TDE Game Engine

User Manual

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

Neil Donnelly (10108823)
5/27/2013

Table of Contents

1. Setup Instructions

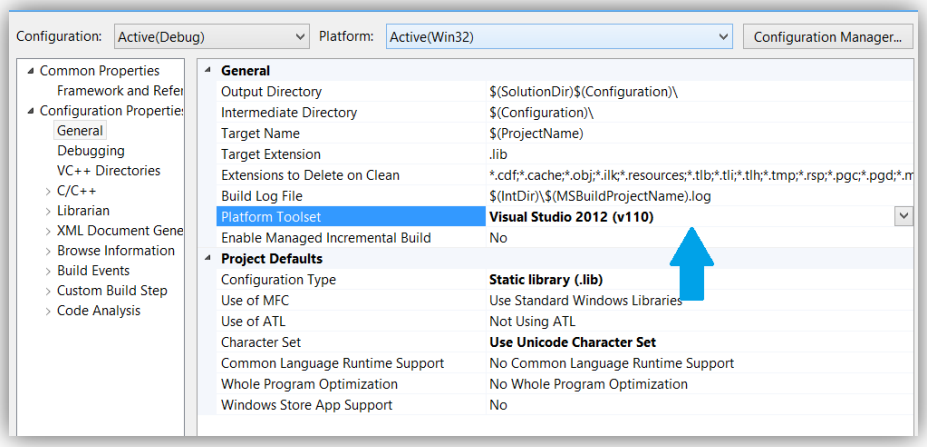
Required:

- A version of Visual Studio (2008 edition or later) on a Windows Machine
- The TDE Engine folder

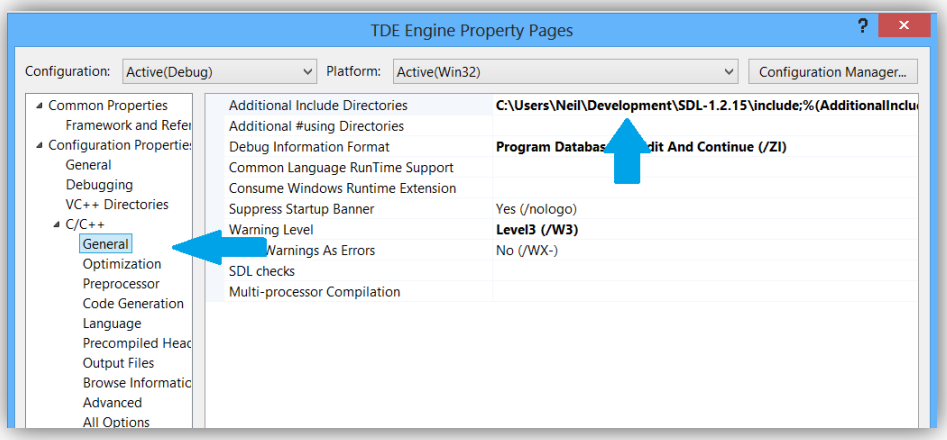
1.1 Visual Studio pre-2012

As the engine was written using Visual Studio 2012 edition, the library is set for development on the same version of Visual Studio. If this is the edition you are using, skip ahead to 1.2 for setting your TDE Project up. If you're using an earlier version of Visual Studio, thee instructions detail how to set up, and compile the TDE Engine code to work with other versions of Visual Studio.

1. Locate the VS Project file in the TDE Project folder.
2. Double clicking this should open your version of Visual Studio which should request permission to try and recreate the project to run on your edition of Visual Studio.
3. Once this is done, open the properties for the project by right clicking the project in the solution explorer.
4. In the General tab in the properties dialogue, make sure that the file being created is a Static Library (.lib)
5. Next, in the Platform Toolset, change the value that should be v110 to v10 or v08 (whichever corresponds to your version of VS)



6. Under the C/C++ tab and the General option, in the field for 'Additional Include Directories', add the file path for the SDL include files, present in the SDL folder in the Project folder.

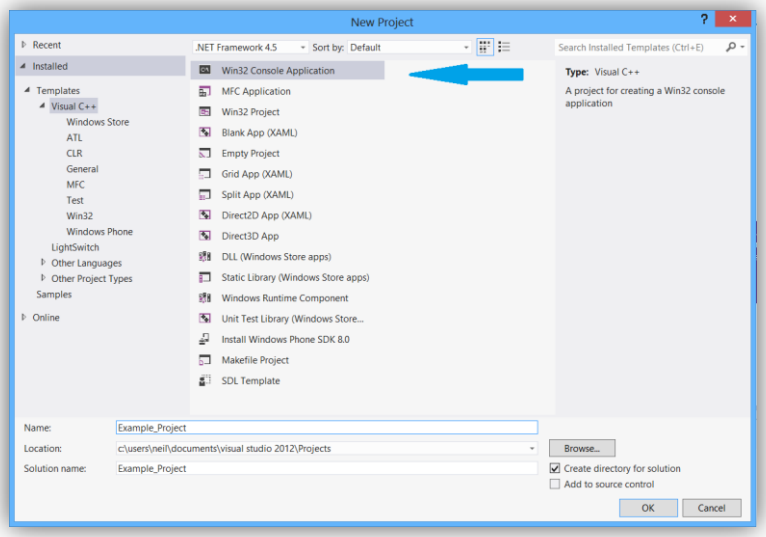


7. Once this is done, apply the changes to the properties and build the solution using the option at the top of the menu or by pressing the F7 key. This should generate the lib file for the engine that will work with your version of Visual Studio. It should be found in the Release or Debug folder in the folder created by Visual Studio.

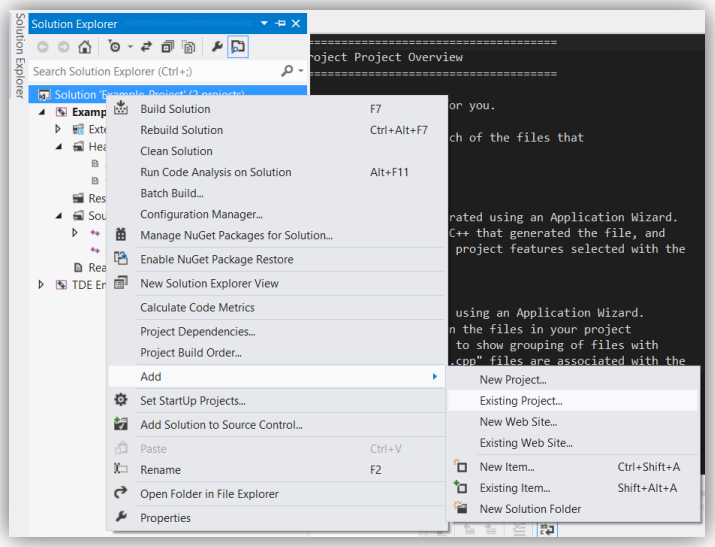
This library file and the files in this project folder can be used for the following section to set up a TDE Project.

1.2 Set up TDE Project

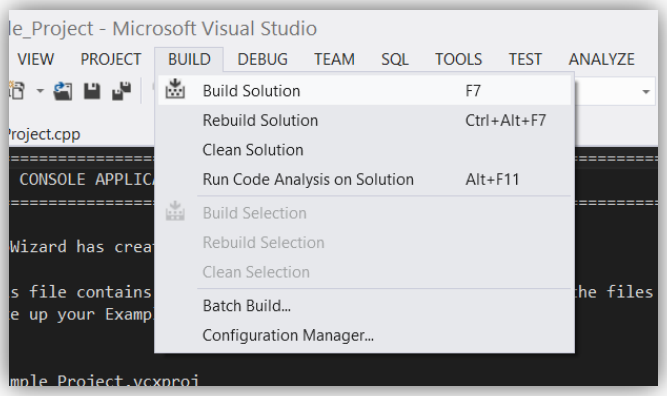
1. Open up Visual Studio and start a new project under the File menu.
2. Choose ‘Win32 Console Application’ as your type of project and give the project a name



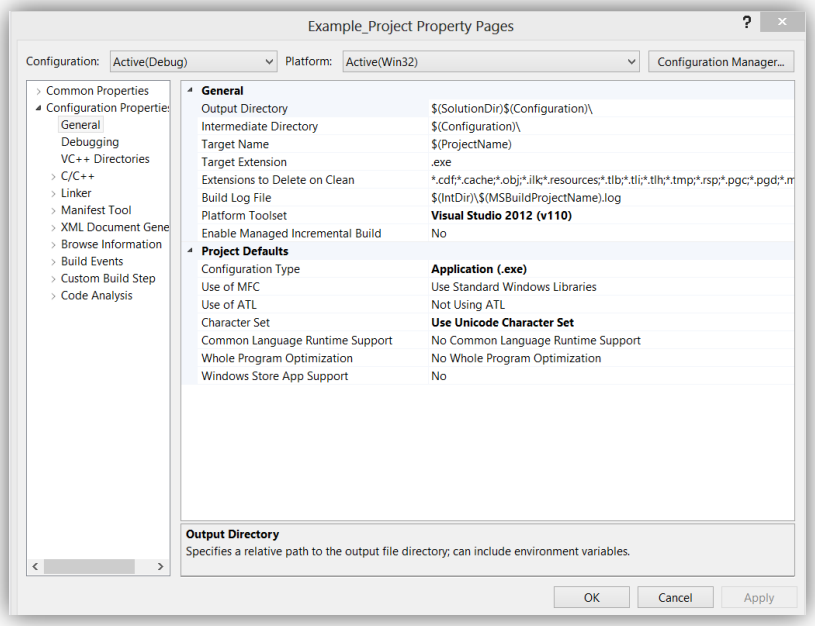
3. *This is an optional step in order so the Engine source may be seen in the project.* Next, under the Solution Explorer window, right click the solution icon, choose Add, then 'Add Existing Project' and navigate to location of TDE Engine Project folder. In this folder, choose where you should fine the Visual Studio project folder.



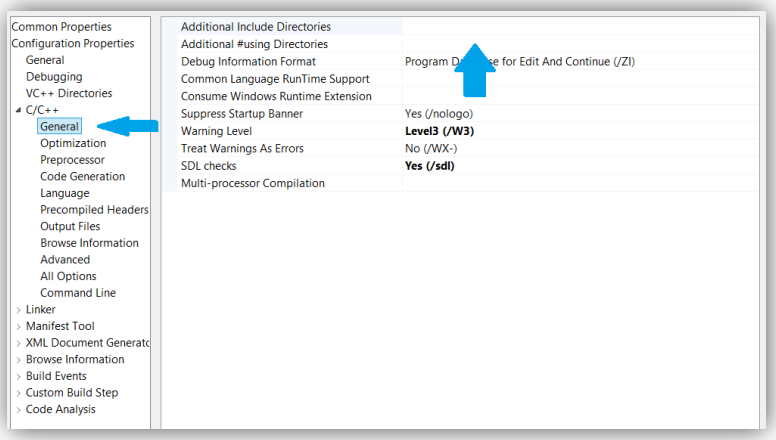
4. Next, click on the Build option at the top of the Window and choose Build Solution. This is to build the generated directories and files necessary to continue. A build pane should open at the bottom detailing a successful build.



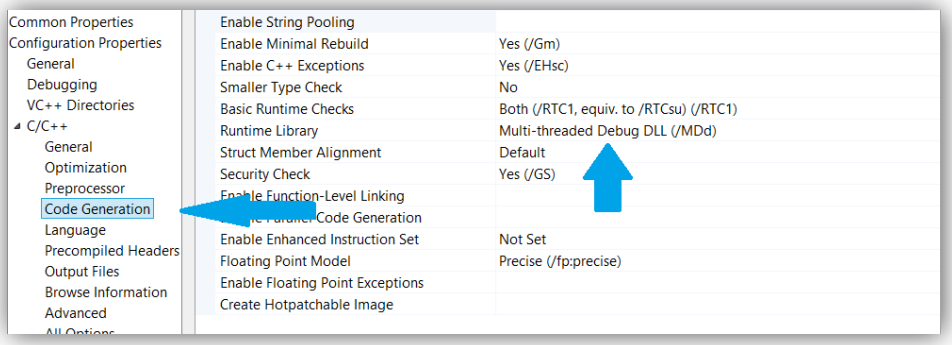
5. Now, in Windows Explorer, copy all the Dynamic Linked Libraries (DLL) files in the DLL folder from the TDE Engine folder into newly generated Debug folder in your Project's directory.
6. Back to Visual Studio. In the Solution Explorer view, right click on your Project's Solution icon (not the project icon above it) and choose click on the 'Properties' option. This should open a Dialog box similar to this:



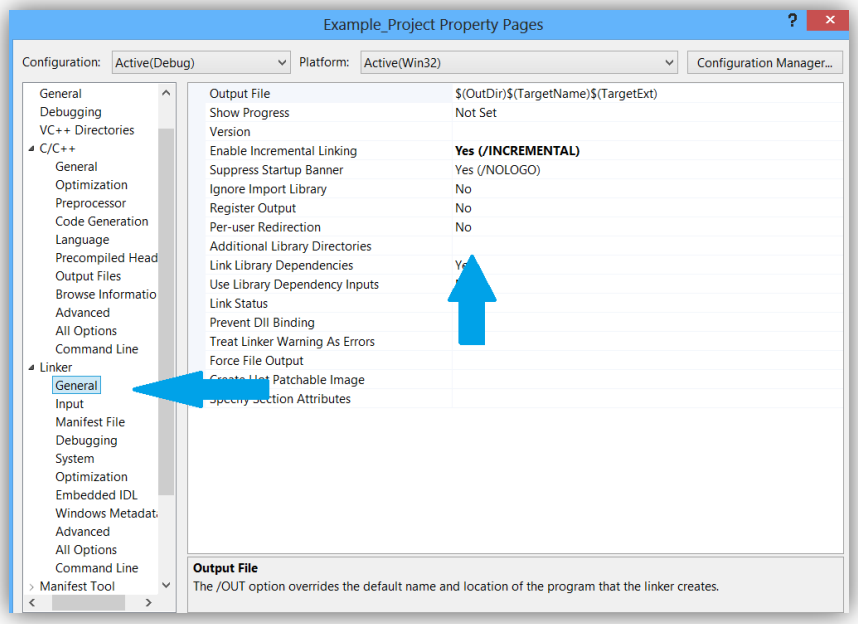
7. Under the C/C++ tab, click the General option. Then in the 'Additional Include Directories' add the file path to the TDE Engine source code in the TDE folder, and the file path to the SDL include files, also in the TDE Folder.



8. Under the Code Generation option, ensure that under the 'Runtime Library' field, 'MultiThreaded Debug DLL (/MDd)' is chosen.



9. In the same properties dialog, open the Linker settings tab. Under the General option, there is a field for “Additional Library Directories”. In his field, add the file path to the static library file (.lib) generated by the Engine. Alternatively, this lib file can be added to your own project folder and add the file path to that location. In the same field, add the file path to the x86 lib files in the SDL folder provided in the TDE Engine. These two entries should be separated by a semi-colon (;).



10. Under the same Linker tab, in the Input options, under the Additional Dependencies field, double click on the field to open another dialog and enter the names of the following lib files:
    - SDL.lib
    - SDLmain.lib
    - SDL_image.lib
    - SDL_ttf.lib
    - opengl32.lib
    - SDL_mixer.lib
    - glu32.lib
    - TDE Engine.lib

2. Beginning the Development

To begin development of the game with the TDE engine, you must first initialise the engine and then run it using a Root Widget. This root widget forms the main point of entry for the engine to control the game.

2.1 Running the Engine

To begin, first create the engine and initialise the graphics portion in order to create the window. To do this, in the file containing your main method which was automatically created when you first created the project, ensure that you are using the TDE namespace by placing this below all your '#include' statements:

```
using namespace TDE;
```

Now, in your main method, create an instance of the TDE Engine:

```
TDE_Engine engine = TDE_Engine();
```

Using this engine object's Graphics object, initialise the window:

```
TDEGraphics* g = engine.GetGraphics();
g->InitGraphics(Width_Of_Screen, Height_Of_Screen, "Window Name");
bool wantFullScreen = false;
g->CreateWin(wantFullScreen);
```

This will create a window of the dimensions specified with the width and height variables and the given name.

Once this is done, a Root Widget needs to be created. The implementation and creation of this widget will be explained further in the next section (3.1.3), but for now, all that is needed is the code necessary to run the engine. In order to run the engine, it will need to pass the Graphics, Animation and Audio manager to the root widget and then use this widget to constantly run and draw the game:

```
RootWidget root = RootWidget(engine.GetInputManager(), g,
    engine.GetAudioManager());
engine.Run(&root, 60);
```

This is all that is needed for the engine to run. The logic of the game will be defined throughout the various widgets.

3. Widgets

Widgets form the basis for all interaction with the player, they control what the player sees, hears and when they hear it. The player will also be interacting with these widgets when they use the keyboard and mouse. All the widgets are placed in a tree structure beginning with the Root Widget. This widget has child nodes which it passes control to when needed,

these children can have children of their own allowing the flow of control to pass through the tree iteratively.

3.1 Basic Widget

All other widgets derive from this most basic widget. This widget is an abstract class and so many of its functions can be overridden. In order to control the widget appropriately, the main two functions to implement are the Update and Draw functions. The Update function is commonly used to complete certain logic tasks and update the state of that portion of the game. For instance, if a game has two players, the widget controlling this game would update the two players with any necessary information (input, time, etc.). The Draw function is used to do any rendering. Each draw function is passed a pointer to the Graphics object held by the Root. A sample header file for the implementation of the Widget class is below:

```
#include "stdafx.h"
using namespace TDE;

class TestWidget : public Widget
{
public:
	TestWidget(int x, int y, int w, int h, ParentWidget* parent);
	TestWidget();
	~TestWidget(void);

	void Update();
	void Draw(TDEGraphics* g);
};
```

This includes a constructor with the dimensions and position of the widget. The ParentWidget parameter will be the parent node of the node created on the tree structure.
The basic widget also has functions that allow the widget to listen to keyboard and mouse input; this process will be explained in Section 4: User Interaction.
Each Widget also has functions which can hide and show the widget. If hidden, the Draw function of the Widget is never shown and so never seen. These functions can be overridden so the Widget can make necessary adjustments to its state if required. Each Widget can also be deactivated and activated using the function:

```
void SetActive(bool isActive);
```

When not active, the Widget is not updated and not drawn. It can also be updated to allow a Widget to make any necessary adjustments before being deactivated.

3.2 Parent Widget

The Parent Widget is a Widget with the potential to contain any other widgets. Any widget that requires this capability needs to be a Parent Widget in order to update and draw the children. It is defined much like the Widget implementation, including the addition of a

Parent Widget which may be the Root Widget. Otherwise, the ParentWidget has the same features as the basic widget. This includes the Hide, Show and SetActive functions, which when used the effect trickles down to the Children of the Parent; so if a Parent is hidden, all of the children are automatically updated as well.

In order to call the Update and Draw functions of the children of the parent, the super function of the Parent Widget needs to be called. This will take care of calling all the parents children. This can be done as follows for drawing the children:

```
void TestParent::Draw(TDE::TDEGraphics* g)
{
	ParentWidget::Draw(g);
}
```

To add or remove children to the Parent, the following functions can be used:

```
virtual void AddChild(Widget* child);
virtual void RemoveChild(Widget* child);
```

The functions require a pointer to the children being added. As a warning when using the functions, be wary of the pointers passed that you do not dereference them on passing them to the function as this will cause a null pointer exception. All of these children are stored in a vector of pointers known as `mChildren` which is protected so only the implementation of the Parent has access.

3.3 Root Widget

This is an extension of the Parent Widget designed to form the basis of the Widget tree. It is used by the engine to drive the game by calling its Update and Draw functions at the proper intervals. Creating and running this widget has already been discussed in Section 2, but to implement there are a few recommendations to be made.

When implementing the Root Widget, the consideration needs to be made that this Widget will never be stopped and will be in control of what widgets on the tree will be run and when, thus in order to keep track of what branch it should be running, a way of keeping track of the state is required from each branch. For instance, the widget for the Start Menu will need to be run from the Root and so some way for the Root to know when to close this branch and what other will need to be implemented. This could possibly be done using a callback function or using an enumeration of various states that allows the Root to query after each cycle.

Another consideration is a performance one. Once the root switches from one branch to another, try to ensure that the previous branch is set to inactive rather than merely hidden and that this branch is never run. Otherwise, these unseen branches can take up valuable

computing cycles. These hidden branches may also be set to pick up on keyboard activity which can cause unexpected consequences if not properly controlled.

3.4 Button Widget

This is a rather simple extension of the Widget class used for the construction of one of the most basic components of a Graphical User Interface. It allows for two images to be specified which make up what the button looks like when not pressed and when it is. It also has two callback functions which are called once the widget detects it is being pressed or released.

The two button images can be set using the constructor:

```
ButtonWidget(int x, int y, int width, int height, ParentWidget* parent,
TDEImage* aUp, TDEImage* aDown);
```

Or this can be done explicitly using the functions SetUpImage() and SetDownImage(). The Up image is compulsory for the button to be drawn but the down image is optional.

4. User Interaction

The TDE Engine has an interaction system based on a subscribed observer system which supports both mouse and keyboard activity. It works by subscribing a Widget to a particular type of input (either Mouse or keyboard), then when input is detected from one, all the widgets subscribed to that input will be notified with the changes. This allows for individual widgets to decide what input pertains to them and ignores other input.

4.1 Subscribing

In order to subscribe a widget to a particular input, the first thing that needs to be done is to retrieve the subject for the Input. This can be either the MouseSubject or KeySubject. This is done quite simply by having the widget call the function to listen for the subject. This takes care of all the necessities to have them subscribe to the input. For the mouse input:

```
Widget->ListenForMouse();
```

And for Keyboard input:

```
Widget->ListenForKeyboard();
```

Once this is done, functions for each of the input can be implemented to pick up the desired input. These functions will be explained in over the next two sections (4.2 and 4.3):

```
void KeyDown(TDE_Key k);
void KeyUp(TDE_Key k);
void MouseClickDown(MouseState &m);
void MouseClickUp(MouseState &m);
void MouseMoved(MouseState &m);
```

In order to have the Widget no longer listen to the input, this can simply be done using the functions:

```
void IgnoreMouse();
void IgnoreKeyboard();
```

The widget is removed from the list of subscribers and will no longer receive any more input updates.

4.2 Keyboard Input

The only functions that need to be implemented for the collection of the key presses are KeyDown and KeyUp. Both these take a TDE_Key as a parameter, which is an enumeration defined by the engine which lists all the possible keys the engine is capable of collecting. The full list is present in the file 'InputKeys.h'. The list includes all the letters, numbers and symbols of the keyboard as well as the control shifting keys such as the Control, Shift and Alt keys. The use of the enumeration makes it possible to use a switch statement to decide the value of the input. A possible implementation of the KeyDown function could be:

```
void TestWidget::KeyDown(TDE_Key k)
```

```
{
	switch(k)
	{
	case: KEY_RIGHT:
		moveRight();
		break;
	case: KEY_UP:
		moveUp();
		break;
	default:
		break;
	}
}
```

4.3 Mouse Input

The Mouse Input is slightly more complicated as it allows for three types of input to be notified about. These are for the mouse buttons being pressed and released but also the movement of the mouse on the screen. All of this information is kept in a structure called MouseState. When one of the functions is called, it provides a reference to the current Mouse state which includes the current position of the mouse and the press status of all the buttons. The following code sample demonstrates how to print out the state of the mouse which is called when a button on the mouse is pressed:

```
void TestWidget::MouseClickDown(MouseState& m)
{
	printf("Position of Mouse: %d, %d\n", m.x, m.y);

	printf(m.leftClicked ? "Left pressed\n":"Left not pressed\n");
	printf(m.rightClicked ? "Right pressed\n":"Right not pressed\n");
	printf(m.middleClicked ? "Middle pressed\n":"Middle not pressed\n");
}
```

5. Graphics

5.1 Drawing Shapes

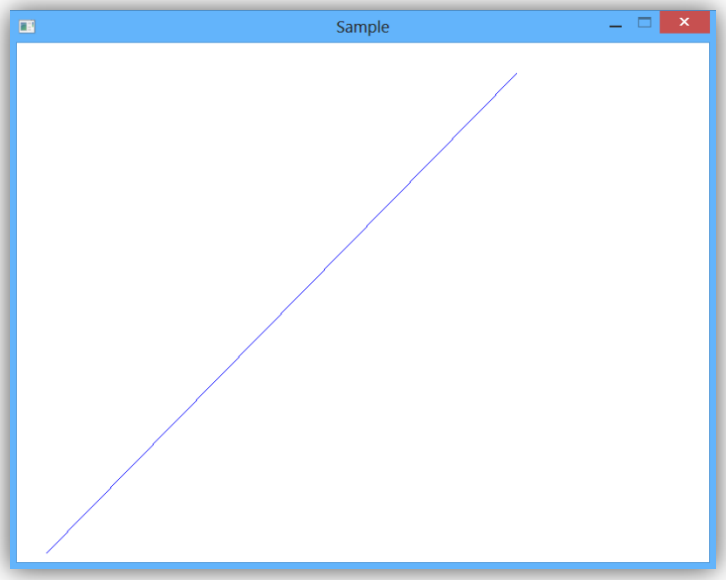
The TDE Engine supports the construction and drawing of primitives such as lines and rectangles but also supports the drawing of irregular shaped polygons. For further details on the following functions, please see the API.

5.1.1 Lines

Drawing a line is a simple process of providing the x and y coordinates using integers or floats. There is also the ability to explicitly state the colour of the line when drawing it.

```
g->DrawLine(lPoints[0], lPoints[1], lPoints[2], lPoints[3],
    TDEColor(0.f,0.f,1.f));
```

Where lPoints is an array of random integers within the limit of the window, this produces:

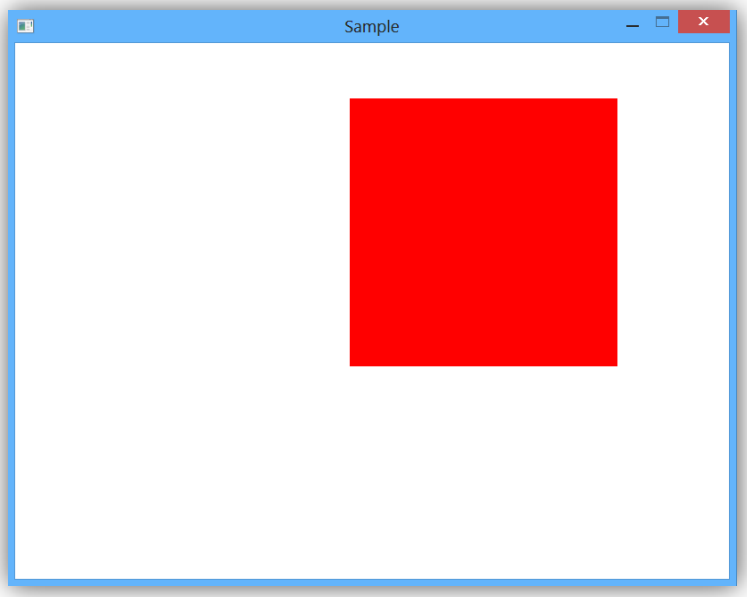


5.1.2 Rectangles

Rectangles can be drawn in two styles. The function DrawRect simply draws the perimeter of the rectangle specified by the arguments. The FillRect function draws the full area and colours the shape with the engine's currently saved colour. The dimensions of the rectangle can be given with floats, integers and using a Rect object, a class provided by the engine to define a Rectangle.

The following code snippet draws a red rectangle of 300px width and 300px height at a random location on the screen, assuming rPoints[] is an array of random points.

```
g->SetColor(1.f,0.f,0.f);
g->FillRect(Rect(rPoints[0],rPoints[1],300,300));
```

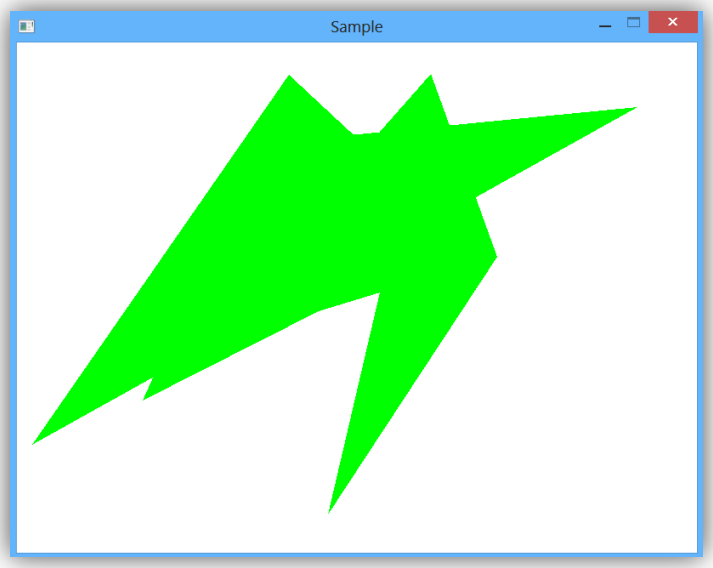


5.1.3 Custom Shapes

The TDE Engine also provides the capability to draw more complicated objects by allowing the construction of a shape using an array of points using the shape primitive of Triangle Strips. This requires an array of float or integer tuple with the x and y values. It also allows for the specification of a colour to draw the shape in. As the engine separates the array of points into the triangles itself, you only need to be concerned with the points provided and the order in which they are provided. It is important to note that the direction the points are drawn defines the face of the object so try to ensure a counter-clockwise direction.

The following draws a green shape of ten random points using an array, pPoints[], which holds 10 random points.

```
g->DrawShape(pPoints, numPoints, TDEColor(0.f,1.f,0.f));
```



5.2 Images

The TDE Engine supports multiple ways of loading in images and textures of various formats and then drawing these images to screen. In the engine, image objects are referred to as TDEImage's.

5.2.1 Loading Individual Images

All images loaded in as textures to be used by the engine later but can dynamically added and removed during run time. The formats available for use are PNG, JPEG, GIF, BMP, TGA and TIFF. Individual images can be loaded using a Graphics object. This is done using the following function:

```
bool LoadTDEImage(std::string path, std::string name);
```

The path variable specifies the location of the image in the directory and the name arguments specifies the name you wish to associate with the loaded image. The function will return true if loaded successfully and false otherwise. The function will fail if the image is not present at the given location or if the name provided is not valid (already used by another image). Any specific errors will be printed to the standard out if game run with console active.

Any image loaded is kept by the engine and a pointer to the loaded image can be retrieved by using the following function:

```
TDEImage* GetImage(std::string name);
```

5.2.2 Loading Atlas of Images

The engine also allows for the loading of multiple textures in one large image, called an Atlas. This Atlas is used to save performance as the engine only requires a single texture to

be kept in memory when using images from the one file. To facilitate this, another program comes packaged with the engine called the TexAtlas tool. This program can be pointed to a particular directory and combine all the images in the directory and its sub directories into large atlases. As it creates these large images, it also creates an XML document which is used by the engine to discover the location of the component images in the Atlas so that they may be used individually. In order to use this XML and Atlas file, all that is needed is to use the following function:

```
bool LoadAtlasFromXML(std::string filePath, TDEColor alphaKey);
```

The filePath argument should specify the location of the XML document which is used to find and load the Atlas files. The other argument, the alphaKey, is an option argument used to specify what (if any) colour is being used to represent the alpha pixels in the document. Usually, this is a colour that is never used in any of the pictures. When using this Atlas system, the name given to each of the images is the same as that of the original image's file name.

5.2.3 Drawing the Images

Once the images have been loaded and a pointer has been retrieved, there are multiple ways to draw the image. The images can be drawn using integers or float arguments which define position and size, or can be defined using one of TDE's rectangles. The position of the image is compulsory but defining the size is optional. If the size is not specified, the image's actual size is used. Using the retangles method also allows for a particular section of the image to be drawn. For instance, a rectangle can be defined to only draw the top right quarter of the image rather than the whole thing using a source Rectangle. More information on these functions can be found in the API. Here are the three most basic functions for drawing an image:

```
void DrawImage(TDEImage* im, int x, int y);
void DrawImage(TDEImage* im, Rect destRect);
void DrawImage(TDEImage* im, Rect destRet, Rect srcRect);
```

Quite often drawing an image perfectly straight is enough however, and so the engine also provides the ability to draw the image at an angle. The angle is specified using a floating point number representing the degrees of rotation. To do so, use the function:

```
void DrawRotatedImage(TDEImage* im, float x, float y, float angle);
```

To allow even more customization, the engine also provides a method of colouring images drawn to the screen. This process, known as colorizing, blends the current colour the engine has saved onto the texture drawn to the screen. To activate and deactivate this feature, the following function can be used:

```
void ColorizeImages(bool cIm);
```

5.2.4 Code Sample

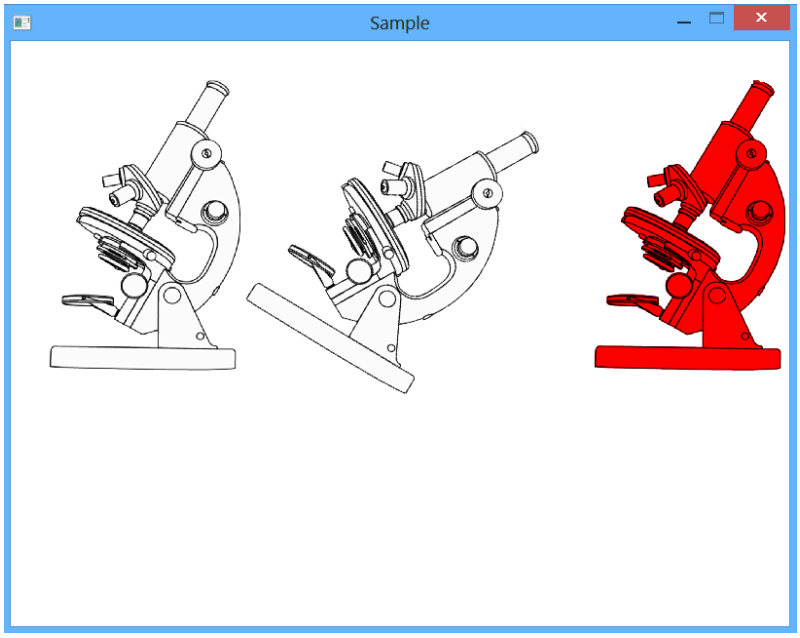
The following sample code illustrates how to load an image, retrieve it and then draw it, rotate it and then colour it.

```
g->LoadTDEImage("Resources\\MICRO.png", "SampleImage");
TDEImage* im = g->GetImage("SampleImage");

g->DrawImage(im, Rect(40,40,200, 300));
g->DrawRotatedImage(im, Rect(300,40,200,300), Rect(0,0,im->GetWidth(),
     im->GetHeight()), 30);

g->ColorizeImages(true);
g->SetColor(255,0,0);
g->DrawImage(im, Rect(600,40,200,300));
g->ColorizeImages(false);
```

This code produces this result:



5.3 Animations

Animations are a crucial part of any game that allows objects to appear as though they are moving fluidly on the screen. The TDE Engine offers a variety of ways to create and control animations. An animation object is referred to as a TDE_Animation.

5.3.1 Loading Animations

There are several ways of loading animations onto the engine. The simplest but least efficient way is to load a series of standard images onto the engine, collect them into an array and then have the engine compile them into an animation. This can be done using the following function:

```
bool CreateAnim(std::string name, TDEImage* cells[], int numCells);
```

Another feasible way is to use a single image to house all the cells of an animation in uniform dimensions. The engine can then use this single image to collect each individual cell and form the animation. This technique can be done with an image already loaded onto the engine or it can be pointed to the location of the image and compile the animation from that.

```
bool LoadAnim(std::string path, std::string name, int cellWidth, int
      cellHeight, int totalWidth, int totalHeight, int numCells);

bool LoadAnimFromImage(std::string name, TDEImage* im, int cellWidth, int
      cellHeight, int numCells);
```

Similarly to the Image loading process, the name given to the animation is the string that can be used to retrieve the animation.

5.3.2 Drawing the Animation

In order to draw the animation, the process works much like that of drawing an image with the same functions, for instance to draw the animation a particular point and rotated by a number of degrees, one can use the function:

```
void DrawRotatedAnim(TDE_Animation* anim, float x, float y, float angle);
```

Drawing an animation to the screen will draw the currently active cell of the animation; once the animation is playing the engine will decide what cell should be drawn so the user need only worry about the location and angle of the animation.

5.3.3 Controlling the Animation

As an animation, there are multiple operations that can be performed on it to control how it runs. Firstly, to play the animation the following function is used from the animation itself:

```
void Play(int loopTime, int startCell, int numRepeats);
```

The first argument indicates how long it should take to cycle through all the cells in milliseconds. Giving an integer of 1000, will mean the animation will play in its entirety once per second. The second argument of the function indicates what cell the animation should start playing from, this ranges from 0 to the number of cells present minus one. Any number outside of this range results in a failure and an Exception. The final argument indicates how many times the animation should run before stopping. Using 0 implies the animation will play fully through all its cells and then stop at the last, 1 will play the animation twice, and 2 will play it three times and so on. To play the animation continuously, use -1.

Once the animation is playing, the animation can also be instructed to pause, stop and reset. Pause interrupts the playing of the animation at the current cell until it is resumed Stop ceases the animation completely and in order to play again requires the Play function be recalled. The Reset function allows the animation to keep playing but returns the

animation to the first cell. Once the animation is running, in order to keep the animation updated, it is recommended that during the Update function of the Widget controlling the animation, the animation's own update function is called.

All the animations on the engine are controlled individually by the user when needed, but this can become cumbersome in certain situations when there may be a variable number of animations. In order to aid the developer in this predicament, the TDE Engine allows for animations to be registered to the Animation Manager using the function:

```
bool RegisterAnim(TDE_Animation* anim);
```

Once registered to the manager, all the registered animations can then be controlled collectively once run. This includes the ability to pause, stop and reset all the animations at once. Any animation registered in this way is updated by the manager without the need for the widget to do so in its Update function.

5.4 Fonts

As well as the ability to draw shapes, image and animations, the TDE Engine also provides the capability to draw text to the screen with any user defined font. The only requirement being that the font is of the True Type Font (TTF) format, the most common font format used across all the most popular operating systems. The engine refers to Font objects as TDE_Fonts.

5.4.1 Loading Fonts

Much like images or animations, fonts can be loaded simply by pointing the engine to the directory location of the TTF font that is to be used. This is done using the function:

```
bool LoadTTF(string name, string path);
```

This loads the TTF font to the engine where it can be retrieved using the name given with the function:

```
void LoadFont(string name);
```

Once loaded, the Graphics engine then stores the font as the font it will then use whenever text is to be drawn.

5.4.2 Drawing Text

Drawing the text is simply done by specifying the location the string is to be drawn, based on its top left corner. The font currently loaded by the engine is the font that will be used to draw the text.

```
void DrawString(std::string str, int x, int y);
```

The colour of the text is defined by the currently loaded colour of the Graphics object. Alternatively, both the font to use as well as the colour can be dynamically assigned to the text by passing them as arguments in the function. Care must be given to how the text is drawn as there is no support for allowing multiple lines, the engine will simply draw the entirety of the text on one line until it is done. To allow for a developer to draw text on multiple lines, the engine allows the height of the font to be retrieved and the width of a string of text will be using the font. The size of the font can be altered using the font's size function:

```
void SetSize(int s);
```

The style of the text can also be controlled using the font's functions for making the text bold, italic or underlined with the functions:

```
void SetBold(bool b);
void SetItalic(bool b);
void SetUnderlined(bool b);
```

In order to check that any changes to a font have not damaged the font and to ensure a successful loading, use the following function to ensure that the object remains valid. If false is returned, something has gone wrong and using the font will lead to an invalid pointer exception:

```
bool validateFont();
```

5.5 Graphics State

As control of the graphics object passes from one Widget to another, one Widget may need settings quite different from another. These settings include the colour, font, translation or scaling being used by the Graphics object when rendering. To allow for a Widget to make its own setting with disrupting the settings of its parent or siblings, the engine isolates these settings into a Graphics State object. These states are placed on a stack to allow for one Widget to push the current settings onto the stack, change them for its own purposes then once done, pop the old state of the stack for the Graphics to adopt again. To do this, the following two functions are used:

```
void PushState();
void PopState();
```

The settings associated with these states are described in the proceeding functions.

5.5.1 Clipping

Clipping is the ability to set an area for the Graphics object in which drawing objects can only be done in this object. Anything beyond the bounds of this area is not drawn to the screen. The TDE engine uses a rectangle for this and can be set by using the function:

```
void SetClipRect(Rect clip);
```

Clipping must be enabled for the clipping to have an effect. By default the clipping rectangle begins as the size of the window.

5.5.2 Colour

The engine uses the RGB model of colour and contains its own colour representation that can be referred to get an integer value between 0 and 255 or as a floating point number between 0 and 1. The colour used by the Graphics object is used in a multitude of ways. It decides what colour any shapes should be, what colour any text drawn to the screen will be and what colour images will be colorized with, if enabled. To set the colour, one of either of these functions can be used:

```
void SetColor(TDEColor col);
void SetColor(int iRed, int iGreen, int iBlue);
```

The alpha is an optional argument that defaults to the value of 255. The default colour used by the engine is white.

5.5.3 Scaling

The scaling setting of the Graphics object changes the size of an image or text. Scaling is represented by a floating point number where 1.0 represents the actual size of the object. 0.5 is half the size and 2.0 is twice the size of the object. Once set, this scaling affects everything drawn to the screen. The scaling can be set using the function:

```
void SetScale(float scale);
```

5.5.4 Translation

Whenever objects are drawn to the screen, they are drawn in relation to the origin. For this engine, the origin is deemed to be the top left of the window. In order to translate the origin to another point on the screen, which may make it easier for drawing a group of objects occupying the same sub space, the following function can be used:

```
void Translate(int newX, int newY);
```

Until this translation is changed, any objects drawn will use the new point provided as its origin.

6. Audio

The Audio of the engine can be broken up into two different types: Sound and Music. Although no difference in the quality of the audio between the two, a sound is considered a chunk of audio played over a short time span, like a sound effect. Music however, is thought of as being much longer in time and is used as background music for the games. They are loaded quite similarly but controlled differently.

Both are controlled using the Audio Manager. This controls the use of all music and sound as well as loading them. The manager needs to be initialised before use, which can be done like this:

```
audioManager->Init();
```

When the audio manager is no longer needed, all the audio memory can be cleared by using the following function:

```
void CleanUp();
```

This deletes all music and sound files currently stored by the audio manager.

6.1 Loading Audio

Loading audio is a simple process and requires the same arguments for loading both music and sound.

```
bool LoadSoundFile(std::string name, std::string path);
bool LoadMusicFile(std::string name, std::string path);
```

The path argument refers to the location of the file and the name argument is a string associated with the loaded object that can be used to retrieve a reference from the audio manager.

6.2 Using Music

All the music loaded is kept by the audio manager but only can play at a time. A Music object is referred to as TDE_Music in the engine. The current music to be played can be set using the function:

```
bool PlayMusic(TDE_Music* m, int repeats);
```

The pointer to the TDE_Music object to be played can also be substituted for a string holding the name of the music that should be played. Similar to playing an animation, the repeats defines how many times the music should be repeated, with -1 meaning continuously, 0 means to play the mouse only once then stop and so on.

Once this music is loaded by the audio manager, it can be paused, resumed and stopped completely. The volume of the music can also be set by steadily increasing the volume, which ranges from 0 to 128, or by setting the volume explicitly using an integer.

```
void SetMusicVolume(int vol);
void IncremenetMusicVolume();
void DecrementMusicVolume();
```

This engine also provides the ability to fade in and out music rather than abruptly start or stop it. This can be done using the functions:

```
bool FadeInMusic(TDE_Music* m, int repeats, int fadeTime);
void FadeOutMusic(int fadeTime);
```

It plays music much like the standard function, but now fades the music in using the fadeTime argument to decide how long it should take to reach the intended volume. The fadeTime is an integer that represents time in milliseconds.

Playing another music object immediately replaces the previous music being played.

6.3 Sounds

Sound objects are pieces of audio intended for short spans of time and are often used as feedback in reaction to a user's interaction or completing an activity (e.g. clicking a button, finishing a level). These sounds are known as TDE_Sounds. As there is a necessity to play multiple sound effects at once, each sound is played on a 'Sound Channel'. This helps separate the sounds and allows the ability to keep control of when a particular sound finishes.

In order to play a sound, one of these functions can be used:

```
bool PlaySound(std::string name, int repeats);
bool PlaySound(TDE_Sound* s, int repeats);
```

Once a sound is playing on a channel it can be paused, resumed and stopped at any time using the name or pointer to the sound object, for instance:

```
void PauseSound(TDE_Sound* s);
```

All the sounds currently on a channel can also be collectively controlled using the audio manager. All the sounds can be paused, resumed and stopped at the same time. For instance to pause all sounds, use this function:

```
void PauseAllSounds();
```

There are a finite number of channels available for sounds. It is not anticipated that a game should ever reach this upper limit as large performance penalties will make this rather infeasible. In case such an eventuality does occur, the TDE engine does provide a queue for waiting sounds if no channel is immediately available.

The volume for these sounds can also be altered individually or collectively using the functions associated with the TDE_Sound object or the Audio Manager.

7. Random Number Generator

The TDE Engine provides a random number generator that can provide a random integer or float. Once created, the number generator will create a random seed to use for generating these numbers. To help with any debugging efforts, the class also allows the ability to retrieve the integer used for the seed and set the seed explicitly. This can be helpful when debugging an area of the code that requires random numbers, if something unexpected happens, the seed can be retrieved and reused to help in reproducing the results.

Once created, this function will set a seed for the generator automatically:

```
void Randomize(void);
```

This however, allows the seed to be assigned:

```
void SetRandomSeed(unsigned int n);
```

Once the seed has been set, the following functions can be used for creating pseudo-random integers and floats respectively:

```
unsigned int Random(unsigned int n);
float Random();
```

The argument for the integer version sets a limit on the size of the returned integer number.

The API for the TDE Engine can be found at the following webpage:

http://student.computing.dcu.ie/~donneln7/fyp/api/apihome.html