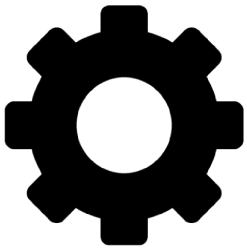
Multi Platform Game Engine

Memòria del Treball de Final de Carrera

Enginyeria Tècnica en Informàtica de Sistemes



RAG ENGINE

Autor: **Iván Sánchez Luque**
Tutor: **Josep Soler Masó**
Gener 2016

Abstract

We have seen a huge success in mobile markets in the latest years. This success have been followed by an increasing demand of applications, which in turn have lead to more and more companies creating apps. Although there are big opportunities, developers compete fiercely with each other. Developers are forced to create great applications that deliver rich multimedia contents and experiences, while targeting multiple platforms at the same time. Instead of create their own technology, it has been a common trend to reuse what are known as *game engines*.

This study is focused on the design and implementation of one game engine, made from scratch using some low-level libraries. The idea is to provide a thin framework on top of OpenGL / C++, to facilitate the development of multi-platform applications, while leaving the doors open for enhancements and access to low-level libraries, providing an excellent performance in terms of low CPU and GPU usage, memory footprint, and disk size, making possible to create applications that are performant even in the lower end mobile devices.

The game engine described integrates well with some 3rd party authoring tools, such as Flash CS, Spine, Tiled, Texture Packer, and others. Some other tools are provided as well, as a Particle Editor, and some scripts to improve the production pipeline. As a final step, in order to demonstrate how the engine works, a simple interactive application has been created. The app is no other than a clone of the Arkanoid game, in which the player controls a ball with a paddle and must break a set of bricks in each screen.

Table of contents

Rag Engine

Introduction

In the latest years we have seen an increasing demand in the field of digital entertainment, specially in mobile platforms, with free titles reaching more than a billion downloads (1) and other titles making easily more than one billion dollars per year (2). With the growing interest in the sector, new developer companies appear and the demand for new both commercial and open source game engines also raises.

The exposed project below consist in the creation of a software library along with all the required documentation and tools to help in the task of develop games and/or interactive applications.

Other Game Engines.

There exists a few engines well established (3) with a quite cumbersome number of features and complexity.

Unity 3D stands out of the competition, being one of the first in offer a commercial solution with a completely free license for small indie developers, featuring 3D and 2D with support for multiple languages including C# and Java Script on top of MonoDevelop (4) for multi-platform one-click deploy.

Another contender which is increasing its popularity is *Cocos2D-X* (5), the multi-platform C++ open source solution featuring most of the common requirements for game development with a lot of successful stories (6).

Unreal Engine, *Marmalade*, *Game Maker Studio*, are other popular choices among a big number of engines, and the list keeps growing.

Why then yet another game engine?

Among all this competition, a question arises; why do we need yet another game engine?

Control: Since we are working from the ground up using some low level basic frameworks, we always have control about *what* we do and exactly *how* we do and *when*. That's often not possible with bigger commercial engines, which offer built-in standard solutions that not always fit with your requirements.

Small footprint: By not using huge abstractions like the other super-flexible generic-purpose engines, we can focus more directly on our task and create experiences with the minimum footprint in the devices. Low usage of memory, CPU and disk size can make a difference with the competition.

Learn: Last but not least, the creation of such a complex task is an excellent academic exercise, even if you're not up to par with the competition, which counts with far more

time and resources. Even knowing that we're not to create the last cutting-edge game engine, the experience of architect, develop and document this project can help us learn a lot.

Game Engine Features

Below we have a list with the main features of the library. An explanation of these features follows, where we go deeper from a technical point of view analyzing them all.

- Multiplatform framework (iOS, Android, Windows).
- Small footprint in memory, disk and CPU.
- Multithreaded resource management.
- Flash CS as editor.
- Skeletal animation (Spine).
- Texture Batch and Texture Atlases.
- Multiple graphic input support: png, jpg, pvr (disabled support for bmp, tga, psd, gif, hdr and pic).
- Bitmap and TTF fonts, unicode support.
- File input/output.
- Touch input.
- Simple sound and music support.
- Dynamic content support.
- Localization.
- Particle editor.
- 3D md5 models with support for skeletal animation with GPU skinning.
- Easily extended.

The engine consists in a serie of C++ headers and source files, organized through some namespaces to keep the structure organized. The engine code can be embedded directly in the application, thus compiling it, or can be linked against as a regular library.

For convenience Windows is used to create the project, although technically is possible to do it from x-code and OS X. An important thing to notice is that all code is cross-platform C++, and some native code is used to glue it together with the target platform. Although the majority of the code, say 90%, is written in C++. A tool (premake) is used to generate the project files for the different targets, so only one project file is required to be maintained, and the others are generated automatically.

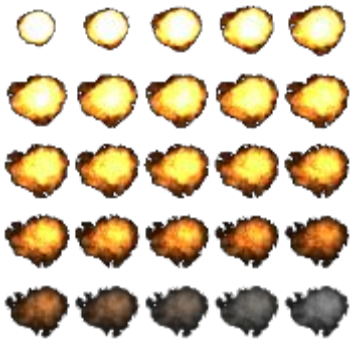
A multithreaded resource manager makes possible to load assets in background while performing other operations. The resource manager create one or various threads where big game *Resources* such as images, 3D-models or sounds are enqueued to be loaded. When the asynchronous loading operation has been done the Resource is notified, so can decide what to do.

The engine supports *Flash CS* project files as long as they are saved as xfl, which is an xml-based format. The supported Flash features, which include scene loading, node hierarchy tree, images, texts, and for all of them, translation, scale, rotation, skew, among others properties, and some basic support for the Flash timeline to create simple

animations. Additionally, to accelerate the process of loading flash files, a binary format is used (and a binarizer provided) that can boost performance loading assets.

Spine is another authoring 3rd party tool to create skeletal animations from 2D sprites, and is integrated with the library to perform efficiently animations.

The images can be loaded one by one, but in order to improve the performance and reduce draw calls, the use of spritesheets is recommended. The library would include support for this kind of images, using an external tool to create the texture atlases.



Texture Atlas example

Images can be loaded in the most common formats, namely .png and .jpg, although compressed cross-platform formats like .pvr are also supported. At the cost of some graphical quality loss, these compressed formats have a minimum impact on memory once loaded in GPU, allowing different types of compression that range from 1 *bpp* (bits per pixel) to 2 or 4 bpp, way better than the 24 or 32 bpp that are normally used with .jpg and .png respectively. Depending on the kind of graphics, the quality loss may be completely acceptable or not, so all formats combinations are available to choose whatever fits better.

Regarding texts and fonts, TTF and OTF types are supported, on top of OpenGL, generating dynamically textures with support for UTF-8 texts. The size of the textures is increased as needed depending on the size of the text and the number of different characters used. Multi-paged font textures are not going to be implemented but could be a good addition for better support of applications with lot of texts in oriental languages. Bitmap fonts are supported as an alternative for fully customizable characters. Text justification, vertical and horizontal alignment and other convenience features are supported.

At the core of the library, support for common game maths is provided through the 3rd party library *glm*, that allows to operate with vectors, matrices, quaternions, etc., in a convenient language that mimics GLSL, the language used in the graphic shaders.

File Input/Output is fully supported for all the platforms through a *File* abstraction that hides the implementation details. You can create, read or write files consistently, no matter what platform you're working with. Such is the case for screen Touch Input, allowing multi-touch and simulating touches for Windows platform.

Sounds and music support is provided with CocosDenshion, which is an open-sourced part of the Cocos 2D-X library. It allows to asynchronously instantiate sounds and music. The sound system API is really simple and would allow only basic operations.

For modern games that are regularly updated, often dynamic content is necessary, so a system to check and download new content is provided to allow actualizations without much hurdle for the players. Also language localization is a common concern, so tools

are provided to allow the application localization.

Another common feature in games are particle effects, used to generate dynamic explosions, smokes, fires, magic, or other effects with simple sprites that are rotated, scaled, translated, colored, blended together in different ways with the goal of give the illusion of the real or cartoonish effect. A particle effect editor and render system is going to be part of the library too.

Lastly, some 3D features are going to be integrated on top of the 2D engine, allowing the use of the *md5* model format introduced when *Doom 3* went open source, to load and animate 3D models with simple texture and illumination.

Game Engine Tools and Environment

In order to use the library, Visual Studio 2015 would be the IDE environment of choice, but theoretically any other modern IDE should be easily adapted. The graphic pipeline would use Flash CS as editor, along with the custom particle editor to create visual layouts that can be used later from code. Extra content data would be stored in Google Spreadsheets and exported to a json or xml format understable by the application.

Some tools to help in the assets pipeline are scripted in python so they can be run in any environment. This tools include the optimization of images, the binarization of Flash files, the creation of code to hard-link code with Flash, the search for missing localized texts and the creation of atlases of images in big textures.

Risks

As the scope for the project is pretty big, some features may fall down if it's not entirely possible to implement them with a good quality standard. As the engine is very modular and apart from the core most of the features are independent from each others, it may be possible to let some features unimplemented if there's no other option.

Methodology

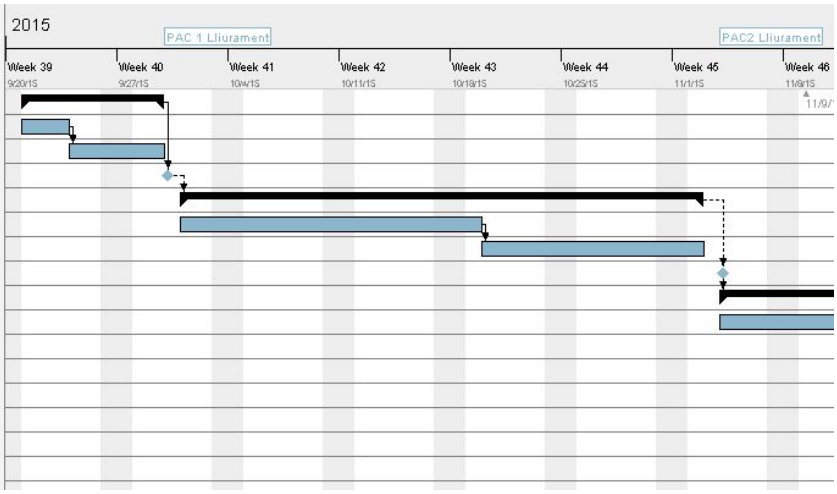
It is intended to provide all the above features explained implemented, documented, and accompanied with Unit Tests that would check for consistency and be part of the product not only for regression testing but also for documenting the code from a practical point of view.

The code would be self-documented, using a tool such as doxygen to generate the documentation from the comments found in the code. Besides, a formal explanation of every feature covering more details will be delivered.

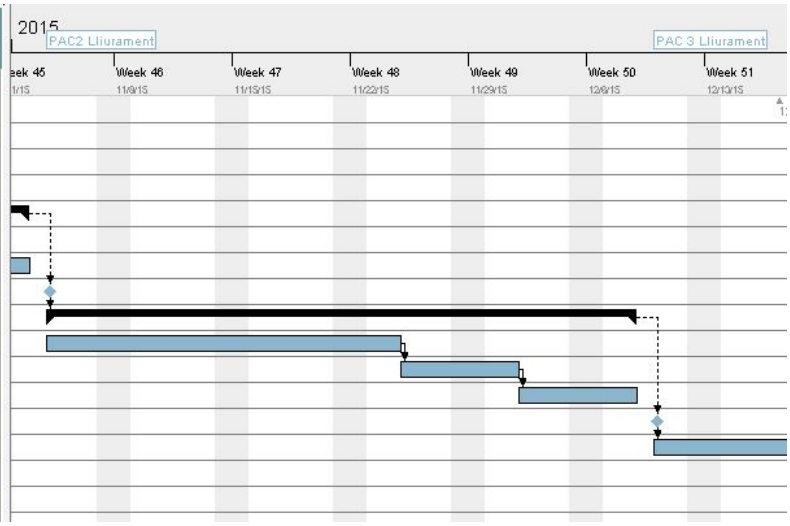
The implementation of an Arkanoid-like game will be released as well as an appendix to demonstrate how an interactive application can be build using the engine.

Gantt Diagram

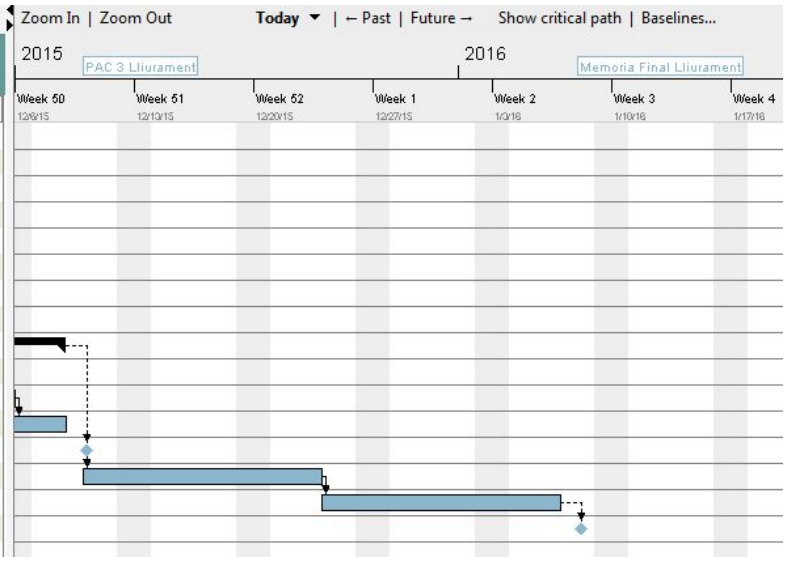
Name	Begin date	End date
Planificació	9/21/15	9/29/15
Descripció del projecte	9/21/15	9/23/15
Redactat del pla de treball	9/24/15	9/29/15
PAC 1 Lliurament	9/30/15	9/30/15
Disseny	10/1/15	11/2/15
Formulació del disseny	10/1/15	10/19/15
Redacció del document de disseny	10/20/15	11/2/15
PAC2 Lliurament	11/4/15	11/4/15
Implementació	11/4/15	12/8/15
Implementació classes	11/4/15	11/24/15
Implementació Unit Test	11/25/15	12/1/15
QA, bugfixing	12/2/15	12/8/15
PAC 3 Lliurament	12/10/15	12/10/15
Conclusions	12/10/15	12/23/15
Redacció Memoria Completa	12/24/15	1/6/16
Memoria Final Lliurament	1/8/16	1/8/16



Name	Begin date	End date
Planificació	9/21/15	9/29/15
Descripció del projecte	9/21/15	9/23/15
Redactat del pla de treball	9/24/15	9/29/15
PAC 1 Lliurament	9/30/15	9/30/15
Disseny	10/1/15	11/2/15
Formulació del disseny	10/1/15	10/19/15
Redacció del document de disseny	10/20/15	11/2/15
PAC2 Lliurament	11/4/15	11/4/15
Implementació	11/4/15	12/8/15
Implementació classes	11/4/15	11/24/15
Implementació Unit Test	11/25/15	12/1/15
QA, bugfixing	12/2/15	12/8/15
PAC 3 Lliurament	12/10/15	12/10/15
Conclusions	12/10/15	12/23/15
Redacció Memoria Completa	12/24/15	1/6/16
Memoria Final Lliurament	1/8/16	1/8/16



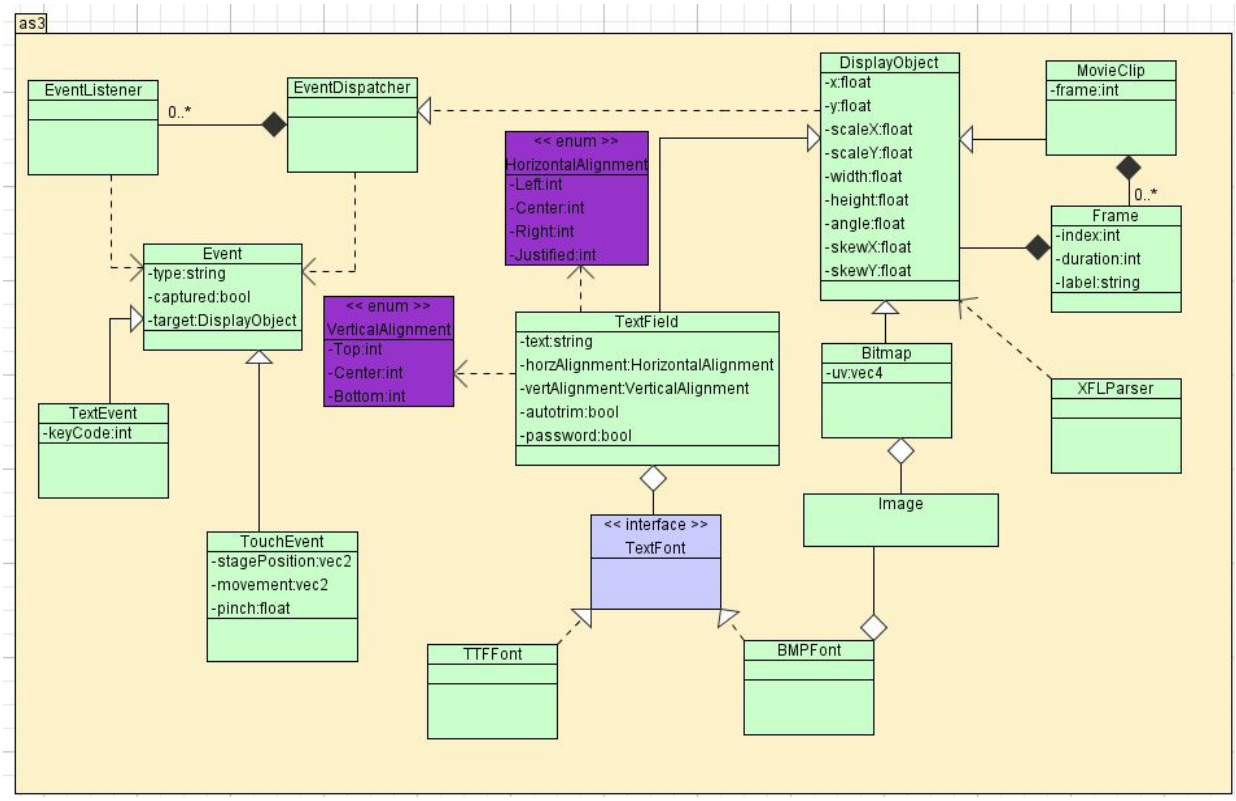
Name	Begin date	End date
Planificació	9/21/15	9/29/15
Descripció del projecte	9/21/15	9/23/15
Redactat del pla de treball	9/24/15	9/29/15
PAC 1 Lliurament	9/30/15	9/30/15
Disseny	10/1/15	11/2/15
Formulació del disseny	10/1/15	10/19/15
Redacció del document de disseny	10/20/15	11/2/15
PAC2 Lliurament	11/4/15	11/4/15
Implementació	11/4/15	12/8/15
Implementació classes	11/4/15	11/24/15
Implementació Unit Test	11/25/15	12/1/15
QA, bugfixing	12/2/15	12/8/15
PAC 3 Lliurament	12/10/15	12/10/15
Conclusions	12/10/15	12/23/15
Redacció Memoria Completa	12/24/15	1/6/16
Memoria Final Lliurament	1/8/16	1/8/16



Subsystems

High-Level classes

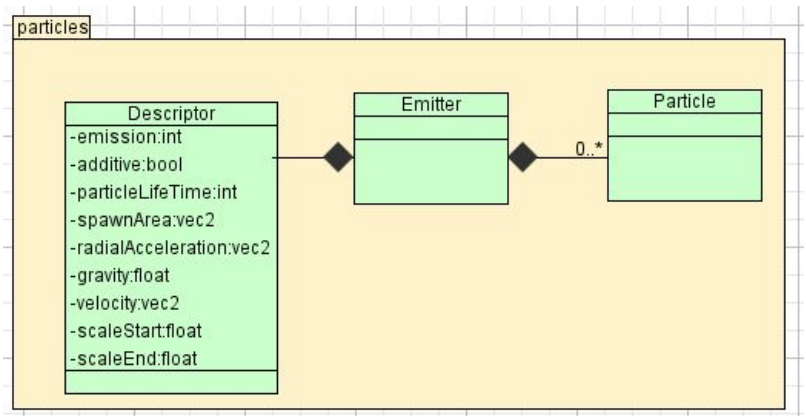
On the top level of abstraction of the library, we found some classes that mimics the *Action Script 3* language API, proprietary from Adobe. This is a design decision to maintain most of the compatibility and terminology from the Flash CS editor which uses AS3 as the scripting language. Indeed, to simplify the design and to ensure the use of a proper, practical and already proven concept, the API for the top elements of the library are almost cloned from those of Adobe. In particular, the classes DisplayObject, Bitmap, MovieClip, TextField, EventDispatcher and EventListener have a very similar API to Adobe classes.



[see full size version]

To understand how the library works, we need to define first the concept of *Display List*. The Display List is a tree graph where each node is a *DisplayObject*. The Display List contains exactly one root node. Other nodes can be child of the root node, and this “one-parent, multiple child” relation is maintained recursively. Display objects have *transforms* (like position, rotation or scale, among others) that are passed to their child's. So when you move a *parent* element all their children are moved with him and preserve their original position with respect his parent. The Display List can grow virtually infinitely in both depth or width to allow the representation of arbitrarily complex structures. Several classes inherits DisplayObject, and those are the basic objects that can be displayed on the screen.

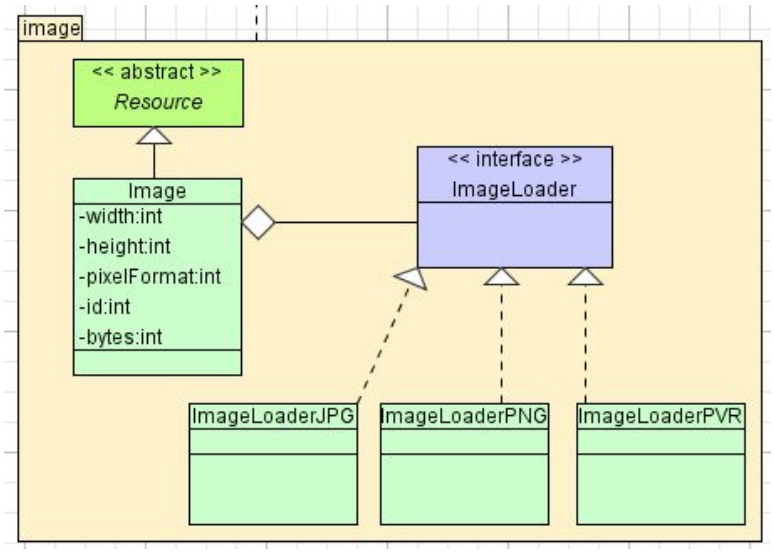
Along with the display list is provided an event system that provides a form of communication between different elements of the display list. We can think of this model as a regular implementation of the *Observer* pattern; classes that inherits *EventDispatcher* can dispatch events, and those events can be listened from *EventListeners*. The *events* are regular structures that can carry data and the user of the library can define its own customized events.
A particle system and an editor is part of the library, providing tools to create visual effects made of tens or hundreds of particles. A basic particle system consist on one emitter, with one system descriptor. When the emitter is fired, particles are spawned from the emitter position, creating an effect that can emulate things like fire, smoke, clouds, explosions, etc.



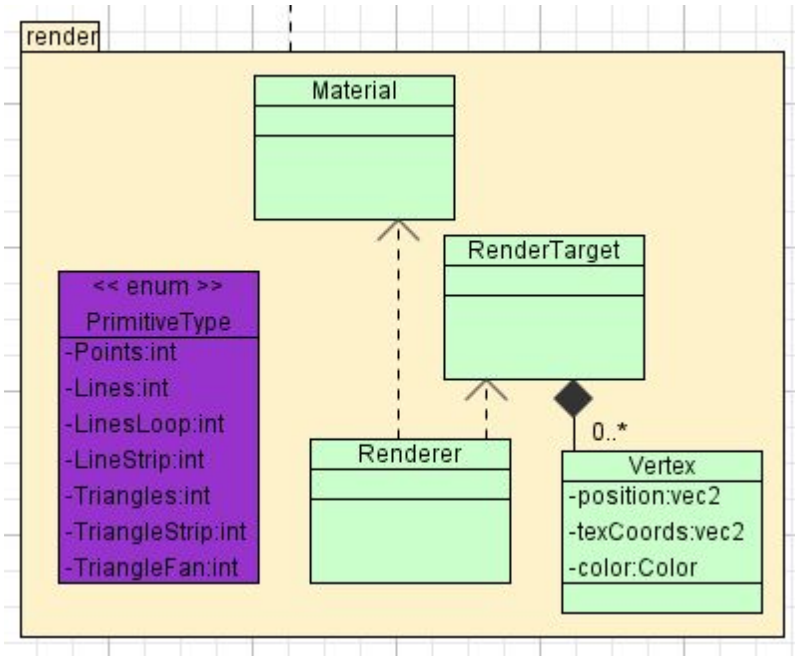
Mid-Level classes

We have separated in this layer classes that are less often required, but are still very useful to perform certain tasks.

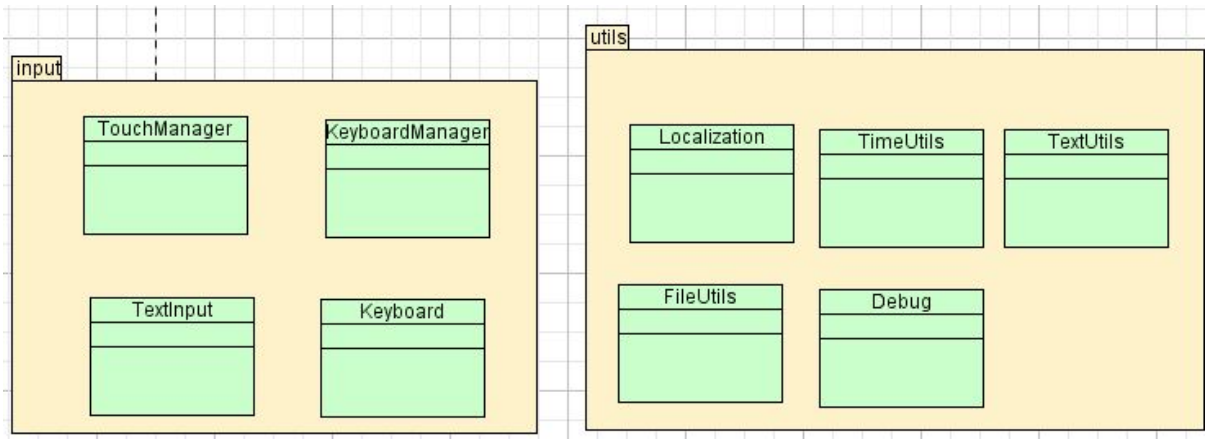
One common thing we need to deal with often in multimedia applications is the management of images. Although high-level classes use images internally, we can also load images on our own, using the Image class. You can load images from disk both synchronously or asynchronously by using the assets path. Formats supported are .png, .jpeg and .pvr.



Regarding the render package, *RenderTarget* is used for batch draw calls, as is well known that number of draw calls is a common bottleneck for graphic engines. Objects are not rendered directly, instead all draw commands are stored in a RenderTarget and are flushed to GPU only when is required (i.e., on changes of Material or when there are no more draw commands). Other than that, Renderer does little more, and allows the use of OpenGL directly without interfere too much.

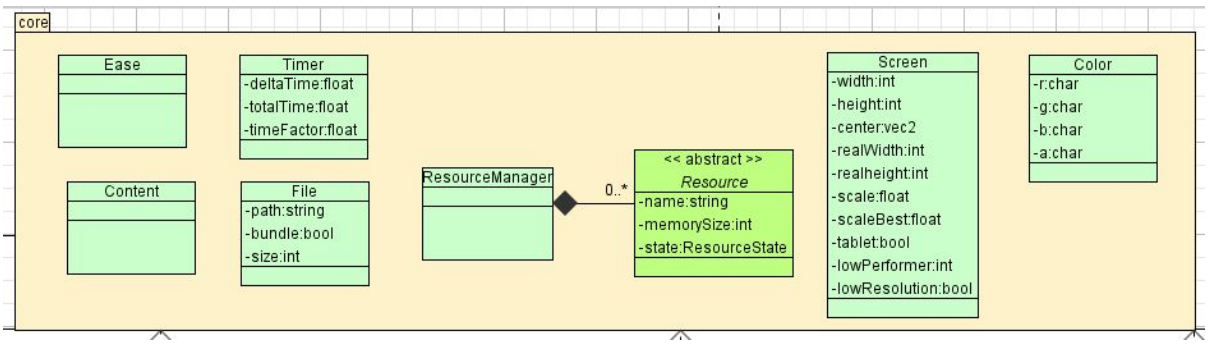


Inside the input package we have some classes to manage input from users. *InputManager* handles all the native input events and there are enqueued and propagated to the Display List at the end of each frame. Some DisplayObjects will dispatch input events that can be listened from other objects to react to the user input.
Utils package contains some other utility helper classes for things such as Localization of the app in different languages, text utilities to format text, join or split strings or other common functions that are not that trivial in C++.



Low-Level classes

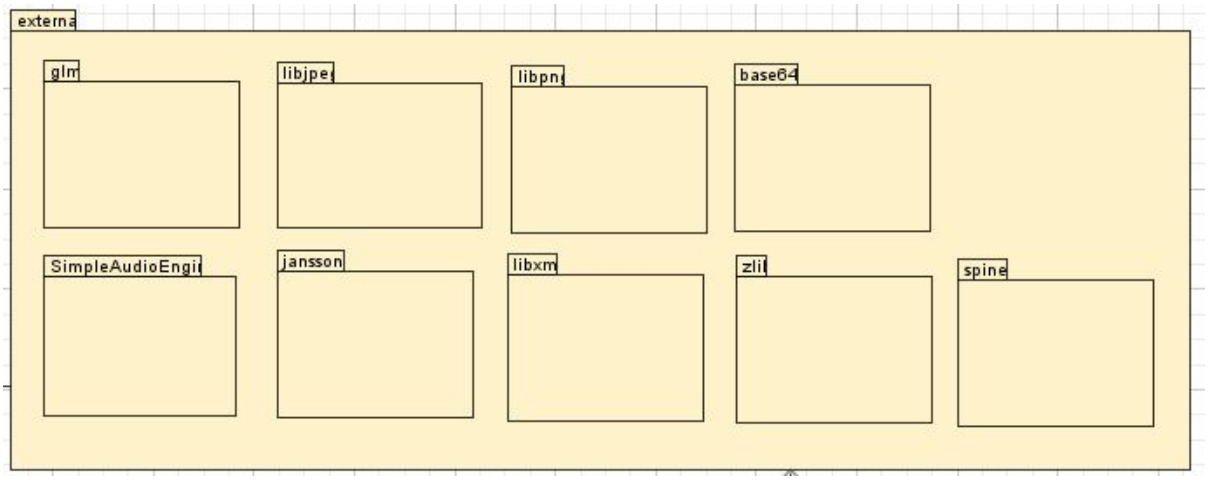
At the bottom we have some low level classes that are frequently used by the internals of the library and that are exposed for the client too. Resource management is done at this level, the client can extend a Resource and start using it seamlessness with the *ResourceManager*. It is important to use *ResourceManager* specially for big assets as images, models or animations to prevent leaks and avoid multiple allocation in memory and multiple loading times for the same asset. Running in a multi threading fashion, the Resource Manager allows to load resources in background while performing other task in the main thread.



Another common functionality required for multimedia apps are Timers, so a class *Timer* provides operations to start, stop and evaluate timers, and along with the Ease class we can create *Tween* animations using different functions very easily.
File give us low-level multiplatform access to read and write files. *Screen* have some handy information about the device we're running the app into, that can be evaluated in execution time, like screen resolution, aspect ratio, etc. that allow the developer to write applications for any screen.

External libraries

The library depends on some other external libraries. The simplest way to use them is to compile them with the library, so the final app and all dependencies are compiled together. However, it is possible to compile the code and link against the generated library, but then we need to link against all the other dependencies and include the binaries for all external libraries in our application. The former will save some application size as compiler can strip out the parts of the code that are not used, and the later can accelerate link times, as we don't need to link every time all the dependencies.



Use Cases

Below we have a list with some of the most common use cases for the library. The cases cover basic functionality such as input/output, rendering graphics, text, etc. Then we go in more detail with some **selected cases**, formally describing each case, pre-conditions, trigger, flow and alternate flows for every case.

Use Case ID	Use Case Name	Primary Actor	Scope	Complexity	Priority
1	Open File	Library client	input	Low	High
2	Open Bitmap (JPEG, PNG, PVR)	Library client	input	Low	High
3	Parse XML	Library client	input	Med	Low
4	Parse JSON	Library client	input	Med	Low
5	Receive Input From Touch Screen	Library client		Med	High
6	Rotate and scale a Bitmap	Library client	input	Low	High
7	Access Bitmap data	Library client	graphics	High	Low
8	Compress Image In Memory	Library client	graphics	Med	Low
9	Use custom pixel shader	Library client	graphics	High	Low
10	Animate Texture	Library client	graphics	High	Low
11	Use texture atlas	Library client	graphics	High	Low
12	Asynchronous load textures	Library client	graphics	Med	Low
13	Load a 3D model with animation	Library client	graphics	High	Low
14	Render text	Library client	text	Med	High
15	Load layout from Flash CS editor	Library client	editor	Med	Med
16	Show Skeletal Animation with sprites	Library client	animation	Med	Med
17	Reproduce Sound and Music	Library client	sound	Low	High

18	Use custom event	Library client	events	Med	High
19	Use Timers and Chronos	Library client	time	Med	High
20	Show Ease animations	Library client	ease	Med	Med
21	Allow user write in textfield	Library client	keyboard	Med	Med
22	View Particle Effects	Library client	particles	Med	High
23	Use custom OpenGL Code	Library client	client	High	High

Open File

Use Case Element	Open a file in disk and read its contents.
Use Case Number	1
Application	input
Use Case Name	Open File
Use Case Description	The client wants to get access to a local file in disk and be able to read file's content. The data retrieved can then be used for any purpose, it can be read as binary or as text and can used as a bitmap, json, xml or whatever the client need. The code is expected to work for all supported platforms.
Primary Actor	Client
Precondition	The file must exists in disk, and must have the proper permissions to be read from the application.
Trigger	Client call to library API to open the file.
Basic Flow	The client creates a File object using the file path as parameter. Then the client can access file data by calling open() method of the file object. When the file object is destroyed, the file is automatically closed. Alternatively, the client can use the convenience method File::load() to do just the same (open, read and close the file) in one single line.
Alternate Flows	If the file doesn't exists an error is shown by console and a return code indicates the caller that the operation went wrong.

Receive Input From Touch Screen

Use Case Element	Receive input from touch screen (or mouse for Windows platform) and interpret the data received.
Use Case Number	5
Application	Input
Use Case Name	Touch Screen Input
Use Case Description	The client wants to know about the user input to perform some action in response.
Primary Actor	Client of the library.
Precondition	The target device must have a functional way of provide input to the application.
Trigger	The application user taps the screen for mobile devices, or clicks the mouse for Windows platform.
Basic Flow	The client of the library adds touch event listeners in one or more *Display Objects*. When the user touches within the bounds of the object, an event will be dispatched so the application can react in consequence. The event itself contains information about the touch, i.e., *screen coordinates* or *velocity* if it was a *drag* gesture.

Rotate and scale a Bitmap

Use Case Element	Rotate and scale a bitmap previously loaded from disk.
Use Case Number	6
Application	Graphics
Use Case Name	Rotate and scale a bitmap
Use Case Description	The client wants to load a bitmap and the show it in the screen rotating and scaling.

Primary Actor	Client.
Precondition	The bitmap must exists in disk, and have a size smaller than the maximum supported texture size for the platform.
Trigger	The client of the library calling the API to perform such action.
Basic Flow	The client creates an instance of a the class *Bitmap* using the path where the image is located as parameter. The Bitmap is created and loaded. Then the client uses the *behaviour* components *Rotate* and *Scale* by adding them to the bitmap. The client can create as many custom behaviours as desired.
Alternate Flows	If there's a problem loading the asset file an error will be shown in the console and the graphic won't be draw.

Render text

Use Case Element	Render text with different fonts, alignments, sizes or shadows
Use Case Number	14
Application	Text
Use Case Name	Write text
Use Case Description	The client wants to show some text in the application being allowed to choose the format to use. Some of the format parameters that can be chosen are the font type, the horizontal and vertical alignment, the size of the text, the size of the "box" where the text is written, and optionally shadows can be added to the text.
Primary Actor	Client
Precondition	The font exists in the assets folder. The size of the text is small enough to fit in a texture.
Trigger	The client of the library calling the API to perform such action.
Basic Flow	The client creates a *TextField* object using the path of the ttf font and the font size as parameters. The *textfield* object must be added to the *Display List* and then the attribute *text* has to be assigned. By the next frame the text will be displayed in the screen.
Alternate	If the font doesn't exists, an error is displayed by the console and the

Flows	text won't be shown.

Reproduce Sound and Music

Use Case Element	Reproduce Sound and Music
Use Case Number	17
Application	Sound
Use Case Name	Sound and Music
Use Case Description	The client wants to reproduce music and on top of the music wants to play sounds.
Primary Actor	Client.
Precondition	The music and sound resources exists within the assets folders.
Trigger	The client calls the library API to perform such action.
Basic Flow	The client can use the shared instance of the *SimpleAudioEngine* to play the music by using the .mp3 file path. Similarly, sounds can be played with the *playEffect* API that would return a handler for the sound. This handler can be used later to stop the sound or modify its volume.
Alternate Flows	If the sound or music files are missing, an error will be shown in the console and the sound or music won't be reproduced.

Class diagram

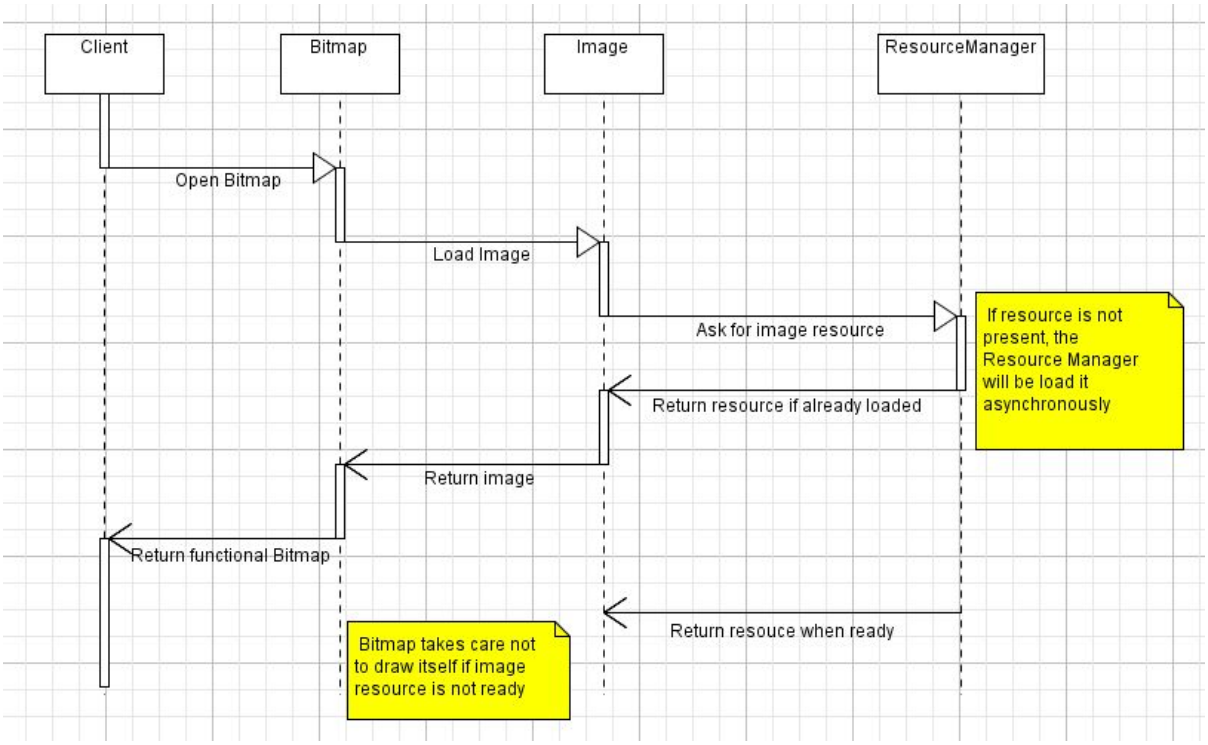
The diagram has been decomposed by systems as shown in the subsystems section, nonetheless in the following link can be found the full-size diagram:

see full size version

Sequence diagrams

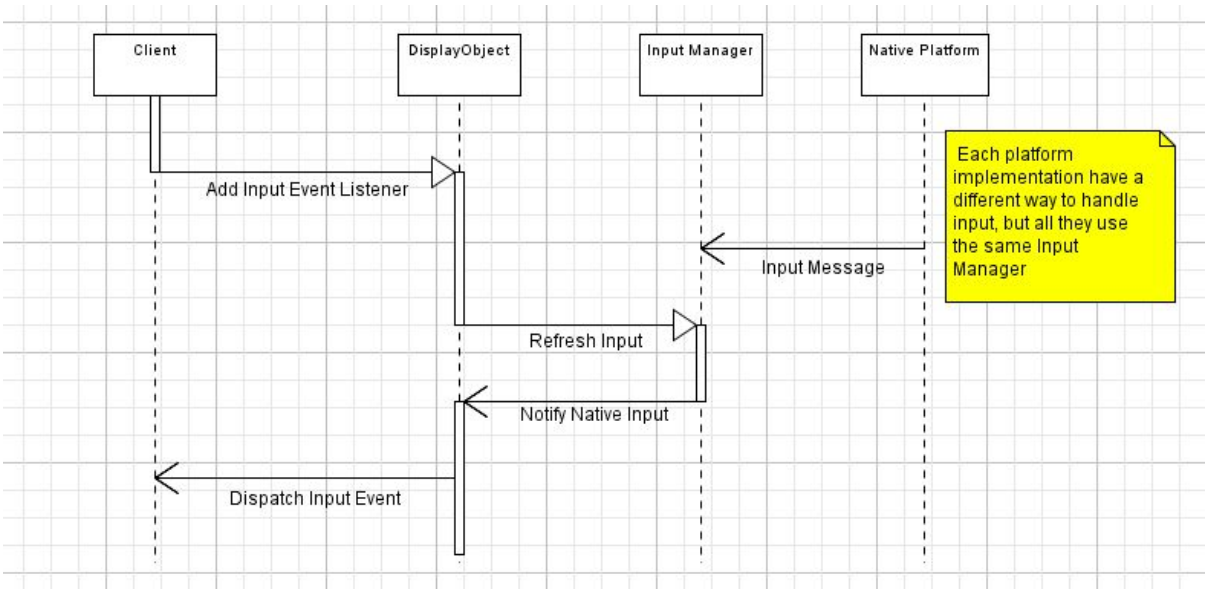
Open Bitmap

Bitmaps are the most common way to display graphics within the library. Then next diagram shows how a client interacts with Bitmap API and behind the scenes, an image is created in collaboration with the ResourceManager that keeps track of the image for potential later use.



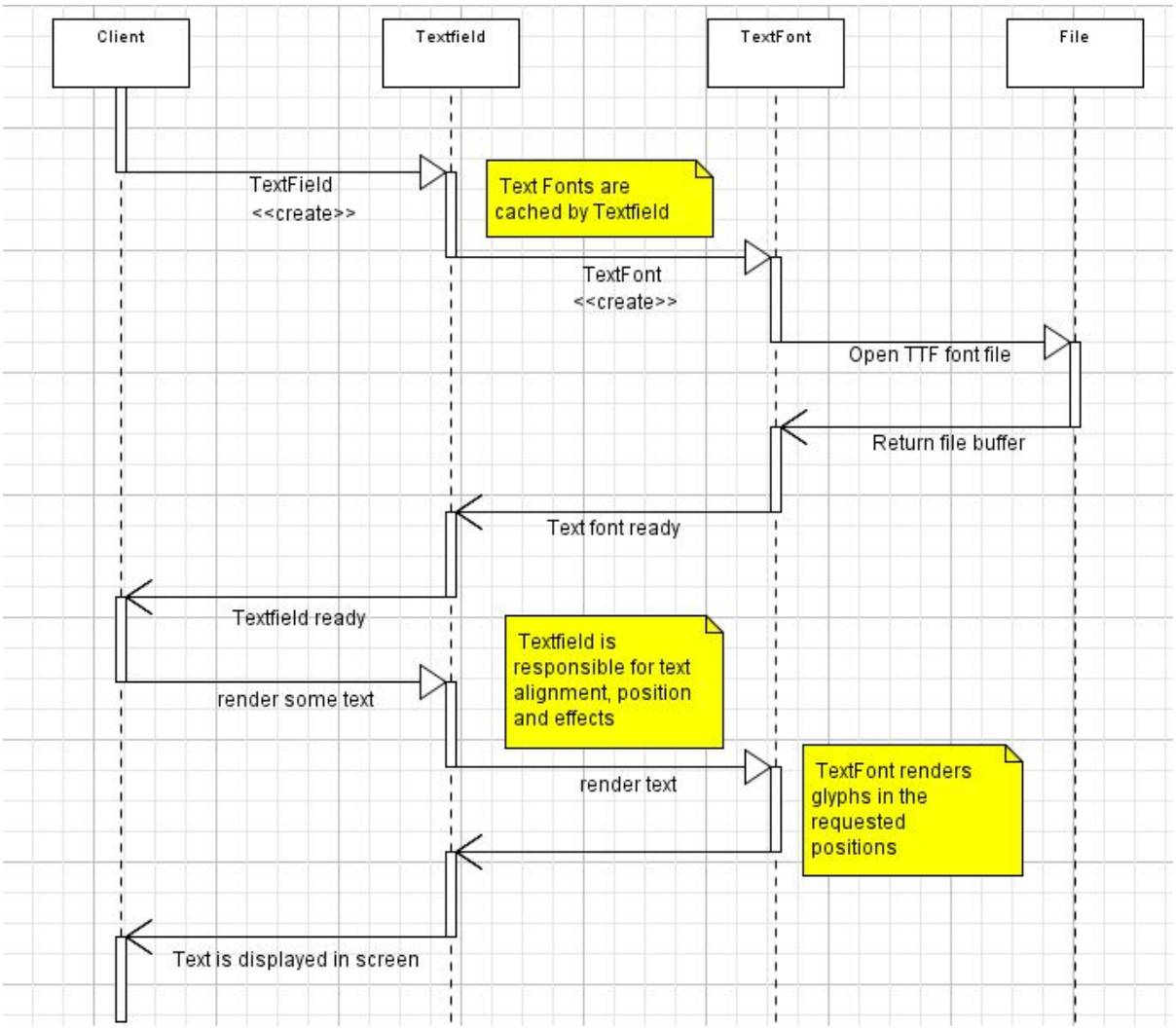
Receive Input From Touch Screen

The system uses an event system to give feedback about the user input. In such model, the client subscribe to certain input events, and then the system dispatches the appropriate events when the user taps the screen. Behind the scenes, the native platform (among the supported ones, e.g., iOS, Android and Windows), feeds the Input Manager with new input, then the display list asks once per frame the Input Manager to process that input. The display list is traversed looking for candidates that if selected will dispatch the touch input event.



Display text with different fonts, alignments, sizes, shadows

Another very common task is show text in screen. To accomplish that, the client can use the *TextField* class. *Textfields* are *DisplayObjects* that can be instanced and placed in the display list. Internally, they use a *TextFont*, who is responsible for draw the text.



Coding conventions

The code is separated in classes, each class is splitted in a header file with the extension .h and c++ code file with the extension .cpp. Small internal classes however can be placed inside .cpp files, but they should never use other namespace than rag. Header files contains header guards, in the form:

```
#ifndef Rag_Foo
#define Rag_Foo

... class declaration ...

#endif
```

The order in which the headers are included should never end in different behaviours, each class should contain all its dependencies.

The code from the rag library is contained inside the `rag` namespace, with the only exception of events, which have a reserved namespace called `events`. The namespace declaration is placed in the header file, wrapping all the class or classes declaration/s. The same applies for the .cpp file.

```
// h file

namespace rag
{
  ... class declaration
} // rag
```

```
// cpp file

namespace rag
{
 ... class definition
} // rag
```

Operator overloading is only used when the meaning is obvious, and disallowed in all other places.

The order of the declarations in a class should start with `public`, then `protected`, then `private`. Inside each section, the order of the declarations should be: constructors, destructors, methods, class members.

When casting it is preferred to use explicit cast like `static_cast<Foo>(my_var)` instead of C cast like `(Foo)my_var`.

C++11 can be used when is appropriate. Being the primary platforms Windows, iOS and

android, with all 3 supporting C++11 there is no reason to not use it.

Naming is an important field and following rules should be applied across all the engine:

- File names should match class names, both starting with a capital letter. Uppercase characters can be used to separate words. Filenames should be descriptive and very concise, with one or two words at most.
- Class, type names and enums should follow the same rules as file names.
- Variable member names, should also be short and concise. Abbreviations are disallowed, as they can be confusing. Variable names starts with lowercase and can have uppercase letters to separate words.
- Method names, starts with lowercase and can use uppercase to separate words.
- Macros although rarely used, should be all in uppercase.

Braces should be followed by a return character, like:

```
struct MyStruct
{ // new line
   int a;
   char b;
};
```

The braces convention is used also for functions, loops or other conditional code and namespaces.

For other topics not covered here, in general it is recommended to be consistent across the code. For example if we need to add some code to an existing file, we should follow the same file conventions, if they are not covered in this section.

List of folders

Below there's a list with the most significant folders and a brief explanation for each one.

```
rag·····························Root folder.
 ├──────android·····················Android platform specifics.
 ├──────doc·························Rag Documentation.
 ├──────external····················External libraries.
 │     ├──────audio·················Cocos Denshion audio library.
 │     ├──────base64················Base64 conversion utilities.
 │     ├──────glm···················OpenGL Mathematics library.
 │     ├──────jansson···············JSON parser.
 │     ├──────libjpeg···············Reading and Writing JPEG images.
 │     ├──────libpng················Reading and Writing PNG images.
 │     ├──────libxml2···············XML parser.
 │     ├──────pvr···················Reading and Writing PVR images.
 │     ├──────spine·················2D Skeletal Animation.
 │     ├──────texture-atlas·········Utilities to generate texture atlas from
sprites.
 │     └──────zlib··················File compression utilities.
 ├──────include·····················Main folder to include when using rag engine.
 │     └──────rag···················All classes from rag are here.
 ├──────ios·························iOS platform specifics.
 ├──────linux·······················linux platform specifics.
 ├──────particles···················Particle engine and editor.
 ├──────shared······················Some other classes in experimental state.
 │     ├──────behaviours···········Display Objects that can modify behaviour of their
parent.
 │     ├──────debug·················Utilities to debug.
 │     └──────utils·················Other miscellaneous code utilities.
 ├──────test························Unit tests for rag code.
 ├──────tools·······················Rag tools and utilities.
 │     ├──────colorama··············External library used for tools.
 │     └──────project_template·····Default content of an empty Rag project.
 └──────win·························Windows platform specifics.
```

As we see, the library is composed of an external folder with all the other libraries, then there are one folder for each specific target platform, e.g., Windows, iOS, Android and linux. The part which is actually written for the engine, is inside *include/rag* folder, which contains most all the classes from rag library. Then there's a folder for unit tests, a folder with the implementation of the particle editor and particle engine, another one for some experimental features which may be part of the library in the future, and one folder for tools. The tools are mainly written in python, and are explained below in the next section.

The doc folder contains documentation in .pdf, .rtf and .html formats. It contains the reference documentation auto-generated from the code.

Engine Tools

In order to help with the production pipeline, several tools accompany the graphic engine. Being a programmer-centric engine, the tools are provided in the form of python scripts that help us in the task of organize and process the contents of the application. Below there's a list of the current tools available along with a summarized explanation for each one. Then we go in detail for each tool in this section.

script name	summary
new_project.py	Creates a new rag project in a specific folder.
premake5.exe	Creates the solution project for different platforms.
binarize_flash.py	Binarizes the flash .xml files, leveraging CPU, memory and disk resources.
flash2code.py	Generates code to access flash in a typified way.
clean_flash.py	Notifies about unused symbols or graphics in order to save space.
atlas-make.py	Creates textures atlases from flash files.

new_project.py

Generates the folder structure from a basic template for a new project. The new project folder will be at the same level than the engine, as it is required to stay there in order to work properly. The project contains the minimum structure to work, but it is required to generate the solution first with premake5.exe tool, explained below.

premake5.exe

This 3rd party tool generates the project files or solution for different platforms, including windows, iOS and (with an extension) Android. The Lua configuration script required is already done by default in the template when generating a new project. Later modifications can be done for the specific project.

After running premake a completely functional project should have been generated, that should compile, link and execute as a simple "Hello World" demo code.

binarize_flash.py

Although the engine can load assets directly from Flash CS, it can be slow in terms of performance to read the long .xml files that the editor generates. It is recommended instead to binarize the output, which can be done with the *binarize_flash.py* tool. The tools traverses the assets folders looking for Flash directories, and converts all .xml files into a binary version, extracting only those parts interesting for the engine, reducing the file size by a factor of x10 approximately.

Using the binarized version impacts on key-aspects for mobile platforms like *speed*, *memory* and *size* in disk. Indeed, the binary files are read faster, because they are smaller, and additionally there's no need to parse any .xml file, which have also a direct impact on memory for the same reasons.

The downside of the flash binarizer is that binarized files can't be opened anymore by the Flash CS editor, so it's recommended to use the plain .xml files under development in order to be able to iterate faster, and binarize files as a final step before deploying a build.

The original sources of the flash files should be kept safe in any other place of the project, because the binarizer deletes the originals.

flash2code.py

Symbols in flash make use of the Display List concept. Symbols may contain other symbols, and this is a powerful and convenient system that allows artist to reuse their contents over different scenes. For instance, an artist can create a button made of a graphic and a text, and may want to use the same button in two or more separated panels. That's precisely what Flash symbols allows them to do. Several symbol instances can be created, and changing the symbol would affect all instances at the same time, reducing the cost of maintaining a big library of assets.

When those assets are referred from C++ code, there are several ways to access symbols. We can load symbols using a string with the exact path of each component and then we can inspect into Display List hierarchy by name, like shown in the example below:

```
rag::XFLParser p;

// May fail if the route is changed by an artist.
auto popupx* = p.load("assets/flash/LIBRARY", "popupx");
addChild(popupx);

// May fail if "text" name changes
setText(popupx->getChildByName("text"), rag::localize("TID_X"));

// May fail if hierarchy changes
popupx->getChildByName("label_1")->getChildByName("item")->addChild(new
rag::Bitmap("assets/star.png"));
```

As we see, there are some issues with this approach. In order to improve stability once a project starts to be well established (meaning there are meaningful amount of assets intertwined with code), this tool can help us a lot by moving this issues from execution-time to compile-time.

What *flash2code.py* does is generate a .cpp and .h file were all paths and the Flash hierarchy is typified, in a way that if something is broken by moving files from one place to another, or by a change of a name, the problem will be noticed at compile-time,

meaning that it's easier to spot issues in a deterministic a fast way.

With the auto generated code, the code for the same purpose will look more like:

```
auto popupx = new flash::popupx();
addChild(popupx);
setText(popupx->text(), rag::localize("TID_X"));
popupx->label_1()->item()->addChild(new rag::Bitmap("assets/star.png"));
```

And the auto generated code looks like this, but is nothing we should worry too much about, since is auto-maintained.

```
class label_1: public rag::DisplayObject
{
public:

        rag::DisplayObject* item();
};

class popupx : public rag::DisplayObject
{
public:
        popupx();

        rag::TextField* text();
        label_1* label_1();  // Typified cross-references between symbols.

};
```

Although is completely optional, the general recommendation except for very small projects is use this tool to generates the code to be safer when referring to symbols coming from Flash editor.

clean_flash.py

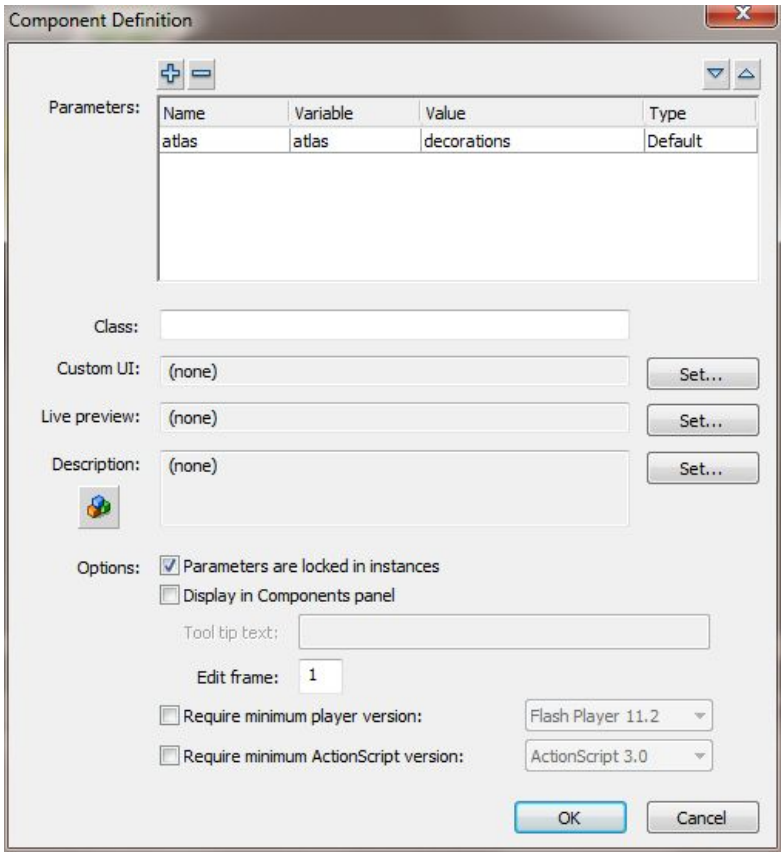
This tool examines the flash folders and looks for symbols and images that are not found in the Flash editor *DOMDocument.xml*. This files are considered missed references that nobody can point to, so for all purposes can be considered trash. The tool warns about this kind of files, and offers an option to clean things up. The tool also notifies about missing references (i.e., files that are referenced from other places but are not found) as potential errors.

atlas-make.py

One common bottleneck for graphic-intensive applications for mobile platforms is the limited amount of draw calls imposed by hardware. It is well known and a well-established standard the use of texture atlas, where different sprites are distributed over a big image, and then the application that interacts with the atlas can render several sprites in one single draw call. This optimization is a must for a performant application where too many draw calls can ruin the performance, leading to a slow and battery-hungry application.

To create a texture atlas first an artist must decide what sprites are going to be part of the atlas. A new symbol should be created and should contain the images and symbols required to be in the atlas. The symbols doesn't need to be in any particular order. The symbol can look more or less like this:

Then we right-click on the symbol, and click on "*Component Definiton…"*. This will open the window where we define a new parameter "atlas" and for "Value" we give the name of the atlas where all this images will be:



We then save all our work in Flash CS and close the editor.
Then we run the tool *atlas-make.py* that would parse the symbols with the “atlas” variable and would generate the corresponding atlases. At the same time, the Flash .xml files are modified to include the extra information about the atlases and texture coordinates.

```
D:\prj\RagProject>python tools\atlas-make.py
parsing.....................................
.....................................

8.70 seconds
```

The generated sprite-sheet looks like this:

And the engine will automatically take care of draw all sprites that use the same texture in a single draw call.

Engine pre-requisites

Rag engine uses a few dependencies, which are already included in the project for convenience. It is however required to install some other software packages depending on the features we want to use from the library. As a minimum it's going to be required an IDE, that may vary depending on the development platform.

Windows: Microsoft Visual Studio (tested with 2013 and 2015)
Mac OS: XCode (Tested with XCode 5 and 6)

It is worth noting that Visual Studio can be used for Android development, although it's also possible to run the projects from the command line, under Mac OS or linux distributions.

Although not strictly required, is it recommended to install Python 2.7 to get all the benefits of the tools under the */tools* folder. It can be downloaded from their distribution site: https://www.python.org/downloads/

Other 3rd party authoring tools that generates content that can be read directly with the engine are:

There are optional software packages in the form of 3rd party authoring tools that generates content that can be read directly with the engine. Those are listed here:

- Flash CS. Create layouts and put bunch of images together. Organize assets.
- Spine Editor. Skeletal 2D Animation.
- General image processing (Photoshop, Gimp, Illustrator, etc.)
- General 3d modelling tools (Blender, 3d Max, ...)

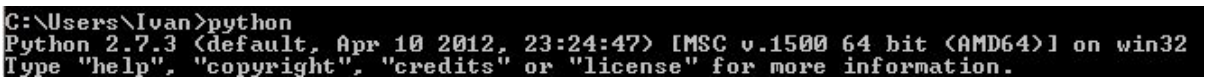
Setup instructions

Step 1: Clone repository

First we need to clone the git repository. We can found git in the url https://git-scm.com/.

```
git clone https://ivan_sanchez@bitbucket.org/ivan_sanchez/rag.git
```

The we need to know if we have python installed, we can simply type “python” in a console and see what happens. If it’s installed, we should see something like:



Step 2: New project folders structure

To generate our first project, we can open a terminal console, move to the rag/tools folder, and then type

```
python new_project.py MyProject
```

A new project besides rag folder will be generated from a standard template with the required items to start working. the project will contain folders for assets, sources and tools, and the .dlls used by the engine.

Step 3: Generate the solution

Now we should run premake5 with the corresponding parameter according to our target platform. For a windows machine, we can switch to the new folder called MyProject/tools and type:

```
premake5.exe vs2015
```

Now we can open the Visual Studio solution in the “build” folder, compile and execute the project.

Future directions

The engine doesn't pretend to compete with the existing commercial products, as is far behind into what a final user expects. It is however an interesting project that fits better in the open-source culture, and that could eventually gain some traction in front of other open-source engines such as Cocos2X. The main difference from other projects is in it's simplicity, as has been an objective from the beginning not to over-complicate things and to create a minimalistic interface on top of some low-level libraries to create games rapidly. It offers also a modern object oriented approach which uses C++11 features, recently set as the standard version of C++.

There are plenty of room for optimizations and improvements in the engine, such as the addition of new features, as 3D support (which is already developed in an experimental state), or other integration with 3rd party tools. There's also work to do with tools, specially with a custom editor to create scenes, to remove the dependencies with the Flash CS editor, which may be not the most comfortable editor for the open-source community.

The maintenance of every feature in a multi-platform, multi-environment way is quite cumbersome, so it could benefit from the additions of the community. The port to new platforms is an interesting addition, indeed it should be easy to adapt the engine to MacOS, but it can be more challenging (although possible) to port it to WebGL, Windows Mobile or others.

Conclusion

We have moved through the process of create an entire little game engine from scratch, using only some standard low-level libraries, and we have learned a lot in the process. Even for a simple 2D engine, dealing with computer graphics in a low level fashion is always challenging. We also have write code that runs multi-threaded, fast and reliably, in multiple platforms, which is far from trivial and give us a big sense of accomplishment and satisfaction.

We have write a lot of documentation for the project, there are some examples along with the code, and also a small game developed with the engine. All this documentation, can help any programmer to rapidly start programming with the engine. We also have the tools for create a functional project within a few seconds.

Despite all the achievements, the engine can be still considered in its infancy for the standard in a software product, and there are many features and improvements that can be done. The decision to open the project to the open-source community has been postponed for a while, but it is the most certain future for the engine.

All in all, the whole process has been a very rewarding experience that have teached us a lot, and give us an insight of what happens under the hood in those big commercial engines that are growing so popular these days, helping us to understanding them better.

References

(1) http://www.pocketgamer.biz/news/49118/the-charticle-the-steady-growth-of-subway-surfers/
(2) http://www.theguardian.com/technology/2014/dec/09/clash-of-clans-billion-dollar-mobile-games
(3) http://www.learn-cocos2d.com/2013/11/mobile-game-engine-popularity-index/
(4) http://www.monodevelop.com/
(5) http://www.cocos2d-x.org/
(6) http://cocos2d-x.org/games

Bibliography

McConnell, Steve. *Code Complete*. Unterschleissheim: Microsoft, 2005. Print.

Freeman, Eric, Elisabeth Freeman, Kathy Sierra, Bert Bates, and Marie-Cécile Baland. *Design Patterns*. Beijing: O'Reilly, 2005. Print.

Astle, Dave, and Kevin Hawkins. *Beginning OpenGL: Game Programming*. Boston: Thompson, 2004. Print.

Hunt, Andrew, and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Reading, MA: Addison-Wesley, 2000. Print.

OpenGL ES Standard Manual, https://www.khronos.org/opengles/sdk/docs/man/

Valve Publications, http://www.valvesoftware.com/company/publications.html

Appendices

Reference Manual

With this document there should be attached the **Reference Manual**, in a file called *refman.pdf*. It is attached apart due to the extension of the file. Contains documentation generated automatically from the comments found in code. It can be found also inside rag/doc/rtf

Arkanoid

Along with this document, it is delivered

Lines of code

In the entire rag root folder, there are more than 450K lines, mostly in C/C++. Most of them are from external libraries.

Language	files	blank	comment	code
C	195	34856	98645	292962
C/C++ Header	498	17305	36574	108694
C++	160	7638	8277	33065
Bourne Shell	13	4312	4526	27967
Python	21	691	711	3582
JSON	1	0	0	2419
Objective C	5	476	397	2199
m4	9	258	46	2068
make	14	203	120	1504
MSBuild script	9	0	0	1186
Objective C++	7	332	117	1048
XML	1	2	24	200
CMake	10	37	20	161
Lua	1	13	11	81
Perl	2	16	0	69
Windows Resource File	1	21	29	33
DOS Batch	2	3	0	21
SUM:	949	66163	149497	477259

If we explore only the include folder, which contains most of the code wrote for this project, the results are like this:

Language	files	blank	comment	code
C++	27	1169	512	5032
C/C++ Header	37	852	1245	1721
Objective C++	1	27	2	73

```
SUM:                            65          2048          1759          6826
--------------------------------------------------------------------------------
```

We can see there's only 6826 lines of code, making the project not that big for the number of features supported.

Full list of folders

A brief list of folders and the content inside each one was explained in section List of folders.
For reference, here is the full list:

```
rag
├─────android
│     ├─────libxml
│     └─────libxml2
│          └─────jni
│               └─────include
│                    └─────libxml
├─────external
│     ├─────audio
│     │     ├─────android
│     │     │     └─────jni
│     │     ├─────include
│     │     ├─────ios
│     │     └─────win32
│     ├─────base64
│     ├─────glm
│     │     ├─────bench
│     │     │     ├─────core
│     │     │     ├─────gtc
│     │     │     └─────gtx
│     │     ├─────glm
│     │     │     ├─────core
│     │     │     ├─────gtc
│     │     │     ├─────gtx
│     │     │     └─────virtrev
│     │     ├─────test
│     │     │     ├─────bug
│     │     │     ├─────core
│     │     │     ├─────external
│     │     │     │     └─────gli
│     │     │     │          ├─────core
│     │     │     │          └─────gtx
│     │     │     ├─────gtc
│     │     │     └─────gtx
│     │     └─────util
│     ├─────jansson
│     ├─────libjpeg
│     ├─────libpng
│     ├─────libxml2
│     │     └─────include
│     │          └─────libxml
│     ├─────NDKHelper
│     ├─────pvr
│     │     └─────OGLES2
│     │          └─────Build
│     │               ├─────Android
│     │               ├─────iOS
│     │               ├─────LinuxGeneric
│     │               └─────WindowsVC2010
│     ├─────spine
│     │     ├─────.settings
│     │     ├─────data
│     │     ├─────Debug
│     │     ├─────include
```

```
│   │   │   └──────spine
│   │   ├──────Release
│   │   └──────src
│   │       └──────spine
│   ├──────texture-atlas
│   │   ├──────src
│   │   │   ├──────png++
│   │   │   │   ├──────example
│   │   │   │   └──────test
│   │   │   │       └──────cmp
│   │   │   ├──────tclap
│   │   │   │   ├──────config
│   │   │   │   ├──────include
│   │   │   │   │   └──────tclap
│   │   │   │   └──────msc
│   │   │   │       └──────examples
│   │   │   └──────texturepacker
│   │   │       └──────.svn
│   │   │           └──────text-base
│   │   └──────TextureAtlas
│   │       ├──────Release
│   │       └──────TextureAtlas
│   │           └──────Release
│   │               └──────TextureAtlas.tlog
│   └──────zlib
├──────include
│   └──────rag
├──────ios
├──────linux
├──────particles
├──────shared
│   ├──────behaviours
│   ├──────debug
│   └──────utils
├──────test
├──────tools
│   ├──────colorama
│   └──────project_template
│       ├──────assets
│       │   ├──────fonts
│       │   └──────shaders
│       ├──────src
│       └──────tools
│           └──────colorama
└──────win
    ├──────AntTweakBar
    │   ├──────include
    │   └──────lib
    ├──────PowerVR
    │   ├──────Include
    │   │   ├──────EGL
    │   │   ├──────EWS
    │   │   ├──────GLES
    │   │   ├──────GLES2
    │   │   ├──────GLES3
    │   │   └──────KHR
    │   ├──────iOS
    │   │   └──────Resources
    │   │       └──────OGLES2
    │   └──────Windows
    │       ├──────Resources
    │       ├──────x86_32
    │       │   └──────Lib
    │       └──────x86_64
    │           └──────Lib
```

```
├──────pthreads
│     ├──────dll
│     │     ├──────x64
│     │     └──────x86
│     ├──────include
│     └──────lib
│           ├──────x64
│           └──────x86
└──────system
```

List of classes

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

Class	Description
Event	Base class for event system
EventDispatcher	Base class used to dispatch events
EventListener	Interface that allows to listen events
KeyboardEvent	Event to handle KeyBoard actions
TextEvent	Dispatched by InputText when user writes one character
TouchEvent	Event for handle input from screen
path	Mimics boost fs::path class with some limited functionality
Bitmap	Provides the ability to show images
BMPFont	Font system based on bitmap fonts
Chrono	Helper class to count time elapsed from a moment in time
Color	Represents RGBA color
Color4B	Color representation using 4 bytes
DisplayObject	Core object used to display things in screen
DropShadowFilter	Shadow effect for TextField instances
Ease	Collection of code-generated curves useful to create procedural tween animations
File	File multiplatform abstraction to read contents of a file
Image	Image object
ImageLoader	Interface to load images
ImageLoaderJPG	Loader for .jpg format
ImageLoaderPNG	Loader for .png format
ImageLoaderPVR	Loader for .pvr compressed format
InputManager	Simple Input Manager
ITextFont	Interface for text fonts

Keyboard	Multiplatform keyboard abstraction
KeyboardManager	Singleton class that dispatches Keyboard events
Material	Represents a render material that would affect how objects will be rendered
MovieClip	Allows to use imported animations created by Flash CS tool
Frame	Internal of MovieClip, represents a single frame
Program	Represents a Shader Program
Rectangle	Represents a Rectangle
Renderer	Contains methods to render objects
RenderTarget	Object where DisplayObject instances with render capability are suposed to render
Resource	Abstract class the represent a game Resource, typically something costly to loaded
ResourceMgr	Handles Resource management, including loading and unloading Resource instances
Screen	Contains information about the current device
Shader	Represents a GL Shader
TextEdit	Creates a native window to edit a text
TextField	High level abstraction to render texts in display list
TextInput	Helper object to add input to a TextField
Timer	Provides time-related functionality
TTFFont	Implementation of ITextFint based on TrueType or OpenType Fonts
TUniformVar	Shader uniform representation
Vertex	Vertex representation
XFLBinaryParser	Binary version of parser for documents generated by the 3rd party editor tool FlashCS
XFLParser	Parser of XFL documents generated by the 3rd party editor tool FlashCS
SkeletonDrawable	Display 2D Skeletal Animations made with the 3rd party tool Spine