C# 2d SDL2 Game Engine Documentation

Sample “Galaxius 2” Included/Available

C# 2d SDL2 Game Engine and Source Code Documentation

C# 2d SDL2 Game Engine Documentation

Introduction (taken from Conflict 3048 manual)

Core Engine Code and Classes

Class Config{}

Class Game{}

Class StreamResources{}

Class Resources{}

Class Fonts{}

Class Audios{}

Class Images{}

Class RenderList{}

Class Network{}

Class Functions{}

Class Program{}

Class Gui{}

Class LocalStorage{}

Class Translation{}

Extension Classes

Class Map()

Class Emitter()

Class Particle()

Game Data Files

Reading In of Data Files

Gui File

Translation File

Coding Conventions and Preferences

Conflict 3048 Source Code Licence

Introduction (taken from Conflict 3048 manual)

Update 12apr2022 - Documentation for Conflict 3048 source code is in the relevant pdf file - this document only contains the documentation for the game engine itself, but does come with an example game ‘Galaxius 2’ in zip format attached with the game engine.

Conflict 3048 is based on the same engine that I used to develop Knights of Grumthorr and its various incarnations. The game engine however has roots that go back farther than that. Originally I developed a number of 2d games for browser using pure javascript. These used a 2d rendering method that is still used in Conflict 3048 and my other Windows games.

So the origin of this engine goes back to around 2014 for myself. From 2018 onwards I began working on more browser games which meant the rendering engine grew more complex. Eventually in 2020 I taught myself C# and ported the engine to use both SDL2 and C#. Later I was able to port it to Java to use with Android.

At the core heart of the 2d graphical engine is a system that works very simply by building up an array each rendering frame of all the drawing commands and then iterating through that array before resetting the index to zero before the next frame.

In this document I hope to describe to you the classes in the engine and the game, as well as the supporting data files that make up the game. This should give you sufficient information to make your own changes and additions if and when you choose to.

For the purposes of this documentation I have split it into the following main areas:

1. Core Engine Code and Classes
2. Game Data Files (text, graphics and audio)

As a loose description, the core engine code handles things like window initialisation, drawing routines, audio routines and the main game loop. The game code and classes handle elements such as the gameplay mechanics of the soldiers in the game, bullets, magical effects and such. The game data area describes the scripts and data files that bring the game code to life. These control the specific details of the gui, the game units, graphical elements and such.

I also include as an appendix the licence I provide to you for using this code I have written.

The simple premise of the licence is that you can do what you want with this code as long as you treat it, and myself, with the respect you might hope another would show to your own work. “The Golden Rule” but for code I suppose.

From
Matt Lloyd.

Core Engine Code and Classes

The Core Engine of Conflict 3048 is a 2d graphical engine with audio and gui support. At its heart is a simple list renderer that takes all the drawing commands issued during the current frame and puts them into an ordered array before iterating through that array to draw each element in turn.

The other features of the engine include classes for managing graphical memory at run time, translating language text, audio and music, initialisation and quitting, as well as resource loading and text file parsing.

Much of both the game engine and game code is controlled by a Config class containing many static variables that are set on initialisation and only sometimes altered afterwards.

Due to some poor decisions at the beginning of the project there are some cross over between game engine and game code that shouldn't really occur - but these are rare.

The classes of the core engine are described as follows.

Class Config{}

The Config class contains a large number of static variables accessible from anywhere in the program. Most of these are set during either the method checkconfig() and applyprefs() which is called before window initialisation. The values in these methods are acquired from relevant config.txt files loaded during initialisation.

Although boolean values are traditionally used for true/false values often I use integers for the same or similar effect, with 0 being equivalent to false and non zero as true. This is because for some variables to expand their definitions is useful to allow other values than a simple binary choice allows.

Many of the values in this class never change and are effectively constant. Also, many of them existed as switches during development to control new features as they were added to the engine, thus changing them from true to false will likely cause those features to disappear from the game in some cases.

For the most part - the values in this class should not be changed except by reading their values in from the config text files.

Class Game{}

The game class holds the various arrays that store references to the images, audio, gui and other elements loaded in the game. For example - every time an image is loaded with the various LoadImage routines it is added to an array of images that can be referenced by an index number at a later point. The same holds true for sounds and other elements.

These arrays grow in size as more elements are added or loaded.

The game class also holds the various IntPtr references that handle the Window, Renderer and other graphical links to the SDL library.

The game class is the most likely of the classes to have cross over between game core engine code and game play code (naughty, naughty Matt!). There are many variables and arrays that refer to specific game features. These should be obvious by their variable names.

This class also handles the mouse coordinates and keyboard input.

The Game class handles quitting the game with a method called quit() that unloads all the previously loaded resources.

Class StreamResources{}

The StreamResources class manages the memory used by all the 2d textures for the game. It is a simple process where each frame it checks the last drawn frame value of each image in the game, and if it is further into the past than a specific tolerance value it frees the image from memory, only reloading it when it is needed again.

During gameplay mode it deliberately avoids unloading graphics to prevent load/unload happening when fast, responsive gameplay is required.

However during parts of the game where the only graphical features are gui related, then it will free and reload graphics as needed.

The purpose of this is to reduce load times initially and to reduce memory footprint by not loading every graphic initially.

By altering the Config.STREAMLOAD variable you can activate or deactivate this feature.

Class Resources{}

The Resources class handles loading all the various data files and resources for the game. The game engine calls a function at the beginning of the game - before the main loop is entered - where all resources are loaded, or attempted to be loaded.

Each resource is loaded from a defining text file. So the gui for example contains a text file that holds all the details of the gui elements in the game. This text file is read, parsed and all relevant images are loaded - or prepared to be loaded.

The Resources class also contains text parsing functions.

Text in the data files is stored in the manner of name=value.
So for example a gui element might have a set of parameters that are listed in their data file as follows:

```
image=media/button.png
title=Play
tooltip=Play the Game
scale=1.5
```

The various readstring(), readint(), readfloat(), readbool() methods will parse those lines of data in the text file into their relevant data types.

Each major element of the game such as the gui, or the game units, or the missions are loaded from their text files and then compared line by line with a switch statement to determine how to handle each line of data in the file.

Values that do not exist in the switch statement are ignored by the parser - meaning you can put as many comments as you like in the data files and they will be ignored.

The various load routines for both data files and Images, Audio and Fonts may be found in the Resources class.

Class Fonts{}

This class is very basic. It merely holds a reference to the font pointer as well as a few numeric values relating to the scaling of the font. Languages other than English often need their own font and these are loaded within the relevant config file for that language.

Class Audios{}

The Audio class handles playing audio including both short clips and music files. Due to the limitations of SDL2 currently, or at least the version I am using, only wav files work consistently. This means your audio folder will be quite large as compressed files such as mp3 or ogg though supposedly supported are not in fact.

Audio files are loaded into an array and referenced by a numeric identifier (integer).

Class Images{}

The Image class handles storing all the 2d images in the game. Both animated and static images can be loaded. Animated images are loaded as a series of cells laid out top to bottom, left to right. A number of key properties are set during load of an image that determine how the image is animated.

Loading of images requires the width, height and number of frames across and down to be specified at load time.

Png and jpg files can be loaded.

If an image is successfully loaded it will return an integer value greater than -1.

Class RenderList{}

RenderList is the class where the 2d images are drawn. It consists of two main types of method. There are the many and varied drawing methods as well as the single render method.

In the drawing methods - you call them by supplying a reference to the image you wish to draw, x/y coordinates and other values. Or if you are drawing text or primitives such as rectangles then you simply supply the relevant values. The drawing methods then put that information into an ordered array. This ordered array is then read through and drawn one by one by the render method.

The RenderList class can handle realtime scaling of the graphical window at runtime.

You might ask what is the benefit of placing all draw commands into a list and then drawing at once, rather than just drawing at the point the command is issued to draw - that’s a good question and the answer is not easy for me to explain, but - there is a benefit to doing things this way and one of those is ease of porting to other programming languages (such as Java or Javascript) in that only the render() method needs changing in most cases.

Class Network{}

The Network class is a simple class for performing https requests. This is useful if you wish to include for example, a high score table or similar that is stored externally on a website.

Currently receiving data from the Network is disabled but can be reenabled with a Config flag.

The usage of this class should be fairly self explanatory from the methods in the code.

Class Functions{}

Functions is a class containing a group of helper functions I originally wrote for the Javascript browser engine. Many of these are not used, but are fairly simple to understand.

However the main function that is able to be used is a simple random number generator.
This function is used throughout the game rather than C#'s own random number generator as it produces predictable results that I am used to.

The isometric functions are quite useful if you are making an isometric game, however I do not use them, or many of the other functions inside this class.

This class mainly exists for historical reasons.

Class Program{}

The Program class contains a group of functions handling the setting up of the game, Main(), checkconfig(), game(), update() and render().

If you are making changes you will most likely make them to the checkconfig() and applyprefs() if you are altering or adding new config values to be loaded.

The actual Main() method contains a loop that tells the game to run, and to reload and run everything again after quitting unless the 'restart' flag is set to false. This allows for things such as changing game window/full screen modes or entering and exiting the Editor after saving a map without quitting the game.

The render() method ultimately just points to the RenderList render method() after clearing the screen buffer and then copying the contents of the screen buffer to the screen.

Inside the update() method is where all the action takes place that calls the various methods relating to game mechanics such as moving units etc.

Class Gui{}

The Gui Class handles updating and drawing the gui in game. As the names might suggest, update() is where the game handles updating interactions with the gui, while draw() handles drawing the gui elements.

Each gui element has an x,y,w,h value that determines its position. Gui Elements also have a scale value. Most gui elements have a title and tooltip.

Each gui element has a 'type' that determines its behaviour. ButtonTypes, ImageTypes and other more specific types are used. Basically if the element can be interacted with it is a button, and if it is static then it is an imagetype.

The index of a gui element determines the frame or cell of an animated image to draw from.

The triggerlink, link, action and command properties determine aspects of the gui element's behaviour in game.

Link is a textual field that is used to define certain types of behaviour.
Command is a specific command that the gui element passes to the game.
Action is specific to button type gui elements and is a number that performs some action.
Triggerlink is a link between the gui element and Game Scripting triggers.

All gui elements have two values, gamestate and nextgamestate. These values may be considered to refer to the Window or Screen or Menu that the gui element belongs to. All elements that share the same gamestate value will appear on the same window. The initial gamestate that the game begins on is gamestate '0'. Internally these gamestates in the gui file have a value of 10,000 added to them, but this is invisible to the user. If a button has a different gamestate in the value 'nextgamestate' then pressing the button will transition the menu to this new gamestate or menu screen. In the update and draw methods for the Gui only those elements in the current gamestate are updated or drawn.

Gui Elements can be hidden from view and interaction by setting the value of the 'hidden' field to 1 or greater.

Gui Elements can be linked to triggers such as chatter triggers in the game by setting the triggerlink value. The triggerlink value can also be used to group panels that toggleon/toggleoff together. Example - the options menu in battle uses a panel that hides/becomes visible with a touch of a button - these are grouped using the triggerlink value to group all the elements that form part of this panel.

Class LocalStorage{}

LocalStorage handles writing all the config changes at runtime to disk. It also handles any other preferences that are set at runtime. It reads/writes data from an array of name=value pairs to a text file in the user's Local App Data folder.

You will find that the game records whether loading of resources is successful with this function.

Also features such as screen mode, windowed or full screen, audio on/off, are set with this function.

Deleting the LocalStorage file or folder it resides in will reset the game state to its original downloaded state.

Class Translation{}

The Translation class handles translating text into various languages. A translation file is able to be loaded which holds name,value pairs for phrases used in the game. This is read at load time and applied to gui elements and such.

Any text that is to be displayed on screen should be put through the static method Translation.TranslateWord() to ensure the correct information is displayed to the user.

Translation of gui elements takes place at load time.

Translation of other text often takes place at run time. A binary search method is used to speed up searching through the list of translation entries.

Extension Classes

Extension classes are classes that have been added to the core engine for use with 2d games. They are not essential to the functioning of the engine but are general enough that they can be used with a number of games.

Class Map()

This is a helper class for the Emitter and Particle code. It contains a reference to CornerX and CornerY which represent the upper left corner of the screen position in your 2d world. You may or may not use this depending on the game you create. Add to this as desired.

Added 13apr2022

Class Emitter()

The Emitter class is used for creating particle emitters in the game and also for loading the definitions of particles from a text file.

The usage is as follows - first call Emitter.Reset(game) somewhere before the first time an emitter is created, then when a particle emitter is desired call Emitter.TYPEOFFSET = 0 to 8 and Emitter.CreateEmitter().

In your main loop you will need an Emitter.UpdateAll(game) command to make sure they perform during the game loop.

The particle engine can be activated by setting the relevant Config.PARTICLEENGINEON value to true.

Added 13apr2022

Class Particle()

The Particle class is used for managing the particle updates and drawing in the game. You will need to call Particle.UpdateAll(game) and Particle.DrawAll(game) in your main loop.

Particles are defined in a text file typically held in media/txt/ptypes.txt which contains definitions for each particle type and emitter used in the game.

This can be reloaded at runtime if you desire as well.

Added 13apr2022

Game Data Files

The Core Data files for the game are the Config, Gui, Campaign, Unit Templates and Tech files.

The typical structure of the game data is arranged into a media folder with subfolders for each of the config, units, gui and graphical and audio elements.

The layout within each file is linear. In other words, each unit is defined one after the other, with the marker for a new unit being referenced by a keyword ‘newtemplate=’ for example.

The same applies for gui elements ‘newgui=’, as well as campaign missions, ‘mission=’ and so forth.

Rather than encapsulating elements with brackets of a ‘start’ and ‘end’ marker only a ‘start’ marker is used and all name=value pairs that come after the ‘start’ marker are assumed to belong to the element just created.

Data files can contain comments, which you do not need to mark especially, though it is advisable for readability.

Missing or incorrectly referenced text files can cause problems / crashes at load time.

Missing or incorrectly referenced graphic and audio files will not cause problems other than the graphic or audio not being shown/played.

Reading In of Data Files

The first data file that is always read in is ‘media/txt/init.txt’. This file contains a single line referencing the relevant config.txt file to read. The purpose of this file with a single line is to allow easy modding of the game by allowing users to create all their data files and then point to them with a single change to the line in init.txt by referencing their new config.txt file.

In the config.txt file, which is read in second, are the title of the game, the internal game title which is used to determine the name of the Local App Data folder, other features such as screen resolution and various parameters that determine the flow of the game engine - such as frame rate. The locations of other files are also referenced inside the config.txt file. Files such as the units files, campaign files and gui files.

The various languages supported by the game each get their own config.txt file - a file that is referred to simply by the same initial config.txt name with the language appended to the end of the filename. So for example “config.txtspanish” is the config.txt file that contains references to the Spanish language data files.

Within the units files, and any ‘extraunits’ files that we reference are all the playable game units and entities. These have both a name that is referenced internally by the engine eg ‘Marine’ and a displayname that is shown to users when describing the unit eg “Space Marine”. This is so that when different languages are used we can retain the internal name, but show a different name in the appropriate language to the user.

There are two campaign files - a core campaign and a custom campaign file that are referenced inside the config.txt file. The core campaign file is where the main game campaign resides. The custom campaign file is copied to the Local App Data folder and inside this new location is where custom campaigns and missions can be placed.

The gui file contains all the gui elements. Each gui element inside the file belongs to a particular screen also known as ‘gamestate’.

Gui File

The gui files contain all the gui elements in the game. Gui Elements are positioned using their x,y,w,h properties as well as scale. Each Gui Element has a gamestate and a nextgamestate value that determines the menu window/screen they belong to. A title and tooltip is often present. An image is also referenced along with its base width, height and number of frames wide and high. The index determines which frame to display.

Action, Link, Triggerlink and Command are special values that determine how an element functions. These are mostly tied back to parts of the code in the Gui Class. However the Triggerlink is linked to triggers in the Triggers for a map.

There are two ways of creating foreign language versions of the gui. One is by creating individual foreign language gui files with the descriptions and textual information in their languages. Another is by creating a base gui file and then referring to the titles and descriptions inside the single translation file that is loaded.

Translation File

Each language may have a translation file that is loaded at startup. This contains a list of name=value pairs that determine words and phrases that get replaced by the engine on display. Where possible these are replaced at load time, not at run time.

The Translation file is also useful for renaming unit abilities, for example ‘woodsman’ in Knights of Grumthorr was replaced with ‘Uses Cover’ in Conflict 3048. ‘Frenzy’ was replaced with ‘+ Damage’. This is an easy way of altering the description of elements.

Coding Conventions and Preferences

My preference for coding convention throughout this engine and game code is to use lower case for everything where possible, except in the case of some constants which are CAPITALISED. However I have not always been consistent with this convention.

My preference for brackets is to begin an open bracket on a new line with indentation, and a close bracket on a single line with indentation.

My preference is to use integers over boolean values for toggle states as for expansion purposes it is easier to redefine integers to have multiple meanings beyond merely true/false.

Floating point numbers are avoided in preference for integers wherever possible.

Double precision values are rarely used, if at all.

Game Objects typically have a reset(), update() and draw() method for most objects.

String objects are avoided for anything related to game logic. Typically string objects are associated with reduced performance and so where possible game mechanic and gameplay features are linked to integers as much as possible over strings.

Comments are made whenever game engine code is changed of the format:

//ddmmmyy

Or
```
//start—-------//ddmmmyy

//end—------//ddmmmyy
```

This is to reference what/when/who/where changes were made to the game code.

Typically a reference to the change is made at the top of the engine comment block indicating what was changed and why.

Note - this has sometimes been simply compressed to a single line of text covering a range of dates where a lot of changes were made for a single project task.

Conflict 3048 Source Code Licence