2D  
Platformer Game Engine

Class & file structure



Heartbroken Snakes | GEA

# Class Structure

## Main\_Engine

When first planning our game engine, we decided that we would eventually try to implement additional boost tasks such as making our engine compatible with the Phyre Engine and PS Vita dev kits. In addition we would also try to implement Arcade machine functionality. The inclusion of these extra aspects of functionality into our project meant that we would have a much larger task at hand and as a result, our class and file structure would become large and quite complex.

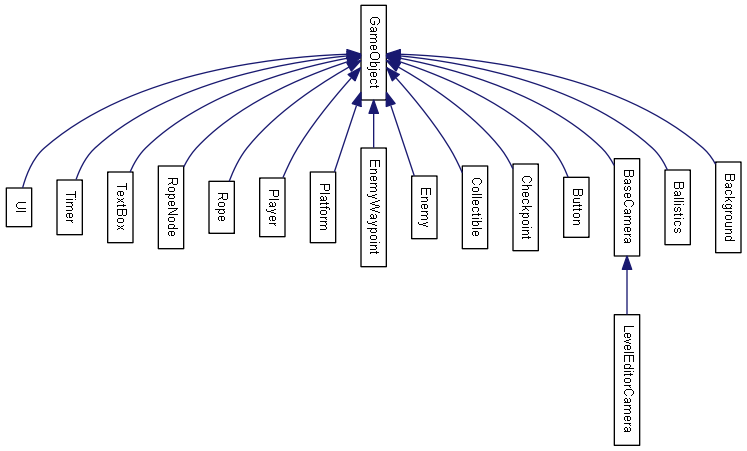
To begin, our application is initialized using the main.cpp file in the Application project. This then initializes the application window before, going into the main game loop. The main game loop is located in our Main\_Engine project which sets up the main aspects of the game. These being:

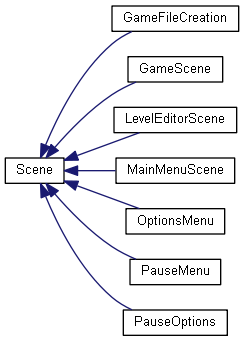
* Input manager
* Renderer
* Collision manager
* Game controller
* Object Factory
* Platforms
* Scene manager

The first diagram shows us some of the game engine’s base classes. These classes are where the game’s main logic exists, each performing an integral task in the overall engine functionality. In some part these classes were created in order to make it easier to implement the additional functionality mentioned previously. However, in order to implement the DirectX and Phyre engine specific code, we created the DXTK wrapper which will be discussed at a later stage.



In addition to the scene class, many of the items in the game are of type Game Object. In our second diagram (below), we can see the extent of this hierarchy and how exactly game objects within our engine, are related. The game object class is by far our largest parent class. This base has allowed us to easily manage and expand our game to include all and more of the elements first shown in our Rapid Prototype build.



Here we can see that the main menu class is of type scene. As a result it is handled by our scene manager class. The Scene manger class is responsible for all of the different scenes in the game including the menus, level editor and main game scene.

During the later stages of our development however, we decided not to include Phyre engine support. However, our implementation at early stages meant that certain features of our game engine are widely expandable.

## DXTK\_Wrapper Project

As mentioned previously because we wanted to implement Phyre engine functionality we created a DirectX oriented project as well as also creating a Phyre engine project, however, this was later deleted.

The DXTK\_Wrapper contains within it most of the main files of our engine. This project is where we can find the CustomMath, DXTKRenderer, GameFileLoader and Input\_manager classes.

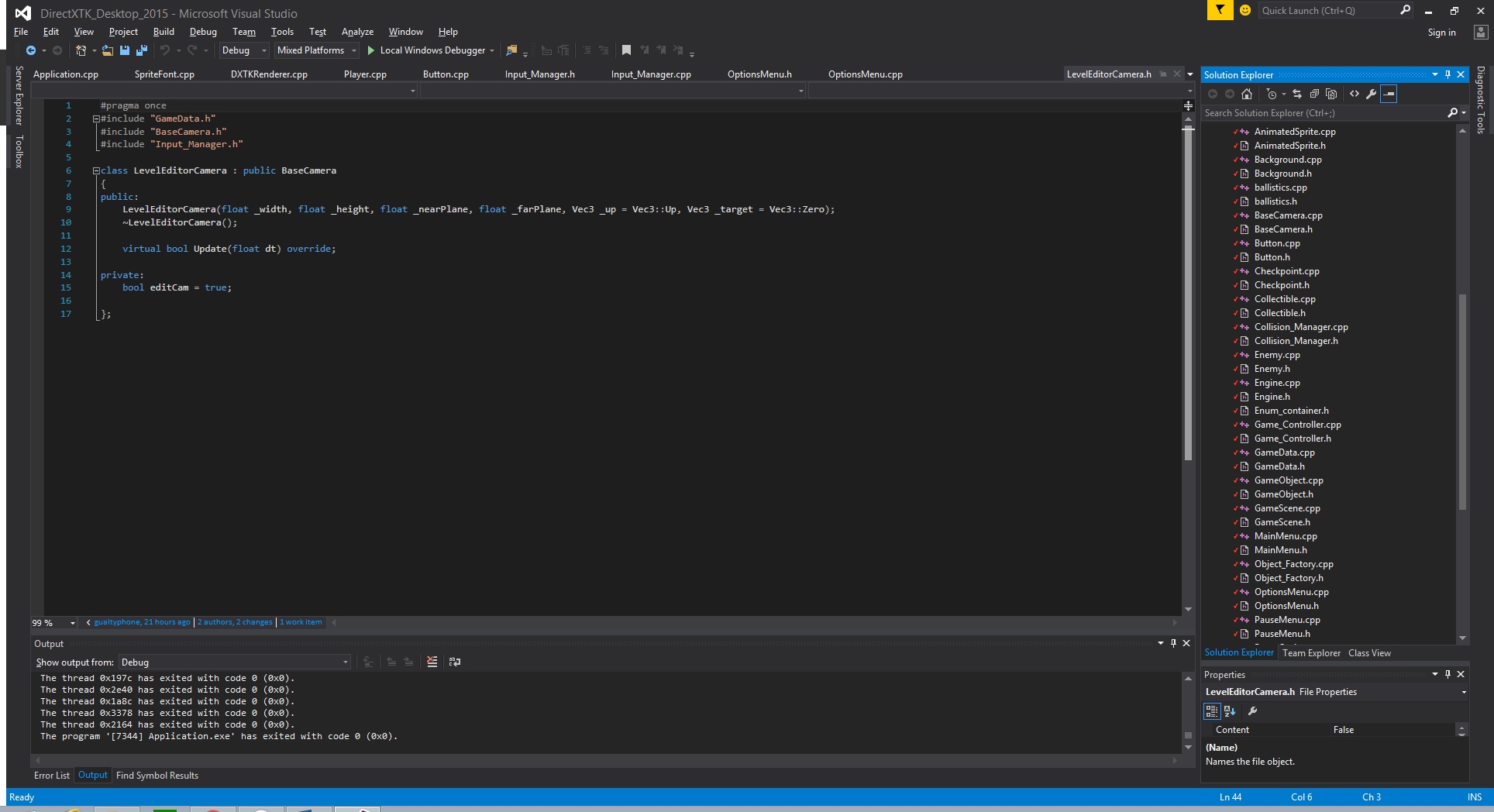
These classes are based off of the main DirectX codebase and allowed us to create the application using either DirectX or Phyre engine code. This originally was defined during the applications initialization. Although, this also became redundant after we decide not to implement Phyre engine functionality.

The files contained within the DXTK\_Wrapper project are used for many other classes throughout the engine. As a result the other projects in the engine are dependent on this one.

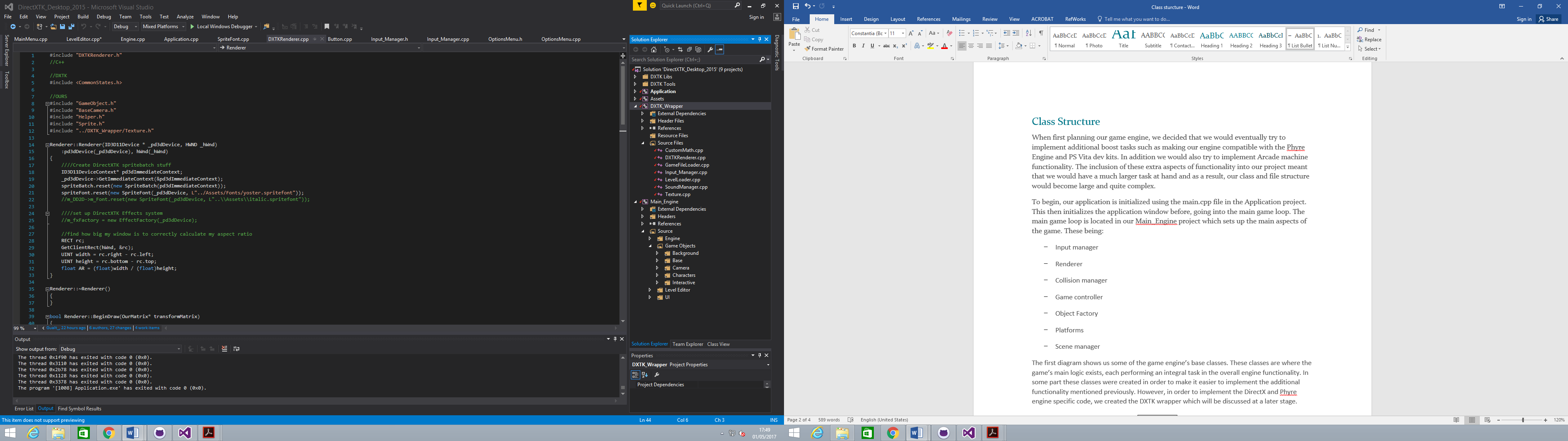
## Pixel Perfect collisions

## Animations

# File Structure

The file structure for our game has evolved over time. As discussed previously, during the early stages of development, we were aiming to implement Phyre Engine functionality. As a result we set about creating four projects:

* PhreEngine
* DXTK\_Wrapper
* Main\_Engine

This inclusion of the Phyre engine code also meant that we had to delete most of the provided codebase as we could not rely solely on DXTK oriented code as this would prove incompatible with that of the Phyre Engine. We then set about moving all DirectX code and custom compatibility items into the DXTK\_Wrapper project. This includes our own Custom Math library.

As this was our main concern during the course of the project development. We did not focus on sorting files into specific folders until a late stage in the project. This meant that for most of the project lifetime our file structure was virtually non-existent. This is shown by the image on the right.

This problem however was rectified near to the final hand in date. Using a more suitable and organised file structure makes the system easier to use, thus, making files easier to locate

# **Loading Files**

Within our game engine, we have a number of managers that handle the loading of different file types.

### Sound manager

#### Audio Engine Initialization

Our sound manager is responsible for loading and managing all of the sound items throughout the entire project.

The sound manager in the DXTK\_Wrapper project and operates in a way similar to a library. This allows the manager class to be called from anywhere in the project, thus, allowing sound files to be applied to all aspects of the engine and game.

To start the constructor of the sound manger consists of the following code:

SoundManager::SoundManager()

{

// This is only needed in Windows desktop apps

CoInitializeEx(nullptr, COINIT\_MULTITHREADED);

DirectX::AUDIO\_ENGINE\_FLAGS eflags = DirectX::AudioEngine\_Default;

#ifdef \_DEBUG

eflags = eflags | DirectX::AudioEngine\_Debug;

#endif

audioEngine = new DirectX::AudioEngine(eflags);

}

This is code shows the initialization of the DirectX audio engine. The audio engine flag created within the #ifdef statement gives the audio engine extra functionality when the application is run in debug mode.

#### Loading Sound Files

In order to allow the engine to determine where our sound files are stored we move to our playSound function.

void SoundManager::playSound(std::string name, bool BG, bool loop)

{

std::string filename = "..\\Assets\\Sounds\\";

filename.append(name);

if (sounds.find(filename) == sounds.end())

{

sounds[filename] = std::make\_unique<DirectX::SoundEffect>(audioEngine, Helper::charToWChar(filename.c\_str()));

}

if (!BG)

{

playingSounds.push\_back(sounds[filename]->CreateInstance());

playingSounds[playingSounds.size()-1]->SetVolume(SFX\_Volume);

playingSounds[playingSounds.size() - 1]->Play(loop);

}

else

{

std::unique\_ptr<DirectX::SoundEffect> soundEffect;

soundEffect = std::make\_unique<DirectX::SoundEffect>(audioEngine, Helper::charToWChar(filename.c\_str()));

auto effect = soundEffect->CreateInstance();

effect->Play(loop);

}

}

This is where most of the logic for the sound manager happens in terms of loading actual sound files.

All of our game sound files are located in the sounds assets folder, as such, we can see the path defined in the first line of this function. This does however mean that sound files stored in other locations cannot be accessed by the engine.

The then uses the passed in sound file name to create a unique\_ptr based sound item within our custom map (sounds). The sound file is then added to a vector array before an instance is created and played by the engine.

#### Volume

In order to set the volume levels of different sounds in our engine we have one sound variable, SFX\_Volume. However, the DirectX audio engine also has a function which allows you to adjust the master volume level, this set all game sounds automatically to the same level. We found that this would not meet all of our needs and so, our SFX\_Volume variable allows us to directly modify the volume level of individual sound effects, thus giving us and the user complete control over the sounds they wish to implement.

### Level Loader

Our engine also uses a Level Loader system which takes pre-made game files and loads them and all of their items into the engine allowing them to be played.

With the nature of the game Jet Set Willy and its multiple room based gameplay, our game consists of a number of levels and so, we decided that a level file management system would the best option to apply the desired functionality into our engine.

struct Level

{

std::string name = "NULL";

std::string path;

Vec2\* playerStartingPosition;

Vec2\* backgroundStartingPos;

std::vector<GameObject\*> go\_list;

GameFile\* gameFile;

int findCollectibles();

};

In the constructor we firstly define and initialize an instance of our “Level” struct (above) This struct sets up our base variable contained within each level file, allowing the most important aspects to be loaded first.

Level\* tmpLevel = new Level();

std::fstream fileStream;

fileStream.open(LevelPath);

if (!fileStream.is\_open())

{

tmpLevel->playerStartingPosition = new Vec2(0.0f, 0.0f);

tmpLevel->backgroundStartingPos = new Vec2(0.0f, 0.0f);

return tmpLevel;

}

tmpLevel->name = getStringFromFile(fileStream);

tmpLevel->playerStartingPosition = getVectorFromFile(fileStream);

tmpLevel->backgroundStartingPos = getVectorFromFile(fileStream);

int ObjNumber = getIntFromFile(fileStream);

Above we can see how the level loader is able to load game files stored within the relevant directory. Using the standard library fStream files are loaded from the applied file paths. These paths are set in the Game File creator which will be discussed in more detail in a later section.

The level loader then reads in the information stored in the file and places it into temporary variables. The one of the most important of these variables is the items type. After the level loader determines the name of an object, it then uses this information to create the desired game object. Below we can see an example of how a game object is stored in the level file.

0:  
Type: (Collectible)  
Name: (Collectible)  
Position: (-972.582 550.77)  
Size: (100 100)  
Rotation: (0)  
(END)

The game file also provides the game objects’ name, size, rotation and finally, its position in the world. Using all of this information the level loader then creates the level.

There are a few game objects which provide the constructor with additional information. The example of this is the, platform object which has a platform type variable, enabling the loader to correctly determine and apply the correct platform behaviors.

All levels are then loaded into an array, ready to be accessed, when the game scene has been created. This functionality is handled by the addLevel function as shown below

void GameFile::addLevel(Level lvl)

{

levels.push\_back(lvl);

collectibles += lvl.findCollectibles();

}

The createLevel function is the responsible for passing all items contained within the level array into our engine’s renderer, thus, rendering the game objects to the screen. This function also shows our “go\_list” vector array. This is located within the GameData class and is responsible for containing all game objects loaded from a scene file.

### Game file creation

#### .lvl

The .lvl file is a text based file, discussed in the Level Loader section of this document. The lvl files are stored within the larger .Game file allowing for easy access. This means that the user is able to neatly store multiple .lvl files. The .lvl files is where the main data for the levels is stored. Items such as platforms ad player position are all defined here.

#### .game

Our custom .game files act as a library of .lvl file paths. This allows the user to add and remove levels as they wish. Giving them complete customisation over their game design which is an invaluable addition to the engine.

.game files and have each with a base of customized levels. In theory this means that the user could play a different game every time should they wish to do so. Below is an example of the contents of a .game file.

1:  
Name:(Intro)  
Path:(.\GameFiles\level.lvl)

The .game file is a text based file and has a number of variables which are processed and loaded by the Game File Loader (see next section).

### Game file Loader

#### Initialisation

As mention before we have a game file system, this is in two stages, the Game File Loader and the Game File Creator. In this section we will discuss the Game File Loader.

The Game File Loader takes in a game file and then sends the contained level file paths to the level loader. The main functionality is shown below

int numberOfLevels = LevelLoader::getIntFromFile(fileStream);

for (int i = 0; i < numberOfLevels; i++)

{

std::string index;

getline(fileStream, index);

std::string name = LevelLoader::getStringFromFile(fileStream);

std::string path = LevelLoader::getStringFromFile(fileStream);

Level\* lvl = LevelLoader::loadLevel(path);

//lvl->name = name;

lvl->path = path;

lvl->gameFile = game;

game->addLevel(\*lvl);

delete lvl;

}

### Saving Files

We have also implemented a save level feature into our engine. This allows the user to save any levels they create in the Level Editor into a .lvl file.

To do this the current array of game objects (stored in go\_list). Is processed and separated into its respective elements.

To accomplish this task, the saveGameFile function also makes use of two other functions located in the Level Loader class. These secondary functions are called SaveStringToFile and SaveIntToFile. These functions are the ones responsible for convertivng the data stored in the go\_list into the correct text format before saving it as a .lvl file.

void GameFileLoader::saveGameFile(GameFile \* game, std::string LevelPath)

{

std::fstream fileStream;

fileStream.open(LevelPath, fstream::out);

if (!fileStream.is\_open())

{

return;

}

LevelLoader::saveIntToFile(fileStream, "Lives:", game->lives);

LevelLoader::saveIntToFile(fileStream, "Collectibles:", game->collectibles);

LevelLoader::saveIntToFile(fileStream, "Levels:", game->levels.size());

for (unsigned int i = 1; i <= game->levels.size(); i++)

{

Level\* lvl = LevelLoader::loadLevel(game->levels[i - 1].path);

fileStream << i << ":\n";

LevelLoader::saveStringToFile(fileStream, "Name:", lvl->name);

LevelLoader::saveStringToFile(fileStream, "Path:", game->levels[i - 1].path);

//LevelLoader::saveLevel(&game->levels[i - 1], game->levels[i - 1].path);

}

}