetTransform()	Retrieves a pointer to the quad's transform
PointInside()	Checks if the given point is inside the quad

Constructors

tgCQuad Quad(); // Default constructor clears the member variables

Use the tgCQuadManager to create a quad.

tgCQuadManager

The *tgCQuadManager* class contains functions for creating and destroying *tgCQuad*s in TenGine. Following functions are available:

FUNCTION	OPERATION
Create()	Creates a *tgCQuad*
Destroy()	Destroys a *tgCQuad*
GetVertexDeclaration()	Retrieves a pointer to the vertex declaration

Constructors

tgCQuadManager QuadManager(); // Default constructor clears the member variables (called when initializing the singleton)

Example code

```
// Get 2d camera width and height
const tgFloat Width          = ( tgFloat )CApplication::GetInstance().Get2DCamera()->GetCamera()->GetViewport().Width;
const tgFloat Height         = ( tgFloat )CApplication::GetInstance().Get2DCamera()->GetCamera()->GetViewport().Height;

// Creates a texture
tgCTexture*  pTexture        = tgCTextureManager::GetInstance().Create( "texture.tga" );

// Creates a quad in the center of the screen and attaches our newly created texture to it
tgCQuad*     pQuad           = tgCQuadManager::GetInstance().Create( "Quad", tgCV2D( Width * 0.5f, Height * 0.5f ), tgCV2D( Width, Height ), TG_RGBA_WHITE, pTexture );


// Render the quad
pQuad->Render();


// Destroy the quad
tgCQuadManager::GetInstance().Destroy( &pQuad );
```

tgCQuaternion

The *tgCQuaternion* class the member varaibles *x, y, z* and *w* of the type *tgFloat.*

FUNCTION	OPERATION
Identity()	Set the quaternion to identity
Rotate()	Creates a rotation matrix around the given axis
Rotate X / Y / Z ()	Creates a rotation matrix around x, y or z axis
RotateXYZ()	Creates a combined rotation matrix
Transform()	Quaternion transformation
DotProduct()	Calculates the DotProduct between 2 quaternions
Length()	Calculates the length of the quaternion
Normalize()	Normalizes the quaternion
Conjugate()	Conjugates the quaternion
Invert()	Inverts the quaternion
InterpolateLinear()	Linear interpolation between two quaternions
InterpolateSpherical()	Slerp interpolation between two quaternions
InterpolateCubic()	Cubic slerp interpolation between two quaternions
CreateFromMatrix()	Creates the quaternion from a *tgCMatrix*
GetMatrix()	Creates a *tgCMatrix* from the quaternion

There are three kinds of *combine types* when using quaternions:

tgQUATERNION_COMBINE_REPLACE – the new transform replaces the existing one.
tgQUATERNION_COMBINE_PRE_MULTIPLY – the new transform will take effect before the existing one (object space).
tgQUATERNION_COMBINE_POST_MULTIPLY - the new transform will take effect after the existing one (world space).

Constructors

tgCQuaternion quaternion0; // Default constructor does nothing = fast
tgCQuaternion quaternion1(matrix); // Creates quaternion2 from matrix
tgCQuaternion quaternion2(0.0f); // Copies value into x, y,z and w
tgCQuaternion quaternion3(0.0f, 0.0f, 0.0f, 1.0f); // Copies all values inte x, y, z and w
tgCQuaternion quaternion4(quaternion3); // Copies all values from quaternion3 into quaternion4

The quaternion class also have a wide selection of operator overloads

Example code

```
// rotating a quaternion
pQuaternion->RotateX( Angles.x, tgQUATERNION_COMBINE_REPLACE );
pQuaternion->RotateY( Angles.y, tgQUATERNION_COMBINE_PRECONCAT );
pQuaternion->RotateZ( Angles.z, tgQUATERNION_COMBINE_PRECONCAT );
```

```
// or the more optimized way
pQuaternion->RotateXYZ( Angles, tgQUATERNION_COMBINE_REPLACE );
```

tgCShader

The *tgCShader* class contains functions for manipulating shaders.
Following functions are available:

FUNCTION	OPERATION
Create()	Creates the shader
GetEffect()	Retrieves the shaders effect
GetUserData()	Retrieves a pointer to the user data
SetUserData()	Sets the user data
GetName()	Retrieves the shaders name

Constructors

tgCShader Shader(); // Default constructor clears the member variables

tgCShaderManager

The *tgCShaderManager* class contains functions for creating and destroying *tgCShader*'s in TenGine. Following functions are available:

FUNCTION	OPERATION
Exists()	Checks if a shader exists
Create()	Creates a *tgCShader*
Destroy()	Destroys a *tgCShader*
SetDefaultShader()	Sets the default shader to use

Constructors

tgCShaderManager ShaderManager(); // Default constructor clears the member variables (called when initializing the singleton)

TODO: Write about the create function

TenGine default shader supports:

Skinned animation
Normal mapping
Specular mapping
Color mapping from uvset 1 and 3 layer maps using uvset 2
Ambient, Directional, Point and Spot lights

tgCSingleton

The *tgCSingleton* class is a template class to be inherited from when making other classes into a singleton.
Following functions are available:

FUNCTION	OPERATION
Initialize()	Initializes the singleton by constructing an instance
Deinitialize()	Deinitializes the singleton by destroying the instance
GetInstance()	Retrieves a reference to the instance
GetInstancePtr()	Retrieves a pointer to the instance

Example code

```
class CApplication : public tgCSingleton< CApplication >
{
public:

		// Constructor / Destructor
		explicit CApplication	( void );
		~ CApplication		( void );
}


// Create the application
CApplication::Initialize();

if( CApplication::GetInstance().Create() )
{
		// Run the application
		CApplication::GetInstance().Run();
}

// Destroy the application
CApplication::Deinitialize();
```

tgCSortedMeshList

The *tgCSortedMeshList* class contains functions for making mesh lists and sort them by distance, for example when doing alpha sorting.
Following functions are available:

FUNCTION	OPERATION
AddMesh()	Adds a mesh to the mesh list
RemoveMesh()	Removes a mesh from the mesh list
SortFront()	Sorts the list from front to back
SortBack()	Sorts the list from back to front
Reset()	Resets the mesh list
GetSortedMesh()	Retrieves a mesh in the mesh array
GetNumMeshes()	Retrieves the amount of meshes in the mesh array

Constructors

tgCSortedMeshList SortedMeshList(); // Default constructor clears the member variables

tgCSphere

The sphere class contains a radius and position variable. All objects (meshes, world sectors and worlds) have their own bounding spheres of type *tgCSphere*.

FUNCTION	OPERATION
Clear()	Clears the member variables
Set()	Sets the member variables
Copy()	Copies data from an other sphere into this sphere
Transform()	Converts a sphere between spaces
Intersect()	Collision testing against the sphere
GetPos()	Get the sphere position
SetPos()	Set the sphere position
GetRadius()	Get the sphere radius
SetRadius()	Set the sphere radius

Constructors

```
tgCSphere sphere0;                  // Default constructor does nothing = fast
tgCSphere sphere1( true );          // Sets the pos to (0.0f, 0.0f, 0.0f) and radius to 0.0f
tgCSphere sphere2( tgCV3D( 2.0, 3.0f, 1.0f ), 5.0f );
                                    // Sets the pos to (2.0f, 3.0f, 1.0f) with radius 5.0f
tgCSphere sphere3( sphere2 );       // Copies data from sphere2 into sphere3
```

tgCSpline

The *tgCSpline* class contains functions for manipulating splines.
Following functions are available:

FUNCTION	OPERATION
AddControlPoint()	Adds a control point to the spline
RemoveControlPoint()	Removes a control point from the spline
RemoveAllControlPoints()	Removes all control points from the spline
SetControlPoint()	Moves a control point to a new position
GetControlPoint()	Retrieves a pointer to the control point
GenerateKnots()	Builds knots which is what turns the control points into a spline
GetName()	Retrieves the spline's name
SetName()	Sets the spline's name
GetNumControlPoints()	Retrieves the amount of control points on the spline
GetKnot()	Retrieves the knot
GetNumKnots()	Retrieves the amount of knots on the spline
GetDegree()	Retrieves the spline's degree
SetDegree()	Sets the spline's degree
GetLoopType()	Retrieves the spline's loop type
SetLoopType()	Sets the spline's loop type
FindPosition()	Retrieves the position on the spline at the specified fraction
FindTangent()	Retrieves the tangent on the spline at the specified fraction
CalculateRenderCurve()	Calculates debug data to be used to render the spline
GetRenderCurvePointArray()	Retrieves the debug data to render the spline
GetNumRenderCurvePoints()	Retrieves the amount of points in the curve point array

Constructors

tgCSpline Spline(); // Default constructor clears the member variables

Use tgCSplineGroup to create a spline.

tgCSplineGroup

The *tgCSplineGroup* class contains functions for creating and destroying *tgCSplines* in TenGine. Following functions are available:

FUNCTION	OPERATION
Create()	Creates a spline
Destroy()	Destroys a spline
DestroyAllSplines()	Destroys all splines in the group
GetNumSplines()	Retrieves the amount of splines in the group
GetSplineFromName()	Retrieves the spline with the specified name
GetSplineFromID()	Retrieves the spline with the specified id
GetName()	Retrieves the spline group's name
SetName()	Sets the spline group's name

Constructors

tgCSplineGroup SplineGroup(); // Default constructor clears the member variables

Use tgCSplineGroupManager to create a spline group.

tgCSplineGroupManager

The *tgCSplineGroupManager* class contains functions for creating and destroying *tgCSplineGroups* in TenGine. Following functions are available:

FUNCTION	OPERATION
CreateEmpty()	Creates an empty spline group
Create()	Creates a spline group
Destroy()	Destroys a spline group
Save()	Saves a spline group to disc

Constructors

tgCSplineGroupManager SplineGroupManager(); // Default constructor clears the member variables (called when initializing the singleton)

Example code

```
// Create the spline group
tgCSplineGroup*          pSplineGroup              =
tgCSplineGroupManager::GetInstance().Create( "splines/doublecurve.tfs" );


// Loop the splines
for( tgUInt32 SplineIndex=0; SplineIndex<pSplineGroup->GetNumSplines(); +
+SplineIndex )
{
        tgCSpline*   pSpline      = pSplineGroup->GetSplineFromID( SplineIndex );

        // Retrieve the position
        tgCV3D       Position;
        pSpline->FindPosition( Fraction, Position );

        // Retrieve the tangent
        tgCV3D       Tangent;
        pSpline->FindTangent( Fraction, Tangent );

        // Create debug items
        rDebugManager.AddLine3D( Position, Tangent, 1.0f, TG_RGBA_GREEN );
}


// Destroy the spline group
tgCSplineGroupManager::GetInstance().Destroy( &pSplineGroup );
```

tgCSprite

The *tgCSprite* class contains functions for using sprites in TenGine.
Following functions are available:

FUNCTION	OPERATION
Create()	Creates the sprite
Update()	Updates the sprite
Render()	Renders the sprite
GetTexture()	Retrieves a pointer to the sprite's texture
SetTexture()	Sets a new texture on the sprite
GetVertices()	Retrieves a pointer to the sprite's vertices
SetPos()	Sets the quad's transform's local matrix position
SetSize()	Sets the sprite's size
SetColor()	Sets the sprite's color
SetTexCoords()	Sets the sprite's texture coordinates
GetName()	Retrieves a pointer to the the sprite's name
GetTransform()	Retrieves a pointer to the sprite's transform

Constructors

tgCSprite Sprite(); // Default constructor clears the member variables

Use tgCSpriteManager to create a sprite.

tgCSpriteManager

The *tgCSpriteManager* class contains functions for creating and destroying *tgCSprites* in TenGine. Following functions are available:

FUNCTION	OPERATION
Create()	Creates a *tgCSprite*
Destroy()	Destroys a *tgCSprite*
GetVertexDeclaration()	Retrieves a pointer to the vertex declaration

Constructors

tgCSpriteManager SpriteManager(); // Default constructor clears the member variables (called when initializing the singleton)

Example code

```
// Creates a texture
tgCTexture* pTexture = tgCTextureManager::GetInstance().Create( "texture.tga" );

// Creates a sprite and attaches our newly created texture to it
tgCSprite* pSprite = tgCSpriteManager::GetInstance().Create( "Sprite",
            tgCV3D( -4.0f, 0.0f, 12.0f ), tgCV2D( 4.0f ), TG_RGBA_WHITE, pTexture );


// Render the sprite
pSprite->Render();


// Destroy the sprite
tgCSpriteManager::GetInstance().Destroy( &pSprite );
```

tgCTexture

The *tgCTexture* class contains functions for manipulating textures.
Following functions are available:

FUNCTION	OPERATION
Create()	Creates the texture
GetD3DTexture()	Retrieves the texture's d3d texture
SetD3DTexture()	Sets the texture's d3d texture
GetUserData()	Retrieves a pointer to the user data
SetUserData()	Sets the user data
GetName()	Retrieves the texture's name
GetImageInfo()	Retrieves the texture's image info
GetFilterMode()	Retrieves the texture's filter mode
SetFilterMode()	Sets the texture's filter mode
GetHasAlpha()	Retrieves if the texture has alpha
SetHasAlpha()	Sets if the texture has alpha

Constructors

tgCTexture Texture(); // Default constructor clears the member variables

TODO: Write about HasAlpha, what happens if you set it true or false.

Filtermodes

The following filter modes are available (in D3D9 version of TenGine):
D3DTEXF_NONE
D3DTEXF_POINT
D3DTEXF_LINEAR
D3DTEXF_ANISOTROPIC
D3DTEXF_PYRAMIDALQUAD
D3DTEXF_GAUSSIANQUAD

Check the header file *d3d9types.h* for more information.

tgCTextureManager

The *tgCTextureManager* class contains functions for creating and destroying *tgCTexture*'s in TenGine. Following functions are available:

FUNCTION	OPERATION
SetPath()	Sets the path's where the manager wil lsearch for textures
Exists()	Checks if a texture exists
Create()	Creates a *tgCTexture*
Destroy()	Destroys a *tgCTexture*
GetQualityReduction()	Retrieves how much texture quality should be reduced when loaded
SetQuailtyReduction()	Sets how much texture quality should be reduced when loaded
GetMipmapEnabled()	Retrieves if textures should be mipmapped when loaded
SetMipmapEnabled()	Sets if textures should be mipmapped when loaded

Constructors

tgCTextureManager TextureManager(); // Default constructor clears the member variables (called when initializing the singleton)

tgCThread

The thread class is for creating and using threads.

FUNCTION	OPERATION
Resume()	Resumes the thread
Suspend()	Suspends the thread

Constructors

tgCThread Thread(Function, THREAD_PRIORITY_NORMAL); // Default constructor creates the thread

The destructor destroys the thread.

tgCTimer

The timer class is for creating and using timers.

FUNCTION	OPERATION
Update()	Updates the timer
GetDeltaTime()	Retrieves the time elapsed between the two last updates
GetLifeTime()	Retrieves the time elapsed since timer was created
GetFramesAlive()	Retrieves the amount of frames elapsed since timer was created

Constructors

tgCTimer Timer(TRUE); // Default constructor creates the timer with high resolution

The destructor destroys the timer.

Example code

```
// Create timer
m_pTimer    = tgNew tgCTimer( TRUE );


// Update the timer
m_pTimer->Update();


// Get the deltatime
const tgFloat DeltaTime    = ( tgFloat )m_pTimer->GetDeltaTime();


// Destroy the timer
if( m_pTimer )
        tgDelete m_pTimer;
```

tgCTransform

The *tgCTransform* is used widely in TenGine. It contains two matrices: a local space matrix and a world space matrix. The transform class will also keep track of its parent and children (if it's in a hierarchy).

All objects that contain the *tgCTransform* can be linked to each other in a hierarchy. For example *tgCCamera*, *tgCModel*, *tgCLight* etc.
If you want a light to always follow the model you can use the *AddChild*() function to create a hierarchy between them.

TODO: Write about *tgCModel* class, hierarchy, groups, joints, meshes

FUNCTION	OPERATION
Clear()	Sets all members to 0.0f and NULL
Copy()	Copies a *tgCTransform*
AddChild()	Adds a child to the transform class. The in-parameter is the child.
RemoveChild()	Removes a specific child
Update()	Updates the transform and its children
GetChild()	Get a pointer to a child by given index
GetParent()	Get a pointer to the parent transform
GetUserData()	Get a pointer to the user data
SetUserData()	Set new user data
GetMatrixLocal()	Get a reference to the local transformation matrix
SetMatrixLocal()	Set a new local transformation matrix
GetMatrixWorld()	Get a reference to the world transformation matrix
SetMatrixWorld()	Set a new world transformation matrix
GetNumChildren()	Get number of children in transform hierarchy

Constructors

```
tgCTransform tf0;                  // Default constructor does nothing = fast
tgCTransform tf1( true );          // Sets the local and world matrices to identity
                                   // and sets pointers and counters to zero
tgCTransform tf2( tf1 );           // Copies tf1 into tf2
```

Update function

When the *Update*() function is executed the local matrix is multiplied with the transforms parent (if it has any) and stores the result in the world matrix. If the transform don't have a hierarchy it will copy the local matrix into the world matrix. All the children will also be updated. Remember that it's the local matrix that you shall change if you manually manipulate the matrices in the *tgCTransform*. The changes you make to the world matrix will be overwritten on next *Update*() call.

tgCTriangle

The *tgCTriangle* class contains three *tgCV3D* vertex variables.

FUNCTION	OPERATION
Clear()	Clears the member variables
Set()	Set the vertices in the triangle
Copy()	Copies a triangle into this triangle
ClosestPoint()	Calculate closest point on triangle from given point
GetVertices()	Get a pointer to the vertex array

Constructors

```
tgCTriangle triangle0;                    // Default constructor does nothing = fast
tgCTriangle triangle1( true );            // Clears the triangle vertices to zero
tgCTriangle triangle2( vertex1, vertex2, vertex3 );
                                          // Sets the three tgCV3D triangle vertices
tgCTriangle triangle3( triangle2 );       // Copies triangle2 into triangle3
```

tgCV2D

The *tgCV2d* class is a two dimensional vector class. It contains the member variables *x* and *y* of type *tgFloat*.

The following functions are provided to manipulate 2D vectors:

FUNCTION	OPERATION
DotProduct()	Calculates the dot product between vectors
Length()	Length of the vector
Between()	Check if a vector is between min/max vectors given
Normalize()	Normalize a vector
InterpolateLinear()	Linear interpolation between two 2D vectors
InterpolateCosine()	Cosine interpolation between two 2D vectors
TransformVector()	Transform a vector with a transform matrix
TransformPoint()	Transform a point with a transform matrix

Constructors:

```
tgCV2D vector0;                 // Default constructor does nothing = fast
tgCV2D vector1( 3.6f );         // Sets both vector coordinates to 3.6f
tgCV2D vector2( 4.2f, 9.7f );   // Sets the x value to 4.2f and y value to 9.7f
tgCV2D vector3( vector2 );      // Copies the vector2 values into vector3
```

The 2d vector class also have a wide selection of operator overloads

The constructors are also usable for input parameters to functions.

Instead of writing:

```
tgCV2D vector4( 3.0f, 5.5f );
pMyClass->Function( vector4 );
```

the constructor can be used:

```
pMyClass->Function( tgCV2D( 3.0f, 5.5f ) );
```

Note that there are two or more sets of the following functions:
DotProduct, *Normalize TransformVector, and TransformPoint*.

Example:

```
vector1.Normalize ();
vector2.Normalize ( vector1 );
```

The first function takes no in-parameter, normalizes vector1 and stores the normalized vector into vector1
The second function takes one in-parameters (vector1), normalizes vector1 and stores the normalized vector into vector2

tgCV3D

The *tgCV3d* class is a three dimensional vector class. It contains the member variables *x*, *y* *and z* of type *tgFloat*.

The following functions are provided to manipulate 3D vectors:

FUNCTION	OPERATION
DotProduct()	Calculates the dot product between two vectors
Length()	Length of the vector
CrossProduct()	Calculates the cross product between two vectors
Between()	Check if a vector is between min/max vectors given
Normalize()	Normalize a vector
InterpolateLinear()	Linear interpolation between two vectors
InterpolateCosine()	Cosine interpolation between two vectors
TransformVector()	Transforms a 3D vector with a transform matrix
TransformPoint()	Transforms a 3D point with a transform matrix

Constructors:

```
tgCV3D vector0;                  // Default constructor does nothing = fast
tgCV3D vector1( 3.6f );          // Sets all vector coordinates to 3.6f
tgCV3D vector2( 4.2f, 9.7f, 6.6f );  // Sets the x = 4.2f y = 9.7f, z = 6.6f
tgCV3D vector3( vector2 );       // Copies the vector2 values into vector3
```

The 3d vector class also have a wide selection of operator overloads

The constructors are also usable for input parameters to functions.

Instead of writing:

```
tgCV3D vector5( 3.0f, 5.5f, 1.0f );
pMyClass->Function( vector5 );
```

the constructor can be used:

```
pMyClass->Function( tgCV3D( 3.0f, 5.5f, 1.0f ) );
```

Note that there are two sets of the following functions:
DotProduct, *Normalize*, *TransformVector* and *TransformPoint*.

Example:

```
vector1.Normalize ();
vector2.Normalize ( vector1 );
```

The first function takes no in-parameter, normalizes vector1 and stores the normalized vector into vector1
The second function takes one in-parameters (vector1), normalizes vector1 and stores the normalized vector into vector2

tgCV4D

The *tgCV4d* class is a four dimensional vector class. It contains the member variables *x*, *y*, *z and w* of type *tgFloat*.

The following functions are provided to manipulate 4D vectors:

FUNCTION	OPERATION
DotProduct()	Calculates the dot product between two vectors
Length()	Length of the vector
Between()	Check if a vector is between min/max vectors given
Normalize()	Normalize a vector
InterpolateLinear()	Linear interpolation between two vectors
InterpolateCosine()	Cosine interpolation between two vectors
TransformPoint()	Transforms a 4D point with a transform matrix

Constructors:
tgCV4D vector0; // Default constructor does nothing = fast
tgCV4D vector1(3.6f); // Sets all vector coordinates to 3.6f
tgCV4D vector2(4.2f, 9.7f, 6.6f, 2.2f); // Sets x = 4.2f, y = 9.7f, z = 6.6f, w = 2.2f
tgCV4D vector3(vector2); // Copies the vector2 values into vector3

The 4d vector class also have a wide selection of operator overloads

The constructors are also usable for input parameters to functions.

Instead of writing:
tgCV4D vector5(3.0f, 5.5f, 1.0f, 4.8f);
pMyClass->Function(vector5);

the constructor can be used:
pMyClass->Function(*tgCV4D*(3.0f, 5.5f, 1.0f, 4.8f));

Note that there are two sets of the following functions:
DotProduct, *Normalize* and *TransformPoint*.

Example:
vector1.Normalize ();
vector2.Normalize (vector1);

The first function takes no in-parameter, normalizes vector1 and stores the normalized vector into vector1
The second function takes one in-parameters (vector1), normalizes vector1 and stores the normalized vector into vector2

tgCWorld

The *tgCWorld* class contains functions for manipulating worlds.
Following functions are available:

FUNCTION	OPERATION
Update()	Updates the world
AddModel()	Adds a model to the world's model mesh list
RemoveModel()	Removes a model from the world's model mesh list
GetSector()	Retrieves a world sector
GetRenderSector()	Retrieves a world render sector
GetCameraSector()	Retrieves a world camera sector
GetUserData()	Retrieves a pointer to the user data
SetUserData()	Sets the user data
GetAlphaMeshList()	Retrieves the world's alpha mesh list
GetSolidMeshList()	Retrieves the world's solid mesh list
GetModelMeshList()	Retrieves the world's model mesh list
GetName()	Retrieves the world's name
GetAABBox()	Retrieves the world axis aligned bounding box
GetBSphere()	Retrieves the world's bounding sphere
GetNumSectors()	Retrieves the world's amount of world sectors
GetNumRenderSectors()	Retrieves the world's amount of world render sectors
GetNumCameraSectors()	Retrieves the world's amount of world camera sectors

Constructors

tgCWorld World(); // Default constructor clears the member variables

TODO: Write about why should add the model to the world.

- **Write about Alpha mesh list, solid mesh list**
- **Write about Render sectors and camera sectors**

tgCWorldManager

The *tgCWorldManager* class contains functions for creating and destroying *tgCWorld*'s in TenGine. Following functions are available:

FUNCTION	OPERATION
CreateEmpty()	Creates an empty *tgCWorld*
Create()	Creates a *tgCWorld*
Destroy()	Destroys a *tgCWorld*
Save()	Saves a world to disc
GetForceMeshRenderCallback()	Retrieves the force mesh render callback
SetForceMeshRenderCallback()	Sets the force mesh render callback
GetDefaultMeshRenderCallback()	Retrieves the default mesh render callback
SetDefaultMeshRenderCallback()	Sets the default mesh render callback

Constructors

tgCWorldManager WorldManager(); // Default constructor clears the member variables (called when initializing the singleton)