Multithreading

Description

Multithreading is utilized in following two processes:

1) Game loading.

Game loading occurs in one thread, while loading screen rendering occurs in another. Loading screen update supports playback of flash animation.

2) Game update and rendering.

Game update and rendering works in single threaded model. However, a number of tasks that are naturally done in parallel are distributed evenly to all available hardware threads. Management of this distribution is performed by class ***JobChief.***

Associated classes and structures

JobChief	Class that handles task distribution across available hardware threads.
r3dD3DVertexBufferTunnel	IDirect3DVertexBuffer9 wrapper class that delegates vertex buffer lock/unlock functions to the rendering thread to support multithreaded loading and rendering
r3dD3DIndexBufferTunnel	IDirect3DIndexBuffer9 class wrapper that delegates index buffer lock/unlock functions to the rendering thread to support multithreaded loading and rendering
r3dD3DTextureTunnel	IDirect3DTexture9 class wrapper that delegates texture lock/unlock functions to the rendering thread to support multithreaded loading and rendering
r3dDeviceTunnel	IDirect3DDevice9 class wrapper that delegates resource creation functions to the rendering thread to support loading and rendering

Associated Source Files

JobChief.h	JobChief class header
JobChief.cpp	JobChief class implementation
r3dRender.h	Among other things, contains declarations of **Tunnel* classes
r3dRender.cpp	Among other things, contains definition of **Tunnel* classes' methods
r3dDeviceQueue.h	Contains declarations of functions which allow delegating IDirect3DDevice9 associated calls to the rendering thread
r3dDeviceQueue.cpp	Contains definitions of functions which allow delegating IDirect3DDevice9 associated calls to the rendering thread

class JobChief

Summary

Handles task distribution across available hardware threads. Main thread is also utilized in task processing.

Important Methods

void Init()

Summary:

Initializes JobChief object.

---

void Close()

Summary:

Frees resources allocated for JobChief

---

void Exec(ExecFunc func, void* data, size_t itemCount)

Summary:

Executes function *func* to work on data pointed by *data* which contains *itemCount* items. Execution is done on all available hardware threads.

Function *func* has the following signature:

*void (*ExecFunc)(void* data, size_t itemStart, size_t itemCount)*

The function receives a pointer to the start of the data and the range it should work on.

Parameters:

func	- pointer to the function to process the data
data	- pointer to the data
itemCount	- number of items (chunks of work) in the data.

---

uint32_t GetThreadCount() const

Summary:

Returns the number of available threads.

Return value:

Number of available hardware threads

---

class r3dD3DVertexBufferTunnel

Summary

Delegates vertex buffer lock/unlock functions to the rendering thread to support multithreaded loading and rendering.

Important methods

void Lock(UINT OffsetToLock, UINT SizeToLock, VOID **ppData, DWORD Flags)

Summary:

Delegates lock function call to the rendering thread in case it is made from loading thread.

Parameters:

Parameters are similar to that of *IDirect3DVertexBuffer9::Lock* function

---

void Unlock( )

Summary:

Delegates unlock function call to the rendering thread in case it is made from loading thread.

Parameters:

Parameters are similar to that of *IDirect3DVertexBuffer9::Unlock* function

---

class r3dD3DIndexBufferTunnel

Summary

Delegates index buffer lock/unlock functions to the rendering thread to support multithreaded loading and rendering.

Important methods

void Lock(UINT OffsetToLock, UINT SizeToLock, VOID **ppData, DWORD Flags)

Summary:

Delegates lock function call to the rendering thread in case it is made from loading thread.

Parameters:

Parameters are similar to that of *IDirect3DIndexBuffer9::Lock* function

---

void Unlock( )

Summary:

Delegates unlock function call to the rendering thread in case it is made from loading thread.

Parameters:

Parameters are similar to that of *IDirect3DIndexBuffer9::Unlock* function

---

class r3dD3DTextureTunnel

Summary

Delegates texture lock/unlock functions to the rendering thread to support multithreaded loading and rendering.

Important methods

void LockRect(UINT Level, D3DLOCKED_RECT *pLockedRect, const RECT *pRect, DWORD Flags);

Summary:

Delegates lock function call to the rendering thread in case it is made from loading thread.

Parameters:

Parameters are similar to that of *IDirect3DTexture9::LockRect* function

---

void UnlockRect(UINT Level)

Summary:

Delegates unlock function call to the rendering thread in case it is made from loading thread.

Parameters:

Parameters are similar to that of *IDirect3DTexture9:: UnlockRect* function

class r3dDeviceTunnel

Summary

Delegates resource creation functions to the rendering thread to support multithreaded loading and rendering.

Important methods

The following IDirect3DDevice9 methods get delegated. Please note that instead of Direct3D resource pointers, these function operate with *Tunnel classes.

```
void CreateTexture(        UINT Width,UINT Height,UINT Levels,DWORD Usage,
                           D3DFORMAT Format,D3DPOOL Pool,
                           r3dD3DTextureTunnel* TextureTunnel  )
```

```
void CreateVolumeTexture(        UINT Width,UINT Height,UINT Depth,
                                 UINT Levels,DWORD Usage,D3DFORMAT Format,
                                 D3DPOOL Pool, r3dD3DTextureTunnel* TextureTunnel )
```

```
void CreateCubeTexture(          UINT EdgeLength,UINT Levels,DWORD Usage,
                                 D3DFORMAT Format,D3DPOOL Pool, r3dD3DTextureTunnel* TextureTunnel )
```

```
void CreateVertexBuffer(         UINT Length,DWORD Usage,DWORD FVF,D3DPOOL Pool,
                                 r3dD3DVertexBufferTunnel* VertexBufferTunnel )
```

```
void CreateIndexBuffer(          UINT Length,DWORD Usage,D3DFORMAT Format,
                                 D3DPOOL Pool, r3dD3DIndexBufferTunnel* IndexBufferTunnel )
```

```
void CreateRenderTarget(         UINT Width,UINT Height,D3DFORMAT Format,
                                 D3DMULTISAMPLE_TYPE MultiSample,DWORD MultisampleQuality,
                                 BOOL Lockable,r3dD3DSurfaceTunnel* SurfaceTunnel )
```

```
void CreateDepthStencilSurface(  UINT Width,UINT Height,D3DFORMAT Format,
                                 D3DMULTISAMPLE_TYPE MultiSample,DWORD MultisampleQuality,
                                 BOOL Discard, r3dD3DSurfaceTunnel* SurfaceTunnel )
```

```
void CreateQuery(                D3DQUERYTYPE Type,IDirect3DQuery9** ppQuery)
```

```
void CreateVertexDeclaration(    const D3DVERTEXELEMENT9* pVertexElements,
                                 IDirect3DVertexDeclaration9** ppDecl )
```

```
void D3DXCreateTextureFromFileInMemoryEx(
        LPCVOID pSrcData, UINT SrcDataSize, UINT Width, UINT Height, UINT MipLevels,
        DWORD Usage, D3DFORMAT Format, D3DPOOL Pool, DWORD Filter, DWORD
        MipFilter, D3DCOLOR ColorKey, D3DXIMAGE_INFO* pSrcInfo,
        PALETTEENTRY* pPalette, r3dD3DTextureTunnel* TextureTunnel,
        const char* DEBUG_NAME, bool async )
```

```
void D3DXCreateVolumeTextureFromFileInMemoryEx(
        LPCVOID pSrcData, UINT SrcDataSize, UINT Width, UINT Height, UINT Depth,
        UINT MipLevels, DWORD Usage, D3DFORMAT Format, D3DPOOL Pool, DWORD
        Filter, DWORD MipFilter, D3DCOLOR ColorKey, D3DXIMAGE_INFO* pSrcInfo,
        PALETTEENTRY* pPalette, r3dD3DTextureTunnel* TextureTunnel, bool async=false)
```

```
void D3DXCreateCubeTextureFromFileInMemoryEx(
        LPCVOID pSrcData, UINT SrcDataSize, UINT Size, UINT MipLevels,
        DWORD Usage, D3DFORMAT Format, D3DPOOL Pool, DWORD Filter,
        DWORD MipFilter, D3DCOLOR ColorKey, D3DXIMAGE_INFO* pSrcInfo,
        PALETTEENTRY* pPalette,
        r3dD3DTextureTunnel* TextureTunnel, bool async = false )
```

```
void SetD3DResourcePrivateData( r3dD3DResourceTunnel* tunnel, const char* data )
```

```
void SetD3DResourcePrivateData( r3dD3DSurfaceTunnel* tunnel, const char* data )
```

The following method performs additional actions:

```
void CreateTextureAutoDownScale(
        UINT* Width,UINT* Height,UINT* Levels,DWORD Usage,
        D3DFORMAT Format,D3DPOOL Pool, r3dD3DTextureTunnel* TextureTunnel  )
```

In case the texture fails to be created because of lack of memory, it gets downscaled until it fits into the available memory

Resulting Width and Height are stored via *Width* and *Height* pointers

---

Device Queue

Summary

Device queue is used to delegate device associated activity to the rendering thread. Delegated activity is described by structure *DeviceQueueItem*.

This structure has the following fields:

Name	Type	Description
Type	*eType*	Specifies type of the device queue item. May be one of the following: *DEFAULT_POOL_RESOURCE_REGISTER* Register a resource that is related to device's **Default Pool.** This registration must be done in main thread to prevent automatic default resource freeing/allocation to occur in contradictory manner with resource loading thread. *DEFAULT_POOL_RESOURCE_CREATE* Create a resource that is related to device's **Default Pool.** *DEFAULT_POOL_RESOURCE_DEREGISTER* Unregister a resource that is related to device's **Default Pool.** *DEFAULT_POOL_RESOURCE_DESTROY* Destroy a resource that is related to device's **Default Pool.** *CUSTOM* Perform custom device related action.
ReadyPtr	*volatile int* *	When DeviceQueueItem is added, the value at this pointer is set to 0. After processing of the item has been complete, it is set to non-zero value.
DefPoolRes	*DefPoolResData*	A structure that contains a pointer to *r3dIResource*. This class is used as a base class for all device's **Default Pool** related reosurces.
Custom	*CustomData*	A structure that contains the following fields: *void (*Func) (void*) ;* Pointer to the function to be executed in the rendering thread *void* Param ;* Parameter to be passed to function *Func*

Following is the list of functions to work with DeviceQueueItem

void InitDeviceQueue()

Summary:

Initializes device queue.

---

void CloseDeviceQueue()

Summary:

Frees resources allocated for device queue.

---

void BlockDeviceQueue()

Summary:

Blocks device queue from adding new items.

---

void UnblockDeviceQuueue()

Summary:

Unblocks device queue and allows adding new items again.

---

void AddDeviceQueueItem(const DeviceQueueItem& item)

Summary:

Adds device queue item.

Parameters:

item	- device queue item to add.

---

void ProcessDeviceQueueItem(const DeviceQueueItem& item)

Summary:

Adds device queue item, then waits for it to be processed in the rendering thread.

Parameters:

item	- device queue item to process.

---

void AddCustomDeviceQueueItem(void (*func)(void *), void* param)

Summary:

Adds custom device queue item

Parameters:

func	- function that executes custom action
param	- parameter to pass to function *func*

---

void ProcessCustomDeviceQueueItem(void (*func)(void *), void* param)

Summary:

Adds custom device queue item and waits for it to be processed.

Parameters:

func	- function that executes custom action.
param	- parameter to pass to function *func*.

---

bool ProcessDeviceQueue(float chunkTimeStart, float maxDuration)

Summary:

Processes accumulated device queue. Must be called at least once per frame.

Parameters:

chunkTimeStart	- time since the start of device queue processing. Normally one needs to substitute current time.
maxDuration	- maximum duration of the queue processing. This duration is checked against the time that has passed since *chunkTimeStart*

Return value:

true if the queue still has items to be processed, **false** otherwise

---

void CreateQueuedResource(r3dIResource* res)

Summary:

Creates d3d pool default components of the resource pointed to by *res*. Creation is queued to be done in rendering thread.

Parameters:

res	- pointer to the resource default pool components of which need to be created.

---

void ReleaseQueuedResource(r3dIResource* res)

Summary:

Releases d3d default pool resources associated with resource pointed to by *res.*

Parameters:

res	- pointer to the resource default pool components of which need to be released.