r3dRenderLayer

Description

r3dRenderLayer is a wrapper class, that makes communication with Direct3D more convenient.

Associated code

r3dRenderLayer	Direct3D wrapper class
r3dRenderLayer::DeviceWrapper	IDirect3DDevice wrapper that does threading checks in non-final build.
r3dIResource	Base class for classes, objects of which contain *D3D_POOL_DEFAULT* related resources.
r3dGPUStats	Class that contains GPU rendering statistics
r3dDeviceInfo	Class that contains various D3D device capabilities and outputs them to file.
r3dDeviceTunnel	Class that helps tunnel IDirect3DDevice9 related calls into the rendering thread. This done during resource loading in separate thread. Please see Multithreading document for detailed description.
r3dD3DSurfaceTunnel	Helper classes that help tunnel respective IDirect3D* calls into the rendering thread. Please see Multithreading document for detailed description.
r3dD3DTextureTunnel	
r3dD3DIndexBufferTunnel	
r3dD3DVertexBufferTunnel	
r3dD3DResourceTunnel	

Associated source files

r3dRender.h	`class r3dRenderLayer` and related classes header.
r3dRender.cpp	`class r3dRenderLayer` and related classes implementation.

class r3dRenderLayer

Description

A wrapper class, that makes communication with Direct3D more convenient.

Notable public fields

Name	Type	Description
HLibWin	*HWND*	Handle to the window which was used during D3D

		initialization
`pd3d`	`IDirect3D9`	Pointer to globally shared `IDirect3D9`
`d3dpp`	`D3DPRESENT_PARAMETERS`	`D3DPRESENT_PARAMETERS` that was used during device creation
`pd3ddev`	`DeviceWrapper` or `IDirect3DDevice9` depending on build configuration	Essentially, a pointer to globally shared `IDirect3D9Device9.` In non-final configurations, this is a wrapper `DeviceWrapper,` which performs threading checks as `IDirect3D9Device9` gets processed.
`d3dCaps`	`D3DCAPS9`	Capabilities of the device created.
`CameraPosition`	`r3dPoint3D`	Position of the camera currently in use.
`ViewMatrix`	`D3DXMATRIXA16`	View matrix of the camera currently in use.
`InvViewMatrix`	`D3DXMATRIXA16`	Inverse of the view matrix of the camera currently in use.
`ProjMatrix`	`D3DXMATRIXA16`	Projection matrix of the camera currently in use.
`ViewProjMatrix`	`D3DXMATRIXA16`	Product of view and projection matrices of the camera currently in use.
`InvProjMatrix`	`D3DXMATRIXA16`	Inverse of the projection matrix of the camera currently in use.
`FrustumCorners`	`D3DXVECTOR3[8]`	World space corners of the view frustum of the camera currently in use.
`FrustumPlanes`	`D3DXPLANE[6]`	World space planes of the view frustum of the camera currently in use.
`NearClip`	`float`	Near clipping plane distance of the camera currently in use.
`FarClip`	`float`	Far clipping plane distance of the camera currently in use.
`CurrentBPP`	`int`	Back buffer bits per pixel
`ScreenW`	`float`	Current render target width
`ScreenH`	`float`	Current render target height
`ScreenW2`	`float`	Half of current render target's width
`ScreenH2`	`float`	Half of current render target's height
`ViewX`	`float`	Current viewport's X (Left)
`ViewY`	`float`	Current viewport's Y (Top)
`ViewW`	`float`	Current viewport's width
`ViewH`	`float`	Current viewport's height
`ViewMinZ`	`float`	Current viewport's minimum z value
`ViewMaxZ`	`float`	Current viewport's maximum z value
`Fog`	`r3dFogState`	Fog parameters for current frame
`AmbientColor`	`r3dColor`	Current ambient color
`Stats`	`r3dGPUStats`	GPU rendering stats
`VertexShaderProfileName`	`char[16]`	Vertex shader profile name, e.g. `"vs_2_0"`, `"vs_3_0"`
`PixelShaderProfileName`	`char[16]`	Pixel shader profile name, e.g. `"ps_2_0"`, `"ps_3_0"`
`CurrentPixelShaderID`	`int`	ID of currently set pixel shader
`CurrentVertexShaderID`	`int`	ID of currently set vertex shader
`SupportsVertexTextureFetch`	`int`	Non-zero value in case d3d device supports vertex texture fetch.
`SupportsR32FBlending`	`int`	Non-zero value in case blending of D3DFMT_R32F render target is supported
`SupportsEventQueries`	`int`	Non-zero value in case `D3DQUERYTYPE_EVENT`

		queries are supported
SupportsStampQueries	*int*	Non-zero value in case stamp queries are supported
SupportsNULLRenderTarget	*int*	Non-zero value in case null render targets are supported. Null render targets allow filling depth stencil render targets without memory consuming render targets of appropriate dimensions being set.
SupportsHWShadowMapping	*int*	Non-zero value in case hardware shadow mapping is supported
SupportsRESZResolve	*int*	Non-zero value in case resolve of RESZ surfaces is supported
SupportsINTZTextures	*int*	Non-zero value in case INTZ textures are supported
DeviceAvailable	*int*	Non-zero value in case device is available for rendering (not lost).
ShadowPassType	*ShadowPassTypeEnum*	Determines which type of shadow pass currently takes place. Can be either *SPT_ORTHO* or *SPT_PROJ* . *SPT_ORTHO* indicates that shadow map for directional light is being constructed *SPT_PROJ* indicates that shadow map for point or spot light is being rendered. For point light, this means that one of the cubemap faces is being constructed.
DefaultCullMode	*D3DCULL*	Defines which cull mode is currently considered default.
CurrentCullMode	*D3DCULL*	Defines, which cull mode is currently set.

Important methods

```
bool Reset()
```

Summary:

Performs *D3D_POOL_DEFAULT* resource deallocation, calls *IDirect3DDevice9::Reset*, then creates and fills *D3D_POOL_DEFAULT* resources again.

Return value:

true if operation was successful, **false** otherwise.

```
int Init(HWND hWindow, const char* displayName)
```

Summary:

Initializes r3dRenderLayer, creates *IDirect3D9* instance.

Parameters:

`hWindow`	– handle to window to render into
`displayName`	– currently ignored

Return value:

Non-zero value if successful, zero value otherwise.

`int   Close()`

Summary:

Destroys all allocated resources.

Return value:

Non-zero value if successful, zero value otherwise.

`int SetMode(int xRes, int yRes, int bpp, int flags, int renderPath)`

Summary:

Adjusts render window size and changes video mode if required.

Parameters:

`xRes`	– New back buffer X resolution
`yRes`	– New back buffer Y resolution
`bpp`	– New back buffer bits per pixel
`flags`	– if *R3DSetMode_Windowed* bit is set, windowed mode is setup. Otherwise, full screen mode is set up.
`renderPath`	– currently unused.

Return value:

Non-zero value if successful, zero value otherwise.

`void  SetViewport ( float x, float y, float w, float h )`

Summary:

Sets new viewport to render into.

Parameters:

x	– new viewport X start
y	– new viewport Y start
w	– new viewport Width
h	– new viewport Height

```
void  ResetViewport()
```

Summary:

Resets the viewport to full current render target.

```
void ResetStats()
```

Summary:

Resets *r3dGPUStats* structure to start statistics for a new frame.

```
void  StartRender(int bClear, bool clearToWhite)
```

Summary:

Resets textures, material, shaders, vertex declaration and viewport and clears back buffer if requested.

Parameters:

bClear	– indicates if current renter and depth stencil targets are to be cleared
clearToWhite	– indicates if the render targets is to be cleared with white color (**true**) or with fog color (**false**)

```
void EndRender( bool present )
```

Summary:

Draws the contents of dynamic geometry buffers and flushes them. Processes the contents of device queue to execute direct3d commands scheduled from non-rendering thread. In case *present* is set to **true,** presents the contents of the back buffer.

Parameters:

present	– **true** if contents of the babk buffer is to be presented (end of rendering frame is to occur).

```
void  StartFrame()
```

Summary:

Resets render target and depth stencil to back buffer render target and depth stencil. Resets D3D state caches, calls *IDirect3DDevice9::BeginScene*

`void  EndFrame()`

Summary:

If there's a pending query scheduled, issues the query. Calls *IDirect3DDevice9::EndScene* , performs threading checks.

`void  SetBackBufferViewport()`

Summary:

Sets the viewport that matches back buffer dimmensions and z range [0..1]

`void  StartRenderSimple(int bClear);`

Summary:

Clears back buffer in case `bClear` is set to non zero value. Resets the viewport to much the whole active render target.

Parameters:

`bClear`	– non zero value in case render target is to be cleared with fog color and depth stencil target is to be cleared with (1.f, 0)

`void EndRenderSimple( bool present )`

Summary:

In case *present* is set to **true** , presents the back buffer contents.

Parameters:

`present`	– **true** if contents of the back buffer needs to be presented

---

`void EndRenderSimple( bool present )`

Summary:

In case *present* is set to **true** , presents the back buffer contents.

Parameters:

present	– **true** if contents of the back buffer needs to be presented

---

```
void  SetCamera( const r3dCamera &cam );
```

Summary:

Sets new camera and recalculates all camera related public fields. Following is the list of recalculated fields:

CameraPosition
ViewMatrix,
InvViewMatrix,
ProjMatrix,
ViewProjMatrix,
InvProjMatrix,
FrustumCorners,
FrustumPlanes,
NearClip,
FarClip

Parameters:

cam	– new camera to set

```
void SetCameraEx( const D3DXMATRIX& view, const D3DXMATRIX& proj, const
                  r3dPoint3D& camPos, float nearD, float farD)
```

Summary:

Sets camera related variables to custom values. The following variables also get recalculated:

InvViewMatrix,
ViewProjMatrix,
InvProjMatrix,
FrustumCorners,
FrustumPlanes,

Parameters:

view	– new view matrix
proj	– new projection matrix
camPos	– new camera position
nearD	– new near clipping distance
farD	– new far clipping distance

```
int   AddPixelShaderFromFile( const char* shaderName,
                              const char* fileName,
                              int bSystem )
```

Summary:

Tries loading compiled pixel shader from cache. If there's no cached version of the shader, compiles it and stores to the cache. Returns the new ID assigned to the pixel shader.

Parameters:

shaderName	–	name of the shader to use in the r3dRenderLayer
fileName	–	file path to the shader
bSystem	–	if newly created shader is to be assigned 'system' attribute

Return value:

ID of the compiled/loaded shader.

```
int   AddPixelShaderFromFile( const char* shaderName, const char* fileName, int
                              bSystem, const ShaderMacros & defines)
```

Summary:

Tries loading compiled pixel shader from cache. If there's no cached version of the shader, compiles it and stores to the cache. Compilation is done with shader *defines*. Returns the new ID assigned to the pixel shader.

Parameters:

shaderName	–	name of the shader to use in the r3dRenderLayer
fileName	–	file path to the shader
bSystem	–	if newly created shader is to be assigned 'system' attribute
defines	–	an array of defines to use when compiling the shader.

Return value:

ID of the compiled/loaded shader.

```
void  ReloadShaderByFileName(const char* fileName)
```

Summary:

Reloads (recompiles) all shaders which were created from file *fileName*. There may be several such shaders because of different defines which may be used with the same shader file.

Parameters:

fileName	–	file name of the shaders to recompile and reload

```
int    AddVertexShaderFromFile(        const char* shaderName,
                                       const char* fileName,
                                       int bSystem )
```

Summary:

Tries loading compiled vertex shader from cache. If there's no cached version of the shader, compiles it and stores to the cache. Returns the new ID assigned to the vertex shader.

Parameters:

shaderName	– name of the shader to use in the r3dRenderLayer
fileName	– file path to the shader
bSystem	– if newly created shader is to be assigned 'system' attribute

Return value:

ID of the compiled/loaded shader.

```
int    AddVertexShaderFromFile(        const char* shaderName, const char*
                                       fileName, int bSystem,
                                       const ShaderMacros & defines)
```

Summary:

Tries loading compiled vertex shader from cache. If there's no cached version of the shader, compiles it and stores to the cache. Compilation is done with shader *defines*. Returns the new ID assigned to the vertex shader.

Parameters:

shaderName	– name of the shader to use in the r3dRenderLayer
fileName	– file path to the shader
bSystem	– if newly created shader is to be assigned 'system' attribute
defines	– an array of defines to use when compiling the shader.

Return value:

ID of the compiled/loaded shader.

```
int    GetShaderIdx(const char* name)
```

Summary:

Searches for shader among pixel and vertex shaders with the name *name*. If no shader is found, returns -1. Otherwise returns the found shader’s ID

Parameters:

name – the name of the shader to look for

Return value:

ID of the pixel or vertex shader with the name *name*. -1 if no such shader was found.

```
int    GetPixelShaderIdx( const char* name )
```

Summary:

Searches for shader among pixel shaders with the name *name*. If no shader is found, returns -1. Otherwise returns the found pixel shader’s ID

Parameters:

name – the name of the pixel shader to look for

Return value:

ID of the pixel shader with the name *name*. -1 if no such shader was found.

```
int    GetVertexShaderIdx( const char* name )
```

Summary:

Searches for shader among vertex shaders with the name *name*. If no shader is found, returns -1. Otherwise returns the found vertex shader’s ID

Parameters:

name – the name of the vertex shader to look for

Return value:

ID of the vertex shader with the name *name*. -1 if no such shader was found.

```
void   SetPixelShader(const char* name)
```

Summary:

Sets pixel shader by name *name*.

Parameters:

`name` – the name of the pixel shader to set

```
void  SetPixelShader(int id)
```

Summary:

Sets pixel shader with id (array index) *id* . *id* may be equal to -1, in which case pixel shader is removed and fixed function pipeline is used for pixel calculations.

Parameters:

`id` – id (index array) of the pixel shader to set

```
void  SetVertexShader(const char* name)
```

Summary:

Sets vertex shader by name *name*.

Parameters:

`name` – the name of the vertex shader to set

```
void  SetVertexShader(int id)
```

Summary:

Sets vertex shader with id (array index) *id* . *id* may be equal to -1, in which case vertex shader is removed and fixed function pipeline is used for vertex calculations.

Parameters:

`id` – id (index array) of the vertex shader to set

```
void  SetDefaultCullMode( D3DCULL cullMode )
```

Summary:

Sets cull mode from now one to be used as default one. D3D render state also gets updated.

Parameters:

`cullMode` – new cull mode to use as the default one and to set D3D state to.

`void SetCullMode( D3DCULL cullMode )`

Summary:

Set D3D cull mode to *cullMode*

Parameters:

`cullMode` – new cull mode to set D3D state to.

`void RestoreCullMode()`

Summary:

Restores cull mode to its default value (set by `SetDefaultCullMode` function).

`int GetCurrentPixelShaderIdx() const`

Summary:

Returns the ID of the pixel shader which is currently set.

Return value:

Id of the pixel shader currently set, -1 if no shader is currently in use.

`int GetCurrentVertexShaderIdx() const`

Summary:

Returns the ID of the vertex shader which is currently set.

Return value:

Id of the vertex shader currently set, -1 if no vertex shader is currently in use.

`int IsTextureFormatAvailable(D3DFORMAT fmt)`

Summary:

Returns non zero value if format *fmt* is supported on current device and settings.

Return value:

Non-zero value if format is supported, 0-value otherwise.

```
void  SetMipMapBias(float bias, int stage )
```

Summary:

Sets mip map bias *bias* for texture stage *stage*. In case *stage* equals to -1 , propagates the mip map bias to all stages.

Parameters:

bias	– mip map bias to set.
stage	- texture stage to apply mip map bias to. -1 in case all stages have to be affected.

```
r3dTexture* AllocateTexture()
```

Summary:

Allocates texture in the internal texture array. The internal contents of the texture is not created. Texture dimensions, format and other parameters are later specified by the user via *r3dTexture* methods

Return value:

Pointer to the newly allocated texture.

```
r3dTexture* LoadTexture(      const char* texFile, D3DFORMAT texFormat,
                              bool bCheckFormat, int downScale,
                              int downScaleMinDim, int systemMem )
```

Summary:

Searches for the texture that was loaded from file `texFile` in the texture cache. In case no instance of the texture is found in the cache, allocates the new texture. Fills the contents of the texture according to function parameters.

Parameters:

texFile	– file name of the texture to load
texFormat	– desired format of the texture. Specify *D3DFMT_UNKNOWN* . To retain format of the file
bCheckFormat	– if this parameter is set to **true,** when search in the texture cache is performed the successful find is accomplished only if the texture format matches the *texFormat* argument
downScale	– a divisor of texture dimmensions when new texture is loaded.
downScaleMinDim	– minimum allowed texture dimmension when downscaling. DXT compression formats suggest the minimum dimmension of 4.
systemMem	– non zero value in case the texture is to be loaded into the system memory (as opposed to video memory)

Return value:

Pointer to the newly allocated and loaded texture.

```
void  ReloadTextureData( const char * fileName )
```

Summary:

Reloads all textures loaded from file *fileName*. Several instances may be loaded, in spite of the caching, because different formats have been specified to *LoadTexture* with *bCheckFormat* being set to **true.**

Parameters:

fileName	– file name of the textures to reload

```
void  DeleteTexture(r3dTexture *tex, int bForceDelete)
```

Summary:

Decreases the reference count of the texture *tex* . In case the reference counter reaches 0, deletes the texture. If *bForceDelete* is set to non-zero value, deletes the texture even if its reference counter is non-zero.

Parameters:

tex	– texture to delete from the cache
bForceDelete	– whether to account for texture's reference counter when deleting the texture.

```
void   SetRenderingMode(int mode)
```

Summary:

Sets combination of render states according to flags specified in *mode*. The flags are listed in the following table.

R3D_BLEND_NZ	No Z, *D3DRS_ZENABLE* is set to **false**.
R3D_BLEND_ZC	Z Check enable, *D3DRS_ZENABLE* is set to **true**.
R3D_BLEND_ZW	Z Write enable, *D3DRS_ZWRITEENABLE* is set to **true**. Without this flag, it is set to **false.**
R3D_BLEND_NOALPHA	Alpha blending disabled, *D3DRS_ALPHABLENDENABLE* is set to **false.**
R3D_BLEND_ALPHA	Alpha blending enabled, *D3DRS_ALPHABLENDENABLE* is set to **true.** *D3DRS_SRCBLEND* is set to *D3DBLEND_SRCALPHA* *D3DRS_DESTBLEND* is set to *D3DBLEND_INVSRCALPHA*
R3D_BLEND_ADD	*D3DRS_ALPHABLENDENABLE* is set to **true.** *D3DRS_SRCBLEND* is set to *D3DBLEND_ONE* *D3DRS_DESTBLEND* is set to *D3DBLEND_ONE*
R3D_BLEND_SUB	*D3DRS_ALPHABLENDENABLE* is set to **true.** *D3DRS_SRCBLEND* is set to *D3DBLEND_ZERO* *D3DRS_DESTBLEND* is set to *D3DBLEND_INVSRCCOLOR*

R3D_BLEND_MODULATE	*D3DRS_ALPHABLENDENABLE* is set to **true.** *D3DRS_SRCBLEND* is set to *D3DBLEND_ZERO* *D3DRS_DESTBLEND* is set to *D3DBLEND_SRCCOLOR*
R3D_BLEND_ADDMODULATE	*D3DRS_ALPHABLENDENABLE* is set to **true.** *D3DRS_SRCBLEND* is set to *D3DBLEND_DESTCOLOR* *D3DRS_DESTBLEND* is set to *D3DBLEND_SRCCOLOR*
R3D_BLEND_COLOR	*D3DRS_ALPHABLENDENABLE* is set to **true.** *D3DRS_SRCBLEND* is set to *D3DBLEND_SRCCOLOR* *D3DRS_DESTBLEND* is set to *D3DBLEND_INVSRCCOLOR*

Use the following flags to perform additional actions.

R3D_BLEND_PUSH	Push previous states into state stack before applying new ones
R3D_BLEND_POP	Pop states from states stack

Parameters:

mode – combination of flags described above

```
void  ResetMaterial()
```

Summary:

Sets NULL texture for texture stages 0..7, removes pixel and vertex shaders.

```
void  SetMaterial(r3dMaterial *mat)
```

Summary:

Makes material *mat* active.

Parameters:

mat – new material to set.

```
void  SetTex(r3dTexture *tex, int stageID )
```

Summary:

Sets texture *tex* at stage *stageID*

Parameters:

tex	– new texture to set
stageID	- new stage to set texture to.

```
void  SetRT( int slot, IDirect3DSurface9* surf)
```

Summary:

Sets new render target at slot *slot*.

Parameters:

surf	– new render target surface to set.
slot	- slot to set new render target to.

```
void  SetDSS( IDirect3DSurface9* dss )
```

Summary:

Sets new render target at slot *slot*.

Parameters:

surf	– new render target surface to set.
slot	- slot to set new render target to.

```
void  SetFog(int fogEnabled)
```

Summary:

Enable or disable fog via D3D render states.

Parameters:

fogEnabled	– non-zero value if the fog is to be enabled

```
void  StretchRect(r3dScreenBuffer* source,
                  r3dScreenBuffer* target, int filter )
```

Summary:

Calls IDirect3DDevice9::StretchRect function to stretch for render target *source* to render target *target*. Apply *D3DTEXF_LINEAR* filter in case *filter* is non-zero. Otherwise apply no filter.

Parameters:

source	– source render target to copy from
target	– target render target to copy to
filter	– non zero value in case linear filtering should be used when stretching.

```
void  Flush()
```

Summary:

Renders all geometry accumulated in dynamic geometry buffers and flushes the buffers.

```
void Render2DPolygon(int numV, R3D_SCREEN_VERTEX *v)
```

Summary:

Append an array of screen vertices pointed to by *v*, with vertex count *numV* to dynamic geometry buffer for 2D triangles.

Parameters:

numV	–	the number of vertices to append
v	–	pointer to the beginning of the array of vertices.

```
void Begin2DPolygon(int numVertsInPoly, int numPoly)
```

Summary:

Reserves space in dynamic geometry buffer for *numPoly* polygons each of which has `numVertsInPoly` vertices. Should be paired with `End2DPolygon` call.

Parameters:

numVertsInPoly	–	number of vertices in each polygon
numPoly	–	total number of polygons

```
void Fill2DPolygon(int numV, R3D_SCREEN_VERTEX *v)
```

Summary:

Appends `numV` vertices to dynamic geometry buffer for 2D triangles, pointed to by *v*. Within space allocated by `Begin2DPolygon`. After all vertices have been appended, `End2DPolygon` call should be made. Each `Begin2DPolygon/End2DPolygon` does single vertex buffer lock.

Parameters:

numV	–	number of vertices to append
v	–	pointer to the array of vertices

```
void End2DPolygon()
```

Summary:

Finalizes vertices appending to the dynamic geometry buffer for 2D triangles rendering. Unlocks the dynamic vertex buffer. This call should be paired with *`Begin2DPolygon`*.

```
void Render3DPolygon (int numV, R3D_DEBUG_VERTEX *v)
```

Summary:

Appends *numV* vertices to the dynamic geometry buffer for rendering 3D tirangles. The vertices to append are pointed to by *v*.

Parameters:

`numV`	–	the number of vertices to append
`v`	–	pointer to the beginning of the array of vertices.

```
void  Render3DLine(int numV, R3D_DEBUG_VERTEX *v)
```

Summary:

Appends *`numV`* vertices to dynamic geometry buffer for rendering 3D lines. The vertices to append are pointed to by *v*. The vertices should constitute a line list.

Parameters:

`numV`	–	the number of vertices to append
`v`	–	pointer to the beginning of the array of vertices.

```
void  BeginFill3DLine(int numV)
```

Summary:

Reserves space in dynamic geometry buffer for *`numV`* 3D lines. Should be paired with *`EndFill3DLine`* call. The vertices should constitute a line list.

Parameters:

`numV`	–	dynamic number of vertices in lines to append.

```
void  Fill3DLine(int numV, R3D_DEBUG_VERTEX *v)
```

Summary:

Appends 3D line vertices pointed to by *v*, numbered *numV*, to dynamic geometry buffer for 3D lines. This should be done within *`BeginFill3DLine/EndFill3DLine`* pair. Several calls may be made, but total amount of vertices should not surpass the reserved amount.

Parameters:

numV – the number of vertices to append
v – pointer to the beginning of the array of vertices.

`void EndFill3DLine()`

Summary:

Finalizes vertices appending to the dynamic geometry buffer for 3D lines rendering. Unlocks the dynamic vertex buffer. This call should be paired with *`BeginFill3DLine`*.

`void Render3DTriangles(int numV, R3D_DEBUG_VERTEX *v)`

Summary:

Appends *`numV`* vertices to dynamic geometry buffer for rendering 3D triangles. The vertices to append are pointed to by *v*. The vertices should constitute a triangle list.

Parameters:

numV – the number of vertices to append
v – pointer to the beginning of the array of vertices.

`int IsBoxInsideFrustum(const r3dBoundBox& bbox)`

Summary:

Determines if the bounding box *bbox* is inside the view frustum of the current camera.

Parameters:

bbox – the bounding box to check against the current view frustum.

Return value:

0 – if the box is fully outside of the frustum
1 – if the box is fully inside the frustum
2 – if the box is partially inside the frustum (visible)

`int IsSphereInsideFrustum(const r3dPoint3D& c, float r)`

Summary:

Determines if the sphere centered in *c*, and with radius *r*, is inside the current view frustum.

Parameters:

c – center of the sphere
r – radius of the sphere

Return value:

0 - if the sphere is fully outside the frustum
1 – if the sphere is fully inside the frustum
2 – if the sphere is partially inside the frustum (visible)

```
bool  DoesBoxIntersectNearPlane (const r3dBoundBox& bbox)
```

Summary:

Determines if the bounding box *bbox* intersects the current near plane.

Parameters:

bbox	- bounding box to check for intersection against the near plane.

Return value:

true if the box intersects the near plane, **false** otherwise

---

```
bool  IsDeviceLost() const
```

Summary:

Allows to retrieve the D3D device lost state variable.

Return value:

true if it has been detected that the device is lost, **false** otherwise

---

```
int   TryToRestoreDevice()
```

Summary:

Tries to restore the device in case it has been lost.

Return value:

non-zero value in case the device has been restored or it is already in non-lost state, 0-value otherwise.

```
void  UpdateSettings()
```

Summary:

Updates the device settings according to new resolution. Performs D3D_POOL_DEFAULT deallocation, device reset, then allocates and fills D3D_POOL_DEFAULT resources back.

```
void  UpdateDimmensions()
```

Summary:

Updates the settings of the viewport, which defines the main rendering area of the back buffer. It is possible to setup the engine to render in ceratin part of the back buffer in order to maintain desired aspect ratio. *UpdateDimmenions* call updates the related engine variables when resolution and/or aspect ratio settings have been changed.

```
void  ChangeForceAspect( float val )
```

Summary:

Changes the forced aspect ratio value. If *val* is equal to 0, no aspect ratio forcing takes place. In case val is non-zero, the value is used as the new force aspect ratio value. Some parts of the screen may be not rendered into as result (left black).

Parameters:

val - value of the force aspect ratio to set (0 to remove aspect ratio forcing)

```
DisplayResolutions GetDisplayResolutions() const
```

Summary:

Retrieves available display resolutions.

Return value:

The array of available display resolutions.

```
void DrawIndexed( D3DPRIMITIVETYPE type, INT baseVertexIndex,
                  UINT minVertexIndex, UINT numVertices, UINT startIndex,
                  UINT primCount )
```

Summary:

This is a wrapper around similar D3D call, which may perform certain engine specific tasks (e.g. D3D profiling calls) in the process.

Parameters:

Similar to those of IDirect3DDevice9::DrawIndexedPrimitive

```
void DrawIndexedUP(      D3DPRIMITIVETYPE primitiveType, UINT minVertexIndex,
                         UINT numVertices, UINT primitiveCount,
                         CONST void* pIndexData, D3DFORMAT indexDataFormat,
                         CONST void* pVertexStreamZeroData,
                         UINT vertexStreamZeroStride )
```

Summary:

This is a wrapper around similar D3D call, which may perform certain engine specific tasks (e.g. D3D profiling calls) in the process.

Parameters:

Similar to those of IDirect3DDevice9::DrawIndexedPrimitiveUP

```
void Draw(  D3DPRIMITIVETYPE primitiveType,UINT startVertex,
            UINT primitiveCount )
```

Summary:

This is a wrapper around similar D3D call, which may perform certain engine specific tasks (e.g. D3D profiling calls) in the process.

Parameters:

Similar to those of IDirect3DDevice9::DrawPrimitive

```
void DrawUP (     D3DPRIMITIVETYPE primitiveType, UINT primitiveCount,
                  CONST void* pVertexStreamZeroData,
                  UINT vertexStreamZeroStride )
```

Summary:

This is a wrapper around similar D3D call, which may perform certain engine specific tasks (e.g. D3D profiling calls) in the process.

Parameters:

Similar to those of IDirect3DDevice9::DrawPrimitiveUP

```
void MarkEssentialD3DCommand()
```

Summary:

Mark essential D3D commands in order to force time stamp queries to be executed. Normally, essential D3D commands are considered to be Draw***Primitive*** calls. This are taken care of by the engine. Any other essential calls should be marked by calling this function.

```
void CheckOutOfMemory( HRESULT hr )
```

Summary:

Checks the result *hr* to be out of memory error. If this is the case, certain actions, to allow the user to successfully launch the game next time, are performed.

Parameters:

`hr` - *`HRESULT`* to check.
