Game framework : code outline, v1.0

Overview

This document provides a summary of the important classes and functions in the provided framework and an outline of how the framework works.

You will implement your program by sub-classing the framework classes and changing existing behaviour or implementing new behaviour for the associated objects. There are really only two main conceptual types of objects - a BaseEngine object (which controls the overall game/program) and an object for each of the moving things in the program. Of course each of these types can be sub-divided, so you may want to have multiple classes for different types of moving objects. You may also want to add new classes to maintain other information, e.g. any level maps or data.

It is important to keep in mind that objects are responsible for themselves. They can ask other objects to do things where necessary, but should not actually directly change the state (data) of other objects. i.e. your data should be private, not public.

Within your program there are different classes, each with its own responsibilities. Each class may inherit functionality from a base class, which is made available to you. You can change the way that functionality works by implementing or changing the virtual functions that are provided. You will need to be aware of the virtual functions which are made available by the base classes for modifying the behaviour of the game, and of the utility functions which are made available to perform specific tasks. These are listed with a brief summary of their purposes and the ways in which they can be used to alter the behaviour of the base classes in the sections of this document. Although the descriptions assume that you will not be changing the framework base classes, this does not prevent you from doing so if you so wish.

In addition to the documentation in this file, you should take a look for yourself at the supplied header files. You will find a lot of useful methods in them, particularly for drawing functions, that are not all described in this document (deliberately). This document aims to give you enough information to get started on your coursework and should not be considered to be complete.

The purposes of the framework can be summarised as follows:

- To wraps up SDL (Simple DirectMedia Layer) so that you don't need to learn SDL functions.
- To hide re-draw issues (to run relatively quickly) allowing you to produce a fast program without needing to spend hours debugging the redraw code.
- To avoid the need for good maths skills from you. Maths ability is not a pre-requisite for the course.

Version

Version 1.0: Initial version.

1 Classes

The available classes are summarised below.

1.1 Basic classes

The basic classes embody the framework of the game. The basic classes are as follows:

Class name	Purpose
BaseEngine	The basic engine for the program. This drives everything and includes: System initialisation and clean-up code. Basic drawing operations (e.g. set pixel, draw string). Input handling functions.
DisplayableObject	Code to draw and move a sprite. Includes basic position calculation code.
FontManager	Code to maintain a set of fonts, so that each is only loaded once.
JPGImage	Code to load and draw a .jpg image.
MovementPosition	A simple class to work out a position for an object, to avoid you having to do the calculations yourself.
TileManager	A simple class to maintain an array of tile information and draw the tiles.

1.2 BouncingBallMain classes

`BouncingBallMain` is a simple demonstration of the use of the framework.

Class name	Purpose
BouncingBallMain	Simple sub-class of BaseEngine. Bounces a ball around the screen.
BouncingBall	Simple sub-class of DisplayableObject. Draws a ball and calculates the new position for it as it moves.

2 Overview of flow of control

2.1 Initialisation steps:

- Initialise SDL
- Create window
- Create framework objects
- Call `GameInit()`

2.2 Main loop

- Update key presses and mouse moves and call appropriate functions:
  - `KeyUp()`
  - `KeyDown()`
  - `MouseDown()`
  - `MouseUp()`
  - `MouseMoved()`
- Call `GameAction()`
- Call `GameRender()` This calls one of the following:
  - `DrawScreen()`. This will call `CopyAllBackgroundBuffer()`, `DrawStrings()` and `DrawChangingObjects()`
  - `DrawChanges()`. This will call `UndrawChangingObjects()`, `DrawStrings()` and `DrawChangingObjects()`.

2.3 Deinitialisation

The Deinitialisation() method just calls the virtual function `CleanUp()`, then frees the memory which was allocated to store the key press array.

3 A procedural view of the framework

From a procedural point of view, the framework goes through the three generic steps:

1. Initialise
2. Game Loops
3. Deinitialise (or clean-up)

These steps are explicitly executed by the start-up code, found in `mainfile.cpp`. The `mainfile.cpp` code has the following steps:

1. Create an object of a sub-class of `BaseEngine`
2. Call `Initialise()` on the `BaseEngine` sub-class object, specifying the window caption, the size of the window to display and the default font to use for text.
3. If initialisation succeeded then call `MainLoop()` on the `BaseEngine` sub-class object.
4. Once the main loop has finished, called `Deinitialise()` on the `BaseEngine` sub-class object.

3.1 Constructor

The constructor for the `BaseEngine` needs to know the maximum number of separate screen regions which will need to be updated at once. It uses this information to create an array of update regions for the screen, to avoid having to keep recreating the array as needed. This number is related to the number of displayable objects that will need to be displayed, since the moving of each one will require a section of the screen to be updated. It is recommended that this value is larger than the maximum number of moving objects on the screen, since each object will need a part of the screen to be redrawn when it moves. If the number of displayable objects exceeds this value then the entire window will be redrawn instead of just the changed parts. It is not a major problem if you set this value to a value larger than required, but is more of a problem is you set it to a value too small.

3.2 The Initialise() function

The `Initialise()` function initialises SDL and creates the initial window. It then loads the font that will be used for displaying text. Finally, it calls the `GameInit()` virtual function.

You should not need to change `Initialise()` since you can override the `GameInit()` virtual function and perform any additional initialisation that you need within the override.

3.2.1 The GameInit() function

`GameInit()` is a virtual function. The base class implementation of `GameInit()` calls two other virtual functions:

`InitialiseObjects()`: This is usually implemented in your sub-class to create all of the objects which move around the screen.

`SetupBackgroundBuffer()`: This is usually implemented in your sub-class to draw the background.

By implementing this virtual function, a sub-class can provide initialisation code which will be called before the `MainLoop()` function.

3.3 The MainLoop() function

The first thing that the `MainLoop()` function does is handle key-presses or mouse events (movement or button presses). SDL has an event system which provides the program with information about events which have happened such as key presses. Rather than require you to access SDL directly and pick up these events, the framework calls `SDL_PollEvent()` to pick up all of the pending events. As far as you are concerned, the only events to worry about are the pressing and releasing of a key on the keyboard, the movement of the mouse or the pressing or releasing of a mouse button. Keyboard events and input are described in section 7.

Following the handling of SDL events, the `MainLoop()` function calls two other functions. Firstly the `GameAction()` virtual function is called to handle any game logic (handling input, moving things on the screen, and so on), then the `GameRender()` function is called to draw (render) the image of the current game state on the screen. The `GameAction()` function is discussed in section 4. The `GameRender()` function is discussed in section 5.

3.4 The Deinitialise() function

The `Deinitialise()` function first calls the virtual function `CleanUp()`. Following the call to the `CleanUp()` function, the memory for the font, key status array and screen background buffer are deleted.

Sub-classes can implement the `CleanUp()` function to free any memory that they allocate. The base class version does nothing.

4 The GameAction() function

The `GameAction()` function is where the game logic, in terms of moving things around and deciding what to do, should be performed. The `GameAction()` function in the base class calls `UpdateAll-Objects()`, which will tell all of the `DisplayableObject` objects to update themselves (by calling `DoUpdate()` on them). This means that (providing the CPU is fast enough), each object will be told to update itself every 10ms.

You may wish to override this method in a sub-class and change the behaviour, or make the behaviour depend upon the current state.

5 The GameRender() function

The `GameRender()` function is where the screen should be drawn. The base class provides a basic function for doing this and sub-classes can change this behaviour if desired.

Note: You should not need to override this function. If you do, then you will need to do all of the drawing for the background and moving objects yourself.

The basic implementation differentiates between needing to redraw the entire screen and just redrawing the moving objects.

The framework maintains a background image for the screen. The idea is that, if the entire window needs to be drawn (for example when the game state is changed) then the background is drawn over the entire window. Otherwise, if the whole screen hasn't changed, then only those parts of the window which have actually changed need to be updated.

Whenever moving objects need to be drawn on the screen, the picture on the screen should consist of the background image, plus the moving objects, which are drawn on top of it. When moving objects are moved, three steps need to be performed:

1. 'Undraw' the objects from their old position.

   (i.e. Draw the background image over the position where the objects were so that the objects are removed.)

   Note that this means that it is important to record the position at which an object is drawn for later use when it is 'undrawn', to know which part of the background to redraw.

2. Move the object to a new location.

3. Draw the object in the new location (and store the location so that it can later be removed).

With this understanding of how the drawing works then it is possible to consider how to use the drawing mechanism.

Initially, the entire screen is drawn. After the initial draw, the full screen will only be redrawn if the `Redraw(true)` function is called. After the initial draw, the whole screen should probably only need to be redrawn when the game state is changed, meaning that the whole window appearance changes. When the game state is changed, the `Redraw(true)` function (or the `RedrawWholeScreen()` function on the `DisplayableObject`, which does the same thing) should be called. This will force an entire refresh of the screen, which will call the `DrawScreen()` function. This can be modified, but will usually copy the background to the foreground, then draw the moving objects on top. The background will have been drawn by a previous call to the `SetupBackgroundBuffer()` function.

Whenever anything on the screen is moved/changed, it is necessary to tell the framework to redraw the moving objects. A call should be made to `Redraw(false)`, which will then force a call to `DrawChanges()`. `DrawChanges()` can be changed, but the basic version will redraw the moving objects via calls to `UndrawChangingObjects()` (to redraw the background over the current moving objects), `DrawStrings()` to draw any text strings, then `DrawChangingObjects()` to draw the moving objects at their new locations.

The previous `DoUpdate()` methods on the moving objects will have stored the old positions of the objects then moved them. The old positions will be used for re-drawing the background and the new positions will be used for drawing the objects in their new positions on the screen.

5.1 Summary

In summary, the base class implementation of this function does one of the following things:

- If `Redraw(true)` was called, then it will redraw the whole window, including setting up the background.

  To do this it will call `DrawScreen()`.

  Note: `Redraw(true)` is effectively called at startup and should be called when the state is changed. As startup, and whenever you call it yourself, `SetupBackgroundBuffer()` is called to draw the background image which `DrawScreen()` will copy onto the screen.

- If `Redraw(false)` was called then this function will undraw and redraw the moving objects.

  To do this it will call `DrawChanges()`, the basic implementation of which will call `UndrawChangingObjects()`, `DrawStrings()` then `DrawChangingObjects()`.

  Note: `Redraw(false)` is normally called when objects are moved from within the object's `DoUpdate()` function.

5.2 Telling SDL to redraw the screen

The reason why the `GetNextUpdateRect()` or `SetNextUpdateRect()` functions sometimes have to be called manually is discussed in this section.

One important aspect of SDL that has not been discussed is that changing the pixels of the memory structure which represents the screen is not actually sufficient to force the screen to be redrawn. The parts of the window are only redrawn under two circumstances: 1) When the operating system asks for it, or 2) when the program says it is necessary.

Typically, the operating system will ask the program to draw the window when it was not previously displayed. For example, if it un-minimises the window, or if another window which was on top or it is moved and the window for this program thus becomes visible. The operating system has no way of knowing that you moved something or changed something unless you tell it so.

SDL hides the redraw from you, and uses the array of pixels which it exposes to the program to do the drawing. This means that, if you change something in the array of pixels, you MUST tell SDL that you did so. The `SDL_UpdateRect()` function is used for this purpose, which `GameRender()` will call for you. When drawing the full window, after calling `SetupBackgroundBuffer()` and `DrawScreen()`, the framework will tell SDL to refresh the entire window. When drawing the moving objects, after asking the objects to undraw and redraw themselves the framework will ask the objects for a rectangle which encloses both the old and new draw positions. It will then tell SDL to redraw the sections of the window which are within these rectangles.

See also section 5.6.

5.3 The SetupBackgroundBuffer() function

`SetupBackgroundBuffer()` is a virtual function and is responsible for drawing the background to the background image. This should be overridden in the sub-class.

You should draw the background for your screen in here and it will be copied to the screen for you as needed.

This function should only set up the background image, rather than the screen itself, since anything you draw to the foreground will be overwritten as soon as the framework draws the background on top.

The `CopyAllBackgroundBuffer()` is later used to copy the entire background image onto the screen.

The `CopyBackgroundPixels( int iX, int iY, int iWidth, int iHeight )` function can be used later to copy a specific part of the background image onto the screen when necessary.

The `FillBackground()` function can be used to fill the entire background with a single colour.

The `SetBackgroundPixel( int iX, int iY, unsigned int uiColour )` function can be used to set the colour of a specific pixel in the background image.

There are many other functions for drawing triangles, rectangles, ovals and polygons to the background or foreground. You should look at the header file for the `BaseEngine` class.

5.4 The DrawScreen() function

`DrawScreen()` is a virtual function and is called by the `GameRender()` function when the entire screen should be drawn. This includes at the start of the game and when the game state is changed.

The basic implementation will first copy all of the background onto the screen, overwriting anything currently showing, by calling `CopyAllBackgroundBuffer()`. It will then call two virtual functions: `DrawStrings()` and `DrawChangingObjects()`. As long as you override `DrawStrings()` to draw any text you need then you will probably not need to override the `DrawScreen()` function.

5.5 The DrawChangingObjects() function

`DrawChangingObjects()` is a virtual function that is called to draw the moving objects on the screen. You should not need to change this function. This function will call `Draw()` on each object in turn, telling them to draw themselves.

This is called in two circumstances:

1. When the entire screen is redrawn. First the background is drawn, then the strings are drawn, then this function is called to draw the moving objects.
2. When drawing moving objects. First the moving objects are 'undrawn', then the strings are drawn, then the moving objects are redrawn.

5.6 The GetUpdateRectanglesForChangingObjects() function

`GetUpdateRectanglesForChangingObjects()` is a virtual function and is called by the framework in response to a call to the `Redraw(false)` function.

This function should record the positions of all moving objects so that the SDL functions can be told to redraw that part of the screen. To do this, the Game framework maintains an array of rectangles on the screen which have changed. In order to notify the framework of a change you should request one of these rectangles then fill in the details of the rectangle which has changed. The `GetNextUpdateRect()` function of `BaseEngine` has been provided to aid with this. Each call to `GetNextUpdateRect()` will return a pointer to an `SDL_Rect` structure. The framework will keep a count of the number which were requested and filled in and will ensure that the SDL functions are notified about these changes. This structure should be filled in with the details of the coordinates of the parts of the screen that should be refreshed. NOTE: You do not need to allocate or free any memory for this, you just fill in the details within the structure which is returned to you.

See also section 5.2.

6 Pixel Colours

Drawing things on the screen means setting the colour of a pixel. You will need to understand how to specify a pixel colour to be able to draw things.

Colours are specified using Red, Green and Blue values as a single number. A four-byte number is used, where one byte is used for each of red, green and blue and the final byte is unused. The reason that pixels are specified like this is because this is the way SDL was told to expose the screen to the user in this way. Other options are available but this is the easiest to understand.

Hexadecimal is extremely useful for specifying numbers which need to be separated into bytes, since each 2 hexadecimal digits represents one byte. This means that a colour can be specified as the number 0xRRGGBB, where each digit R, G or B is a hexadecimal digit from 0 to F. The amount of each colour is a number from 0 to 255, or 0x00 to 0xFF. You may find it useful to create constants for some of these values rather than putting a hard-coded value into the program.

The following table may give you more ideas:

Colour	Value for dark	Value for medium	Value for light
White	0xFFFFFF	0xFFFFFF	0xFFFFFF
Grey	0x404040	0x808080	0xB0B0B0
Black	0x000000	0x000000	0x000000
Red	0x800000	0xB00000	0xFF0000
Green	0x0080ff	0x00B000	0x00FF00
Blue	0x000080	0x0000B0	0x0000FF
Magenta	0x800080	0xB000B0	0xFF00FF
Cyan	0x008080	0x00B0B0	0x00FFFF
Yellow	0x808000	0xB0B000	0xFFFF00

You should also investigate the `GetColour()` method of the `BaseEngine` class, which will give you a colour value for a number of fixed colours. You may find this useful to keep consistent colour values. Alternatively, you could set up constants (or enums?) for colours you use a lot.

7 Handling player input

You have two choices for handling user input via key presses. Either you can handle the fact that a key was pressed or released as soon as the event is received (by implementing the virtual functions `KeyDown()` and `KeyUp()`), or you can test whether a key is pressed at the time you need to know (by calling the `IsKeyPressed()` function).

If you want to handle a key press inside a `DisplayableObject` (e.g. inside a pacman to decide which way to turn), then you probably want the `IsKeyPressed()` function. You could use this to determine whether to increase or decrease the speed and to determine whether the movement direction should be changed. i.e. the logic is to determine whether a key is pressed at the time rather than to catch the event that a key was pressed.

The big conceptual difference between the two functions is whether you want to force the user to have to press the key at the appropriate time, or whether you just want to check whether the key is pressed down at the appropriate time.

7.1 The KeyDown() function

`KeyDown()` is a virtual function and is called whenever a key event is picked up. Note that this is called from within the `MainLoop()` function, and the code for the key that was pressed is passed as a parameter to the function. The easiest usage of this function is to `switch` on the value of the key code and perform the appropriate operations to handle the key press.

You will want to implement this function in your sub-class if you want to handle key presses as soon as they are received by the application. It is assumed that the rest of the code should execute quickly enough that events are picked up almost as soon as they are posted. You should be aware that there may be a noticeable delay between pressing a key and it being handled if you do a lot of work within `GameAction()` or `GameRender()`.

7.2 The KeyUp() function

`KeyUp()` is a virtual function and is called whenever a key event is picked up. It is analogous to the `KeyDown()` function but is called for releasing a key rather than for pressing a key.

7.3 Knowing whether a key is pressed down

You do not have to handle the key pressed event as soon as it occurs. The `MainLoop()` function maintains an array specifying which keys are currently pressed. As soon as a key pressed event is handled (prior to calling `KeyDown()`), the relevant array entry is set, and as soon as a key released event is handled (prior to calling `KeyUp()`) the array entry is cleared.

It is possible to check the key pressed array to determine whether a key is pressed at that time. You can call the `IsKeyPressed()` event to determine whether a specific key is currently pressed.

7.4 Handling mouse presses

For the mouse interaction you have no option but to handle the event as it happens. There are three functions which you may override in your sub-class:

`MouseMoved( int iX, int iY )`: This will be called when the mouse is moved and will tell you the new X and Y position of the mouse on the screen.

`MouseDown( int iButton, int iX, int iY )`: This will be called when a mouse button is pressed and will tell you the button which was pressed and the position on the screen at which it was pressed. Note that you still get `MouseMoved()` events while the button is pressed so you could handle dragging if you wished.

`MouseUp( int iButton, int iX, int iY )`: This will be called when a mouse button is released and will tell you the button which was released and the position on the screen at which it was pressed.

You do not need to handle mouse input when it happens. The framework will cache the latest values for you. This means that you can call a number of functions to ask for the current (last recorded) position of the mouse if that is easier. You should investigate the following functions: `GetMouseXClickedDown()`, `GetMouseYClickedDown()`,`GetMouseXClickedUp()`,`GetMouseYClickedUp()`,`GetCurrentMouseX()`, `GetCurrentMouseY()`,`GetCurrentButtonStates()`.

8 Other useful functions of the BaseEngine

The BaseEngine class is responsible for maintaining the state of the program and for drawing the background of the program. Drawing the moving objects is delegated to the moving object classes themselves.

8.1 int GetTime()

This is a function which will return an increasing time value. You can use it to work out how often to do something, or to calculate where something should be drawn if you know when and where it started and when and where it should end its movement.

This function could be used to determine which frame of an animation to display, or whether it is time to perform some action yet.

For example, the following code will generate a value to cycle from 0 to 5, with an increase every 5 ticks of the `GetTime()` function. Thus, by calculating the frame number to use every time an item must be drawn, and drawing the frame that is specified by the calculated number, the item will appear to be animated. The example coursework answer uses this to work out the radius of the chasers, so that they appear to grow and shrink.

```
int iTicksPerFrame = 4; // Number of ticks per frame
int iNumberFrames = 6; // Frames will cycle from 0 to 5 then back to 0
int iFrame = (GetTime()/iTicksPerFrame)%iNumberFrames;
```

8.2 void SetTimeToAct(int iDelay)

A function which will implement a delay. Specify the required delay before the next action (i.e. the frequency with which your program should act) and pass it to this function, then call TimeToAct() to determine whether it is time yet. `TimeToAct()` will not return `true` until this delay has expired.

8.3 bool TimeToAct()

A small function which will return true if it is time to act now, or false otherwise.

8.4 void Redraw(bool bAllScreen)

This is used to indicate that either the screen or just specific regions of it must be redrawn. Calling this will change the behaviour of the `GameRender` function.

If you specify a parameter of `true` then the `GameRender` function will request the background image for the entire window to be updated and will then copy the image over onto the screen. It will call the `DrawScreen()` function to do this.

If you specify a parameter of `false` then the `GameRender` function will just attempt to undraw and redraw the moving objects.

See also sections 4, 5, and especially section 5.1.

You can use the functions `RedrawObjects()` or `RedrawWholeScreen()` on the `DisplayableObject` class instead if you prefer. They do the same thing.

8.5 void SetExitWithCode(int iCode)

Should be called with an exit code of 0 or greater! Set an exit code and terminate the program. When set this will make the various game loops exit, which will effectively end the program!

8.6 int GetScreenWidth()

Return the screen/window width, in pixels.

8.6.1 int GetScreenHeight()

Return the screen/window height, in pixels.

8.7 void SafeSetScreenPixel(int iX, int iY, unsigned int uiColour)

This function will set the colour of a pixel on the screen (i.e. in the window, not in the background). You can use it to draw moving objects. Since the background image is unaffected, the object can be removed by re-drawing the background image.

This function is exactly the same as `SetScreenPixel()` except that it checks that the pixel coordinates which you specify are within the screen area. Since these checks take time, you should instead use the `SetScreenPixel()` function if you are sure that the coordinates are valid.

8.8 void SetScreenPixel(int iX, int iY, unsigned int uiColour)

This function is exactly the same as `SafeSetScreenPixel()` except that it does not validate the supplied coordinates. This function should be much faster because of the lack of checking, but is more dangerous since it potentially allows you to overwrite arbitrary memory by supplying bad coordinates, and thus crash or corrupt your program.

8.9 void CopyAllBackgroundBuffer()

Copy all of the background buffer onto the screen (foreground), filling the display window with the background image.

8.10 void CopyBackgroundPixels(int iX, int iY, int iWidth, int iHeight)

Copy the background pixels in the specified region onto the screen, overwriting what is within the specified window. The x and y coordinates of the top left of the rectangle need to be specified, along with the width and height of the rectangle.

8.11 void DrawScreenString(int iX, int iY, const char* pText, unsigned int uiColour, Font* pFont)

Use this to draw a string to the foreground. You should specify the pixel coordinates of the top left corner of the string, the text to draw and the colour to draw it. The string will be drawn to the screen NOT the

background. This means that any refresh of the background will remove the string. Note that you also need to tell SDL that the area of the screen has changed.

8.12 GetNextUpdateRect(), SetNextUpdateRect()

When a part of the window contents are changed, it is important to tell SDL about this. In order to notify the framework of a change you should request one of these rectangles then fill in the details of the rectangle which has changed. See section 5.6 for more information. The `GetNextUpdateRect()` function will get you a pointer to a struct to fill in. The `SetNextUpdateRect()` will get the struct and fill it in for you using the values you provide, so you don't have to handle the struct yourself.

8.13 SetSDLUpdateRectImmediately(int iX, int iY, int iWidth, int iHeight)

This can be used to tell SDL manually that a specific rectangle on the screen should be redrawn. Using this should be unnecessary for redrawing moving objects since you can use `GetNextUpdateRect()` instead, however you will need to call it if you draw strings outside of a `DrawScreen()` call. See `DrawString()` above.

9 The DisplayableObject class important functions

9.1 void Draw()

Draw the object. The override of this function should implement the actual drawing of the correct object.

9.2 void DoUpdate(int iCurrentTime)

Handle the update action, moving the object and/or handling any game logic. For example, for the player this should handle input and set a new movement location. For any enemies, this would perhaps make some kind of random or pre-determined move.

If the object is moved (defined as either the old or new screen coordinates having changed) then the `Redraw(false)` function must be called to force a redraw of the screen so that the movement appears.