Pixie and Trixie's Adventure

Derik Mahan
Version 1.0
Mon Dec 6 2021

Table of Contents

Table of contents

Pixie and Trixie's Adventure

A small 2d platform game based on the game Thomas Was Alone on the Steam library. Two friends work together to make their way through the platforms, over fire and water to get to the end. Contains a clock to count down time for each level, the ability to split screens so one player can control both characters. And each level is designed in text files and are completely customizable. This is much shorter than the game it is based on but can still be very challenging and give you lots of fun.

Hotkeys:

- "Esc" for an exit from the game
- "Enter" to start/pause game
- "Q" switches camera from Pixie to Trixie and vice versa in fullscreen mode
- "W", "A", "D" for movement of Pixie
- "Up arrow", "Left arrow", "Right arrow" for movement of Trixie
- "F1" for enable/disable the split-screen mode

Requirements:

- C++
- SFML (x32 version)

My Notes:

This game was fun to build and learn with. I had to do a lot of research to make it work and had to learn how to use SFML to really get what I wanted out of this. Like I said I did a lot of extra research and external resources to help me get through this but am very happy with the way it turned out in the end. I hope as I continue through this journey of programming I can come up with some more original games of my own and not have to base them off other games. Also I chose to do this in x32 that way anyone can run it (just a side note).

Hierarchical Index

Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Class Index

Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

File Index

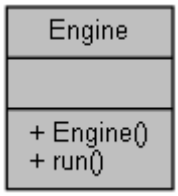
File List

Here is a list of all files with brief descriptions:

Class Documentation

Engine Class Reference

`#include <Engine.h>`
Collaboration diagram for Engine:



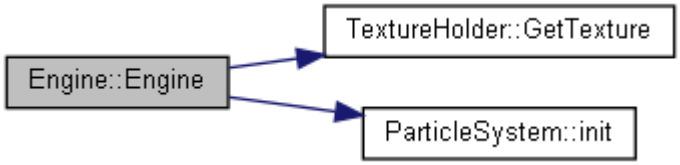
Public Member Functions

- **Engine** ()
- void **run** ()

Constructor & Destructor Documentation

Engine::Engine ()

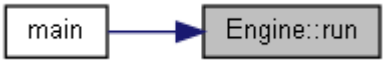
Here is the call graph for this function:



Member Function Documentation

void Engine::run ()

Here is the caller graph for this function:

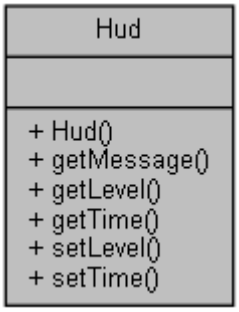


The documentation for this class was generated from the following files:

- **Engine.h**
- **DetectCollisions.cpp**
- **Draw.cpp**
- **Engine.cpp**
- **Input.cpp**
- **LoadLevel.cpp**
- **PopulateEmitters.cpp**
- **Update.cpp**

Hud Class Reference

`#include <HUD.h>`
Collaboration diagram for Hud:



Public Member Functions

- **Hud** ()
- Text **getMessage** ()
- Text **getLevel** ()
- Text **getTime** ()
- void **setLevel** (String text)
- void **setTime** (String text)

Constructor & Destructor Documentation

Hud::Hud ()

Member Function Documentation

Text Hud::getLevel ()

Text Hud::getMessage ()

Text Hud::getTime ()

void Hud::setLevel (String *text*)

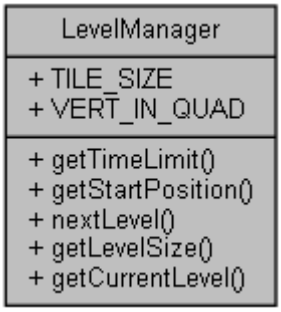
void Hud::setTime (String *text*)

The documentation for this class was generated from the following files:

- **HUD.h**
- **HUD.cpp**

LevelManager Class Reference

`#include <LevelManager.h>`
Collaboration diagram for LevelManager:



Public Member Functions

- float **getTimeLimit** ()
- Vector2f **getStartPosition** ()
- int ** **nextLevel** (VertexArray &rVaLevel)
- Vector2i **getLevelSize** ()
- int **getCurrentLevel** ()

Public Attributes

- const int **TILE_SIZE** = 50
- const int **VERT_IN_QUAD** = 4

Member Function Documentation

int LevelManager::getCurrentLevel ()

Vector2i LevelManager::getLevelSize ()

Vector2f LevelManager::getStartPosition ()

float LevelManager::getTimeLimit ()

int ** LevelManager::nextLevel (VertexArray & *rVaLevel*)

Member Data Documentation

const int LevelManager::TILE_SIZE = 50

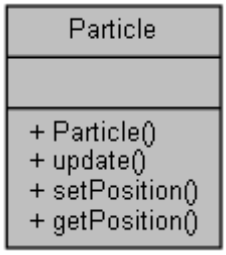
const int LevelManager::VERT_IN_QUAD = 4

The documentation for this class was generated from the following files:

- **LevelManager.h**
- **LevelManager.cpp**

Particle Class Reference

`#include <Particle.h>`
Collaboration diagram for Particle:



Public Member Functions

- **Particle** (Vector2f direction)
- void **update** (float dt)
- void **setPosition** (Vector2f position)
- Vector2f **getPosition** ()

Constructor & Destructor Documentation

Particle::Particle (Vector2f *direction*)

Member Function Documentation

Vector2f Particle::getPosition ()

void Particle::setPosition (Vector2f *position*)

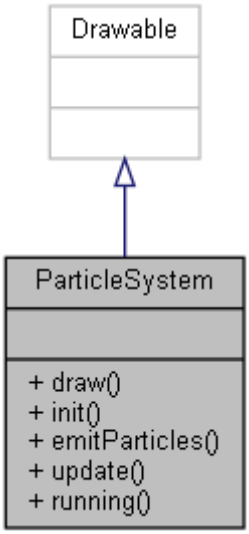
void Particle::update (float *dt*)

The documentation for this class was generated from the following files:

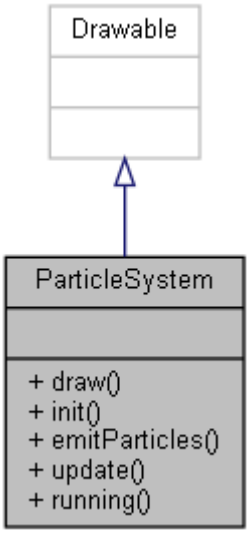
- **Particle.h**
- **Praticle.cpp**

ParticleSystem Class Reference

`#include <ParticleSystem.h>`
Inheritance diagram for ParticleSystem:



Collaboration diagram for ParticleSystem:



Public Member Functions

- virtual void **draw** (RenderTarget &target, RenderStates states) const
- void **init** (int count)
- void **emitParticles** (Vector2f position)
- void **update** (float elapsed)
- bool **running** ()

Member Function Documentation

void ParticleSystem::draw (RenderTarget & *target*, RenderStates *states*) const `[virtual]`

void ParticleSystem::emitParticles (Vector2f *position*)

void ParticleSystem::init (int *count*)

Here is the caller graph for this function:

bool ParticleSystem::running ()

void ParticleSystem::update (float *elapsed*)

The documentation for this class was generated from the following files:

- **ParticleSystem.h**
- **ParticleSystem.cpp**

Pixie Class Reference

`#include <Pixie.h>`

Inheritance diagram for Pixie:

Collaboration diagram for Pixie:

Public Member Functions

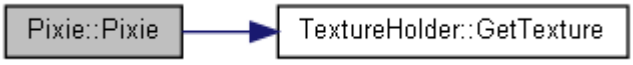
- **Pixie** ()
- virtual bool **handleInput** ()

Additional Inherited Members

Constructor & Destructor Documentation

Pixie::Pixie ()

Here is the call graph for this function:



Member Function Documentation

bool Pixie::handleInput () `[virtual]`

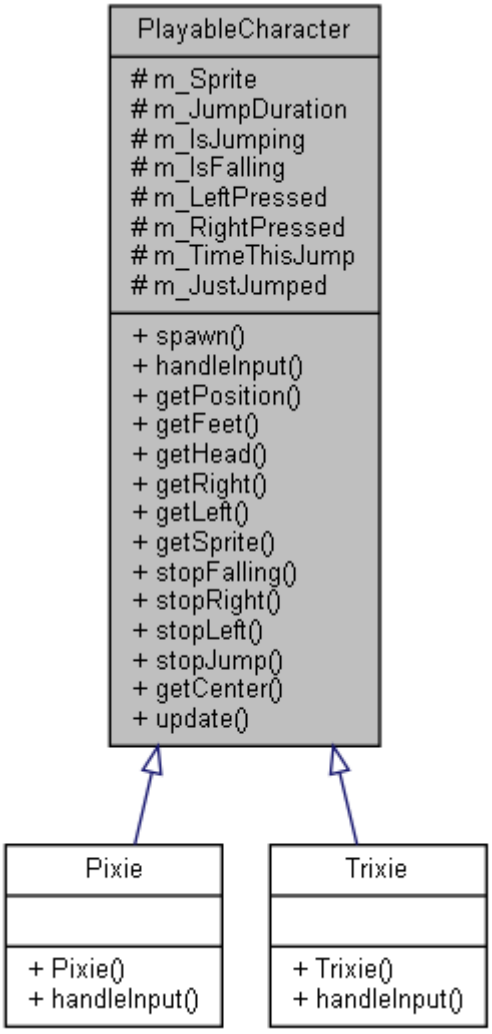
Implements **PlayableCharacter** (*p.17*).

The documentation for this class was generated from the following files:

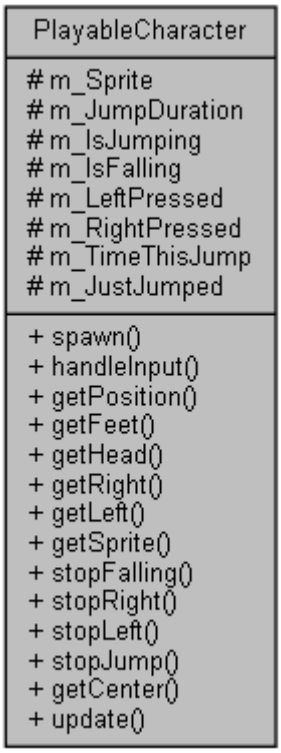
- **Pixie.h**
- **Pixie.cpp**

PlayableCharacter Class Reference

`#include <PlayableCharacter.h>`
Inheritance diagram for PlayableCharacter:



Collaboration diagram for PlayableCharacter:



Public Member Functions

- void **spawn** (Vector2f startPosition, float gravity)
- virtual bool **handleInput** ()=0
- FloatRect **getPosition** ()
- FloatRect **getFeet** ()
- FloatRect **getHead** ()
- FloatRect **getRight** ()
- FloatRect **getLeft** ()
- Sprite **getSprite** ()
- void **stopFalling** (float position)
- void **stopRight** (float position)
- void **stopLeft** (float position)
- void **stopJump** ()
- Vector2f **getCenter** ()
- void **update** (float elapsedTime)

Protected Attributes

- Sprite **m_Sprite**
- float **m_JumpDuration**
- bool **m_IsJumping**
- bool **m_IsFalling**
- bool **m_LeftPressed**
- bool **m_RightPressed**
- float **m_TimeThisJump**
- bool **m_JustJumped** = false

Member Function Documentation

Vector2f PlayableCharacter::getCenter ()

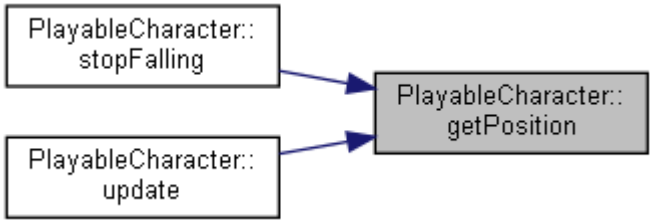
FloatRect PlayableCharacter::getFeet ()

FloatRect PlayableCharacter::getHead ()

FloatRect PlayableCharacter::getLeft ()

FloatRect PlayableCharacter::getPosition ()

Here is the caller graph for this function:



FloatRect PlayableCharacter::getRight ()

Sprite PlayableCharacter::getSprite ()

virtual bool PlayableCharacter::handleInput () `[pure virtual]`

Implemented in **Pixie** (*p.13*), and **Trixie** (*p.22*).

void PlayableCharacter::spawn (Vector2f *startPosition*, float *gravity*)

void PlayableCharacter::stopFalling (float *position*)

Here is the call graph for this function:

void PlayableCharacter::stopJump ()

void PlayableCharacter::stopLeft (float *position*)

void PlayableCharacter::stopRight (float *position*)

void PlayableCharacter::update (float *elapsedTime*)

Here is the call graph for this function:

Member Data Documentation

bool PlayableCharacter::m_IsFalling `[protected]`

bool PlayableCharacter::m_IsJumping `[protected]`

float PlayableCharacter::m_JumpDuration `[protected]`

bool PlayableCharacter::m_JustJumped = false `[protected]`

bool PlayableCharacter::m_LeftPressed `[protected]`

bool PlayableCharacter::m_RightPressed `[protected]`

Sprite PlayableCharacter::m_Sprite `[protected]`

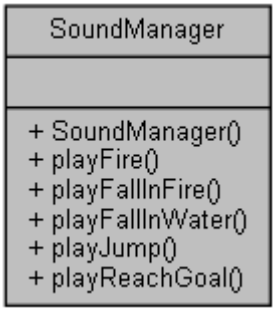
float PlayableCharacter::m_TimeThisJump `[protected]`

The documentation for this class was generated from the following files:

- **PlayableCharacter.h**
- **PlayableCharacter.cpp**

SoundManager Class Reference

`#include <SoundManager.h>`
Collaboration diagram for SoundManager:



Public Member Functions

- **SoundManager** ()
- void **playFire** (Vector2f emitterLocation, Vector2f listenerLocation)
- void **playFallInFire** ()
- void **playFallInWater** ()
- void **playJump** ()
- void **playReachGoal** ()

Constructor & Destructor Documentation

SoundManager::SoundManager ()

Member Function Documentation

void SoundManager::playFallInFire ()

void SoundManager::playFallInWater ()

void SoundManager::playFire (Vector2f *emitterLocation*, Vector2f *listenerLocation*)

void SoundManager::playJump ()

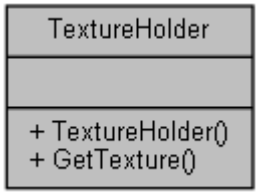
void SoundManager::playReachGoal ()

The documentation for this class was generated from the following files:

- **SoundManager.h**
- **SoundManager.cpp**

TextureHolder Class Reference

`#include <TextureHolder.h>`
Collaboration diagram for TextureHolder:



Public Member Functions

- **TextureHolder** ()

Static Public Member Functions

- static sf::Texture & **GetTexture** (std::string const &filename)

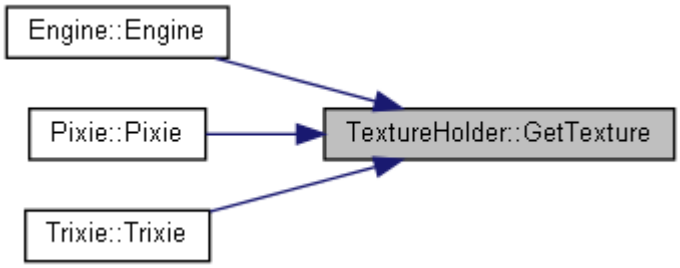
Constructor & Destructor Documentation

TextureHolder::TextureHolder ()

Member Function Documentation

sf::Texture & TextureHolder::GetTexture (std::string const & *filename*) `[static]`

Here is the caller graph for this function:



The documentation for this class was generated from the following files:

- **TextureHolder.h**
- **TextureHolder.cpp**

Trixie Class Reference

`#include <Trixie.h>`

Inheritance diagram for Trixie:

Collaboration diagram for Trixie:

Public Member Functions

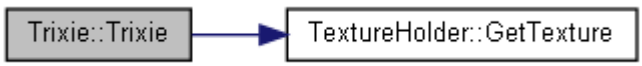
- **Trixie** ()
- virtual bool **handleInput** ()

Additional Inherited Members

Constructor & Destructor Documentation

Trixie::Trixie ()

Here is the call graph for this function:



Member Function Documentation

bool Trixie::handleInput () `[virtual]`

Implements **PlayableCharacter** (*p.17*).

The documentation for this class was generated from the following files:

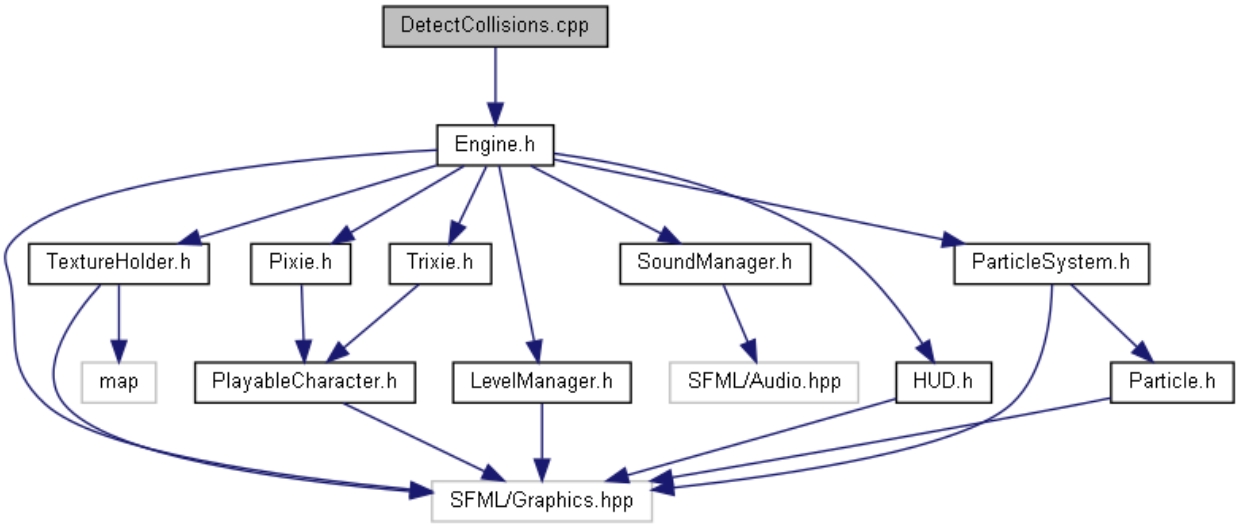
- **Trixie.h**
- **Trixie.cpp**

File Documentation

DetectCollisions.cpp File Reference

```
#include "Engine.h"
```

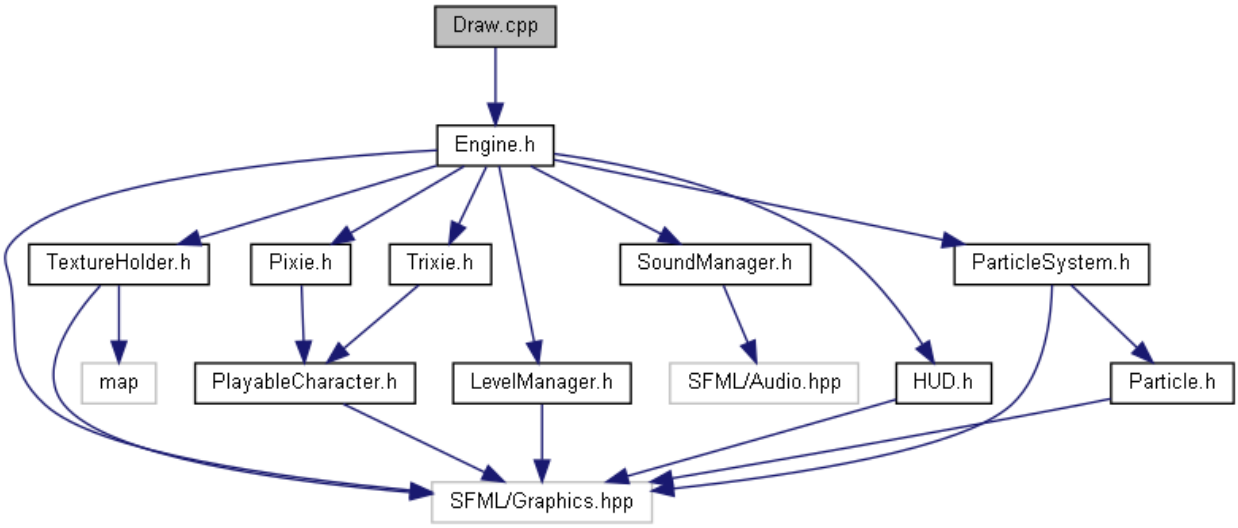
Include dependency graph for DetectCollisions.cpp:



Draw.cpp File Reference

```
#include "Engine.h"
```

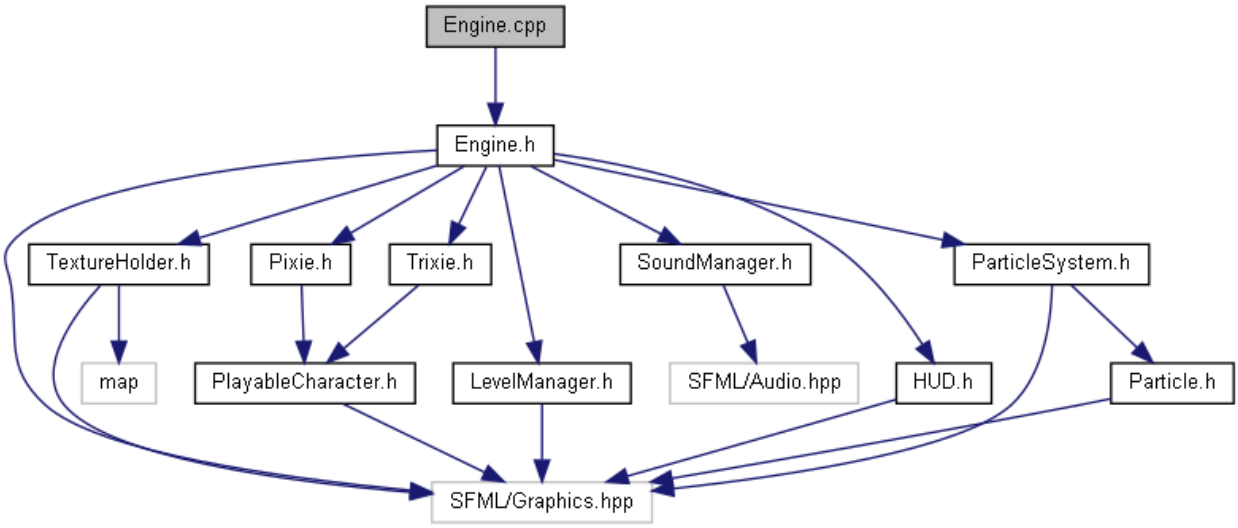
Include dependency graph for Draw.cpp:



Engine.cpp File Reference

```
#include "Engine.h"
```

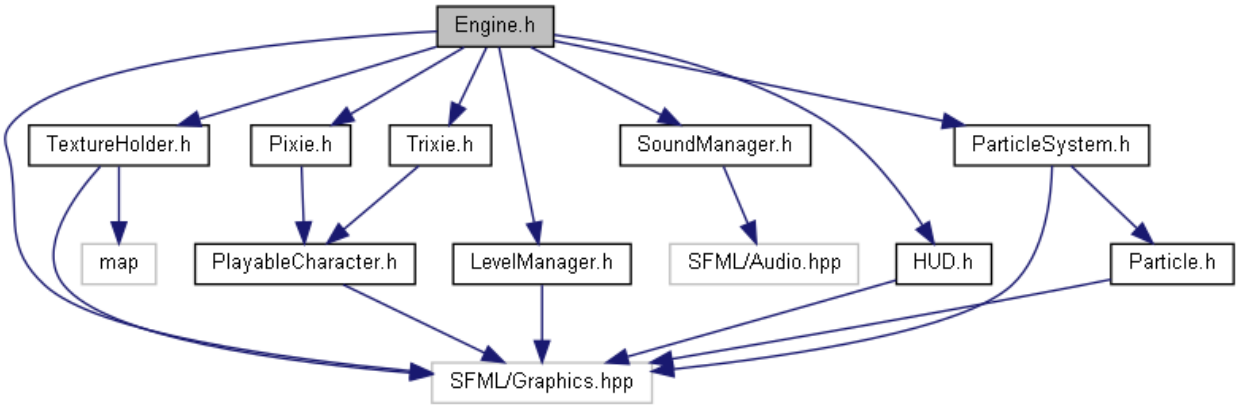
Include dependency graph for Engine.cpp:



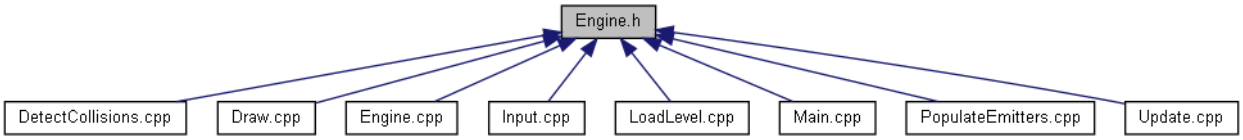
Engine.h File Reference

```
#include <SFML/Graphics.hpp>
#include "TextureHolder.h"
#include "Pixie.h"
#include "Trixie.h"
#include "LevelManager.h"
#include "SoundManager.h"
#include "HUD.h"
#include "ParticleSystem.h"
```

Include dependency graph for Engine.h:



This graph shows which files directly or indirectly include this file:



Classes

- class **Engine**

Engine.h

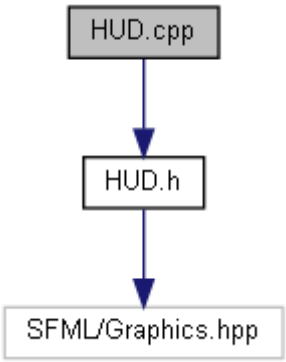
```
Go to the documentation of this file.1 #pragma once
2 #include <SFML/Graphics.hpp>
3 #include "TextureHolder.h"
4 #include "Pixie.h"
5 #include "Trixie.h"
6 #include "LevelManager.h"
7 #include "SoundManager.h"
8 #include "HUD.h"
9 #include "ParticleSystem.h"
10
11 using namespace sf;
12
13 class Engine
14 {
15 private:
16     // The texture holder
17     TextureHolder th;
18
19     // Create a particle system
20     ParticleSystem m_PS;
21
22     // Pixie and her friend, Trixie
23     Pixie m_Pixie;
24     Trixie m_Trixie;
25
26     // A class to manage all the levels
27     LevelManager m_LM;
28
29     // SoundManager
30     SoundManager m_SM;
31
32     // The Hud
33     Hud m_Hud;
34     int m_FramesSinceLastHUDUpdate = 0;
35     int m_TargetFramesPerHUDUpdate = 500;
36
37     const int TILE_SIZE = 50;
38     const int VERTS_IN_QUAD = 4;
39
40     // The force pushing the characters down
41     const int GRAVITY = 300;
42
43     // A regular RenderWindow
44     RenderWindow m_Window;
45
46     // The main Views
47     View m_MainView;
48     View m_LeftView;
49     View m_RightView;
50
51     // Three views for the background
52     View m_BGMainView;
53     View m_BGLeftView;
54     View m_BGRightView;
55
56     View m_HudView;
57
58     // Declare a sprite and a Texture for the background
59     Sprite m_BackgroundSprite;
60     Texture m_BackgroundTexture;
61
62     // Declare a shader for the background
63     Shader m_RippleShader;
64
65     // Is the game currently playing?
66     bool m_Playing = false;
67
68     // Is character 1 or 2 the current focus?
69     bool m_Character1 = true;
70
71     // Start in full screen mode
72     bool m_SplitScreen = false;
73
```

```
74      // How much time is left in the current level
75      float m_TimeRemaining = 10;
76      Time m_GameTimeTotal;
77
78      // Is it time for a new/first level?
79      bool m_NewLevelRequired = true;
80
81      // The vertex array for the level tiles
82      VertexArray m_VALevel;
83
84      // The 2D array with the map for the level
85      // A pointer to a pointer
86      int** m_ArrayLevel = NULL;
87
88      // Texture for the level tiles
89      Texture m TextureTiles;
90
91      // Private functions for internal use only
92      void input();
93      void update(float dtAsSeconds);
94      void draw();
95
96      // Load a new level
97      void loadLevel();
98
99      bool detectCollisions(PlayableCharacter& character);
100
101      // Makes a vector for the best place to emit sounds from
102      void populateEmitters(vector <Vector2f>& vSoundEmitters,
103          int** arrayLevel);
104
105      // vector of Vector2f for teh fire emitter locations
106      vector <Vector2f> m_FireEmitters;
107
108 public:
109      // The Engine constructor
110      Engine();
111
112      // Run will call all the private functions
113      void run();
114
115 };
```

HUD.cpp File Reference

```
#include "HUD.h"
```

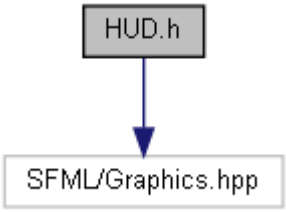
Include dependency graph for HUD.cpp:



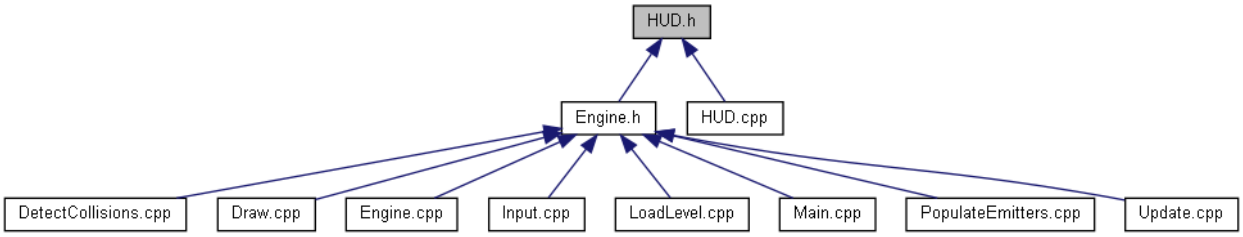
HUD.h File Reference

```
#include <SFML/Graphics.hpp>
```

Include dependency graph for HUD.h:



This graph shows which files directly or indirectly include this file:



Classes

- class **Hud**

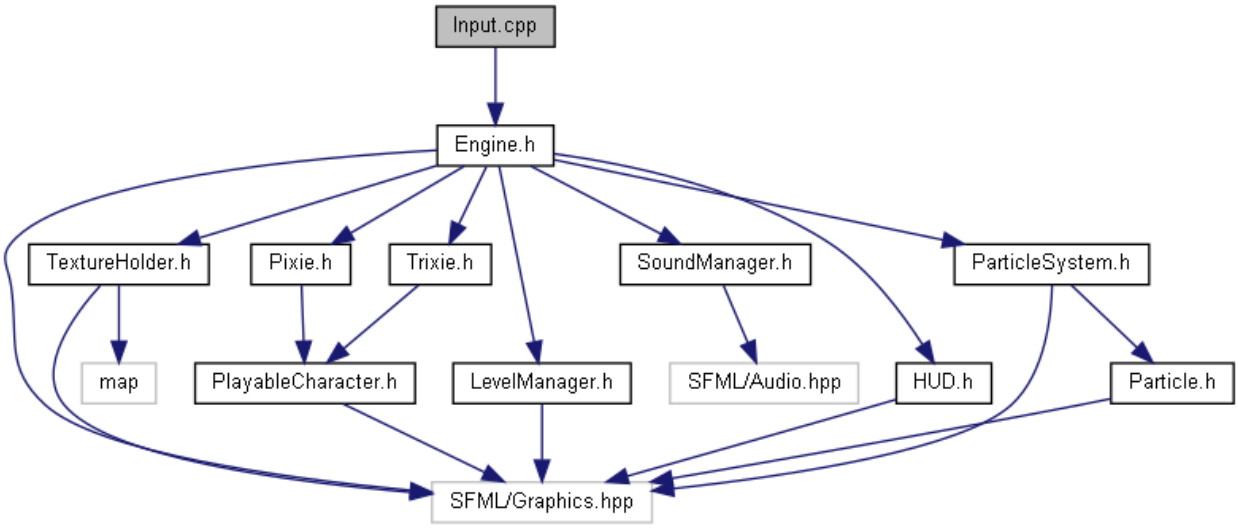
HUD.h

```
Go to the documentation of this file.1 #pragma once
2 #include <SFML/Graphics.hpp>
3
4 using namespace sf;
5
6 class Hud
7 {
8 private:
9     Font m_Font;
10     Text m_StartText;
11     Text m_TimeText;
12     Text m_LevelText;
13 public:
14     Hud();
15     Text getMessage();
16     Text getLevel();
17     Text getTime();
18
19     void setLevel(String text);
20     void setTime(String text);
21 };
```

Input.cpp File Reference

```
#include "Engine.h"
```

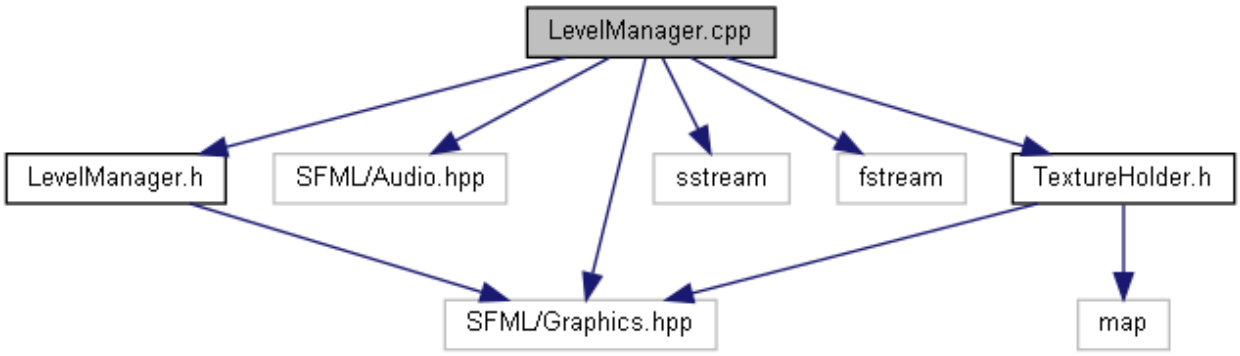
Include dependency graph for Input.cpp:



LevelManager.cpp File Reference

```
#include <SFML/Graphics.hpp>
#include <SFML/Audio.hpp>
#include "TextureHolder.h"
#include <sstream>
#include <fstream>
#include "LevelManager.h"
```

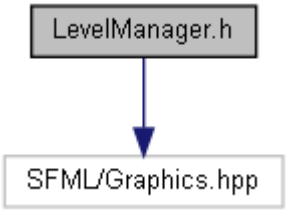
Include dependency graph for LevelManager.cpp:



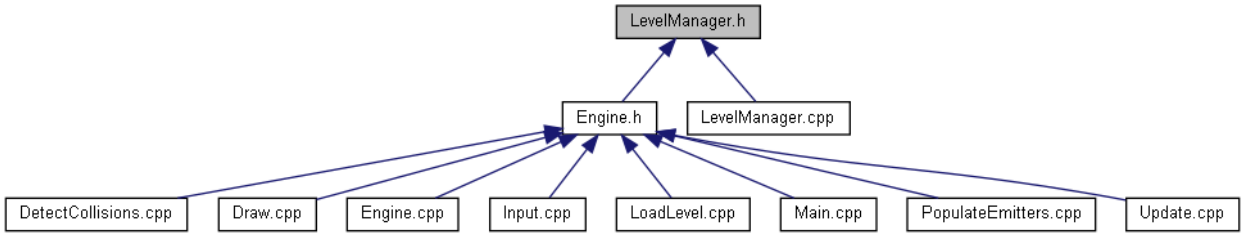
LevelManager.h File Reference

```
#include <SFML/Graphics.hpp>
```

Include dependency graph for LevelManager.h:



This graph shows which files directly or indirectly include this file:



Classes

- class **LevelManager**

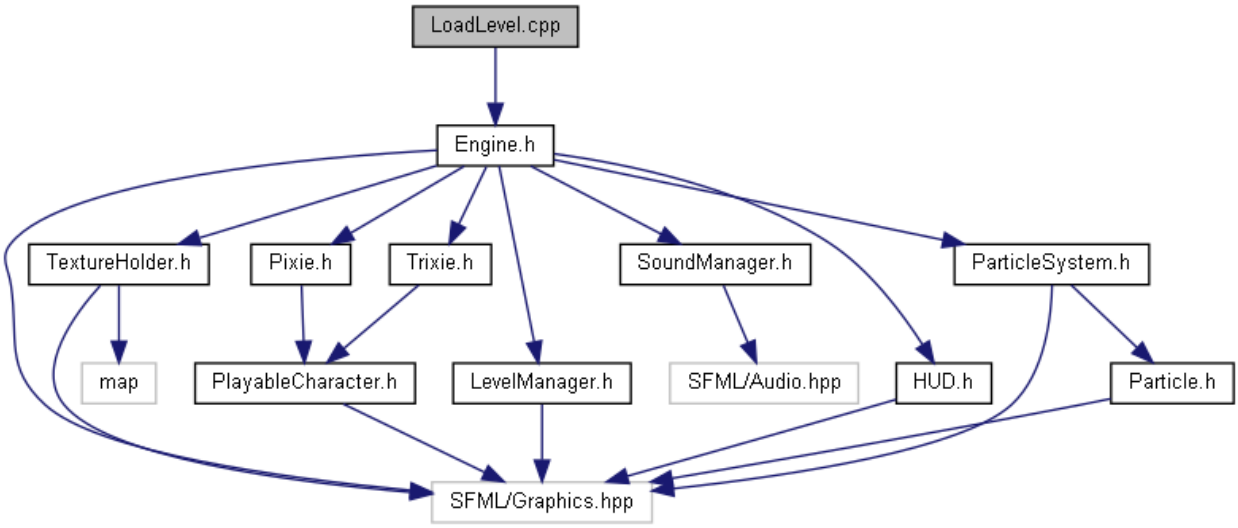
LevelManager.h

```
Go to the documentation of this file.1 #pragma once
2
3 #include <SFML/Graphics.hpp>
4 using namespace sf;
5 using namespace std;
6
7 class LevelManager
8 {
9 private:
10     Vector2i m_LevelSize;
11     Vector2f m_StartPosition;
12     float m_TimeModifier = 1;
13     float m_BaseTimeLimit = 0;
14     int m_CurrentLevel = 0;
15     const int NUM_LEVELS = 4;
16
17 public:
18     const int TILE_SIZE = 50;
19     const int VERT_IN_QUAD = 4;
20
21     float getTimeLimit();
22
23     Vector2f getStartPosition();
24
25     int** nextLevel(VertexArray& rVaLevel);
26
27     Vector2i getLevelSize();
28
29     int getCurrentLevel();
30 };
```

LoadLevel.cpp File Reference

```
#include "Engine.h"
```

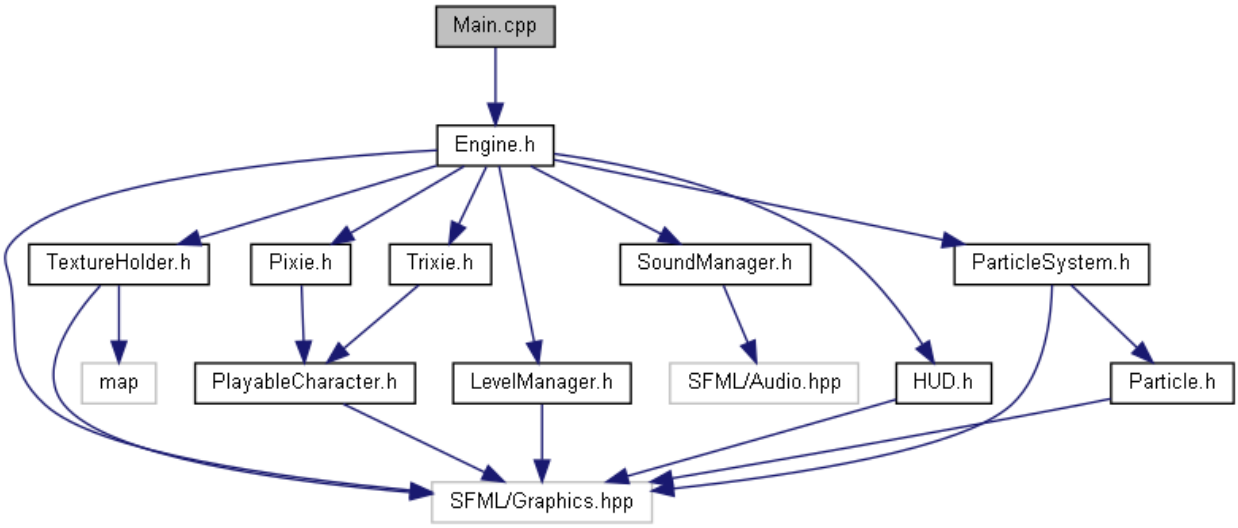
Include dependency graph for LoadLevel.cpp:



Main.cpp File Reference

```
#include "Engine.h"
```

Include dependency graph for Main.cpp:



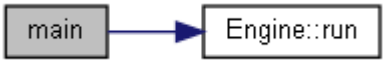
Functions

- int **main** ()

Function Documentation

int main ()

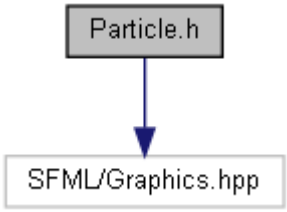
Here is the call graph for this function:



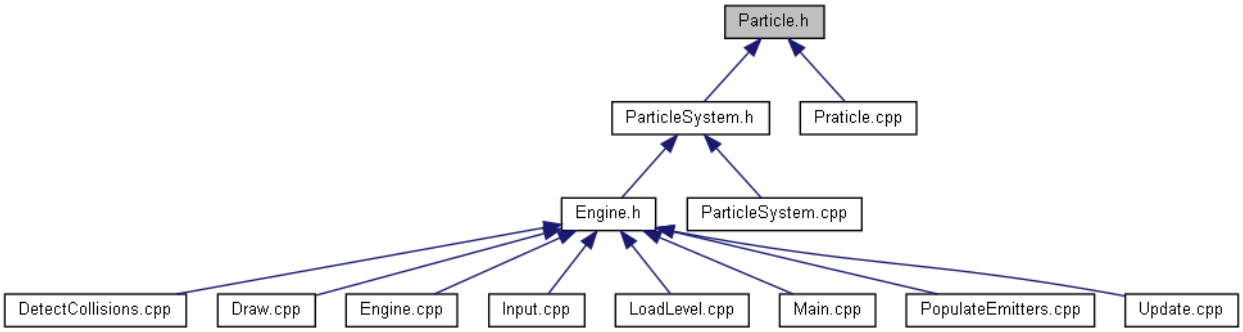
Particle.h File Reference

```
#include <SFML/Graphics.hpp>
```

Include dependency graph for Particle.h:



This graph shows which files directly or indirectly include this file:



Classes

- class **Particle**

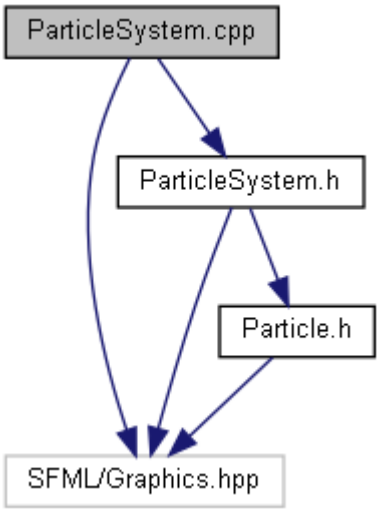
Particle.h

```
Go to the documentation of this file.1 #pragma once
2 #include <SFML/Graphics.hpp>
3
4 using namespace sf;
5
6 class Particle
7 {
8 private:
9     Vector2f m_Position;
10     Vector2f m_Velocity;
11
12 public:
13     Particle(Vector2f direction);
14
15     void update(float dt);
16
17     void setPosition(Vector2f position);
18
19     Vector2f getPosition();
20 };
```

ParticleSystem.cpp File Reference

```
#include <SFML/Graphics.hpp>
#include "ParticleSystem.h"
```

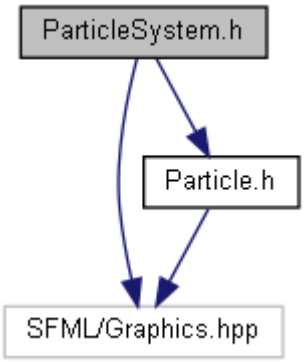
Include dependency graph for ParticleSystem.cpp:



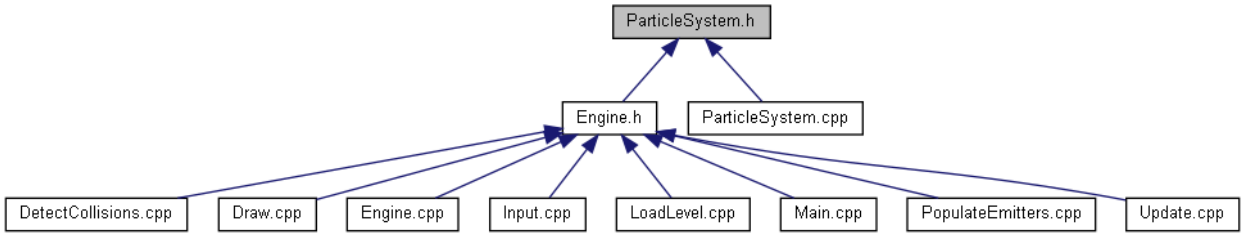
ParticleSystem.h File Reference

```
#include <SFML/Graphics.hpp>
#include "Particle.h"
```

Include dependency graph for ParticleSystem.h:



This graph shows which files directly or indirectly include this file:



Classes

- class **ParticleSystem**

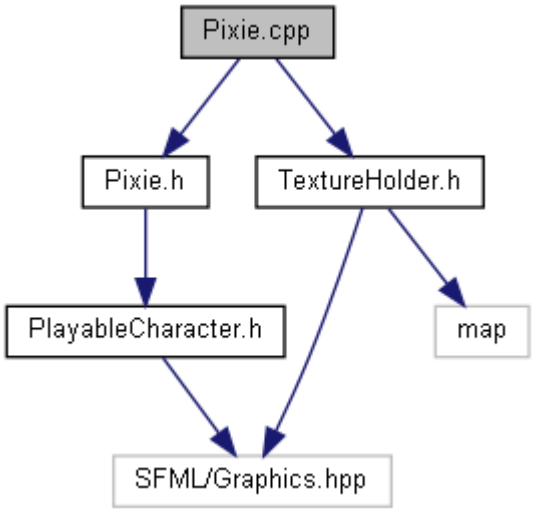
ParticleSystem.h

```
Go to the documentation of this file.1 #pragma once
2 #include <SFML/Graphics.hpp>
3 #include "Particle.h"
4
5 using namespace sf;
6 using namespace std;
7
8 class ParticleSystem : public Drawable
9 {
10 private:
11     vector<Particle> m_Particles;
12     VertexArray m_Vertices;
13     float m_Duration;
14     bool m_IsRunning = false;
15
16 public:
17     virtual void draw(RenderTarget& target,
18         RenderStates states) const;
19
20     void init(int count);
21
22     void emitParticles(Vector2f position);
23
24     void update(float elapsed);
25
26     bool running();
27 };
```

Pixie.cpp File Reference

```
#include "Pixie.h"
#include "TextureHolder.h"
```

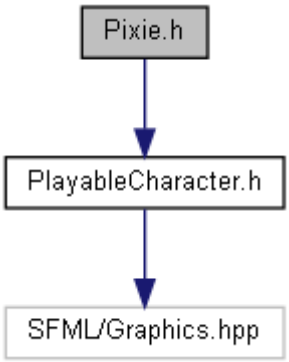
Include dependency graph for Pixie.cpp:



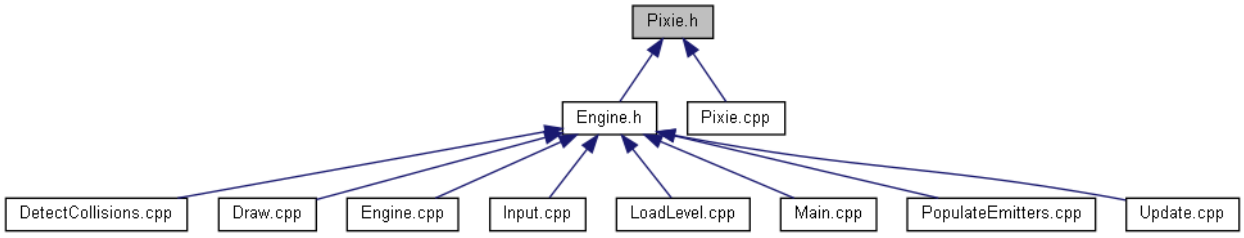
Pixie.h File Reference

```
#include "PlayableCharacter.h"
```

Include dependency graph for Pixie.h:



This graph shows which files directly or indirectly include this file:



Classes

- class **Pixie**

Pixie.h

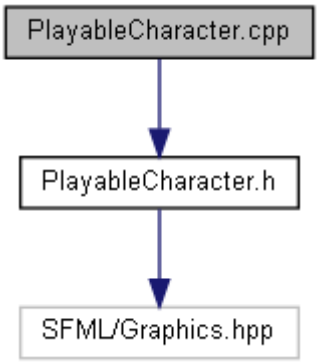
Go to the documentation of this file.

```
1 #pragma once
2 #include "PlayableCharacter.h"
3
4 class Pixie : public PlayableCharacter
5 {
6 public:
7     // A constructor specific to Pixie
8     Pixie();
9
10     // The overriden input handler for Pixie
11     bool virtual handleInput();
12
13 };
14
```

PlayableCharacter.cpp File Reference

```
#include "PlayableCharacter.h"
```

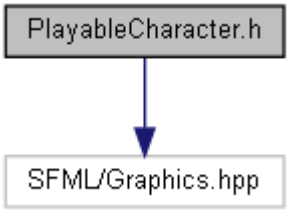
Include dependency graph for PlayableCharacter.cpp:



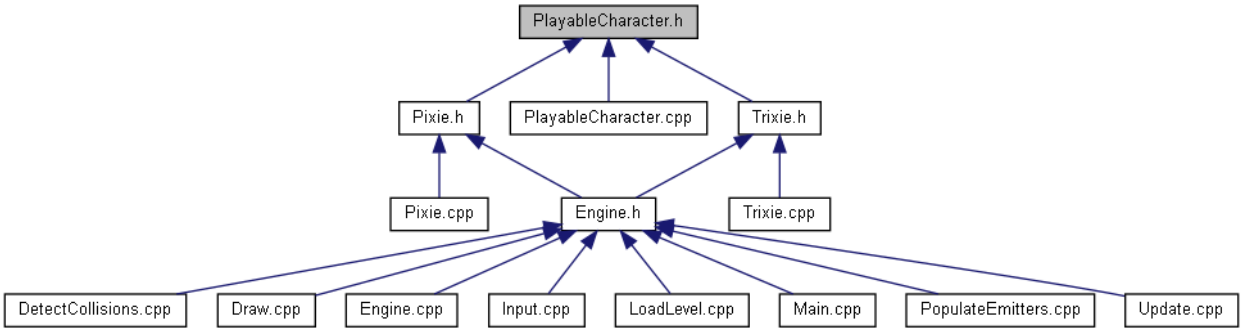
PlayableCharacter.h File Reference

```
#include <SFML/Graphics.hpp>
```

Include dependency graph for PlayableCharacter.h:



This graph shows which files directly or indirectly include this file:



Classes

- class **PlayableCharacter**

PlayableCharacter.h

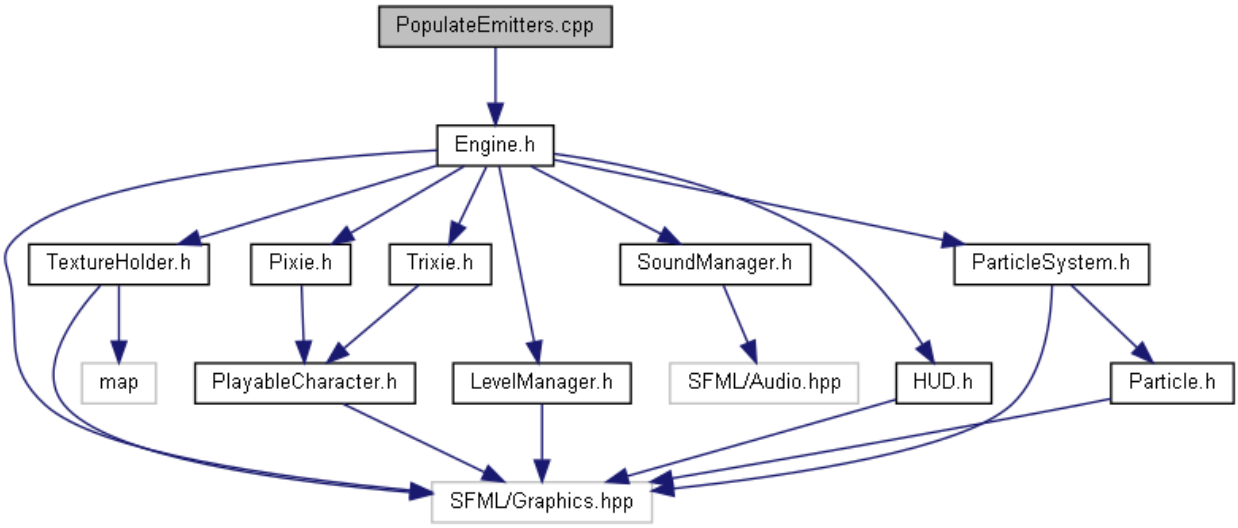
```
Go to the documentation of this file.1 #pragma once
2 #include <SFML/Graphics.hpp>
3
4 using namespace sf;
5
6 class PlayableCharacter
7 {
8 protected:
9     // Of course we will need a sprite
10     Sprite m_Sprite;
11
12     // How long does a jump last
13     float m_JumpDuration;
14
15     // Is character currently jumping or falling
16     bool m_IsJumping;
17     bool m_IsFalling;
18
19     // Which directions is the character currently moving in
20     bool m_LeftPressed;
21     bool m_RightPressed;
22
23     // How long has this jump lasted so far
24     float m_TimeThisJump;
25
26     // Has the player just initialted a jump
27     bool m_JustJumped = false;
28
29     // Private variables and functions come next
30 private:
31     // What is the gravity
32     float m_Gravity;
33
34     // How fast is the character
35     float m_Speed = 400;
36
37     // Where is the player
38     Vector2f m_Position;
39
40     // Where are the characters various body parts?
41     FloatRect m_Feet;
42     FloatRect m_Head;
43     FloatRect m_Right;
44     FloatRect m_Left;
45
46     // And a texture
47     Texture m_Texture;
48
49     // All our public functions will come next
50 public:
51
52     void spawn(Vector2f startPosition, float gravity);
53
54     // This is a pure virtual function
55     bool virtual handleInput() = 0;
56     // This class is now abstract and cannot be instanciated
57
58     // Where is the player
59     FloatRect getPosition();
60
61     // A rectangle representing the position of different parts of the sprite
62     FloatRect getFeet();
63     FloatRect getHead();
64     FloatRect getRight();
65     FloatRect getLeft();
66
67     // Send a copy of the sprite to main
68     Sprite getSprite();
69
70     // Make the character stand firm
71     void stopFalling(float position);
72     void stopRight(float position);
73     void stopLeft(float position);
```

```
74      void stopJump();
75
76      // Where is the center of the character
77      Vector2f getCenter();
78
79      // We will call this function once every frame
80      void update(float elapsedTime);
81  };
82
```

PopulateEmitters.cpp File Reference

```
#include "Engine.h"
```

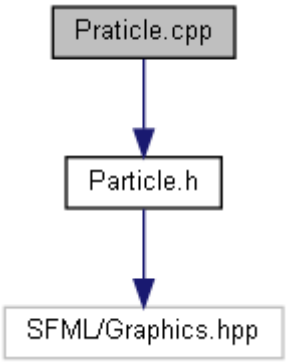
Include dependency graph for PopulateEmitters.cpp:



Praticle.cpp File Reference

```
#include "Particle.h"
```

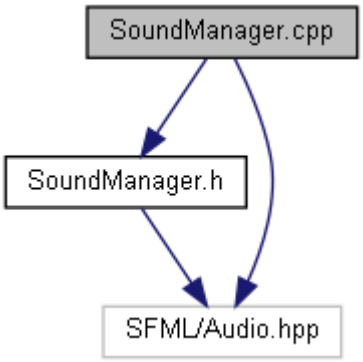
Include dependency graph for Praticle.cpp:



SoundManager.cpp File Reference

```
#include "SoundManager.h"
#include <SFML/Audio.hpp>
```

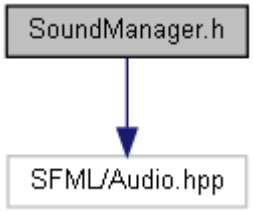
Include dependency graph for SoundManager.cpp:



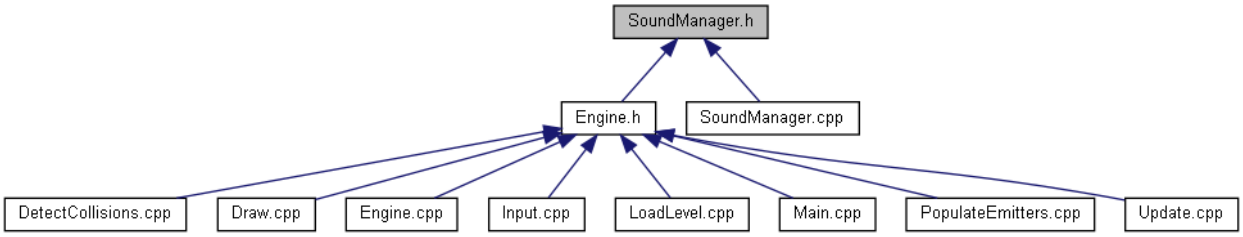
SoundManager.h File Reference

```
#include <SFML/Audio.hpp>
```

Include dependency graph for SoundManager.h:



This graph shows which files directly or indirectly include this file:



Classes

- class **SoundManager**

SoundManager.h

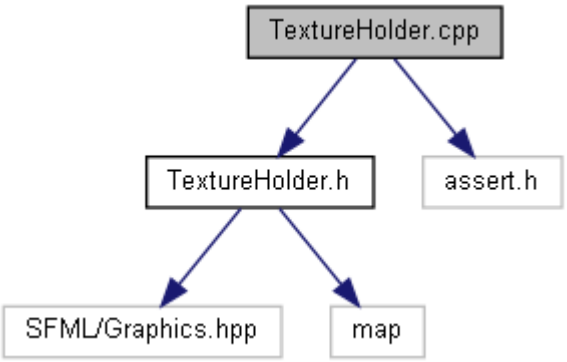
Go to the documentation of this file.

```
1 #pragma once
2 #include <SFML/Audio.hpp>
3
4 using namespace sf;
5
6 class SoundManager
7 {
8 private:
9     // The buffers
10     SoundBuffer m_FireBuffer;
11     SoundBuffer m_FallInFireBuffer;
12     SoundBuffer m_FallInWaterBuffer;
13     SoundBuffer m_JumpBuffer;
14     SoundBuffer m_ReachGoalBuffer;
15
16     // The sounds
17     Sound m_Fire1Sound;
18     Sound m_Fire2Sound;
19     Sound m_Fire3Sound;
20     Sound m_FallInFireSound;
21
22     Sound m_FallInWaterSound;
23     Sound m_JumpSound;
24     Sound m_ReachGoalSound;
25
26     // What sound to use next
27     int m_NextSound = 1;
28
29 public:
30     SoundManager();
31
32     void playFire(Vector2f emitterLocation,
33         Vector2f listenerLocation);
34
35     void playFallInFire();
36     void playFallInWater();
37     void playJump();
38     void playReachGoal();
39 };
```

TextureHolder.cpp File Reference

```
#include "TextureHolder.h"
#include <assert.h>
```

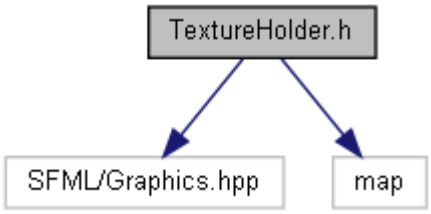
Include dependency graph for TextureHolder.cpp:



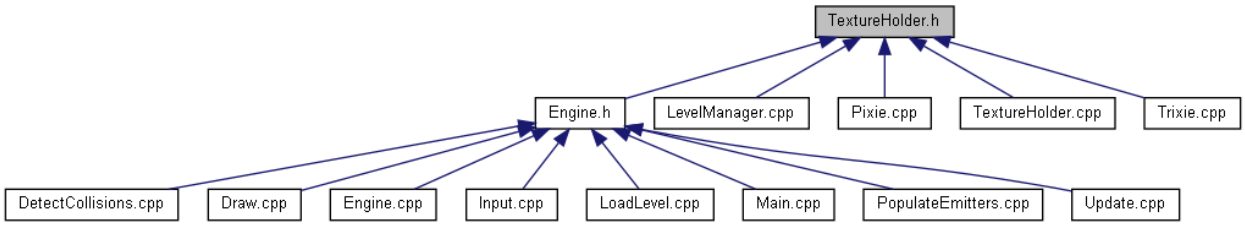
TextureHolder.h File Reference

```
#include <SFML/Graphics.hpp>
#include <map>
```

Include dependency graph for TextureHolder.h:



This graph shows which files directly or indirectly include this file:



Classes

- class **TextureHolder**

Macros

- #define **TEXTURE_HOLDER_H**

Macro Definition Documentation

#define TEXTURE_HOLDER_H

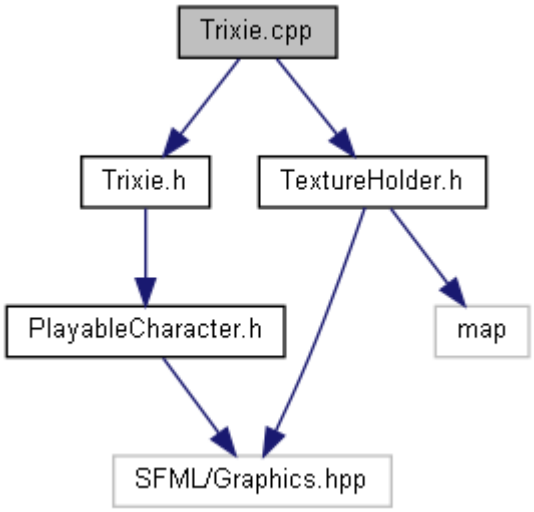
TextureHolder.h

```
Go to the documentation of this file.1 #pragma once
2 #ifndef TEXTURE_HOLDER_H
3 #define TEXTURE_HOLDER_H
4
5 #include <SFML/Graphics.hpp>
6 #include <map>
7
8 class TextureHolder
9 {
10 private:
11     // A map container from the STL,
12     // that holds related pairs of String and Texture
13     std::map<std::string, sf::Texture> m_Textures;
14
15     // A pointer of the same type as the class itself
16     // the one and only instance
17     static TextureHolder* m_s_Instance;
18
19 public:
20     TextureHolder();
21     static sf::Texture& GetTexture(std::string const& filename);
22
23 };
24
25 #endif
26
```

Trixie.cpp File Reference

```
#include "Trixie.h"
#include "TextureHolder.h"
```

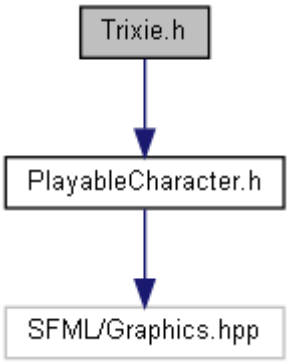
Include dependency graph for Trixie.cpp:



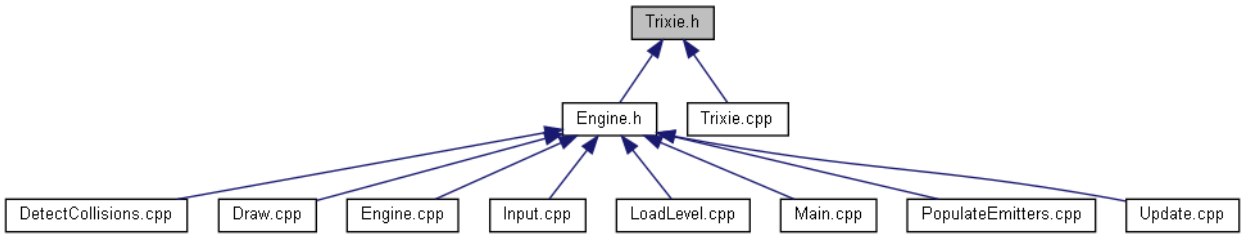
Trixie.h File Reference

```
#include "PlayableCharacter.h"
```

Include dependency graph for Trixie.h:



This graph shows which files directly or indirectly include this file:



Classes

- class **Trixie**

Trixie.h

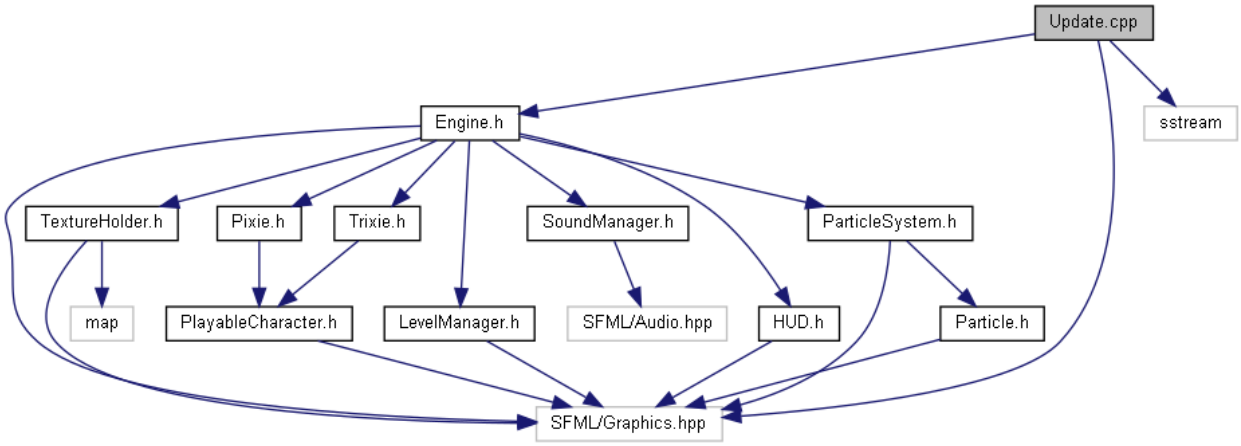
Go to the documentation of this file.

```
1 #pragma once
2 #include "PlayableCharacter.h"
3
4 class Trixie : public PlayableCharacter
5 {
6 public:
7     // A constructor specific to Trixie
8     Trixie();
9
10     // The overriden input handler for Trixie
11     bool virtual handleInput();
12
13 };
14
15
```

Update.cpp File Reference

```
#include "Engine.h"
#include <SFML/Graphics.hpp>
#include <sstream>
```

Include dependency graph for Update.cpp:

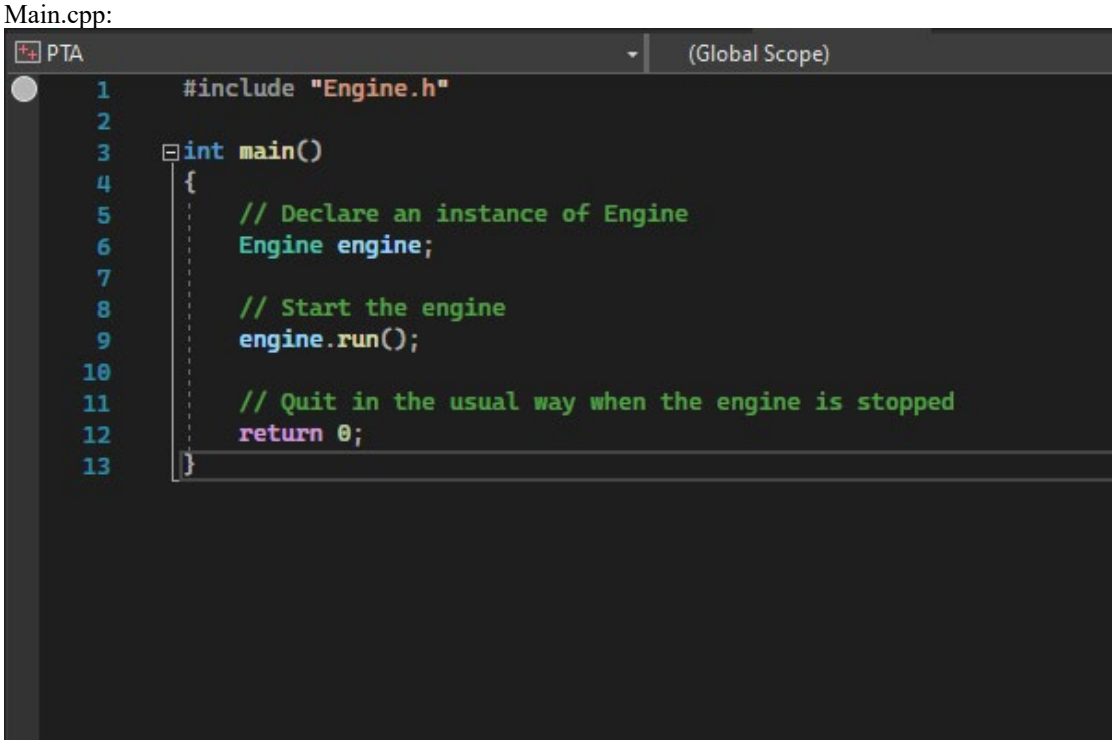


Index

INDEX

Screen Shots of All the Code:

Main.cpp:



Engine.h:

```
#pragma once
#include <SFML/Graphics.hpp>
#include "TextureHolder.h"
#include "Pixie.h"
#include "Trixie.h"
#include "LevelManager.h"
#include "SoundManager.h"
#include "HUD.h"
#include "ParticleSystem.h"

using namespace sf;

class Engine
{
private:
    // The texture holder
    TextureHolder th;

    // Create a particle system
    ParticleSystem m_PS;

    // Pixie and her friend, Trixie
    Pixie m_Pixie;
    Trixie m_Trixie;

    // A class to manage all the levels
    LevelManager m_LM;

    // SoundManager
    SoundManager m_SM;

    // The Hud
    Hud m_Hud;
    int m_FramesSinceLastHUDUpdate = 0;
    int m_TargetFramesPerHUDUpdate = 500;

    const int TILE_SIZE = 50;
    const int VERTS_IN_QUAD = 4;
```

```
39
40      // The force pushing the characters down
41      const int GRAVITY = 300;
42
43      // A regular RenderWindow
44      RenderWindow m_Window;
45
46      // The main Views
47      View m_MainView;
48      View m_LeftView;
49      View m_RightView;
50
51      // Three views for the background
52      View m_BGMainView;
53      View m_BGLeftView;
54      View m_BGRightView;
55
56      View m_HudView;
57
58      // Declare a sprite and a Texture for the background
59      Sprite m_BackgroundSprite;
60      Texture m_BackgroundTexture;
61
62      // Declare a shader for the background
63      Shader m_RippleShader;
64
65      // Is the game currently playing?
66      bool m_Playing = false;
67
68      // Is character 1 or 2 the current focus?
69      bool m_Character1 = true;
70
71      // Start in full screen mode
72      bool m_SplitScreen = false;
73
74      // How much time is left in the current level
75      float m_TimeRemaining = 10;
76      Time m_GameTimeTotal;
```

```
77
78      // Is it time for a new/first level?
79      bool m_NewLevelRequired = true;
80
81      // The vertex array for the level tiles
82      VertexArray m_VALevel;
83
84      // The 2D array with the map for the level
85      // A pointer to a pointer
86      int** m_ArrayLevel = NULL;
87
88      // Texture for the level tiles
89      Texture m_TextureTiles;
90
91      // Private functions for internal use only
92      void input();
93      void update(float dtAsSeconds);
94      void draw();
95
96      // Load a new level
97      void loadLevel();
98
99      bool detectCollisions(PlayableCharacter& character);
100
101     // Makes a vector for the best place to emit sounds from
102     void populateEmitters(vector <Vector2f>& vSoundEmitters,
103         int** arrayLevel);
104
105     // vector of Vector2f for teh fire emitter locations
106     vector <Vector2f> m_FireEmitters;
107
108 public:
109     // The Engine constructor
110     Engine();
111
112     // Run will call all the private functions
113     void run();
114
```

Engine.cpp:

```cpp
#include "Engine.h"


Engine::Engine()
{
    // Get the screen resolution and create an SFML window and View
    Vector2f resolution;
    resolution.x = VideoMode::getDesktopMode().width;
    resolution.y = VideoMode::getDesktopMode().height;

    m_Window.create(VideoMode(resolution.x, resolution.y),
        "Pixie was late",
        Style::Fullscreen);

    // Initialize the full screen view
    m_MainView.setSize(resolution);
    m_HudView.reset(
        FloatRect(0, 0, resolution.x, resolution.y));

    // Inititialize the split-screen Views
    m_LeftView.setViewport(
        FloatRect(0.001f, 0.001f, 0.498f, 0.998f));

    m_RightView.setViewport(
        FloatRect(0.5f, 0.001f, 0.499f, 0.998f));

    m_BGLeftView.setViewport(
        FloatRect(0.001f, 0.001f, 0.498f, 0.998f));

    m_BGRightView.setViewport(
        FloatRect(0.5f, 0.001f, 0.499f, 0.998f));

    // Can this graphics card use shaders?
    if (!sf::Shader::isAvailable())
    {
        // Time to get a new PC
        m_Window.close();
    }
```

```
39
40          // Can this GPU use shaders
41          if (!sf::Shader::isAvailable())
42          {
43              // BOOOO trash pc
44              // remove all the shader related code :(
45              m_Window.close();
46          }
47          else
48          {
49              // Load two shaders, a vertex and a fragment.
50              m_RippleShader.loadFromFile("shaders/vertShader.vert",
51                  "shaders/rippleShader.frag");
52          }
53
54          m_BackgroundTexture = TextureHolder::GetTexture(
55              "graphics/background.png");
56
57          // Associate the sprite with the texture
58          m_BackgroundSprite.setTexture(m_BackgroundTexture);
59
60          // Load the Texture for the background vertex array
61          m_TextureTiles = TextureHolder::GetTexture(
62              "graphics/tiles_sheet.png");
63
64          // Initialize the particle system
65          m_PS.init(1000);
66
67      }
68
69      void Engine::run()
70      {
71          // Timing
72          Clock clock;
73
74          while (m_Window.isOpen())
75          {
76              Time dt = clock.restart();
77              // Update the total game time
78              m_GameTimeTotal += dt;
79              // Make a decimal fraction from the delta time
80              float dtAsSeconds = dt.asSeconds();
81
82              input();
83              update(dtAsSeconds);
84              draw();
85          }
86      }
```

PlayableCharacter.h:

```
#pragma once
#include <SFML/Graphics.hpp>

using namespace sf;

class PlayableCharacter
{
protected:
    // Of course we will need a sprite
    Sprite m_Sprite;

    // How long does a jump last
    float m_JumpDuration;

    // Is character currently jumping or falling
    bool m_IsJumping;
    bool m_IsFalling;

    // Which directions is the character currently moving in
    bool m_LeftPressed;
    bool m_RightPressed;

    // How long has this jump lasted so far
    float m_TimeThisJump;

    // Has the player just initialted a jump
    bool m_JustJumped = false;

    // Private variables and functions come next
private:
    // What is the gravity
    float m_Gravity;

    // How fast is the character
    float m_Speed = 400;

    // Where is the player
    Vector2f m_Position;
```

```
39
40      // Where are the characters various body parts?
41      FloatRect m_Feet;
42      FloatRect m_Head;
43      FloatRect m_Right;
44      FloatRect m_Left;
45
46      // And a texture
47      Texture m_Texture;
48
49      // All our public functions will come next
50  public:
51
52      void spawn(Vector2f startPosition, float gravity);
53
54      // This is a pure virtual function
55      bool virtual handleInput() = 0;
56      // This class is now abstract and cannot be instanciated
57
58      // Where is the player
59      FloatRect getPosition();
60
61      // A rectangle representing the position of different parts of the sprite
62      FloatRect getFeet();
63      FloatRect getHead();
64      FloatRect getRight();
65      FloatRect getLeft();
66
67      // Send a copy of the sprite to main
68      Sprite getSprite();
69
70      // Make the character stand firm
71      void stopFalling(float position);
72      void stopRight(float position);
73      void stopLeft(float position);
74      void stopJump();
75
76      // Where is the center of the character
77      Vector2f getCenter();
78
79      // We will call this function once every frame
80      void update(float elapsedTime);
81  };
82
83
```

PlayableCharacter.cpp:

```cpp
#include "PlayableCharacter.h"

void PlayableCharacter::spawn(Vector2f startPosition, float gravity)
{
    // Place the player at the starting point
    m_Position.x = startPosition.x;
    m_Position.y = startPosition.y;

    // Initialize the gravity
    m_Gravity = gravity;

    // Move the sprite in to position
    m_Sprite.setPosition(m_Position);

}

void PlayableCharacter::update(float elapsedTime)
{

    if (m_RightPressed)
    {
        m_Position.x += m_Speed * elapsedTime;
    }

    if (m_LeftPressed)
    {
        m_Position.x -= m_Speed * elapsedTime;
    }


    // Handle Jumping
    if (m_IsJumping)
    {
        // Update how long the jump has been going
        m_TimeThisJump += elapsedTime;

        // Is the jump going upwards
        if (m_TimeThisJump < m_JumpDuration)
```

```
            {
                // Move up at twice gravity
                m_Position.y -= m_Gravity * 2 * elapsedTime;
            }
            else
            {
                m_IsJumping = false;
                m_IsFalling = true;
            }

        }

        // Apply gravity
        if (m_IsFalling)
        {
            m_Position.y += m_Gravity * elapsedTime;
        }

        // Update the rect for all body parts
        FloatRect r = getPosition();


        // Feet
        m_Feet.left = r.left + 3;
        m_Feet.top = r.top + r.height - 1;
        m_Feet.width = r.width - 6;
        m_Feet.height = 1;

        // Head
        m_Head.left = r.left;
        m_Head.top = r.top + (r.height * .3);
        m_Head.width = r.width;
        m_Head.height = 1;

        // Right
        m_Right.left = r.left + r.width - 2;
        m_Right.top = r.top + r.height * .35;
        m_Right.width = 1;
```

```
77          m_Right.height = r.height * .3;
78
79          // Left
80          m_Left.left = r.left;
81          m_Left.top = r.top + r.height * .5;
82          m_Left.width = 1;
83          m_Left.height = r.height * .3;
84
85          // Move the sprite into position
86          m_Sprite.setPosition(m_Position);
87
88      }
89
90      FloatRect PlayableCharacter::getPosition()
91      {
92          return m_Sprite.getGlobalBounds();
93      }
94
95      Vector2f PlayableCharacter::getCenter()
96      {
97          return Vector2f(
98              m_Position.x + m_Sprite.getGlobalBounds().width / 2,
99              m_Position.y + m_Sprite.getGlobalBounds().height / 2
100         );
101     }
102
103     FloatRect PlayableCharacter::getFeet()
104     {
105         return m_Feet;
106     }
107
108     FloatRect PlayableCharacter::getHead()
109     {
110         return m_Head;
111     }
112
113     FloatRect PlayableCharacter::getLeft()
114     {
```

```
115         return m_Left;
116     }
117
118     FloatRect PlayableCharacter::getRight()
119     {
120         return m_Right;
121     }
122
123     Sprite PlayableCharacter::getSprite()
124     {
125         return m_Sprite;
126     }
127
128
129
130     void PlayableCharacter::stopFalling(float position)
131     {
132         m_Position.y = position - getPosition().height;
133         m_Sprite.setPosition(m_Position);
134         m_IsFalling = false;
135     }
136
137     void PlayableCharacter::stopRight(float position)
138     {
139
140         m_Position.x = position - m_Sprite.getGlobalBounds().width;
141         m_Sprite.setPosition(m_Position);
142     }
143
144     void PlayableCharacter::stopLeft(float position)
145     {
146         m_Position.x = position + m_Sprite.getGlobalBounds().width;
147         m_Sprite.setPosition(m_Position);
148     }
149
150     void PlayableCharacter::stopJump()
151     {
152         // Stop a jump early
153          m_IsJumping = false;
154          m_IsFalling = true;
155     }
156
157
158
159
```

Pixie and Trixie.h:

```h
#pragma once
#include "PlayableCharacter.h"

class Pixie : public PlayableCharacter
{
public:
    // A constructor specific to Pixie
    Pixie();

    // The overriden input handler for Pixie
    bool virtual handleInput();

};
```

```h
#pragma once
#include "PlayableCharacter.h"

class Trixie : public PlayableCharacter
{
public:
    // A constructor specific to Trixie
    Trixie();

    // The overriden input handler for Trixie
    bool virtual handleInput();

};
```

Pixie.cpp:

```cpp
#include "Pixie.h"
#include "TextureHolder.h"

Pixie::Pixie()
{
    // Associate a texture with the sprite
    m_Sprite = Sprite(TextureHolder::GetTexture(
        "graphics/pixie.png"));

    m_JumpDuration = .45;
}

// A virtual function
bool Pixie::handleInput()
{
    m_JustJumped = false;

    if (Keyboard::isKeyPressed(Keyboard::W))
    {

        // Start a jump if not already jumping
        // but only if standing on a block (not falling)
        if (!m_IsJumping && !m_IsFalling)
        {
            m_IsJumping = true;
            m_TimeThisJump = 0;
            m_JustJumped = true;
        }
    }
    else
    {
        m_IsJumping = false;
        m_IsFalling = true;

    }
    if (Keyboard::isKeyPressed(Keyboard::A))
    {
        m_LeftPressed = true;
```

```
39          }
40          else
41          {
42              m_LeftPressed = false;
43          }
44
45
46          if (Keyboard::isKeyPressed(Keyboard::D))
47          {
48              m_RightPressed = true;
49          }
50          else
51          {
52              m_RightPressed = false;
53          }
54
55          return m_JustJumped;
56      }
```

Trixie.cpp:

```cpp
#include "Trixie.h"
#include "TextureHolder.h"

Trixie::Trixie()
{
    // Associate a texture with the sprite
    m_Sprite = Sprite(TextureHolder::GetTexture(
        "graphics/trixie.png"));

    m_JumpDuration = .25;
}

bool Trixie::handleInput()
{
    m_JustJumped = false;

    if (Keyboard::isKeyPressed(Keyboard::Up))
    {

        // Start a jump if not already jumping
        // but only if standing on a block (not falling)
        if (!m_IsJumping && !m_IsFalling)
        {
            m_IsJumping = true;
            m_TimeThisJump = 0;
            m_JustJumped = true;
        }

    }
    else
    {
        m_IsJumping = false;
        m_IsFalling = true;

    }
    if (Keyboard::isKeyPressed(Keyboard::Left))
    {
        m_LeftPressed = true;
```

```
39          }
40          else
41          {
42              m_LeftPressed = false;
43          }
44
45
46          if (Keyboard::isKeyPressed(Keyboard::Right))
47          {
48
49              m_RightPressed = true;;
50
51          }
52          else
53          {
54              m_RightPressed = false;
55          }
56
57          return m_JustJumped;
58      }
59
```

HUD.h:

```
1   #pragma once
2   #include <SFML/Graphics.hpp>
3
4   using namespace sf;
5
6   class Hud
7   {
8   private:
9       Font m_Font;
10      Text m_StartText;
11      Text m_TimeText;
12      Text m_LevelText;
13  public:
14      Hud();
15      Text getMessage();
16      Text getLevel();
17      Text getTime();
18
19      void setLevel(String text);
20      void setTime(String text);
21  };
```

HUD.cpp:

```cpp
#include "HUD.h"

Hud::Hud()
{
    Vector2u resolution;
    resolution.x = VideoMode::getDesktopMode().width;
    resolution.y = VideoMode::getDesktopMode().height;

    // Load the font
    m_Font.loadFromFile("fonts/Roboto-Light.ttf");

    // When paused
    m_StartText.setFont(m_Font);
    m_StartText.setCharacterSize(50);
    m_StartText.setFillColor(Color::Red);
    m_StartText.setString("Pixie & Trixie's Adventure, Enter to Start!");

    // Position text
    FloatRect textRect = m_StartText.getLocalBounds();

    m_StartText.setOrigin(textRect.left +
        textRect.width / 2.0f,
        textRect.top +
        textRect.height / 2.0f);

    m_StartText.setPosition(
        resolution.x / 2.0f, resolution.y / 2.0f);

    // Time
    m_TimeText.setFont(m_Font);
    m_TimeText.setCharacterSize(60);
    m_TimeText.setFillColor(Color::Cyan);
    m_TimeText.setPosition(resolution.x - 150, 0);
    m_TimeText.setString("------");

    // Level
    m_LevelText.setFont(m_Font);
    m_LevelText.setCharacterSize(60);
```

```
39          m_LevelText.setFillColor(Color::Magenta);
40          m_LevelText.setPosition(25, 0);
41          m_LevelText.setString("1");
42      }
43
44      Text Hud::getMessage()
45      {
46          return m_StartText;
47      }
48
49      Text Hud::getLevel()
50      {
51          return m_LevelText;
52      }
53
54      Text Hud::getTime()
55      {
56          return m_TimeText;
57      }
58
59      void Hud::setLevel(String text)
60      {
61          m_LevelText.setString(text);
62      }
63
64      void Hud::setTime(String text)
65      {
66          m_TimeText.setString(text);
67      }
68
```

Input.cpp:

```cpp
#include "Engine.h"

void Engine::input()
{
    Event event;
    while (m_Window.pollEvent(event))
    {
        if (event.type == Event::KeyPressed)
        {


            // Handle the player quitting
            if (Keyboard::isKeyPressed(Keyboard::Escape))
            {
                m_Window.close();
            }

            // Handle the player starting the game
            if (Keyboard::isKeyPressed(Keyboard::Return))
            {
                m_Playing = true;
            }

            // Switch between Pixie and Trixie
            if (Keyboard::isKeyPressed(Keyboard::Q))
            {
                m_Character1 = !m_Character1;
            }

            // Switch between full and split-screen
            if (Keyboard::isKeyPressed(Keyboard::E))
            {
                m_SplitScreen = !m_SplitScreen;
            }
        }
    }

    // Handle input specific to Pixie
    if (m_Pixie.handleInput())
    {
        // Play a jump sound
        m_SM.playJump();
    }

    // Handle input specific to Trixie
    if (m_Trixie.handleInput())
    {
        // Play a jump sound
        m_SM.playJump();
    }
}
```

Draw.cpp

```cpp
#include "Engine.h"

void Engine::draw()
{
    // Rub out the last frame
    m_Window.clear(Color::White);

    // Update shader parameters
    m_RippleShader.setUniform("uTime",
        m_GameTimeTotal.asSeconds());

    if (!m_SplitScreen)
    {
        // Switch to background view
        m_Window.setView(m_BGMainView);

        // Draw the background
        //m_Window.draw(m_BackgroundSprite);

        // Draw the background, complete with shader effect
        m_Window.draw(m_BackgroundSprite, &m_RippleShader);

        // Switch to m_MainView
        m_Window.setView(m_MainView);

        // Draw the level
        m_Window.draw(m_VALevel, &m_TextureTiles);

        // Draw Pixie
        m_Window.draw(m_Pixie.getSprite());

        // Draw Trixie
        m_Window.draw(m_Trixie.getSprite());

        // Draw the particle system
        if (m_PS.running())
        {
            m_Window.draw(m_PS);
```

```
39          }
40      }
41      else
42      {
43          // Split-screen view is active
44
45          // First draw Pixie' side of the screen
46
47          // Switch to background view
48          m_Window.setView(m_BGLeftView);
49          // Draw the background
50          //m_Window.draw(m_BackgroundSprite);
51
52          // Draw the background, complete with shader effect
53          m_Window.draw(m_BackgroundSprite, &m_RippleShader);
54
55          // Switch to m_LeftView
56          m_Window.setView(m_LeftView);
57
58          // Draw the level
59          m_Window.draw(m_VALevel, &m_TextureTiles);
60
61          // Draw Trixie
62          m_Window.draw(m_Trixie.getSprite());
63
64          // Draw Pixie
65          m_Window.draw(m_Pixie.getSprite());
66
67          // Draw particle system
68          if (m_PS.running())
69          {
70              m_Window.draw(m_PS);
71          }
72
73          // Now draw Trixie's side of the screen
74
75          // Switch to background view
76          m_Window.setView(m_BGRightView);
```

```
77          // Draw the background
78          //m_Window.draw(m_BackgroundSprite);
79
80          // Draw the background, complete with shader effect
81          m_Window.draw(m_BackgroundSprite, &m_RippleShader);
82
83          // Switch to m_RightView
84          m_Window.setView(m_RightView);
85
86          // Draw the level
87          m_Window.draw(m_VALevel, &m_TextureTiles);
88
89          // Draw Pixie
90          m_Window.draw(m_Pixie.getSprite());
91
92          // Draw Trixie
93          m_Window.draw(m_Trixie.getSprite());
94
95          // Draw particle system
96          if (m_PS.running())
97          {
98              m_Window.draw(m_PS);
99          }
100
101     }
102
103     // Draw the HUD
104     // Switch to m_HudView
105     m_Window.setView(m_HudView);
106     m_Window.draw(m_Hud.getLevel());
107     m_Window.draw(m_Hud.getTime());
108     if (!m_Playing)
109     {
110         m_Window.draw(m_Hud.getMessage());
111     }
112
113     // Show everything we have just drawn
114     m_Window.display();
```

LevelManager.h

```h
#pragma once

#include <SFML/Graphics.hpp>
using namespace sf;
using namespace std;

class LevelManager
{
private:
    Vector2i m_LevelSize;
    Vector2f m_StartPosition;
    float m_TimeModifier = 1;
    float m_BaseTimeLimit = 0;
    int m_CurrentLevel = 0;
    const int NUM_LEVELS = 4;

public:
    const int TILE_SIZE = 50;
    const int VERT_IN_QUAD = 4;

    float getTimeLimit();

    Vector2f getStartPosition();

    int** nextLevel(VertexArray& rVaLevel);

    Vector2i getLevelSize();

    int getCurrentLevel();
};
```

LevelManager.cpp

```cpp
#include <SFML/Graphics.hpp>
#include <SFML/Audio.hpp>
#include "TextureHolder.h"
#include <sstream>
#include <fstream>
#include "LevelManager.h"

using namespace sf;
using namespace std;

int** LevelManager::nextLevel(VertexArray& rVaLevel)
{
    m_LevelSize.x = 0;
    m_LevelSize.y = 0;

    // Get the next level
    m_CurrentLevel++;
    if (m_CurrentLevel > NUM_LEVELS)
    {
        m_CurrentLevel = 1;
        m_TimeModifier -= .1f;
    }

    // Load the appropriate level from a text file
    string levelToLoad;
    switch (m_CurrentLevel)
    {
    case 1:
        levelToLoad = "levels/level1.txt";
        m_StartPosition.x = 100;
        m_StartPosition.y = 100;
        m_BaseTimeLimit = 30.0f;
        break;

    case 2:
        levelToLoad = "levels/level2.txt";
        m_StartPosition.x = 100;
        m_StartPosition.y = 3600;
```

```
39          m_BaseTimeLimit = 100.0f;
40          break;
41
42      case 3:
43          levelToLoad = "levels/level3.txt";
44          m_StartPosition.x = 1250;
45          m_StartPosition.y = 0;
46          m_BaseTimeLimit = 30.0f;
47          break;
48
49      case 4:
50          levelToLoad = "levels/level4.txt";
51          m_StartPosition.x = 50;
52          m_StartPosition.y = 200;
53          m_BaseTimeLimit = 50.0f;
54          break;
55
56      }// End Switch
57
58      ifstream inputFile(levelToLoad);
59      string s;
60
61      // Count the number of row in the file
62      while (getline(inputFile, s))
63      {
64          ++m_LevelSize.y;
65      }
66
67      // Store the length of the rows
68      m_LevelSize.x = s.length();
69
70      // Go back to the start of the file
71      inputFile.clear();
72      inputFile.seekg(0, ios::beg);
73
74      // Prepare the 2D array to hold the int values from the file
75      int** arrayLevel = new int* [m_LevelSize.y];
76      for (int i = 0; i < m_LevelSize.y; i++)
```

```
77          {
78              // Add a new array into each array element
79              arrayLevel[i] = new int[m_LevelSize.x];
80          }
81
82          // Loop through the file and store all
83          // the values in the 2D array
84          string row;
85          int y = 0;
86          while (inputFile >> row)
87          {
88              for (int x = 0; x < row.length(); x++) {
89                  const char val = row[x];
90                  arrayLevel[y][x] = atoi(&val);
91              }
92
93              y++;
94          }
95
96          // Close the file
97          inputFile.close();
98
99          // What type of primitive is being used
100         rVaLevel.setPrimitiveType(Quads);
101
102         // Set the size of the vertex array
103         rVaLevel.resize(m_LevelSize.x *
104             m_LevelSize.y * VERT_IN_QUAD);
105
106         // Start at the beginning of the vertex array
107         int currentVertex = 0;
108
109         for (int x = 0; x < m_LevelSize.x; x++)
110         {
111             for (int y = 0; y < m_LevelSize.y; y++)
112             {
113                 // Position each vertex in the current quad
114                 rVaLevel[currentVertex + 0].position =
```

```
                Vector2f(x * TILE_SIZE,
                    y * TILE_SIZE);
                rVaLevel[currentVertex + 1].position =
                    Vector2f((x * TILE_SIZE) + TILE_SIZE,
                        y * TILE_SIZE);
                rVaLevel[currentVertex + 2].position =
                    Vector2f((x * TILE_SIZE) + TILE_SIZE,
                        (y * TILE_SIZE) + TILE_SIZE);
                rVaLevel[currentVertex + 3].position =
                    Vector2f((x * TILE_SIZE),
                        (y * TILE_SIZE) + TILE_SIZE);

                // Which tile from the sprite sheet should we use
                int verticalOffset = arrayLevel[y][x] * TILE_SIZE;

                rVaLevel[currentVertex + 0].texCoords =
                    Vector2f(0, 0 + verticalOffset);

                rVaLevel[currentVertex + 1].texCoords =
                    Vector2f(TILE_SIZE, 0 + verticalOffset);

                rVaLevel[currentVertex + 2].texCoords =
                    Vector2f(TILE_SIZE, TILE_SIZE + verticalOffset);

                rVaLevel[currentVertex + 3].texCoords =
                    Vector2f(0, TILE_SIZE + verticalOffset);

                // Position ready for the next four vertices
                currentVertex = currentVertex + VERT_IN_QUAD;
            }
        }

        return arrayLevel;
    }

    Vector2i LevelManager::getLevelSize()
    {
        return m_LevelSize;
    }

    int LevelManager::getCurrentLevel()
    {
        return m_CurrentLevel;
    }

    float LevelManager::getTimeLimit()
    {
        return m_BaseTimeLimit * m_TimeModifier;

    }
    Vector2f LevelManager::getStartPosition()
    {
        return m_StartPosition;
    }
```

LoadLevel.cpp

```cpp
#include "Engine.h"

void Engine::loadLevel()
{
    m_Playing = false;

    // Delete the previously allocated memory
    for (int i = 0; i < m_LM.getLevelSize().y; i++)
    {
        delete[] m_ArrayLevel[i];
    }
    delete[] m_ArrayLevel;

    // Load the next 2D array with map for the level
    // and repopulate the vertex array as well
    m_ArrayLevel = m_LM.nextLevel(m_VALevel);

    // Prepare the sound emitters
    populateEmitters(m_FireEmitters, m_ArrayLevel);

    // How long is this new time limit
    m_TimeRemaining = m_LM.getTimeLimit();

    // Spawn Pixie and Trixie
    m_Pixie.spawn(m_LM.getStartPosition(), GRAVITY);
    m_Trixie.spawn(m_LM.getStartPosition(), GRAVITY);

    // Make sure this code isn't ran again
    m_NewLevelRequired = false;
}
```

TextureHolder.h

```h
#pragma once
#ifndef TEXTURE_HOLDER_H
#define TEXTURE_HOLDER_H

#include <SFML/Graphics.hpp>
#include <map>

class TextureHolder
{
private:
    // A map container from the STL,
    // that holds related pairs of String and Texture
    std::map<std::string, sf::Texture> m_Textures;

    // A pointer of the same type as the class itself
    // the one and only instance
    static TextureHolder* m_s_Instance;

public:
    TextureHolder();
    static sf::Texture& GetTexture(std::string const& filename);

};

#endif
```

TextureHolder.cpp

```cpp
#include "TextureHolder.h"
#include <assert.h>

using namespace sf;
using namespace std;

 TextureHolder* TextureHolder::m_s_Instance = nullptr;

TextureHolder::TextureHolder()
{
    assert(m_s_Instance == nullptr);
    m_s_Instance = this;
}

sf::Texture& TextureHolder::GetTexture(std::string const& filename)
{
    // Get a reference to m_Textures using m_S_Instance
    auto& m = m_s_Instance->m_Textures;
    // auto is the equivalent of map<string, Texture>

    // Create an iterator to hold a key-value-pair (kvp)
    // and search for the required kvp
    // using the passed in file name
    auto keyValuePair = m.find(filename);
    // auto is equivalent of map<string, Texture>::iterator


    // Did we find a match?
    if (keyValuePair != m.end())
    {
        // Yes
        // Return the texture,
        // the second part of the kvp, the texture
        return keyValuePair->second;
    }
    else
    {
        // File name not found
        // Create a new key value pair using the filename
        auto& texture = m[filename];
        // Load the texture from file in the usual way
        texture.loadFromFile(filename);

        // Return the texture to the calling code
        return texture;
    }
}
```

ParticalSystem and Particle.h

```h
#pragma once
#include <SFML/Graphics.hpp>
#include "Particle.h"

using namespace sf;
using namespace std;

class ParticleSystem : public Drawable
{
private:
    vector<Particle> m_Particles;
    VertexArray m_Vertices;
    float m_Duration;
    bool m_IsRunning = false;

public:
    virtual void draw(RenderTarget& target,
        RenderStates states) const;

    void init(int count);

    void emitParticles(Vector2f position);

    void update(float elapsed);

    bool running();
};
```

```h
#pragma once
#include <SFML/Graphics.hpp>

using namespace sf;

class Particle
{
private:
    Vector2f m_Position;
    Vector2f m_Velocity;

public:
    Particle(Vector2f direction);

    void update(float dt);

    void setPosition(Vector2f position);

    Vector2f getPosition();
};
```

ParticleSystem.cpp

```cpp
#include <SFML/Graphics.hpp>
#include "ParticleSystem.h"

using namespace sf;
using namespace std;

void ParticleSystem::init(int numParticles)
{
    m_Vertices.setPrimitiveType(Points);
    m_Vertices.resize(numParticles);

    // Create the particles

    for (int i = 0; i < numParticles; i++)
    {
        srand(time(0) + i);
        float angle = (rand() % 360) * 3.14f / 180.f;
        float speed = (rand() % 600) + 600.f;

        Vector2f direction;
        direction = Vector2f(cos(angle) * speed,
            sin(angle) * speed);

        m_Particles.push_back(Particle(direction));
    }
}

void ParticleSystem::update(float dt)
{
    m_Duration -= dt;
    vector<Particle>::iterator i;
    int currentVertex = 0;

```

```
34      for (i = m_Particles.begin(); i != m_Particles.end(); i++)
35      {
36          // Move the particle
37          (*i).update(dt);
38          // Update the vertex array
39          m_Vertices[currentVertex++].position = i->getPosition();
40      }
41
42      if (m_Duration < 0)
43      {
44          m_IsRunning = false;
45      }
46  }
47
48  void ParticleSystem::emitParticles(Vector2f startPosition)
49  {
50      m_IsRunning = true;
51      m_Duration = 2;
52
53      int currentVertex = 0;
54
55      for (auto it = m_Particles.begin();
56          it != m_Particles.end();
57          it++)
58      {
59          m_Vertices[currentVertex++].color = Color::Yellow;
60          it->setPosition(startPosition);
61      }
62  }
63
64  void ParticleSystem::draw(RenderTarget& target,
65      RenderStates states) const
66  {
67      target.draw(m_Vertices, states);
68  }
69
70  bool ParticleSystem::running()
71  {
72      return m_IsRunning;
73  }
```

Particle.cpp

```cpp
#include "Particle.h"

Particle::Particle(Vector2f direction)
{
    // Determine the direction
    m_Velocity.x = direction.x;
    m_Velocity.y = direction.y;
}

void Particle::update(float dtAsSeconds)
{
    // Move the particle
    m_Position += m_Velocity * dtAsSeconds;
}

void Particle::setPosition(Vector2f position)
{
    m_Position = position;
}

Vector2f Particle::getPosition()
{
    return m_Position;
}
```

SoundManager.h

```
#pragma once
#include <SFML/Audio.hpp>

using namespace sf;

class SoundManager
{
private:
    // The buffers
    SoundBuffer m_FireBuffer;
    SoundBuffer m_FallInFireBuffer;
    SoundBuffer m_FallInWaterBuffer;
    SoundBuffer m_JumpBuffer;
    SoundBuffer m_ReachGoalBuffer;

    // The sounds
    Sound m_Fire1Sound;
    Sound m_Fire2Sound;
    Sound m_Fire3Sound;
    Sound m_FallInFireSound;

    Sound m_FallInWaterSound;
    Sound m_JumpSound;
    Sound m_ReachGoalSound;

    // What sound to use next
    int m_NextSound = 1;

public:
    SoundManager();

    void playFire(Vector2f emitterLocation,
        Vector2f listenerLocation);

    void playFallInFire();
    void playFallInWater();
    void playJump();
    void playReachGoal();
};

```

SoundManager.cpp

```cpp
#include "SoundManager.h"
#include <SFML/Audio.hpp>

using namespace sf;

SoundManager::SoundManager()
{
    // Load the sound in to the buffers
    m_FireBuffer.loadFromFile("sound/fire1.wav");
    m_FallInFireBuffer.loadFromFile("sound/fallinfire.wav");
    m_FallInWaterBuffer.loadFromFile("sound/fallinwater.wav");
    m_JumpBuffer.loadFromFile("sound/jump.wav");
    m_ReachGoalBuffer.loadFromFile("sound/reachgoal.wav");

    // Associate sounds with buffers
    m_Fire1Sound.setBuffer(m_FallInFireBuffer);
    m_Fire2Sound.setBuffer(m_FallInFireBuffer);
    m_Fire3Sound.setBuffer(m_FallInFireBuffer);
    m_FallInFireSound.setBuffer(m_FallInFireBuffer);
    m_FallInWaterSound.setBuffer(m_FallInWaterBuffer);
    m_JumpSound.setBuffer(m_JumpBuffer);
    m_ReachGoalSound.setBuffer(m_ReachGoalBuffer);

    // When player is close sound is full volume
    float minDistance = 150;
    // The sound reduces as the player moves away
    float attenuation = 15;

    // Setting attenuation levels
    m_Fire1Sound.setAttenuation(attenuation);
    m_Fire2Sound.setAttenuation(attenuation);
    m_Fire3Sound.setAttenuation(attenuation);

```

```
34      // Setting min distance
35      m_Fire1Sound.setMinDistance(minDistance);
36      m_Fire2Sound.setMinDistance(minDistance);
37      m_Fire3Sound.setMinDistance(minDistance);
38
39      // Loop for all the fire sounds when played
40      m_Fire1Sound.setLoop(true);
41      m_Fire2Sound.setLoop(true);
42      m_Fire3Sound.setLoop(true);
43  }
44
45  void SoundManager::playFire(
46      Vector2f emitterLocation, Vector2f listenerLocation)
47  {
48      // Who is the listener? Pixie
49      Listener::setPosition(listenerLocation.x,
50          listenerLocation.y, 0.0f);
51
52      switch (m_NextSound)
53      {
54      case 1:
55          // locate and move the source
56          m_Fire1Sound.setPosition(emitterLocation.x,
57              emitterLocation.y, 0.0f);
58
59          if (m_Fire1Sound.getStatus() == Sound::Status::Stopped)
60          {
61              // Play the sound if not already
62              m_Fire1Sound.play();
63          }
64          break;
65
66      case 2:
```

```
            // Same as the first sound but for 2
            m_Fire2Sound.setPosition(emitterLocation.x,
                emitterLocation.y, 0.0f);

            if (m_Fire2Sound.getStatus() == Sound::Status::Stopped)
            {
                // Play the sound if not already
                m_Fire2Sound.play();
            }
            break;

        case 3:
            // Again for the 3rd
            m_Fire3Sound.setPosition(emitterLocation.x,
                emitterLocation.y, 0.0f);

            if (m_Fire3Sound.getStatus() == Sound::Status::Stopped)
            {
                // Play the sound if not already
                m_Fire3Sound.play();
            }
            break;
        }

        // Increment to the next sound
        m_NextSound++;

        // Loop back to 1 after the 3rd has played
        if (m_NextSound > 3)
        {
            m_NextSound = 1;
        }
    }
```

```
100
101 void SoundManager::playFallInFire()
102 {
103     m_FallInFireSound.setRelativeToListener(true);
104     m_FallInFireSound.play();
105 }
106
107 void SoundManager::playFallInWater()
108 {
109     m_FallInWaterSound.setRelativeToListener(true);
110     m_FallInWaterSound.play();
111 }
112
113 void SoundManager::playJump()
114 {
115     m_JumpSound.setRelativeToListener(true);
116     m_JumpSound.play();
117 }
118
119 void SoundManager::playReachGoal()
120 {
121     m_ReachGoalSound.setRelativeToListener(true);
122     m_ReachGoalSound.play();
123 }
124
```

PopulateEmitters.cpp

```
1  #include "Engine.h"
2
3  using namespace sf;
4  using namespace std;
5
6  void Engine::populateEmitters(
7      vector <Vector2f>& vSoundEmitters,
8      int** arrayLevel)
9  {
10     // Empty the vector
11     vSoundEmitters.empty();
12
13     // Track the emitter to no make to many
14     FloatRect previousEmitter;
15
16     // Search for fire in level
17     for (int x = 0; x < (int)m_LM.getLevelSize().x; x++)
18     {
19         for (int y = 0; y < (int)m_LM.getLevelSize().y; y++)
20         {
21             if (arrayLevel[y][x] == 2) // fire is present
22             {
23                 // Skip any fire tiles next to previous emitter
24                 if (!FloatRect(x * TILE_SIZE,
25                     y * TILE_SIZE,
26                     TILE_SIZE,
27                     TILE_SIZE).intersects(previousEmitter))
28                 {
29                     // Add coordinates of water block
30                     vSoundEmitters.push_back(
31                         Vector2f(x * TILE_SIZE, y * TILE_SIZE));
32
33                     // make rectangle 6x6 to not make to many emitters close to this one
```

```
                    previousEmitter.left = x * TILE_SIZE;
                    previousEmitter.top = y * TILE_SIZE;
                    previousEmitter.width = TILE_SIZE * 6;
                    previousEmitter.height = TILE_SIZE * 6;
                }
            }
        }
    }
    return;
}
```

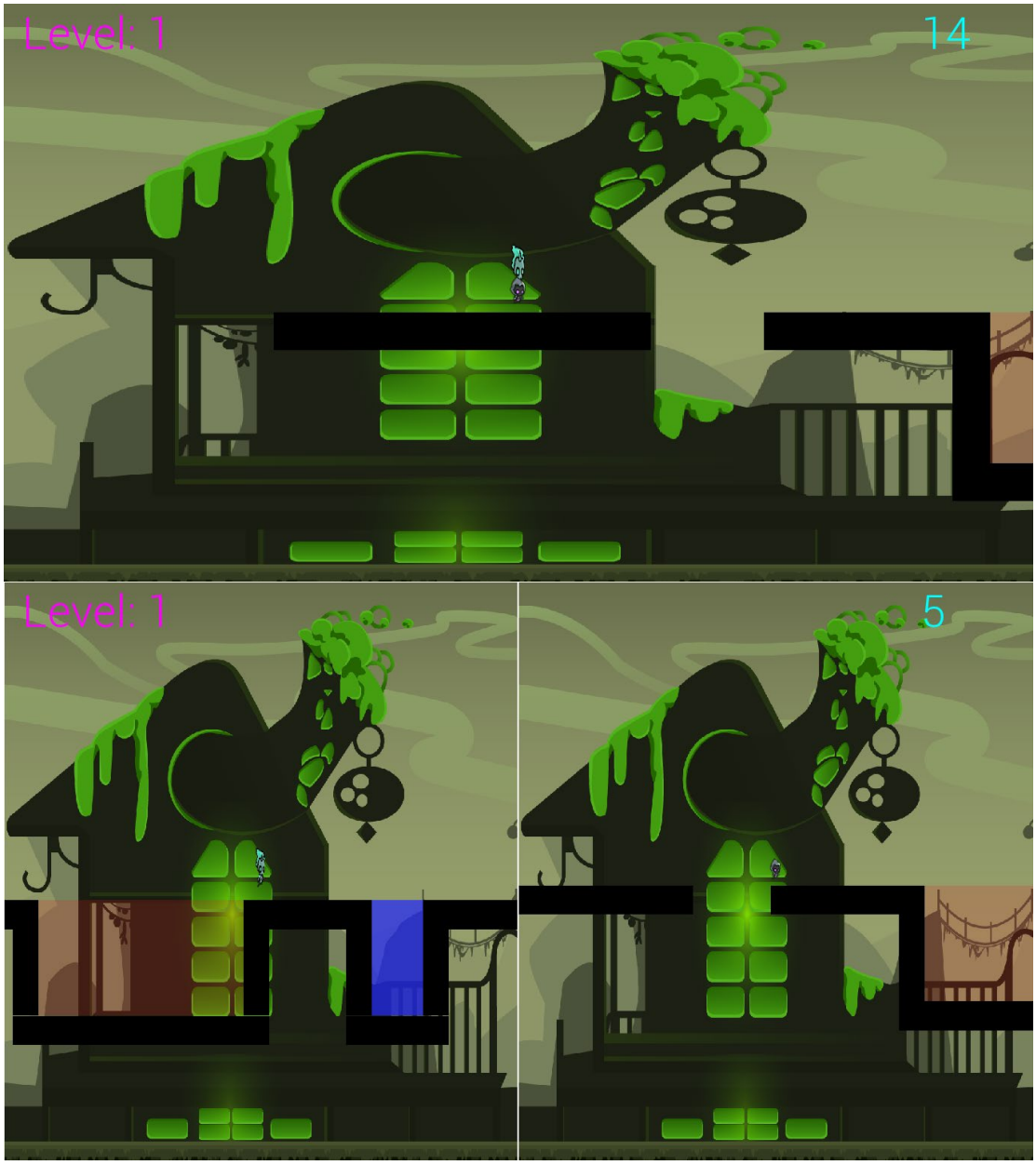
Update.cpp

```
#include "Engine.h"
#include <SFML/Graphics.hpp>
#include <sstream>

using namespace sf;

void Engine::update(float dtAsSeconds)
{
    if (m_NewLevelRequired)
    {
        // Load a level
        loadLevel();
    }

    if (m_Playing)
    {
        // Update Pixie
        m_Pixie.update(dtAsSeconds);

        // Update Trixie
        m_Trixie.update(dtAsSeconds);

        // Detect collision and if the player reached goal,
        // second part is only if Pixie reached goal.
        if (detectCollisions(m_Pixie) && detectCollisions(m_Trixie))
        {
            // New level required
            m_NewLevelRequired = true;

            // Play sound for goal
            m_SM.playReachGoal();

        }
```

```
        else
        {
            // Trixie's collision detection
            detectCollisions(m_Trixie);
        }

        // Let the two jump on each other
        if (m_Trixie.getFeet().intersects(m_Pixie.getHead()))
        {
            m_Trixie.stopFalling(m_Pixie.getHead().top);
        }
        else if (m_Pixie.getFeet().intersects(m_Trixie.getHead()))
        {
            m_Pixie.stopFalling(m_Trixie.getHead().top);
        }

        // Count down the time the player has left
        m_TimeRemaining -= dtAsSeconds;

        // Have Pixie and Trixie run out of time?
        if (m_TimeRemaining <= 0)
        {
            m_NewLevelRequired = true;
        }

    }// End if playing

    // Check if fire sound needed
    vector<Vector2f>::iterator it;
    // Iterate through the vector
    for (it = m_FireEmitters.begin(); it != m_FireEmitters.end(); it++)
    {
        // Find and store emitter
        float posX = (*it).x;
        float posY = (*it).y;

        // Is the emitter near the player and make 500 pixel rect around the emitter
        FloatRect localRect(posX - 250, posY - 250, 500, 500);

        // Check if player is inside localRect
        if (m_Pixie.getPosition().intersects(localRect))
        {
            // Play sound and pass in location
            m_SM.playFire(Vector2f(posX, posY), m_Pixie.getCenter());
        }
    }

    // Set the appropriate view around the appropriate character
    if (m_SplitScreen)
    {
        m_LeftView.setCenter(m_Pixie.getCenter());
        m_RightView.setCenter(m_Trixie.getCenter());
    }
    else
    {
        // Centre full screen around appropriate character
        if (m_Character1)
        {
            m_MainView.setCenter(m_Pixie.getCenter());
        }
        else
        {
            m_MainView.setCenter(m_Trixie.getCenter());
        }
    }

```

```
100     // Time to update the Hud
101     // Increment the number of frames since
102     // the last HUD calculation
103     m_FramesSinceLastHUDUpdate++;
104
105     // Update the HUD every m_TargetFramesPerHUDUpdate frames
106     if (m_FramesSinceLastHUDUpdate > m_TargetFramesPerHUDUpdate)
107     {
108         // Update game HUD text
109         stringstream ssTime;
110         stringstream ssLevel;
111
112         // Update time text
113         ssTime << (int)m_TimeRemaining;
114         m_Hud.setTime(ssTime.str());
115
116         // Update level text
117         ssLevel << "Level: " << m_LM.getCurrentLevel();
118         m_Hud.setLevel(ssLevel.str());
119         m_FramesSinceLastHUDUpdate = 0;
120     }
121
122     // Update the particles
123     if (m_PS.running())
124     {
125         m_PS.update(dtAsSeconds);
126     }
127 } // End of update function
```

Game Play!!!


Level: 1
14
Level: 1
5