Requirements to build

The proper version of Windows 10 SDK is required to build the project. If the windows 10 SDK is not already installed use the link below

Build in **x64 configuration**

If building in RELEASE then include the entire Assests folder and all the ddls in the Release directory

Windows 10 SDK : https://developer.microsoft.com/en-us/windows/downloads/windows-10-sdk

DirectX11

Demo Tutorial (Use headphones for best experience)

The demo starts off with a **third person camera** attached to the spaceship. Once the demo is closed a **log report** is created by the **profiler** at the root in a **"log.txt"** file.

Use **W or S** to move the ship forward, or backward

Use **A or D** to turn/rotate the ship. **Left mouse click and moving** the mouse rotates the ship as well.

If the **ship collides with the helix** it'll **lose a life** indicated by the colored **blocks on the top right**

Press **Tab** to play/change the currently playing music. The sound fades in over 3 seconds

Press **O** to stop the playing background music. The sound fades out over 5 seconds

Click on the **play UI button** to play sound on the sphere object

Click on the **stop UI button** to stop sound on the sphere object

Use the **arrow keys** to move the sphere object up, down left or right. If audio is playing on the object then the audio orientation would also change

Click on the **Reverb UI button** to change the environmental reverb. The current environment is shown on the top left corner. Go near the sound source to really hear the difference in the reverberation.

Click on the **Tool UI button** to launch the sound editor. Choose the sound track you want to alter and hit update to update the values. In the demo hit tab to cycle to the sound with the updated properties.

Press **' ~ ' (tilde)** to open the debug window. A full list of available debug commands is included in the documentation

Type in **BoundingBoxOn** to turn on visible bounding boxes and **BoundingBoxOff** to turn off bounding boxes. Type **ChangeSky** to change the current skybox.

Type in **FreeCam** in the debug window to change the camera to a free form camera. The **ThirdPersonCam** command will switch to a third person camera.

The free form camera controls are

W, A, S, D for moving forward, backward, strafe left and strafe right

Spacebar and X to move the camera Up and down

Left mouse click and move to rotate the camera

Use Arrow Keys to move the sphere. The sphere collides with the helix which prevents it from moving

CodeEngine Usage instructions

To make modifications to the demo scene the Game.cpp file can be altered. Refer documentation for more details on each class.

Profiling

To start profiling a function add PROFILE macro at the beginning of the function

*To load a new resource (Texture,Models,shaders,sounds) alter the "**config.txt**" in the **Assets folder**.*

The **resource manager only returns sounds in the sound properties list**. All other sounds must loaded though the audio engine (refer documentation)

The configuration file format should be as follows

<ResourceName> = <Full path of resource>

The resource name value should be one of the following:

- Texture
- Mesh
- VShader
- PShader
- Sound (These sounds will have to be played through code – refer documentation)

The sounds played in the demo with its properties should be given in the "**soundConfigList.txt**" file located in **Assets/Sounds.** Only these sounds can be played by pressing **TAB** in the demo and altered by the sound editor.

The format of sound properties must be the following:

<Full path of sound file><comma><volume><comma><pitch>

The **resouce manager instance** in the **Game** class is **rm.**

Use the following commands in Game::CreateGameObjects to get the respective resources

rm-> GetMeshByName(const std::string& _name)

rm->GetTextureByName(const std::wstring& _name);   // all textures should be long strings

rm->GetVShaderByName(const std::wstring & _name);   // all shader files name s should be long strings

rm->GetPShaderByName(const std::wstring & _name);   // all shader files name s should be long strings

rm->GetSoundByName(const std::string& _name);

Create a new material with the given texture and normal map (normal map is optional)
materials.push_back(new Material(<vertexShader>, <pixelShader>,<texture>,<normal map>));

The shaders and textures should be obtained from the resource manager

Add new Mesh

In Game::CreateGameObjects() do the following :

meshes.push_back(rm->GetMeshByName("<name with extension (.obj) >"));

Add new Game Object

In Game::CreateGameObjects() do the following :

// no attached sound

gameObjects.push_back(new GameObject(meshes[<index>], materials[<index>], bBoxOn, bSphereOn));

// with attached sound

gameObjects.push_back(new GameObject(meshes[<index>], materials[<index>], <vector of sound names>, audioEngine, bBoxOn, bSphereOn));

Transform game objects

// Check documentation for more details

gameObjects[<index>]->SetObjectPosition(XMFLOAT3(<value>, <value>, <value>));

gameObjects[<index>]->SetObjectRotation(XMFLOAT3(<value>, <value>, <value>));

gameObjects[<index>]->SetObjectScale(XMFLOAT3(<value>, <value>, <value>));

// To view any change the UpdateWorldMatrix function mush be called on the object

gameObjects[<index>]->UpdateWorldMatrix();

Move game objects

In Game::Update() do the following :

// multiply velocity by deltatime

gameObjects[<index>]->Move(XMFLOAT3(<velocity>, <velocity>, <velocity>));

Add new sound

In the demo background sounds can be added and played by adding the full path of the sounds to the soundConfig.txt file in a new line in the given format.

To load a sound directly do the following :

audioEngine->LoadSound(<full path name>, true, true); // 3D sound and looping On

To play a loaded sound do the following:

audioEngine->PlaySounds(< full path name>, AudioVector3{ <pos x>, <pos y>, <pos z> }, audioEngine->VolumeToDecibal(<float volume>), <fade in time in secods>);

Add new Tool

First **register** the tool using the following instruction:

ToolsManager::Instance().RegisterTool("< full file path and name>");

Start a tool

ToolsManager::Instance().StartTool("< Tool name with extension (.exe)>");

Add new texture without the resource manager (not recomended)

In Game::CreateGameObjects() do the following :

materials.push_back(new Material(vertexShader, pixelShader));

materials[<enter new texture index>]->LoadTexture(L"< new texture file path as png or jpg>", device, context);

Add new mesh without the resource manager (not recommended)

In Game::CreateGameObjects() do the following :

// only accepts OBJ files

// raw mesh data can be entered as well – refer documentation

meshes.push_back(new Mesh("<enter OBJ file path here>", device));

CodeEngine Documentation

Classes

Mesh

The Mesh class contains the mesh data of the objects that are loaded into the engine. It holds values for the vertex buffer and the index buffer of the mesh as well as the number of indices in the mesh.

Mesh(Vertex* vertexArray, int numOfVertices, unsigned int* indexArray, int numOfIndices, ID3D11Device* device)

This constructor is used to create a mesh with raw vertex and index data.

Mesh(const char* fileName, ID3D11Device* device)

This constructor is used to create a mesh from an OBJ file. It extracts the vertex data and index data from the OBJ file and assigns it to the vertex buffer and index buffer.

void CreateBuffers(Vertex* vertexArray, int numOfVertices, unsigned int* indexArray, int numOfIndices, ID3D11Device* device);

The create buffers function creates a DirectX11 vertex buffer and a DirectX11 index buffer from the vertex array and index array parameters.

ID3D11Buffer* GetVertexBuffer()

Returns the vertex buffer of the mesh

ID3D11Buffer* GetIndexBuffer()

Returns the index buffer of the mesh

int GetIndexCount()

Returns the index count of the mesh

void CalculateTangents(Vertex* verts, int numVerts, unsigned int* indices, int numIndices)

Calculate the tangents to the mesh surface for each vertex

XMFLOAT3 GetMinVertex()

Return the minimum vertex position on the mesh

XMFLOAT3 GetMaxVertex()

Return the maximum vertex position in the mesh

GameObject

This class holds data relevant to a game object or game entity. It contains data such as the position, rotation and scale of the object. It also contains a list of sounds attached to that game object.

GameObject(Mesh* mesh, Material* material, bool box, bool sphere)

Initializes a game object with a mesh and a material. The material represents the shaders applied to the object and the mesh represents the shape of the object. Setting the box flag will initialize a bounding box for the object and setting the sphere flag will initialize a bounding sphere for the object.

GameObject(Mesh * mesh, Material* mat, std::vector<std::string> soundfiles, AudioEngine * sManager, bool box, bool sphere)

Initializes a game object with a mesh and a material along with sounds that are attached to the object. The material represents the shaders applied to the object and the mesh represents the shape of the object. It takes in an instance of the audio engine manager for controlling the audio attached to it. Setting the box flag will initialize a bounding box for the object and setting the sphere flag will initialize a bounding sphere for the object.

private void LoadSound()

It loads all the sounds attached to the object into the cache.

void SetObjectWorldMatrix(DirectX::XMMATRIX)

Set the world matrix of the game object.

void SetObjectPosition(DirectX::XMFLOAT3)

Set the position of the object in left hand coordinate system

void SetObjectRotation(DirectX::XMFLOAT3)

Set the rotation of the object

void SetObjectScale(DirectX::XMFLOAT3)

Set the scale of the object

DirectX::XMFLOAT4X4 GetObjectWorldMatrix()

Returns the world matrix of the object

DirectX::XMFLOAT3 GetObjectPosition()

Returns the current position of the object

DirectX::XMFLOAT3 GetObjectRotation()

Returns the current rotation of the object

DirectX::XMFLOAT3 GetObjectScale()

Returns the scale of the object

Mesh* GetObjectMesh()

Returns the mesh of the object

Material* GetMaterial()

Returns the material attached to the object

void UpdateWorldMatrix()

Update the world matrix with the current position, rotation and scale.

void Move(DirectX::XMVECTOR velocityVector)

Move the game object given a three dimensional velocity vector.

void Move(float x, float y, float z)

Move the game object given velocities in individual directions as floats.

void Rotate(float x, float y, float z)

Rotate the object along the X, Y, or Z axis.

void PrepareMaterial(DirectX::XMFLOAT4X4 viewMatrix, DirectX::XMFLOAT4X4 projectionMatrix)

Prepare material attached to the object for presenting to the screen by setting the shader values.

void PlayObjectSound(int Id, float fadeIn = 0.0f)

Play sounds attached to the object if it exists. The fadeIn parameter is optional. If specified the sound fades in from zero volume to max volume over the given time in seconds.

void StopObjectSound(int Id, float fadeOut = 0.0f)

Stop a sound playing which is attached to the object. The fadeOut parameter is optional. If specified the sound fades out to zero volume over the given time in seconds.

BoundingOrientedBox GetBoundingBox()

Return the bounding box for the object. If it is not initialized it will return a default bounding box.

BoundingSphere GetBoundingSphere()

Return a bounding sphere for the object. If it is not initialized it will return a default bounding sphere.

Mesh* SetBboxMesh(ID3D11Device * device)

Initialize a bounding box mesh for displaying on screen. This function needs to be called right after an object is created before and translations.

Mesh* GetBboxMesh()

Return the bounding box mesh for drawing on screen.

bool hasBbox()

Returns true if a bounding box is initialized otherwise returns false.

bool hasBsphere()

Returns true if a bounding sphere is initialized otherwise returns false.

Material

This class specifies the properties of a material.

Material(SimpleVertexShader* vertexShader, SimplePixelShader* pixelShader)

This constructor is user to initialise a material with a vertex and pixel shader without a texture attached to it.

Material(SimpleVertexShader* vertexShader, SimplePixelShader* pixelShader, ID3D11ShaderResourceView* srv,ID3D11SamplerState* sampler)

This constructor is used to initialise a material with a vertex shader, pixel shader, srv texture and a sampler for that texture.

Material(SimpleVertexShader* vertexShader, SimplePixelShader* pixelShader, Texture* _texture)

Initialize a material with a diffuse texture which has been loaded in and no normal map texture.

Material(SimpleVertexShader* vertexShader, SimplePixelShader* pixelShader, Texture* _texture, Texture* _normalTex)

Initialize a material with a diffuse texture which has been loaded in along with a normal map texture.

SimplePixelShader* GetPixelShader()

Returns the pixel shader attached to the material

SimpleVertexShader* GetVertexShader()

Returns the vertex shader attached to the material

ID3D11ShaderResourceView* GetSRV()

Returns the texture as a SRV attached to the material

ID3D11SamplerState* GetSampler()

Returns the sampler for the texture attached to the material

ID3D11ShaderResourceView* GetNormalSRV()

Returns a pointer to the normal map SRV which is attached to this material.

void LoadTexture(const wchar_t* filePath, ID3D11Device* device, ID3D11DeviceContext* context)

Loads a texture from a path and creates an SRV of the texture and a sampler to sample it. The sampler is set to anisotropic filtering as default and the texture wraps the UVs as default.

void LoadTexture(Texture* texture)

Sets a loaded in texture to the diffuse SRV and sampler.

Renderer

This class renders the game objects to the screen

Renderer(ID3D11DepthStencilView *depthStencilView,ID3D11RenderTargetView *backBufferRTV,

ID3D11DeviceContext *context, IDXGISwapChain *swapChain, Camera *camera)

Initialize the renderer with the directX 11 context, current depth stencil view, back buffer, swap chain and camera.

void Draw(GameObject * gameobject)

Draw the given game object to the render target.

void Clear(const float color[4])

Clear the render target with a new colour.

void PresentSwapChain()

Present the swap chain to the screen. This step has to be done at the end of draw to view the objects in the render target.

void RenderSky(Mesh *skymesh, SimplePixelShader *skyBoxPS, SimpleVertexShader *skyBoxVS, ID3D11SamplerState* sampler, ID3D11ShaderResourceView* skySRV,ID3D11RasterizerState* skyRastState,ID3D11DepthStencilState* skyDepthState)

This function is used to render the sky box to the current render target

void DrawLines(GameObject * gameObject, ID3D11Device* device, DirectX::XMFLOAT4 color = XMFLOAT4(0.0f,0.0f,1.0f,0.0f))

This function will draw the bounding box of a gameobject if it exists. It draws the game object as a line list as opposed to as triangles. The color value decides what value the color of the lines will be.

void DrawSphere(GameObject* gameObject, XMFLOAT4 color = DirectX::XMFLOAT4(0.0f, 0.0f, 1.0f, 0.0f))

This function will draw the bounding sphere of a gameobject if it exists. The color value decides what value the color of the lines will be. The bounding sphere is drawn as three rings on the three planes namely XY, YZ and ZX. All three rings are drawn in one draw call.

void DrawRing(FXMVECTOR origin, FXMVECTOR majorAxis, FXMVECTOR minorAxis)

This function will draw a single ring aligned along the given major and minor axis.

void ToggleWireframeMode()

This function is used to swap between turning on and off the rasterizer state from rendering wireframes to solid mesh.

Camera

This class contains the properties of the camera. It holds data relating to the camera's position, rotation, direction, speed and rotation sensitivity as well as its view and projection matrix. It also contains a type variable which determines if the camera is freeform or third person and locked onto a target. The third person camera requires a third person object to attach to otherwise would fail.

Camera(float x, float y, float z)

Initialize a camera with an XYZ position in left hand coordinate space

DirectX::XMFLOAT4X4 GetViewMatrix()

Returns the view matrix of the camera

DirectX::XMFLOAT4X4 GetProjectionMatrix()

Returns the projection matrix of the camera

void SetPosition(DirectX::XMFLOAT3)

Set the XYZ position of the camera in left hand coordinate space

DirectX::XMFLOAT3 GetPosition()

Return the position of the camera

void SetDirection(DirectX::XMFLOAT3)

Set the direction the camera is facing (forward direction). It faces the along the positive Z axis by default

DirectX::XMFLOAT3 GetDirection()

Return the forward direction of the camera.

void SetRotationX(float rotX)

Set the rotation of the camera along the X axis.

float GetRotationX()

Return the rotation of the camera along the X axis void SetRotationY(float rotY)

Set the rotation of the camera along the Y axis

float GetRotationY()

Return the rotation of the camera along the Y axis

void Rotate()

Rotates the camera with the current rotation values.

void UpdateFromInput(float deltatime)

Update the position of the camera according to the input from the user.

- W – move forward
- S – move backward
- A – strafe left
- D – strafe right
- SpaceBar – move up
- X – move down

void UpdateViewMatrix()

Updates the view matrix of the camera based on the current position and rotation

void UpdateProjectionMatrix(int width, int height)

Update the projection matrix based on the width and height of the window

void MoveRelative(float x, float y, float z)

Move the camera relative to its current orientation. X, Y, and Z represent the velocity in the respective direction.

void MoveAbsolute(float x, float y, float z)

Move the camera in the world space and not local space. X, Y, and Z represent the velocity in the respective direction.

void UpdateViewMatrix3P(DirectX::XMFLOAT3 target, DirectX::XMFLOAT3 rot, DirectX::XMVECTOR targetfDir)

Update the view matrix of the third person camera based on a target position and target direction. The third person camera is locked to an object and will always look at the target.

UserInputHandler

This header contains classes and structures that handle user input from the keyboard. It enables the user to use windows VK_Keycodes, direct ASCII characters or key codes defined in this header as input to the functions.

KEYPRESSED(vk_code)

This is a global macro function that continuously polls the keyboard and returns true if the key corresponding to vk_code is pressed at that instant.

KEYRELEASED(vk_code)

This is a global macro function that continuously polls the keyboard and returns true if the key corresponding to vk_code is released at that instant.

KeyToggle

This is a helper class used to recognize toggling keys. It can used to check for a single press release of a key. For each key that needs to be toggled a new KeyToggle object needs to be initialized which accepts the corresponding keycode.

Example:

```
KeyToggle(VK_SPACE) toggleSpace;

If ( toggleSpace)

    // do action here
```

KeyInput

This class is used to handle keyboard inputs. It maintains an array of 256 key and their states as either down, pressed or released. The keys states are set depending on windows event messages.

static bool IsKeyDown(uint32_t keyCode)

Checks whether a key is pressed down and retutns true if a key is pressed down.

static bool WasKeyPressed(uint32_t keyCode)

Checks if a key was pressed down in the last frame and returns true if it was pressed.

static bool WasKeyReleased(uint32_t keyCode)

Checks if a key was released in the last frame and returns true if it was released.

static void SetKey(uint32_t keyCode, bool isDown, bool isReleased, bool isPressed)

This function is used to set the state of a key as either down, released or pressed.

static void ResetKeys()

This is used to reset the state of a key.

static std::wstring GetKeyPressedWString()

Gets the wide string version of a key press

Configuration

This class is used to load engine configuration files. It holds a list of all external data such as meshes, textures, sounds, compiled vertex shaders and pixel shaders that are to be loaded into the game engine.

A consolidated config file holding all the external resource data called "**config.txt**" can be used to load in all the resources. This file must be located in **/Assests directory.** This is also the only file where shaders can be loaded in from.

The sound configurations file must be named "**soundConfig.txt**" and be in the **Assests/Sounds directory.** The loader supports MP3, WAV and OGG formats.

The mesh configurations file must be named "**meshConfig.txt**" and be in the **Assests/Models directory**

The texture configurations file must be named "**textureConfig.txt**" and be in the **Assests/Texture directory**

void LoadSoundConfig()

This function is used to load sound names(paths) to a list of sound names from an external configuration file and use those names to load the corresponding files to the cache

std::vector<std::string> GetSoundFileList()

Returns the list of sounds in the configuration file

int GetSoundFileCount()

Returns the number of sounds in the sound list

void LoadMeshConfig()

This function is used to load model names(paths) to a list of model names from an external configuration file and use those names to load the corresponding files to the cache

void LoadTextureConfig()

This function is used to load texture names(paths) to a list of texture names from an external configuration file and use those names to load the corresponding files to the cache

void LoadSoundConfigList()

This function is used to load sound properties to a list of sound properties from an external configuration file.

std::vector<std::string> GetMeshFileList()

Returns the list of models read from the configuration file

std::vector<std::string> GetTextureFileList()

Returns the list of textures read from the configuration file

std::vector<std::string> GetVShaderFileList()

Returns the list of vertex shaders read from the configuration file

std::vector<std::string> GetPShaderFileList()

Returns the list of pixel shaders read from the configuration file

std::vector<SoundProperties> GetSoundProp()

Returns the list of sound properties read from the configuration file

unsigned __int64 GetMeshFileCount()

Return the number of models read from the config file.

unsigned __int64 GetTextureFileCount()

Return the number of texture files read from the config file.

unsigned __int64 GetVShaderFileCount()

Return the number of vertex shader files read from the config file.

unsigned __int64 GetPShaderFileCount()

Return the number of pixel shader files read from the config file.

bool file_exists(const char* filename);

Audio Engine

The audio engine class holds all the implementation details for the audio engine

Reverb properties

This structure holds the data needed to represent the audio reverb environment. It has the following values

float DecayTime;

float EarlyDelay;

float LateDelay;

float HFReference;

float HFDecayRatio;

float Diffusion;

float Density;

float LowShelfFrequency;

float LowShelfGain;

float HighCut;

float EarlyLateMix;

float WetLevel;

Implementation

This structure represents an instance of the audio engine implementation. It holds the cache for all the sounds, channels, banks and events.

Implementation()

Initializes the low level api from FMOD to access audio recourses.

Update()

Updates the implementation instance and caches in the instance.

Audio Engine Class

static void Init()

It initializes a new instance of the audio engine implementation by allocating space for the caches required and setting up FMOD

static void Update()

Updates the current instance of the audio engine

static void ShutDown()

Deallocates memory used for the audio engine and releases FMOD resources

static int ErrorCheck(FMOD_RESULT result)

Used to check for errors within FMOD studio low level API calls

void LoadBank(const string& strBankName, FMOD_STUDIO_LOAD_BANK_FLAGS flags)

Load sound bank files to the audio engine cache. strBankName should be a valid bank path.

void LoadEvents(const string& eventName)

Load FMOD studio events into the audio engine cache.

void LoadSound(const string& soundName,bool _3D=true,bool bLooping = false,bool bStream = false)

Load sound files to the audio engine cache. The sounds can be loaded as 3D sounds or 2D sounds, as a stream or as a sound file and with looping turned on or off. By default the sounds are loaded as 3D non streaming files with looping turned on.

void UnLoadSound(const string& soundName)

Unload sounds from the engine cache effectively releasing the resources allocated for it

void Set3dListenerAndOrientation(const AudioVector3& position,const AudioVector3& look, const AudioVector3& up)

Set the position, forward vector and up direction of the 3D listener active in the current engine instance.

int PlaySounds(const string& soundName, const AudioVector3& position = {0,0,0}, float volume = 0.0f, float fadeInTime = 0.0f)

Plays sounds that have been loaded into the cache. If the sound file hasn't been loaded in it will attempt to load the file before playing it. The position argument holds the position in space where the file will be playing from. The volume argument specifies the volume of the playing sound and fadeInTime is the time over which the sound will into that volume when playing.The volume has to be converted from a decibel value to float (Use DecibalToFloat()).

Returns the ID of the channel the sound is playing in.

void PlayEvents(const string& eventName)

Plays events that have been loaded into the cache.

void StopChannel(int channelId,float fadeOutTime = 0.0f)

Stops a playing channel. The fadeOutTime argument specifies the time over which the sound will fade out before stopping.

void StopEvent(const string& eventName, bool immediate = false)

Stops and event which is playing. If immediate is set to true the playback is stopped immediately otherwise a fadeout is applied before stopping.

void GetEventParameter(const string& eventName,const string& eventParameter,float* parameter)

Returns the parameters of an event

void SetEventParameter(const string& eventName, const string& eventParameter, float value)

Sets the parameters of an event

void StopAllChannels()

Stops all playing channels immediately without any fadeout.

void SetChannel3DPosition(int channelId,const AudioVector3& position)

Set the position of a channel in 3D space.

void SetChannelVolume(int channelId, float volume)

Set the volume of a channel. The volume has to be converted from a decibel value to float (Use DecibalToFloat()).

bool IsPlaying(int channleId) const

Checks if the given channel is playing or not and returns true if it is playing.

bool IsEventPlaying(const string& eventName)const

Checks if the given event is playing or not and returns true if it is playing.

float DecibalToVolume(float dBVolume)

Converts a decibel volume to a float

float VolumeToDecibal(float volume)

Converts a float volume to a decibel value

FMOD_VECTOR VectorToFmod(const AudioVector3& vposition)

Converts AudioVector3 to FMOD vectors

void MoveChannel3DPosition(int channelId, const AudioVector3& velocity)

This function is used to move a channel which id playing to a new position with a given velocity.

void FadeOutChannel(int channelId, float fadeOutTime = 0.0f, float fadeOutVolume = 0.0f)

This function is be used to fade sounds out to given fade out volume. The default fade out time is 0 (no fade) and volume is 0 and the sound stops playing.

void FadeInChannel(int channelId, float fadeInTime = 0.0f, float fadeInVolume = 1.0f)

This function is be used to fade sounds into given fade in volume. But default fade in time is 0 (no fade) and volume is 1.

void SetPitch(int channelId, float pitch = 1.0f)

Sets the pitch of the audio channel to specified value. Pitch value, 0.5 = half pitch, 2.0 = double pitch, etc .The default pitch is 1.

float GetPitch(int channelId)

Return the current pitch value of the given channel.

void SetFrequency(int channelId, float frequency)

Manually set the frequency of the given channel.

float GetFrequency(int channelId)

Return the current frequency of a given channel.

void SetLowPassGain(int channelId, float gain = 1.0f)

Set the Low pass gain value for a given channel.

float GetLowPassGain(int channelId)

Return the high pass gain for a given channel.

void ActivateReverb(bool active)

Activate or deactivate the reverb environment for the audio engine instance.

bool GetReverbState()

Return the current reverb state of the audio engine instance.

FMOD_REVERB_PROPERTIES SetReverbProperties(const ReverbProperties& reverbProp)

Getthe reverb properties of the audio engine with new reverb properties.

void SetEnvironmentReverb(const FMOD_REVERB_PROPERTIES reverbProperties,const AudioVector3& position,float minDist,float maxDist)

Set a 3D reverb zone having the given reverb properties. The zone will be a sphere centered at the given position. The reverb effect will be heard from min distance from the center to the max distance

Shader

This class is a container for the loaded shaders. This container is used by the resource manager to load in shaders.

VertexShader

VertexShader(const std::wstring& filePath,ID3D11Device *device, ID3D11DeviceContext *context)

Loads a vertex shader from the given path into memory and stores it as a simple vertex shader object.

SimpleVertexShader* Get()

Returns the simple vertex shader which was loaded in from the file.

PixelShader

PixelShader(const std::wstring& filePath, ID3D11Device* device, ID3D11DeviceContext* context)

Loads a pixel shader from the given path into memory and stores it as a simple pixel shader object

SimplePixelShader* Get() const { return pixelShader; }

Returns the simple pixel shader which was loaded in from the file.

Texture

This class is a container for holding textures that are loaded in. It holds a reference to the shader resource view that is obtained from the texture and a basic sampler that is created for it. If the file is a dds texture it is stored as a skymap/cubemap texture and a skymap/cubemap sampler is created.

Texture(const std::wstring& filePath, ID3D11Device * _device, ID3D11DeviceContext * context)

Load in a texture from the given file and create a sampler for it. If the file is a dds texture it is stored as a skymap/cubemap texture and a skymap/cubemap sampler is created.

ID3D11ShaderResourceView * GetSRV() const

Return the shader resource view of the loaded texture.

ID3D11SamplerState* GetSampler() const

Return the sampler of the loaded texture.

void CreateCubeSampler(ID3D11Device * _device)

Create a cubemap sampler for skyboxes.

void CreateTextureSampler(ID3D11Device * _device)

Create a basic sampler for normal textures. By default, the filtering mode is set to anisotropic filtering.

ThirdPersonObject

This is inherited from the gameobject class and hold data relevant to an object which has a third person camera attached to it. It holds data such as movement speed, rotation speed, forward direction, life and a reference to the third person camera.

ThirdPersonObject(Mesh* mesh, Material *mat, Camera* cam, bool box = false, bool sph = false)

This constructor initalises the object with a mesh, material, camera and bounding boxes.

void SetCamera(Camera* _cam)

Sets the camera to which the object is attached to.

void Update(float deltaTime)

Updates the position of the third person object based on keyboard input. The W key moves forward, S moves backward, A and D rotates the object to the left and right. It also updates the third person camera to look at the object.

void SetIntersecting(bool intersecting)

Sets the boolean which says if the object is currently intersecting some object.

bool GetIntersecting()

Returns a boolean to indicate if the object is currently intersecting.

void TakeDamage()

Reduce the life of the object.

int GetLife()

Return the life of the object.

Tool

This is a container for the external tools that can be loaded into the engine. It holds data related to the tool such as the application name, path, startup information, process information and is running; The tool has to be a complied binary executable file.

Tool()

Initialize the startup information and process information to zero memory.

void SetPath(LPCTSTR _path)

Set the path the executable of the tool is located.

void SetName(std::string _name)

Set the name of the tool.

void SetStartInfor(STARTUPINFO _si)

Set the startup information of the tool

void SetProcessInfo(PROCESS_INFORMATION _pi)

Set the process information of the tool

void SetStatus(bool status)

Set the running status of the object.

bool isRunning()

Return the running status of the tool

LPCTSTR GetPath()

Return the path of the tool

std::string GetName()

Return the name of the tool

STARTUPINFO GetStartInfo()

Return the startup information of the tool

PROCESS_INFORMATION GetProcessInfo()

Return the process information of the tool.

Tools Manager

This class manages all the tools registered to the engine. It holds a list of all the registered tools and functions for registering and unregistering tools as well as starting and closing tools correctly.

void RegisterTool(LPCTSTR fullPath)

Creates a new tool from the given full path and uses the name of the executable as the name of the tool which is then registered to the tools manager.

void StartTool(LPCSTR toolName)

Start a registered tool in a new process and thread using the windows api create process function. It sets the registered tool's startup information and process information and sets the tool to running status.

void CloseTool(LPCSTR fullPath)

Closes a tools that has been started and terminates its process.

void UnRegisterTool(LPCTSTR toolName)

Unregister a tool from the tools manager if it is already registered.

void UnregisterAll()

Unregister all the registered tools in the list of the registered tools in the tools manager.

std::shared_ptr<Tool> GetTool(LPCSTR toolName)

Return a reference of the tool corresponding to the given tool name if it is registered to the tools manager.

ResourceFactory

Resource class

This is a templated container for all the external resources in the project. It is used by the resource manager to manage the resources in the engine. It holds the unique identity of the resource, name of the resource and a reference to the actual resource.

Resource(std::string _UID, std::string _name, T *_resource) : UID(_UID), name(_name), resource(_resource) {}

Initializes the resource with a a unique ID and name.

T* GetResource()

Returns the reference to the resource.

Resource Manager

This class manages all the resources in the engine. It defines maps which contain a name for the resource and its actual componenet. It is able to manage all textures, meshes, vertex and pixel shaders and sounds (sounds can be loaded only through the consolidated resource file). The resource manager supports only synchronous loading. Every resource will have a unique existance and the resource manager will ensures that no duplicates are made.

ResourceManager(ID3D11Device * _device, ID3D11DeviceContext* _context, AudioEngine*_audioEngine)

This initializes an instance of the resource manager with a device and context as well as an audio engine to load sounds.

~ResourceManager()

Deallocats memory and release all resources loaded through the manager.

void LoadMesh(const std::string& filePath)

Load a model/mesh into the engine. If load is completed successfully a reference to the loaded resource along with its name is added to the appropriate resource map. The model loader is loaded with the help of the OBJ_Loader. If the same resource if already loaded it ensures to only have a unique instance of the resource.

void LoadTexture(const std::wstring& filePath)

Load a texture into the engine. If load is completed successfully a reference to the loaded resource along with its name is added to the appropriate resource map. If the same resource if already loaded it ensures to only have a unique instance of the resource.

void LoadVertexShader(const std::wstring& filePath)

Load a compiled vertex shader into the engine. If load is completed successfully a reference to the loaded resource along with its name is added to the appropriate resource map. If the same resource if already loaded it ensures to only have a unique instance of the resource.

void LoadPixelShader(const std::wstring& filePath)

Load a compiled pixel shader into the engine. If load is completed successfully a reference to the loaded resource along with its name is added to the appropriate resource map. If the same resource if already loaded it ensures to only have a unique instance of the resource.

void LoadFromConfig()

Load all a resources from a consolidated config file having a list of all the resources that need to be loaded. The config file is read using the config manager. Each resource will have a unique existence and no duplicates will be created.

void UnloadAll()

Unload all resources from the resource manager and release the memory allocated for them.

bool UnloadResource(std::string _name)

Unload a specific resource from the resource manager and release the memory allocated to it.

Mesh* GetMeshByName(const std::string& _name)

Return the mesh which has been loaded in with the given name.

Texture* GetTextureByName(const std::wstring& _name)

Return the texture which has been loaded in with the given name. The name has to be given as a wide string.

SimpleVertexShader * GetVShaderByName(const std::wstring & _name)

Return the simple vertex shader which has been loaded in with the given name. The name has to be given as a wide string.

SimplePixelShader * GetPShaderByName(const std::wstring & _name)

Return the simple pixel shader which has been loaded in with the given name. The name has to be given as a wide string.

Sound GetSoundByName(const std::string& _name)

Return the sound which has been loaded in with the given name. The name has to be given as a wide string.

std::wstring StrToWstr(const std::string& s)

Helper function to convert from a normal string to a wide string.

std::string WstrToStr(const std::wstring& s)

Helper function to convert from a wide string to a normal string.

Profiler

Helps profile the engine and logs the stats in an external file.

PerformanceData

This hold all the performance data related to a function. It holds values for file name, line number the profiler is called at, function name, number of cycles in the spent in the function and number of times that function is called.

Profiler

This class actually keeps track of all the profiler data. It holds a performance log having all the performance details of the monitoring functions and a function stack to keep track of the current function the profiler is in.

static Profiler * GetInstance()

Return the unique instance of the profiler.

static void DeleteInstance(int64 deleteTime)

Delete the unique instance of the profiler.

void Start(std::string _fName, std::string _funcName, int lineNum)

Start monitoring a function and tracking its performance details. If a a nested function is being tracked this function adds the parent function to the stack, pauses its tracking and starts tracking the child function. Once the child is finished the parent function tracking is resumed.

void End()

Stops tracking a function and updates it's performance data and the performance log.

LARGE_INTEGER GetClock()

Return the current clock.

void LogPerformance(int64 globalTime)

Logs the performance data in a sorted manner to an external "log.txt file"

~Profiler()

Deletes all memory allocated for the profiler.

bool CheckLog(std::string)

Return true if given function is in the log.

void Sort()

Sorts the log by descending order of time/cycles taken by a function.

ProfilerStart

This class indicates a profiler start point in the engine. Different instances of this class starts tracking different functions

ProfilerStart(std::string fileName, std::string functionName, int lineNumber)

Starts a profiler in the function that is calling it.

~ProfilerStart()

Stops the profiler that is running and logs the data.

void endMark()

Stops the profiler that is running and logs the data.

Game

This class is where the engine is put together. All the logic an implementation of the demo is done here

void LoadResources()

Loads the resources used in the game with the help of the resource manager.

void CreateCamera()

Creates an instance of the camera and initializes it.

void CreateGameObjects()

Initializes the objects and resources used in the game and creates the different game entities. This is where the materials, gameobjects, textures and sounds are loaded and initialized.

void CreateRenderer()

Initializes and sets up a renderer for rendering the graphics.

void CreateLights()

Creates the lights in the scene

void CreateCanvas()

Creates the canvas and UI elements

void PlaySounds()

This is where the sound demo logic lies.

void PlayObjectSound()

Called when the UI play button is clicked and plays the sound playing in the background

void StopObjectSound()

Called when the UI stop button is clicked and stops the sound playing on the game object

void Update()

Updates the objects in the game during each frame.

void Draw()

Draws all the graphics elements to the screen through the renderer.

void CreateDebugFunctions()

Create the functions the debug window accepts.

void SetReverbPresets()

Set the reverb presets used in the demo and store the names of the presets into a vector for reference.

void BBoxOn()

Turn on bounding boxes so that they can be viewed on screen.

void BBoxOff()

Trun off visible bounding boxes and stop drawing them.

void ChangeSky()

This function cycles through all the skyboxes loaded into the demo.

void MoveShip(float deltaTime)

This function handles the movement update of the ship in the demo.

void ThirdPCam()

This function changes the current camera to a third person camera.

void FreeCam()

This function changes the current camera to a free form camera.

void StartTool()

This fucntion starts the sound editor tool with the help of a tools manager.

void ChangeSoundEnvironment()

This functions cycles through the reverb environment presets.