AUDIO, INPUT & COLLISION

Overview

- Graphics isn't everything. Music and sound effects can make the game as well. For this the *Handmade Game Engine* supports ***Music & Voice*** audio as well as ***Sound Effects***
- Before creating and using audio objects, the audio files will need to be loaded into memory, and for this we use the *Audio Manager* :

```
#include "AudioManager.h"
```

Overview

- The *LoadFromFile()* function is used to load the audio file into memory and store it for later use :

```
TheAudio::Instance()->LoadFromFile(name_of_file, type, tag_name);
```

- The arguments passed are the name of the audio file, which should be situated somewhere in the *Assets* folder, the type of audio you wish to load (music, sound effect or voice), and the tag name, which will label the audio with a name
- Regarding the type of audio being loaded, the flags **MUSIC_AUDIO**, **SFX_AUDIO** and **VOICE_AUDIO** are available. This just helps organise the audio data properly

Overview

- To unload the audio again, you can once again use one of the **3** flags to remove the correct audio type
- Simply use the *Audio Manager's UnloadFromMemory()* function :

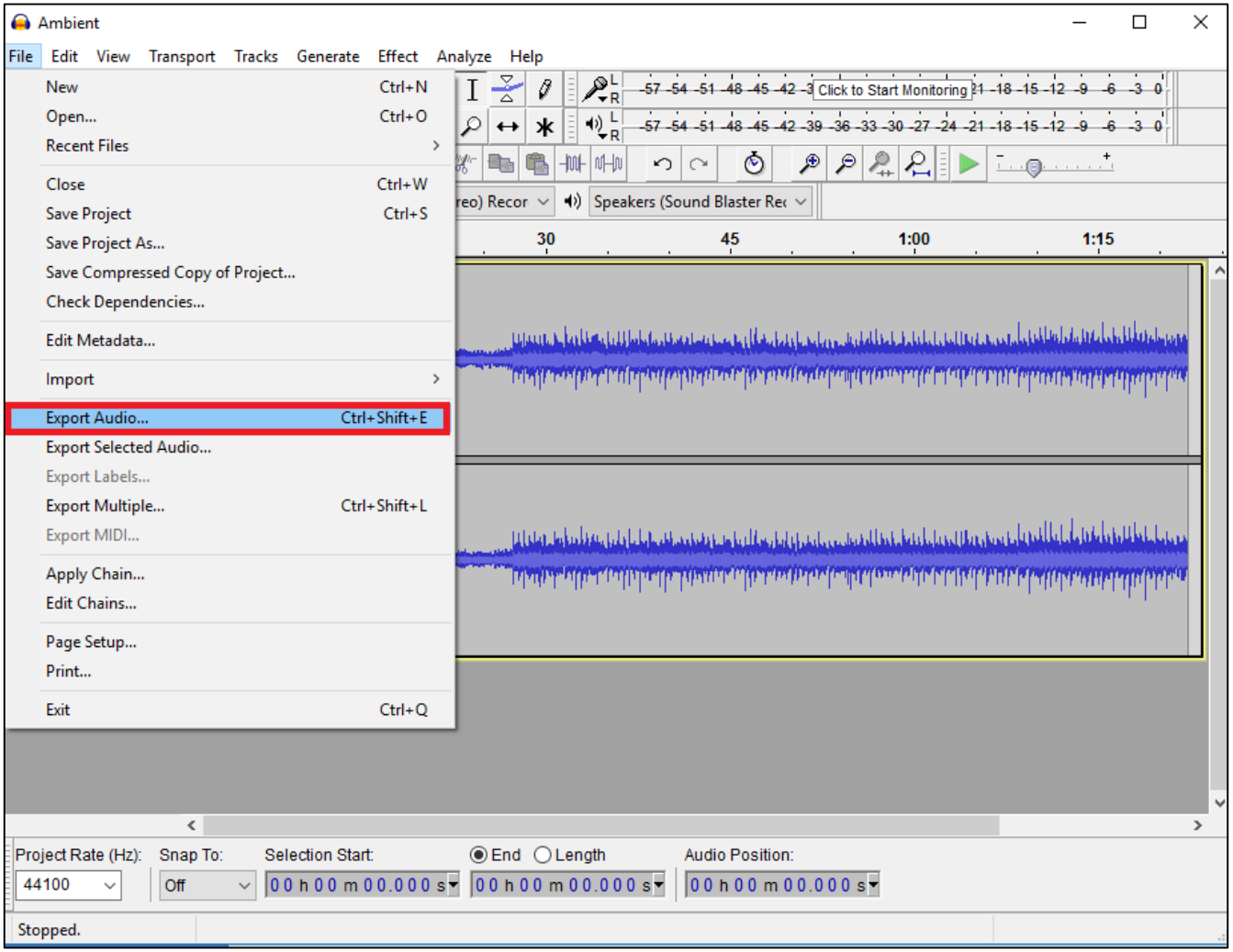
```
//remove all music audio from memory
TheAudio::Instance()->UnloadFromMemory(AudioManager::MUSIC_AUDIO, AudioManager::ALL_AUDIO);

//remove a specific voice file from memory
TheAudio::Instance()->UnloadFromMemory(AudioManager::VOICE_AUDIO, AudioManager::CUSTOM_AUDIO, "MY_VOICE");

//remove all sound effect audio files from memory
TheAudio::Instance()->UnloadFromMemory(AudioManager::SFX_AUDIO, AudioManager::ALL_AUDIO);
```

Overview

- When loading **MP3** files, there seems to be a licensing issue preventing you from playing them without a particular **DLL** file
- To overcome this issue, it's best to use **WAV**, **OGG** or **AIFF** audio files instead.
- If you have a vast amount of **MP3** files and you wish to use them, you could opt to convert the files from **MP3** to another format.
- For that the perfect tool to use would be *Audacity* :

  http://www.audacityteam.org/
- After installing and running *Audacity*, simply load up the **MP3** file and select ***Export Audio*** under the ***File*** menu option.
- Choose which format you wish to export to, choose the appropriate file location and hit ***Save***

Overview


Ambient
File Edit View Transport Tracks Generate Effect Analyze Help
New Ctrl+N
Open... Ctrl+O
Recent Files
Close Ctrl+W
Save Project Ctrl+S
Save Project As...
Save Compressed Copy of Project...
Check Dependencies...
Edit Metadata...
Import
Export Audio... Ctrl+Shift+E
Export Selected Audio...
Export Labels...
Export Multiple... Ctrl+Shift+L
Export MIDI...
Apply Chain...
Edit Chains...
Page Setup...
Print...
Exit Ctrl+Q
Click to Start Monitoring
Speakers (Sound Blaster Re
30
45
1:00
1:15
Project Rate (Hz):
Snap To:
Selection Start:
End
Length
Audio Position:
44100
Off
00 h 00 m 00.000 s
Stopped.

Music & Voice

- Music and good voice-overs can make a game sound and feel so much better
- This includes atmospheric background music, or any speech that in-game characters or NPCs may say
- To create and use music and voice audio, we need to use the *Audio* class and create an object of this type :

```
#include "Audio.h"

Audio music;
Audio voice;
```

Music & Voice

- Take a look once again at the *Background* class and note it also has an internal *Audio* component
- In the constructor, the audio file is loaded into memory and the properties are set up accordingly
- The *Background* class has **2** member functions that *play* and *stop* the music from playing, respectively
- The destructor removes the audio data from memory again
- Try loading in your own audio file (**OGG** or **WAV**) so that you can set your own background music. Do this in the *Play* state's *OnEnter()* function

Music & Voice

- We will create another game object called *MusicBox*, which will be rendered in our scene and will play a tune for us
- We will place both an *Audio* and *Sprite* component in here so that we can see the object and hear it :

```
class MusicBox : public GameObject
{

public:

    MusicBox();
    virtual ~MusicBox();

public:

    virtual void Update() {}
    virtual bool Draw();

private:

    Audio m_music;
    Sprite m_image;

};
```

Music & Voice

- Find a suitable tune and add it into the Audio folder and then load the object's music into memory :

```
TheAudio::Instance()->LoadFromFile("Assets\\Audio\\MusicBox.ogg", AudioManager::MUSIC_AUDIO, "MUSIC");
```

- All that's left to do now is link the audio object with the correct music audio and play!

```
m_music.SetAudio("MUSIC", Audio::MUSIC_AUDIO);
m_music.SetVolume(100);
m_music.Play();
```

- ***Note : Voice audio works exactly the same, except we use the VOICE_AUDIO flag***

Music & Voice

- The *SetAudio()* function links the loaded music or voice data with the audio object.
- The *SetVolume()* function takes in any value between **0** and **128** and adjusts the music volume accordingly
- The *Play()* function will play the audio. By default the music is played in an endless loop. You can choose to play it only once (suited for voice data) :

```
m_music.Play(Audio::PLAY_ONCE);
```

- The available flags for playing the audio are **PLAY_ONCE** and **PLAY_ENDLESS**

Music & Voice

- Music and voice files can also be paused, resumed and stopped altogether :

```
//play the music on an endless loop
m_music.Play(Audio::PLAY_ENDLESS);

//pause the music
m_music.Pause();

//resume the music after pausing
m_music.Resume();

//stop playing the music altogether
m_music.Stop();
```

Sound Effects

- Similarly, good sound effects can make a game more exciting as well
- This could be anything like breaking glass, cracking wood or smashing bricks
- To create and use sound effects, we need to use the *SFX* class and create an object of this type :

```
#include "SFX.h"

SFX sfx1;
SFX sfx2;
```

Sound Effects

- We will reuse our *Explosion* game object and add a sound effect in there
- First we need to load the *Explosion.wav* sound effect into memory :

```
TheAudio::Instance()->LoadFromFile("Assets\\Audio\\Explosion.wav", AudioManager::SFX_AUDIO, "EXPLOSION");
```

- Now we link the audio data, set the volume and we're ready to go :

```
m_sfx.SetSFX("EXPLOSION");
m_sfx.SetVolume(128);
```

- In the *Explosion* class' *Draw()* function, we can play the sound effect there :

```
m_anim.Draw(500, 435);
m_sfx.Play();
```

Sound Effects

- The *Play()* function will play the sound effects. By default the effect is played only once, but you can play it for as many times as you wish :

```
//play explosion once only
m_sfx.Play();

//play splash effect four times
m_sfx.Play(3);
```

- To avoid the sound effect playing repeatedly each draw call, do this :

```
static bool isPlaying = false;

if (!isPlaying)
{
    m_sfx.Play();
    isPlaying = true;
}
```

Input

- For our games to be more interactive, we will need some way for the user to control things on screen
- For this we can either read key presses or mouse movements to tell the game and all its world object what to do
- Reading input can be achieved anywhere in the game code, and is handled through the *Input Manager*. For this we need to include the following header file :

```
#include "InputManager.h"
```

- ***Note : The Handmade Game Engine supports both keyboard input and mouse input***

Keyboard Input

- To determine if a key has been pressed, use the *IsKeyPressed()* function. This will return a *bool* value based on if a key is *up* or *down*
- Ideally, this function can be used in a *if-else* statement, like so :

```
if (TheInput::Instance()->IsKeyPressed())
{
    //a key was pressed
}

else
{
    //a key was released
}
```

Keyboard Input

- The above function is sometimes too generic, as it never tells us which key was pressed or released.
- For more specific keyboard input, we need to determine which exact key was pressed
- Internally the *Input Manager* stores an array which maintains which keys are pressed and which are not
- We need to get this array from the *Input Manager* and for this we first need to declare the following pointer :

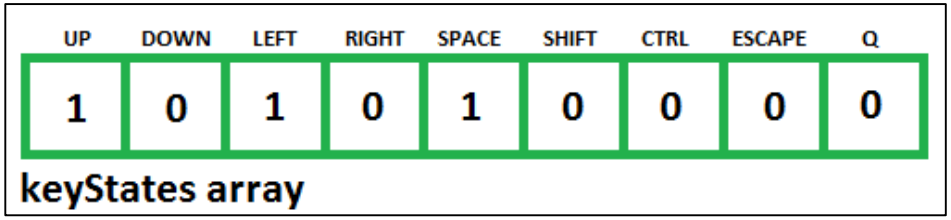
```
const Uint8* keyStates;
```

Keyboard Input

- Now we can call the *GetKeyStates()* function to acquire the array of key states and store it in our pointer :

```
keyStates = TheInput::Instance()->GetKeyStates();
```

- Internally, a small portion of the *keyStates* array may look like this :



- ***Note : Each array element represents a key and for each element with the value 1, a key is pressed and 0 means that the key is released.***

Keyboard Input

- Using the above array in an *if-statement* and indexing it using constant values that represent each key, we can determine if that key was pressed or not.
- If the index value queried returns a **1** (or *true*), the key is pressed, if it returns **0** (or *false*) it is not pressed

```
if (keyStates[SDL_SCANCODE_ESCAPE])
{
    //the ESCAPE key was pressed
}
```

- ***Note : For a complete list of supported key codes, click on the link below :***

  https://wiki.libsdl.org/SDL_Scancode

Keyboard Input

- Let's head back to our *Player* game object and add some keyboard input handling in the *Update()* function :

```
const Uint8* keyStates = TheInput::Instance()->GetKeyStates();

if (keyStates[SDL_SCANCODE_LEFT])
{
    m_position.x -= 2;
}

else if (keyStates[SDL_SCANCODE_RIGHT])
{
    m_position.x += 2;
}
```

Mouse Input

- We can also read mouse motion and clicks and for this we also make use of the *Input Manager*
- To see if any particular mouse button is pressed or released we can use either of the *GetButtonState()* functions, like so :

```
if (TheInput::Instance()->GetLeftButtonState() == InputManager::DOWN)
{
    //left mouse button is clicked
}

if (TheInput::Instance()->GetRightButtonState() == InputManager::UP)
{
    //right mouse button is released
}

if (TheInput::Instance()->GetMiddleButtonState() == InputManager::DOWN)
{
    //middle mouse button is clicked
}
```

Mouse Input

- To see how much the mouse has moved, we use the *GetMouseMotion()* routine and store the returned value in a *vec2* object

```
glm::vec2 mouseMotion = TheInput::Instance()->GetMouseMotion();
```

- The x and y values returned correspond to how much the mouse has moved on its x (*left/right*) and y (*up/down*) axis
- The x mouse motion value will be ***negative*** for *left* and ***positive*** for *right* movement
- The y mouse motion value will be ***negative*** for *up* and ***positive*** for *down* movement
- ***Note : The more rigorously you move the mouse, the higher the returned values will be***

Mouse Input

- Another important functionality of the *Input Manager* is the *GetMousePosition()* routine, which returns the position the mouse cursor is at :

```
glm::vec2 mousePosition = TheInput::Instance()->GetMousePosition();
```

- The x and y values returned correspond to where on screen the mouse cursor is positioned
- The x value will be between **0** and the resolution *width*
- The y value is between **0** and the resolution *height*
- ***Note : The top left corner of the game window is position (0,0)***

Mouse Input

- The mouse cursor can also be manually set, like so :

```
TheInput::Instance()->SetMousePosition(500, 400);
```

- Furthermore, the mouse cursor image can be set to something other than an arrow by using the *SetMouseCursorType()* routine :

```
//create a hand cursor
TheInput::Instance()->SetMouseCursorType(InputManager::HAND);

//create a crosshair cursor
TheInput::Instance()->SetMouseCursorType(InputManager::CROSSHAIR);
```

Mouse Input

- The **X** in the top right corner of a windowed game can also be checked whether is has been clicked.
- For this we use the *IsXClicked()* routine and this is great for ending a game, like so :

```
if (TheInput::Instance()->IsXClicked())
{
    //end the game!
}
```

Collisions

- Collisions are important if we want to set boundaries and obstacles in our game so that players and enemies don't move "through" anything
- The *Handmade Game Engine* supports ***bounding box*** and ***sphere collisions***
- These collisions are meant to be implemented as components of our existing game objects and will represent box and sphere-like bounding volumes accordingly

Bounding Box Collisions

- These types of collisions represent an *Axis-Aligned Bounding Box*, or *AABB* for short
- They use a min and max value on the x and y axis to form a box-like bound and will check if collisions between two boxes are happening
- To use a bounding box, we need to add a *AABB* component into our game object, and for that we need the following header :

```
#include "AABB.h"
```

- Now, somewhere in our game objects, we can add the following :

```
AABB boxBound;
```

Bounding Box Collisions

- We will need to set up the bounding box so that the *AABB* can update and determine if it collides with another box later on :

```
boxBound.SetDimension(width, height);
boxBound.SetPosition(x, y);
```

- The *SetDimension()* routine will set the bounding volume of the box
- The *SetPosition()* function places the *AABB* in the game world so that it can calculate if it collides with other game objects
- ***Note : If the object is moving, the position will need to be set all the time. If the object changes size, its dimension needs to be adjusted***

Bounding Box Collisions

- Now all we need to do is add a *AABB* to another object and check if the two objects collide :

```
boxBound.IsColliding(anotherBound);
```

- The *IsColliding()* function takes in an argument for another *AABB* box and will calculate if the two objects collide
- ***Note : To properly check for collisions between two game objects, we will likely need to call the IsColliding() routine somewhere in the PlayState.cpp code. For this we will need to get the bounds from both game objects before performing the checks***

Bounding Box Collisions

- Let's try and add a *AABB* to both our *Player* and *MusicBox* game objects and see if they collide
- In each game object, we will need to set the bounding box properties accordingly :

```
//set bounding box properties for player
m_bound.SetDimension(125, 250);
m_bound.SetPosition(500, 435);

//set bounding box properties for music box
m_bound.SetDimension(100, 100);
m_bound.SetPosition(300, 450);
```

- Because the player is moving, we need to update his bounds in the *Update()* routine :

```
m_bound.SetPosition(m_position.x, m_position.y);
```

Bounding Box Collisions

- We also need to create a getter function for both game objects so that the bounds can be accessed in order to check for collisions
- Add the following line of code to both the *Player* and *MusicBox* classes' definition :

```
AABB GetBound() { return m_bound; }
```

- Now in the *Play* state, we simply instantiate the *Player* and *MusicBox* objects individually :

```
m_player = new Player;
m_musicBox = new MusicBox;
```

Bounding Box Collisions

- In the *PlayState's Update()* method we will want to check if collisions are happening, and these simple lines of code will do the trick :

```
if (m_player->GetBound().IsColliding(m_musicBox->GetBound()))
{
    //they collide!!
}

else
{
    //they DO NOT collide!!
}
```

Bounding Box Collisions

Bounding Box Collisions

Sphere Collisions

- Spherical collision volumes work in a similar way to their bounding box counterparts, except that they use a sphere bounding volume instead
- To use a sphere collision component, we first need to include the header file :

```
#include "Sphere.h"
```

- To create the *Sphere* object, we do this :

```
Sphere sphereBound;
```

Sphere Collisions

- We will need to set up the bounding sphere so that the *Sphere* can update and determine if it collides with another sphere later on :

```
sphereBound.SetRadius(radius);
sphereBound.SetPosition(x, y);
```

- The *SetRadius()* routine will set radius of the bounding spherical volume
- The *SetPosition()* function places the *Sphere* in the game world so that it can calculate if it collides with other game objects
- ***Note : If the object is moving, the position will need to be set all the time. If the object changes size, its radius needs to be adjusted***

Sphere Collisions

- Now all we need to do is add a *Sphere* to another object and check if the two objects collide :

```
sphereBound.IsColliding(anotherBound);
```

- The *IsColliding()* function takes in an argument for another *Sphere* and will calculate if the two objects collide
- ***Note : To properly check for collisions between two game objects, we will likely need to call the IsColliding() routine somewhere in the PlayState.cpp code. For this we will need to get the bounds from both game objects before performing the checks***

Mouse Collisions

- The *Input Manager* comes with **2** built-in routines, called *IsColliding()*, that can check if the mouse cursor collides with a *AABB* or a *Sphere* object
- This is great for box-like or spherical objects like menu buttons to be checked if the mouse hovers over them or is clicked while on them
- This functionality can be used in a menu, or to make a point-and-click type of game