TDE Game Engine

Technical Specification

http://student.computing.dcu.ie/blogs/donneln7/

Neil Donnelly (10108823)
5/27/2013

Table of Contents

1. Overview

The goal for this project was the design, development and use of a 2D game engine. It was built with C++ through Microsoft's Visual Studio 2012. A game engine is a software framework designed to help developers create and run games. This engine is aimed to specifically aid in the development of 2D games. The engine developed has been named the TDE engine and gives users access to functions relating to graphics, audio, user input and directing the flow of control in the game with widgets. The engine is targeted towards software engineers interested in the development of 2D games which have seen a dramatic rise in popularity with the rise of smartphones and tablets.

So why would one want to use an engine? Although most games are quite different, they all boil down to need the same basic functions. These are the ability to load in images, apply and draw textures created from these images, use user input to control the game and load and play various audio in response to user interaction. An engine isolates these functions and attempts to provide a usable framework that is both easy to use and understand but also efficient and robust. The TDE engine provides some added functionality to this end including support for texture atlases, shape drawing, two forms of audio to use, font support, a timer and a random number generator.

The main challenge when building such a framework is ensuring the engine is both flexible and robust. As one of the main goals for an engine is for it to be suitable to develop multiple games of various types, the engine needs to cater for a multitude of different techniques and functions while maintaining a standard of efficiency. As well as this, as games are so heavily dependent on the engine to run, it is expected that the engine is both durable to high number of operations but also incredibly reliable and failure resistant. An engine that can fail under even a small list of possibilities will not be of much use to a game developer trying to sell their wares. Both of these features can only be achieved through thorough designing and rigorous testing.

The TDE engine went through several design iterations in order to try and achieve these maxims. As well as that, the testing of the engine was quite extensive for the individual components of the engine, although perhaps not as extensive a testing phase as would be required before full deployment onto a public market. In order to demonstrate the flexibility

and robustness of the TDE engine, two games were developed to run on the engine. The two games are quite different and yet both run equally smooth.

2. Glossary

Animation Cell

This is a single frame or image of an animation made up of multiple images cycled iteratively.

Clipping

The technique used to any object or object segment that exists outside of a designated region in the world.

Polygon

This is a shape of more than one edge. Each line of the shape is connected to another line on either side to from a closed path.

OpenGL

The OpenGL library is a 2D and 3D graphics API that works on a variety of platforms. It has become one of the most widely used graphical API's in use for a multitude of purposes, such as in games and graphic design.

SDL

Simple DirectMedia Layer is a library designed to give access to a machine's audio, input, and video on multiple platforms. It is written in C, but provides native support for C++.

Texture

A texture is a digital representation of an image that be laid upon a 2D or 3D object in the world.

Texture Atlas

A texture atlas is a selection of images grouped together to form a single composite image. This allows for the use of numerous textures without the need to swap different textures to and from memory.

Widget

This is a UI component that displays information which can also be interacted with by the user. As the GUI in a game is almost entirely interactive, widgets usually make up the entire UI system. These widgets can be broken up into separate widgets, each displaying separate information at the same time.

3. Motivation

Currently there has been a revolution of 2D games in the games industry due to the recent emergence of mobile devices such as smartphones and tablets. With this rise in popularity there has been a desire for a quick and easy way to prototype and create 2D games on both these devices and on PC. However, as there is a vast number of products that help fill this void, this has led to many working on games to miss out on the lower level structures that power their games. This understanding of how an engine works can help aid in the implementation, optimization and debugging of the game. This understanding was one of the main motivations behind the goal of this project.

The challenge of building such a system was also an intriguing prospect that helped to make the decision to adopt this project. There are many elements to a graphics engine and these must work simultaneously together in an efficient manner in order for the game to run smoothly. The variety of different media to work with and the effort to combine all this work to create this system was enticing.

4. Research

Before even preparing a proposal for the project, there was a great deal of research made to discover the feasibility of completing an engine in the time available as well as the necessary components required for a complete 2D game engine. There are innumerable sources concerning the construction of a game engine with many different recommendations for choice of language, graphics API and platform. Based on the value of experience, C++ and OpenGL were chosen as the language and graphics API respectively. It was also decided that the target platform would be the PC, but as OpenGL was chosen, it lent itself to being easily ported to multiple other platforms.

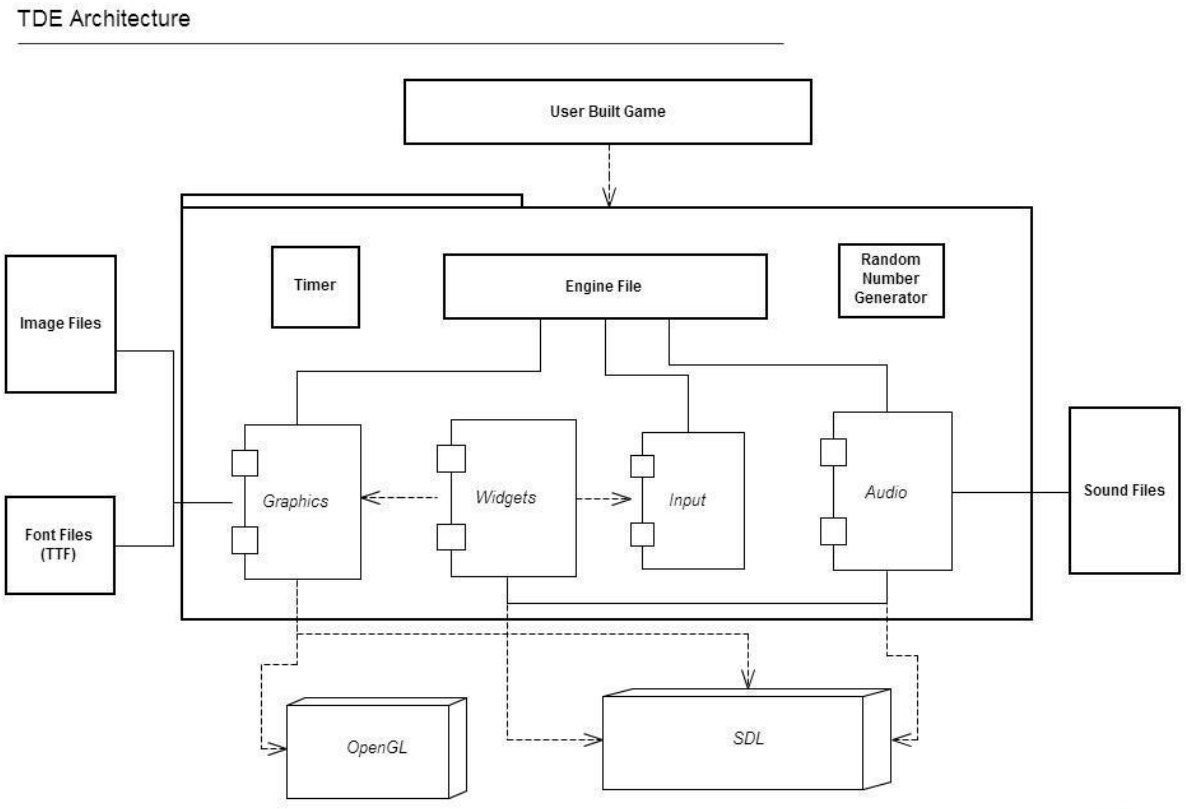
The main source used when designing the engine was the 4th edition of Game Coding Complete, written by Mike Shaffry and David Graham. This book contained guidelines and tips for how to structure and implement a 3D graphics engine for high end games using the DirectX API. Although meant for building a 3D graphics engine and using a different API, it held invaluable information on coding procedures to adopt and possible approaches to breaking up the graphics component of the engine as well as the user interaction components.

For the use of OpenGL with C++, both the OpenGL SuperBible (Haemal, et al.) and the online set of tutorials at nehe.gamedev.net were useful for both the implementation techniques and the broader understanding of how OpenGL operates with the hardware available. In order to implement the audio, input and elements of the graphics components, the online SDL APIs and help sections were quite invaluable, as the sample code illustrated good practices and how best to implement the functions for the necessary tasks.

5. High Level Design

The basic structure of the engine can be broken into four main areas of concern, these are: Audio, Graphics, Input and Widgets (Output). Then with these four large areas, other smaller features such as the random number generator and timer were also added. The graphics, audio and the input system can all exist independently and rely on nothing but the underlying engines. The widget system however, is highly dependent on both the graphics and input sections. The input system drives the power of the widget and forms the basis for how the player interacts with a game. The widgets depend on the graphics engine so it can present a graphics representation of itself on the screen for the player to interact with.

5.1 Architecture Diagram



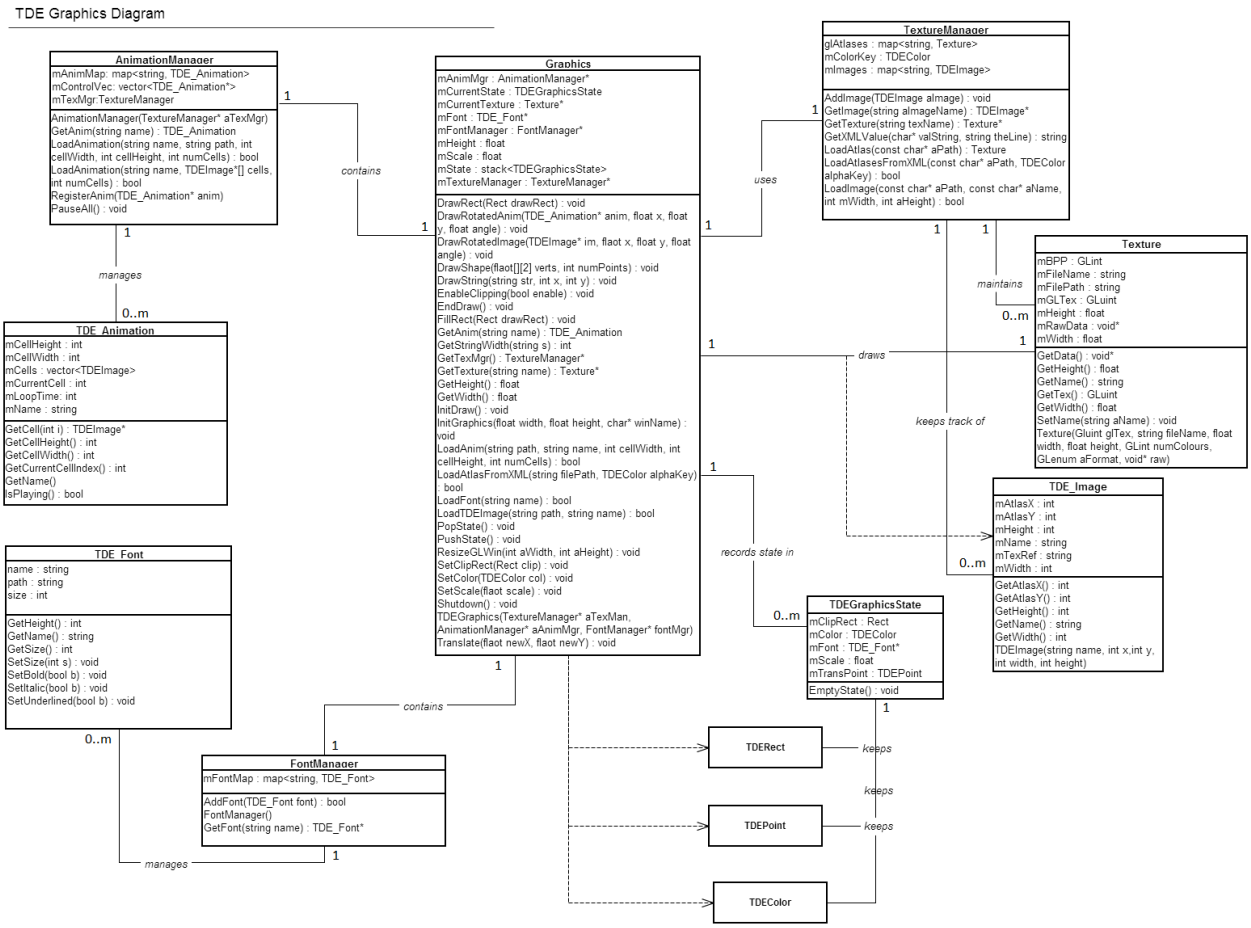
As can be seen in the above diagram, the engine is separated into 4 major components. Each of which is used by the Engine file to power the game. As well as these four most important components, there is also a Timer and Random Number Generator. The timer acts as a helper class for the user to simply incorporate time sensitive actions into their game. The timer is accurate to milliseconds and is based off the SDL tick system but with functions allowing the user to pause, resume and reset the timer whenever needed. The

random number generator is a similar helper class for the user to generate pseudo random integers and floating points. The specifics of the system are discussed in the implementation section of this document (6.1 Implementation Strategy).

The graphics component of the engine interacts with both the OpenGL and SDL library to implement its functions. These are used for loading and drawing images, animations, shapes and text. The TTF files are TrueType Font files and the engine can load these and use them as the font when drawing text. The user can point to any TTF file to be used. The Graphics portion of the engine can also load image files of a variety of formats provided by the user.

The Audio component uses music files of a variety of formats for producing the audio also. The Audio uses the SDL_mixer library in order to complete this task. The input system relies on SDL for catching user input and passing the input to the input system so that it can pass the input to the widgets. The widget component has no external dependencies but relies on the input and graphics component in order to operate correctly. The engine file possesses a manager object of each component which it controls and then passes to the widget system so that the widgets may utilize their functions.

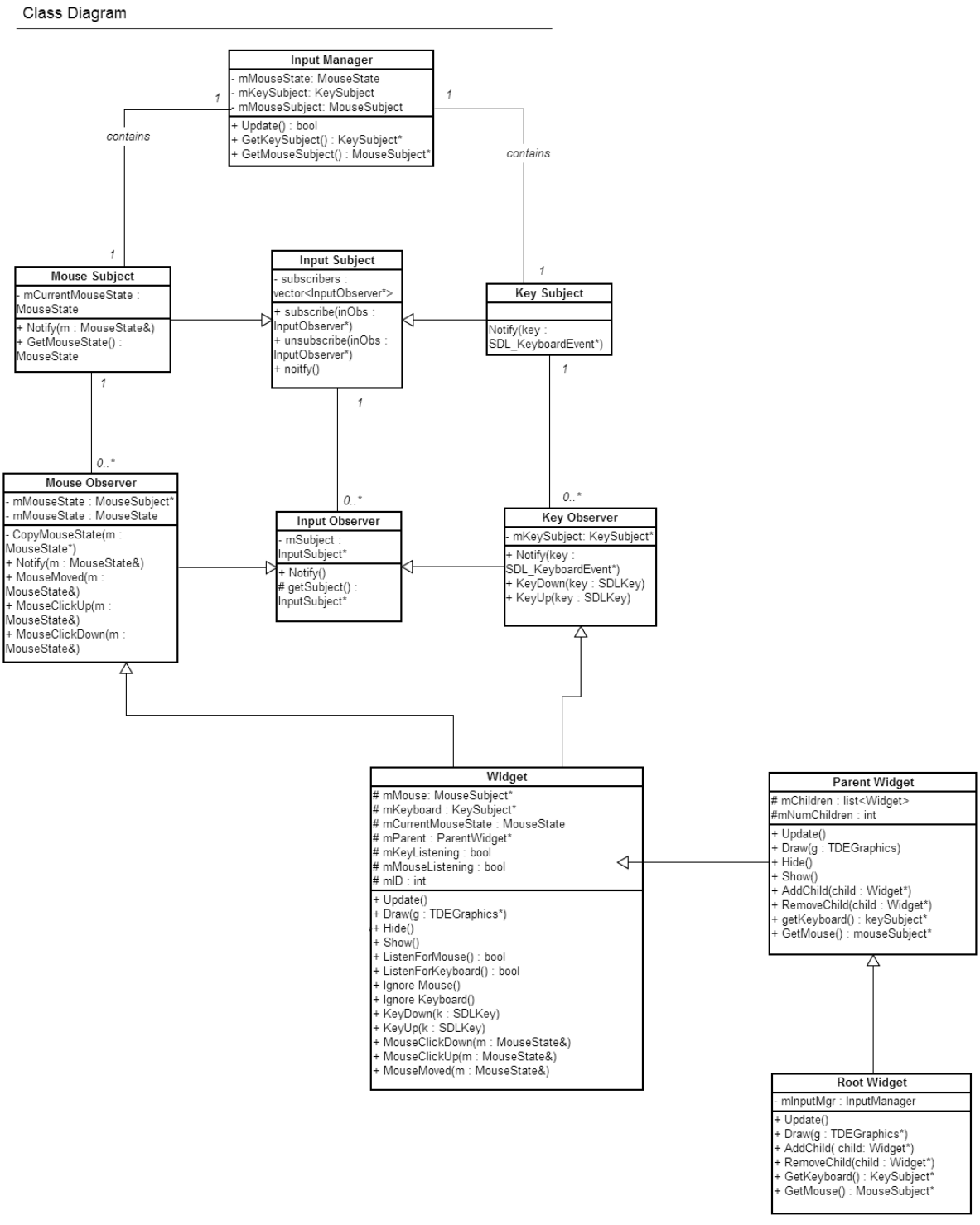
5.2 Graphics Design



1 Contains subset of functions due to unwieldy number of functions present in class

The Graphics class is the point of access for the user; it provides all the functions needed for drawing things to the screen. It contains a pointer to the animation, font and texture managers which deal with loading, storing and retrieving the object used. Each manager holds a map of their respective objects which use a string as the key. Once the user knows the name of the image, animation or font they wish to use they can use it to acquire a reference to the object itself. The texture manager contains two maps, one for images and for textures. All the images contain a string reference to the texture that contains their pixel actual pixel information. The image class simply holds this reference and the dimensions of the image while the texture is what is actually drawn to the screen. There also three simple structures that are used by the engine to simplify information and improve clarity. These are TDERect, TDEPoint and TDEColour, which can also be used by the user.

5.3 Widget and Input Design

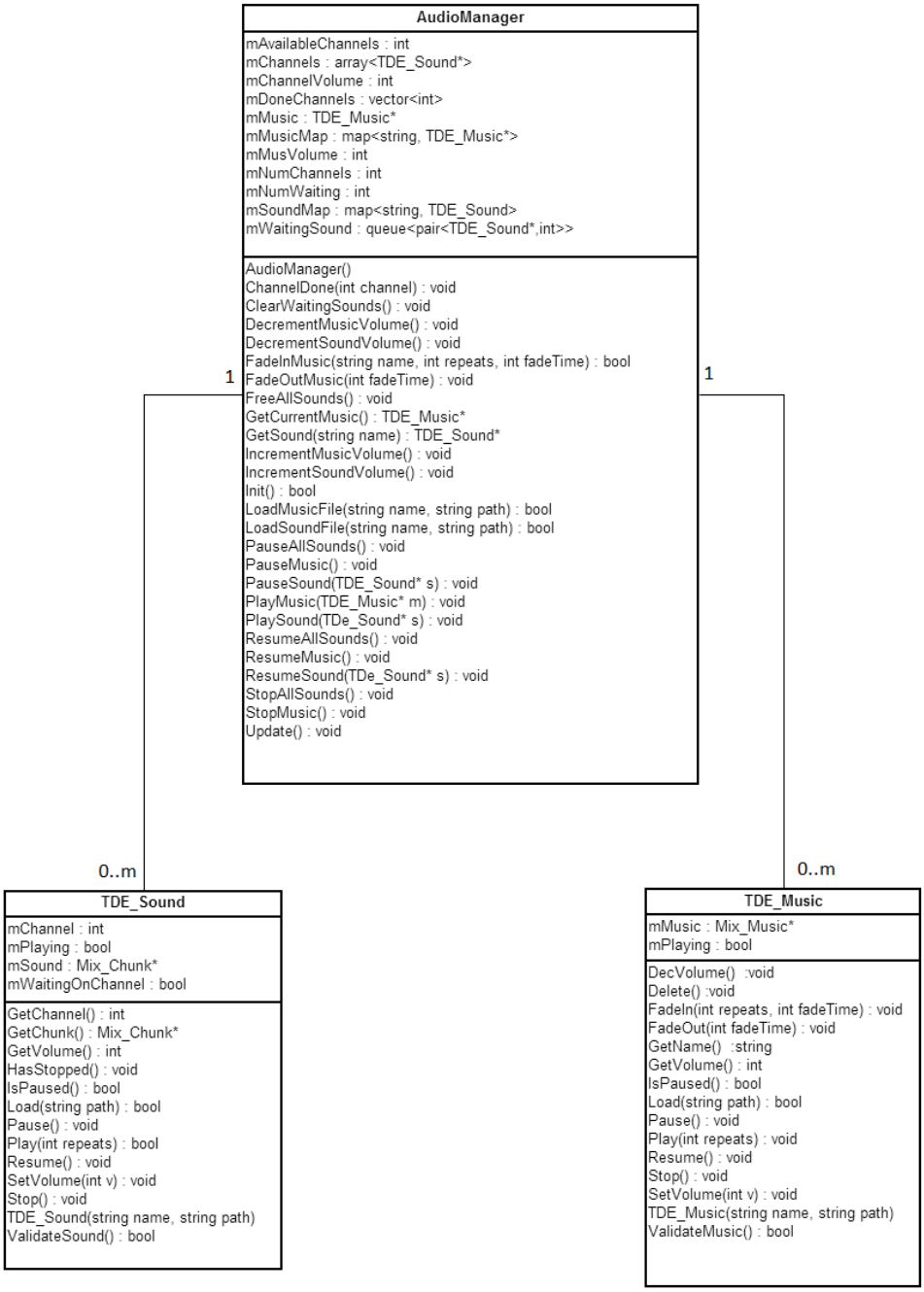


The observer patter was used when implementing the Widget and Input system. The input manager creates a subject for keyboard and mouse which are given the updates from the input. Key observers and mouse observers can then subscribe to these subjects in order to

be notified of any updates. The Widget class extends each of these observers so any widget can subscribe and receive input from the subjects.

The widgets form a hierarchical tree structure. The basic Widget forms a leaf on the tree while the Parent Widget is a branch on the tree that can have multiple leaves or branches stemming from it. The Root Widget is an extension of the Parent Widget that forms the root of the hierarchy. The Root widget is given references to the input manager, audio manager and graphics so that it can pass these to its children when needed. In this way, the use of the individual utilities can be passed down one by one to each widget in depth-first fashion. This system works quite well for graphics as the latest drawing calls are placed above earlier requests, therefore the children can be drawn above their parent as would be expected from a widget system.

5.4 Audio Design



The audio system is quite a simple design. The manager loads, stores and controls all the audio for the game. These come in two varieties, Sound and Music. Music is primarily meant for background music, or audio that is intended to span a large amount of time (60+ second). Sounds, on the other hand, are for audio that takes up very little time, usually used for sound effects or short dialogue. Only one piece of music can be played at a time, but multiple sounds can be played at once over channels controlled by the audio manager. The audio manager can control the volume and state of all the music and sounds.

6. Implementation

6.1 Implementation Strategy

The engine was built using a bottom-up implementation mentality with emphasis on prioritising the most critical elements of the engine. The system was the broken up into individual components with each being implemented separately and then integrated piece by piece.

The graphics component of the system was separated into four distinct areas. These sections were texture loading and drawing; shape drawing and graphics state management (clipping, scaling, etc.); font loading and text drawing; animation creation and playing. These were not all built consecutively. The animation section was developed near the end of the development cycle as it was deemed less critical than the other sections.

The widgets and the input were built almost simultaneously as they were built making the use of the observer pattern. In this pattern, the input manager (subject) contains information, that when updated needs to notify any widgets with registered interest (observers). Utilising this pattern allowed for quite an elegant and abstract approach to allowing each widget to decide whether it wanted to receive updates on the input as well as what variety, either keyboard or mouse. As the two sections were forming either side of a design pattern, the skeleton of the system with basic implementations of each side were built simultaneously, then once the design pattern was finished, each section could be fleshed out with their full implementation and functionality.

Although the mentality for developing the system was to concentrate on the most critical aspects of the engine first, on some occasions logical practicalities out-favoured this method. Both the Timer and Random Number Generator were created ahead of schedule as whilst testing and implementing other features the need for a timer and random numbers became apparent.

The random number generator algorithm implemented for the engine was the Merseene Twister algorithm. This pseudo-random number generator was originally developed in 1998 by Makuto Matsumoto and Takuji Nishimura. The algorithm is based on recurring relations

between certain matrices over a finite binary range. The code used in the projects implementation is derived from the Game Coding Complete book.

6.2 Problems and Resolutions

6.2.1 Drawing Text

The files used for supporting various fonts were TTF (True Type Fonts) files which are loaded by the SDL library SDL_ttf. However, in order for OpenGL to draw this text to the screen, they needed to be of a type that the library could work with. To get around this, the same technique for converting SDL images to OpenGL textures was used. When the function for drawing a string is called, it builds an image of the string with the font on a SDL image. When this is done, the image is iterated over pixel by pixel and converted to an OpenGL texture. Each pixel that did not contain any necessary information on the text image was made completely transparent. This image was then drawn as a rectangle at the designated area by OpenGL.

6.2.2 Control of Animation

One of the features of the engine in terms of the animation system is the ability to control all the animations in one function call. However, this caused some problems with how the animations were originally being stored. When first implemented, the Animation Manager kept all the animations in a single map, when an object wished to use an animation, they were returned a pointer to the original animation. But of course, if an object or multiple objects wished to draw several of these animations as will likely happen for certain animations, they were all the same animation object. So if one animation was paused or reset, they were all paused or reset.

In order to fix this, instead of returning a pointer to the animation object, a copy of that object was returned instead. Once this was done though, the animation manager lost all control of the animations. This instead was turned into a feature. If an object wished for an animation to exist beyond the control of the animation manager, all it needed to do was retrieve the animation object it wanted. If an object wished to have the animation controlled by the manager then it would simply subscribe that animation. Once subscribed, any action the manager took on its list of controlled animations would affect the subscribed animation.

6.2.3 Sound Queuing

One of the more difficult problems encountered during the process of implementing the audio was the issue of making sound channels available once the sounds playing on them were complete. This was an important feature as the goal was to allow a developer to be able to find out what channel each sound was playing on. This proved troublesome as the SDL preferred to clean the channels itself and trying to intervene with this process often leads to faults occurring as a sound finishes. To combat this, a small class to handle what happens when a sound finishes with a channel was constructed. SDL was then given a pointer to a member function of this helper class. When called, it would receive the index of the channel that was finished. This number was then placed on an array. Then each time the audio manager was updated, it would iterate over this array of finished channels and update the song object with the fact that it was finished. It would then place any songs that happened to be on the waiting queue onto the newly freed channel before continuing. This allowed the developer to always stay abreast of what songs were currently being played and when a song was finished.

6.2.4 Widgets and Input

As described in the implementation section of this document, the observer patter was adopted when implementing the Widget and Input systems of the TDE engine. This was actually in response to a problem that was faced early in the original implementation of the widgets. The problem occurred when trying to discover a technique of passing input information to widgets that were interested in that particular type. The plan at first was to have the root widget receive any input from the input manager and have it trickle down the tree so each widget could decide for itself if it was interested. This proved an unwise approach as for many widgets this information was not wanted. Even if the widget did want input updates, it may not have been the input type it was looking for.

To tackle this problem, the observer pattern was applied which allowed each widget to register its own interest in the input. This fit well into the design pattern as every widget would be at the very least, a subclass of the basic widget class. Therefore, without the need for parents or children to be registered as well, each and every widget could decide independently if it required input and the input subjects would deliver this information to each interested party individually.

7. Testing and Validation

7.1 Testing Strategy

With such a large inter-reliant framework, testing was a quite important aspect of ensuring the success of the engine. As the engine is a framework rather than a full application, it can be difficult to detect problems later on in the development cycle without it becoming too late to effectively fix the issue. Even the magnitude of small bugs or rare faults can be magnified greatly when the problem is only detected at user testing and validation as the problem can be costly to find and cause a lot of problems to fix. With this is in mind, testing began quite early on in the process with test cases and reports being drafted and filled in as each unit was being completed.

Unit tests were performed on some of the basic objects of the engine such as the classes for TDE_Image, TDE_Animation, TDE_Sound and so on. Then once these had been tested and integrated with their respective managing classes, integration testing was performed to ensure that the objects and their managers were working together as planned. The integration testing was the last round of official testing that could be completed. Each component of the engine was tested using test drivers. These test drivers ran the functions multiple times with different arguments for each and wrote to standard output the result. The process of writing drivers was made immensely easier using the widgets themselves as drivers. A single program was written that would run a widget and so for each component, a widget could be created to test the functions and plugged in to the program. Although not implemented completely, it also allows for the potential to have a testing suite that can change which widget to work on dynamically at run time.

The main focus for the test cases was testing the boundaries of viable values and null values. With the number of pointers being passed from one section to another, null checks became incredibly important as any null pointer error leads to a full system crash so every function which has a pointer parameter required a null check and so testing had to be sure this was present and working correctly.

With the sheer number of functions to be tested, System testing became unfeasible. Instead, the validation process of developing two games, were merged with the system tests. Smoke tests were completed to check the most important aspects of the system and

see that they could work together without issue, for instance ensuring the use of animations and music together would not cause any problems for either. Thus the work on the first game with the engine became an unofficial system test of the engine, the second game developed then acted more as what the initial validation plan was to be.

7.2 Test Cases and Reports

The following is a sample of the test cases that were filled out when completing the unit testing. Each test case was a test on a function, often a function would have multiple test case associated with it. Each test case followed the same template: name of the file holding the function; the date on which the test was done; the declaration of the header; the input provided for the test; the expected output; the actual output and the name of the test report if the test failed. The resolutions were not worked on as soon as the problem was found, instead placed on a list of items to be fixed. Once a resolution was in place, the item would be regression tested as well as any influenced items; if it failed a new test report was created. The following is all the test cases run on the function to load a single image from a directory pointed to by a string:

File	TextureManager
Date	14/11/2012
Function	LoadImage(char* path, char* name)
Input	LoadImage(“”, name)
Expected Output	Nothing is loaded
Status	Passed

File	TextureManager
Date	14/11/2012
Function	LoadImage(char* path, char* name)
Input	LoadImage(path, “”)
Expected Output	Nothing is loaded
Status	Loaded the image without a valid name provided. IT001

File	TextureManager
Date	14/11/2012
Function	LoadImage(char* path, char* name)
Input	LoadImage(path, “Name”); LoadImage(path, “Name”);
Expected Output	Only first image is loaded, a name shouldn’t be able to be reentered
Status	Loaded the image despite the name already existing. IT002

File	TextureManager
Date	14/11/2012
Function	LoadImage(char* path, char* name, int width, int height)
Input	LoadImage(path, name, 250, 250)
Expected Output	Image is loaded that has the height and width of 250 and name given
Status	Passed

File	TextureManager
Date	14/11/2012
Function	LoadImage(char* path, char* name, int width, int height)
Input	LoadImage(path, name, 0, 0)
Expected Output	No image is loaded
Status	Added image with invalid width and height. IT003

File	TextureManager
Date	14/11/2012
Function	LoadImage(char* path, char* name, int width, int height)
Input	LoadImage(path, name, INT_MAX, INT_MAX)
Expected Output	Image loaded but only width actual image dimensions, no attempt to use values beyond image bounds
Status	Created image with invalid width. IT003

Each test case and report was given a unique identifier. If a test case failed, it referenced the test report filed that contained the status of the bug and how/if the problems was fixed. The following is a sample test report (IT003) referenced by two of the test cases previously.

Testing ID
IT003
File
TextureManager
Date
26/10/13
Function
LoadImage(string path, string name, int width, int height);
Input
LoadImage(path, aName, 0, 0) LoadImage(path, aName2, INT_MAX, INT_MAX)

Expected Output
Neither image is loaded due to invalid dimensions
Resultant Output
Both created
Status
Fixed
Fix
Added a check to ensure width and height are above zero but equal to or less than total width and height of full image. If exceeds width and height, the image is loaded with the width and height of the actual image

A record of the test cases and reports made during the unit and integration testing can be found at the following DCU student site:

http://student.computing.dcu.ie/~donneln7/fyp/testing/testinghome.html

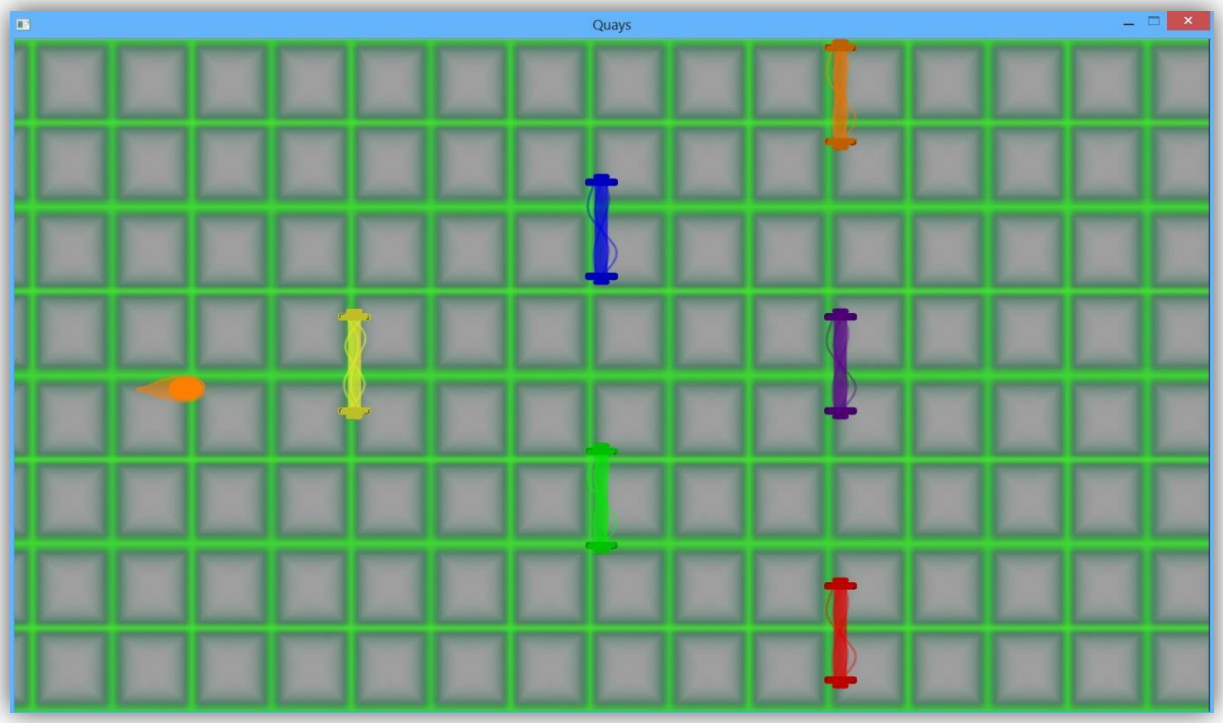
7.3 Validation

For the purpose of validating the purpose of the engine, two games were developed that would use the TDE engine. The reason for building two was to demonstrate the versatility and flexibility of the engine. The main purpose of an engine is that it should facilitate the development of multiple games rather than just one; therefore two very different 2D games were created. Both games had very short development times due to time constraints but still managed to convey the purpose and major design ideas behind the games, which is the whole purpose of the prototype.

7.3.1 Quays

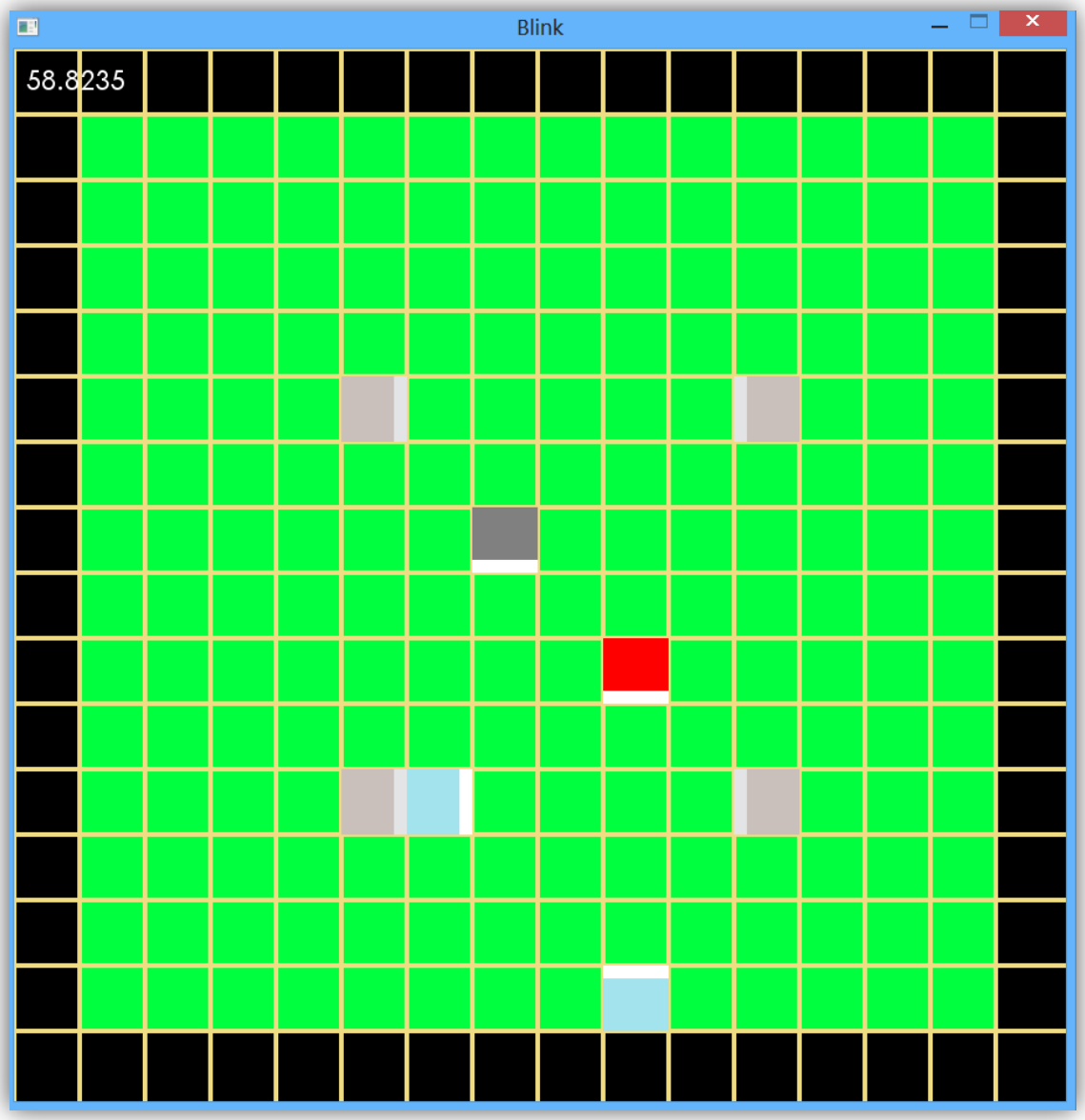
Quays was the first game developed for the engine and as such was used to validate the potential of using the engine for prototyping a game. The game is a 2D side-scrolling game in which the player controls a photon travelling across the screen. The basis of the game is to try and match the colour of the locked door at the end of the level by passing through light gates which change the colour of the photon. Ignoring the horrendous programmer art and incomplete gameplay, the game itself ran smoothly and required very little

development time to have running when compared to the time perhaps needed without an engine.



7.3.2 Blink

The second game developed for the TDE engine was designed to be quite different from the first game to showcase the engine's versatility. It is a top-down puzzle game in which the player controls a character moving around a 14x14 grid. The grid is also populated by enemies that if they get too close will kill the main character and end in game over. However, while the main character or even another enemy is looking at them, the enemy cannot move. The grid also contains four mirrors, which can be rotated around and if an enemy sees itself in the mirror, it will freeze. The goal of the game is to try and freeze all the enemies of the game. The following is a screenshot of the game without the proper art assets. The game works as it should, but no textures have been applied as they are still waiting completion:


Blink
58.8235

8. Backup and Recovery Strategy

There were several techniques used in order to ensure that the project was backed up safely and could be recovered quickly if needed. In fact this strategy was needed midway through the development process following a system failure on the machine used to develop the project.

The main utility used for backing up the system was GitHub. This is an online web based service offering version control on software projects. The project is kept on one of their servers and can keep a track of updates and offers the ability to rollback to previous versions of the project if something were to go awry. This allowed for the latest version of the project to be recovered following a machine malfunction that wiped the local hard drive on the main development machine. In order to keep a project private, GitHub usually requires a subscription but by proving it was for the use of a college project, a free subscription was awarded allowing the project to maintain privacy.

As well as the use of GitHub, the project was also kept on an external hard drive which was update after every two weeks with the latest documentation and project files. For the purpose of keeping testing files backed up and any other relevant documentation, Dropbox was used. This is a cloud based service that offers space on it servers for users. Using all these mechanisms, all data relevant to the project was kept backed up regularly and was easily recovered from remote servers. If these servers were unavailable for some reason, a semi-recent version was kept on an external hard drive.

9. Results

9.1 Project Success

With this project now complete, the engine has met its list of requirements and can accomplish all functionality it was intended for and more. The engine has the capability to load images of various different formats and draw these images to the screen in a multitude of ways. Lines, rectangles and any user defined shape can be drawn to the screen. Clipping scaling and translation can be applied as well as the ability to apply any RGB colour to shape, object or text on the screen. Text can be drawn to the screen using any font pointed to by the user and with any style or size. Animations can be created from unloaded and pre-loaded images, they can be played at any time interval and controlled with as much control as one can expect from a 2D engine.

The engine has a fully implemented widget and input system that allows versatility and flexibility in the development of prototype games using the engine. Any keyboard or mouse input can be collected, and widgets of any size, shape or design can be created and applied one on top of the other with control over each. The audio manager of the engine allows for the loading of music and sound of a multitude of different formats. Music can be played, stopped, swapped and paused and even faded. Sounds can played and controlled individually as well as collectively.

Two fully operational games have been developed for and run on the engine. Although requiring larger, more complex games to fully test the robustness and capability of the engine, it has passed the necessary requirement for the project's deadline of being able to handle this task. Still, the engine performed admirably and the games run smoothly. Not only this, but also the quick development time for each game is a testament to the functionality of the engine. The speed at which prototypes can be put together is quite an important trait for an engine so it was pleasing to see the games being developed so quickly with largely vary little issue.

9.2 Lessons Learned

Throughout the development process of the engine, there have been many priceless lessons that have been learned. For one, the value of spending time on creating a thorough and accurate design and strategy has been instrumental to the successful completion of the

project. The design has not changed dramatically since the completion of the functional specification. With such a complex system of multiple components, the design created early on helped form the classes in a manner that made inter-class communication as easy as possible. It also had a positive effect on the resulting engine on the behalf of the user. As the design was clear, the classes were written with a clear understanding of what functions were needed to be made available and how they would affect the other classes.

Another important lesson is the importance of continuous and early testing. With such a large system and with so many inter-dependencies a bug in one class can cause major problems for other classes that may rely on it. This is particularly important for a system such as an engine or framework where others software relies heavily on the robustness of the underlying software. A single small discretion buried deep in the engine can cause major issues on a game that relies on the engine behaving consistently. As such, testing should be done as early and thoroughly as possible so a great focus on unit testing was made as each component was being completed.

10. Future Work

There are many things that can be accomplished in expanding and improving the TDE Engine. For instance, as the engine was built using OpenGL and SDL (both very portable libraries) it is possible to port the engine to iOS and Android quite comfortably. The engine does not rely too heavily on any system calls that would not be available through the OpenGL ES library (a version of OpenGL designed for mobile devices) and so the engine could be made multi-platform. As the current trend in the game industry is for 2D games to be more successful on smartphones and tablets, porting the engine could be a valuable extension to the engine.

The engine could also be expanded to accommodate the development of 3D games. Much of what is already build on the engine will not be required to change much, except the Graphics component which will merely need to be expanded. There are a few new techniques that would need to be included, for instance the use of a camera, the clipping techniques available, 3D object loaders, a lighting and shading model and a perspection system. The mouse input would also need to be changed slightly to accommodate the potential to click on distinct 3D objects and trace its screen position onto an object in the 3D virtual world. This would take quite a bit of research and development time but still quite possible.

During the process of implementation, a week was taken out of the schedule in order to investigate the potential to implement a physics system into the engine with virtual gravity and a collision detection mechanism. However, although some headway was made into the collision detection implementation, the further development of a physics component to the engine was scrapped due to time constraints but the potential to reignite this endeavour could also be a potential area for expansion of the engine.

As there was not enough time to build a game of any great complexity and size, the engine has of yet, not really been truly pushed in any way to test any potential performance loss. As such, to improve the engine on a whole, it may be beneficial in the future to run more in depth performance tests on the engine and test the limits of what the engine can really handle.

11. Resources and References

OpenGL website - http://www.opengl.org/

SDL website - http://www.libsdl.org/

SDL wiki - http://wiki.libsdl.org/moin.cgi/Tutorials

Graphics tutorial for OpenGL and SDL - http://nehe.gamedev.net/

Online tutorial for collision detection - http://www.metanetsoftware.com/technique/tutorialA.html

GitHub - https://github.com/

Dropbox - https://www.dropbox.com/

Mike McShaffry and David 'Rez' Graham, 2012. *Game Coding Complete Fourth Edition.* Course Technology PTR.

Richard S. Wright, Nicholas Haemel, Graham Sellers and Benjamin Lipchak, 2010. *OpenGL SuperBible: Comprehensive Tutorial and Reference (5th Edition).* Boston: Addison-Wesley Professional.