CO1301: Games Concepts
Worksheet 1 - Introduction to the TL-Engine

Introduction

The *'TL-Engine'* is a special **Game Engine** to help teach games programming. It has been developed at the University of Central Lancashire specifically for teaching games programming. The TL-Engine has been written by Laurent Noel. Essentially it is a library of functions which can be called from within a C++ program. In this module, you will be writing C++ programs which call the TL-Engine to make 3D games.

The TL-Engine comes in two versions.

- **TL-X**. TL-X is the modern version. It is version 2 of the engine. It will not run on the ordinary University computers because it requires a decent graphics card. The whole of the engine has been built by Laurent. We will use this version within the Games lab and it is the version we suggest you use at home.
- **TL-E**. This was version 1 of the engine. It was built around the *Irrlicht* engine. It will run on the University machines but it is less efficient and it does not contain some of the functionality of Version 2.

The TL-Engine was been designed for use with Visual Studio. You need to use *version 2.3 beta 4* with Visual Studio 2015.

- *We will use Visual Studio 2015 in the Games Lab*

Installing the TL-Engine

- A copy of the TL-Engine *should* be on the root directory of the D: drive. However, it has been very recently updated, and the version there may not be the latest one.
- You can download it from the games website.

Note that you need the ***latest*** version of the TL-Engine: *TL-Engine Programming Interface V2.3 Beta 4.* It can be found here:

http://www.gamesnorthwest.net/resources/tl_engine/downloads.html

The TL-Engine should already be installed on the machines in the Games Lab. (The second years have been using it for a few weeks). If the latest version isn't installed on your machine, install it now – use the default options. However, when it asks "Which components should be installed?" ensure that full model library is installed, i.e. that the radio button is set to ***Full*** not *Lite*.

For all but the very latest version of the TL-Engine, you need to configure the TL-Engine to work with the DirectX SDK. Follow the "instructions for setting directories in Visual Studio" on the download page.

Starting a new TL-Engine project

First make a new folder for your work on the desktop and give it a unique name so you can identify it easily.

> At the end of each session in the games lab you must save your work to a pen drive or a copy it to your account via "Remote Access". Do not assume that your work will be safe on the computer for later.
> Furthermore, since your work is stored locally on the hard-drive of the machine, anything you leave there is accessible by anyone else who logs onto the machine. BE CAREFUL not to leave assignment work there for others to steal!

Now from the 'Start' menu select:

'All Programs'->'TL-Engine'->'Create New TL-Engine Project…'

A TL-Engine dialog box will appear – type a descriptive name in the *'Name'* edit box and browse to the folder you made earlier in the *'Location'* edit box. *Visual Studio 2015* is selected by default.

Working with Visual Studio 2015

The TL-Engine wizard creates a fully working Visual Studio project to use a 3D engine. You can see the project structure in the *"Solution Explorer"* window on the left. Open the project by clicking on the '+' signs, and view the program code by double-clicking the on the file ending in '.cpp' (this is a C++ source file). You can run the program straight away:

- Click the play icon to the left of the word *'Debug'* near the menus (or press F5)
- Answer 'Yes' when asked about building the project
- The program will compile and run…
- First a console window appears and tells you what is happening
- Then a second blank grey window appears - this second window is a 3D engine showing an empty scene
- Press 'Alt'+F4 to close this window, and return to the code

Game Program Structure

- The TL-Engine wizard provides you with some initial source code
  - Don't delete it – it is useful!
- You can run the program code already – press the play button (near the top)
  - But you will only see an empty scene
- This template source code contains the instructions to start up and shut down the game engine
- It also has two sections for you to work in:
  - The **Game Set-up** section
  - The **Game Loop** section

- After the Game Engine has started we need to prepare everything needed for the game.
- First we need to load (from the CD or hard-drive) all the game objects
  - Primarily 3D models, but also sprites, sounds, fonts etc.
- Then we need to set the initial positions & settings for all the objects.
  - e.g. position all the 3D characters and objects, set initial statistics (like health or armour), etc.

- We also need these steps to restart a game

The template code provides a general game structure for your project. A game displayed in the same way as a film or animation.

- A sequence of static images (called frames) is displayed very quickly.
- Objects change position very slightly from frame to frame.
- The viewer does not perceive the separate images, and instead sees an animated scene

The Game Loop performs this operation:

- A static image of the scene is drawn
- Then the objects are moved slightly (if necessary)
- This process repeats as long as the game is running

So all games have, at their core, the same basic program structure:

```
#include <game engine libraries>
Start game engine

Load game objects from disk
Set initial positions for game objects

while (game is playing)
{
     Draw the game scene on screen
     Move/Animate game objects
}

Stop game engine
End program
```

Game setup

Game loop

The key sections that you will need to fill in are the *Game Set-up* section and the *Game Loop* section.

TL-Engine – New C++ Syntax

We need to introduce some C++ features to use the TL-Engine: and a style called "object-oriented programming". We will not introduce these features formally yet (you will be taught them later as part of another module), but they are the standard in games development so it is important to get used to them early.

There are two new things to look out for:

You may have already seen variables declared with standard types:

```
int length; // int is a standard C++ type (along with float, char etc.)
```

In the TL-Engine you will see variables declared with custom types:

```
I3DEngine* myEngine; // I3Dengine is a TL-Engine specific type
```

The type *'I3DEngine'* is not a C++ type; it is specific to the TL-Engine. Also it is being used with a *"pointer"* – the symbol *'*'*. You will be introduced to both custom types and pointers later in CO1401. For now just remember that *'I3DEngine*'* is a type just like an int or a float.

The TL-Engine has many *"functions"* that you can use. A function is a block of program code that performs a particular calculation or task. You can use a function in your own code when you want to perform that calculation or task – it saves writing the code yourself. There are some C++ functions that you *may* have seen already:

```
float squareRoot = sqrt( 16 );      // sqrt is a standard C function to
                                    // calculate the square root of a number
```

```
system( "pause" );      // system is a function that issues a special system
                        // command to the computer
```

In the *'sqrt'* example notice how the number 16 is *"passed"* to the function (in brackets). Also notice that the *'sqrt'* function calculates a result – and this result is put into the variable *'SquareRoot'*. In the second example, the string "pause" is passed to the *'System'* function, but there is no result – this function simply performs a task (waits for a key press).

You will see functions like this in the TL-Engine. Some are passed values (sometimes several values), and some are not. Also some return results, although again, some don't.

The use of each function will be explained to you in the worksheets, with examples given. Extra functions for advanced work are described in appendices. You will see functions shortly in CO1404 too.

Finally, most functions in the TL-Engine must be used *through* a variable:

```
      myEngine->DrawScene();  // myEngine is a variable (see above)
            // DrawScene is a function ("method") of myEngine
```

Here the function *'DrawScene'* is used on the variable *'myEngine'*. It means "use myEngine to draw the scene". This kind of function is called a "method". Notice the use of the '->' symbol to indicate this usage.

The TL-Engine Template – Walkthrough

Now look at the TL-Engine template program:

```
// First Project.cpp: A program using the TL-Engine

#include <TL-Engine.h>  // TL-Engine include file and namespace
using namespace tle;
```

After the opening comment there is the include file for the TL-Engine, this contains all the information needed to use the engine. Following the include statement is a *'using namespace'* statement: all of the TL-Engine is "hidden" inside a *'namespace'* so it does not get mixed up with standard C or other libraries. This statement says that we want to expose the contents of the namespace so we can use its contents easily. These opening lines must begin all programs using the TL-Engine, you do not need to understand their detail, just make sure they are there.

In the next lines we see the familiar *'main'* function:

```
void main()
{
```

The first code lines inside *'main'* prepares a 3D engine for us:

```
      // Create a 3D engine (using TL-X) and open a window for it
      I3DEngine* myEngine = New3DEngine( kTLX );
      myEngine->StartWindowed();
```

Here we see one of the new C++ features: a variable called *'myEngine'* is declared and its type is an *'I3DEngine*'*. All you need to know now is that the variable *'myEngine'* will be your interface to the TL-Engine - you will use engine features "through" this variable.

After declaring the *'myEngine'* variable, it is initialised by using a function/method: *'New3DEngine(kIrrlicht)'*. The TL-Engine uses the Irrlicht engine you saw last week to provide some of its features. This function prepares a new Irrlicht 3D engine to be used through the 'myEngine' variable.

Let's summarise this first line in plain English, it says: **"I want to use a 3D engine of the TLX variety, and I will call it myEngine".**

The next line is the first use of this new engine we have prepared. See how we use the `->` symbol to say**, "I want myEngine to start up in a window".**

The next line is:

```
      // Add default folder for meshes and other media
      myEngine->AddMediaFolder( "[Install Folder]\\TL-Engine\\Media" );
```

where *[Install Folder]* is the folder chosen when the TL-Engine was installed. This tells the engine to look in this folder for any files (e.g. 3D objects) that you load later on.

Adding your own code

The next line is a comment - this is the start of the *Game Set-up* area:

```
	/**** Set up your scene here ****/
```

We need to add code here to load/create models so we have something to see. So, below this comment type the following code:

```
	IMesh* cubeMesh;
	IModel* cube;
```

Here we are declaring two variables, a mesh and a model. Note how they use the same pointer syntax as before, these variables will be used in the same way as 'myEngine'.

A mesh is a 3D object generated by an artist and saved in a file. A model is a single copy of a mesh in our scene. At first, this might seem similar - but remember that there can be several copies of a particular mesh in the scene (e.g. we may have one mesh of a tree, but make a forest in our game using 20 models created from this mesh). Consider a mesh as a 'blueprint' for real models. Now type:

```
	cubeMesh = myEngine->LoadMesh( "Cube.x" );
	cube = cubeMesh->CreateModel();
```

The first line here uses the 3D engine to load a file into our mesh variable. Notice how we need to go 'through' the variable 'myEngine' again using `->`. You can think about this as asking the 3D engine to perform a service for us (loading a mesh) - the 3D engine must do this job, as it needs to keep track of all new meshes. So in English:

" Use myEngine to load a mesh called "cube.x" that we will call 'cubeMesh' "

Then we use this mesh as a blueprint to create a real model in the scene. We use the same kind of syntax as before, but this time using the mesh. English:

" Use the cubeMesh blueprint to create a model in the scene that we will call 'cube' "

Finally add these two lines of code:

```
	ICamera* myCamera;
	myCamera = myEngine->CreateCamera( kFPS );
```

We first declare a variable to hold a camera. Then we ask the 3D engine to create a camera to look at the scene. We specify a FPS type camera that can be moved around with the mouse and cursor keys. The code style is the same as before. In English:

" Use myEngine to create an FPS camera that we will call 'myCamera' "

Walkthrough continued…

Now we will look at the last few lines of code – here is the *Game Loop*:

```
	// The main game loop, repeat until engine is stopped
	while (myEngine->IsRunning())
	{
		// Draw the scene
		myEngine->DrawScene();  // myEngine is a variable (see above)
		// DrawScene is a function ("method") of myEngine


		/**** Update your scene each frame here ****/

	}
```

This 'while' loop repeats until the engine stops (generally, the engine only stops when its window is closed). Inside the loop the 3D engine draws the scene using the viewpoint of our camera - the scene will be drawn into the window we opened earlier. We asked for a controllable FPS camera, so the image drawn will update and change as we move the camera around.

There is also space here to update the (objects in the) scene. The FPS camera has a slight problem - it is far too fast when using the new machines. Find the following line of code:

```
		/**** Update your scene each frame here ****/
```

Immediately after this line of code I want you to place the following delay loop:

```
		/**** Update your scene each frame here ****/
		for(int delay=0; delay<1000000; delay++) { /* empty body */ }
```

The final line occurs after the loop has ended (the engine has stopped). We need to delete the 3D engine that we created at the beginning:

```
	// Delete the 3D engine now we are finished with it
	myEngine->Delete();
```

If we don't do this then the resources used by the 3D engine will remain in the computer's memory after the program is finished, but no other program will be able to access them. This is called a memory leak - it will affect the performance of the computer, so this line is important.

The First Run

Now we have walked through the code and added some lines to make it do something more interesting. Run the program again (press the play icon or F5). You should now see a model and you can 'fly' around using the mouse and cursor keys.

Introduction to Programming a 3D Engine - Exercises

Here is a list of mesh filenames that can be loaded with the 'LoadMesh' function that we added above:

```
"Cube.x"
"Torus.x"
"Sphere.x"
"Grid.x"
```

1. Update the code to display the 'Torus.x' mesh instead of 'Cube.x'. Run the program.
2. Now display both the 'Cube.x' and 'Torus.x'. You will need to add new mesh and model variables (with sensible new names).

Positioning

When using 'CreateModel' you can pass three values to set the initial position of the model, e.g.

```
cube = cubeMesh->CreateModel( 10, 0, 20 );
```

This positions the model using 'Cartesian coordinates', with x = 10, y = 0 and z = 20. It is easier to understand Cartesian coordinates by loading an example model:

- Update your solution to question 2 above to display 'Grid.x' and 'Cube.x'
- Run the program.
- You will see a cube placed in the centre of a grid showing 'Cartesian coordinates'.

Now update the 'CreateModel' call for the cube (**not** for the grid) to pass the values above (10, 0, 20) and run the program again. You will see that the cube is now positioned 10 along the X line and 20 along the Z line (each of these lines is called an 'axis', the plural is 'axes' – pronounced 'ack-sees').

1. Experiment with a few different x and z positions for cube. Then try altering the y position. Keep all the values less than 50.
2. See if you can stack two cubes on top of each other.
3. Now stack two cubes on a sphere.
4. Save your project at this point by selecting 'File->Save All'

Movement

In the ***Game Loop*** of the program, we can add code to update the scene. The main use for this is to move our models. If we move a model by a small amount inside the game loop, then it will move continuously as the loop repeats.

- Create a new project using the TL-Engine wizard and repeating the steps from earlier. Save and start a new project each time you move to a new section.
- Write code to display a grid and a cube (copy some of your code from the previous project)
- Add this line inside the game loop:

```
cube->RotateY( 0.05 );
```

This will make the cube rotate by a small amount each time the loop repeats. This effectively spins the cube. Notice the cube spins around its Y-axis as indicated by the name 'RotateY'.

Here is a list of some of the movement functions available for models. Note that the word *'void'* on the left means that these functions don't calculate a value, they just perform a task. The word *'float'* in the brackets means you must pass a float value to these functions (the amount to move or rotate). The examples on the right show how these functions should appear in your code:

```
void RotateX( float );  // e.g. cube->RotateX( 0.05 );
void RotateY( float );  // e.g. sphere->RotateY( -0.1 );
void RotateZ( float );  // e.g. torus->RotateZ( 0.3 );
void MoveX( float );    // e.g. torus->MoveX( 0.1 );
void MoveY( float );    // e.g. cube->MoveY( -0.2 );
void MoveZ( float );    // e.g. cube->MoveZ( 0.4 );
```

1. Rotate the cube around the other two axes.
2. Try to use the MoveX function to move the cube to the right. What happens?
3. Save your work

Further Positioning and Movement

There are more functions available for positioning and movement. Note that the Get functions calculate a float value (indicated by the word *'float'* on the left), look carefully at the example usage:

```
// The Set functions allow you to set the X, Y or Z coordinates of a model to a
// given value
void SetX( float );     // e.g. cube1->SetX( -20 );
void SetY( float );     // e.g. cube2->SetY( 10 );
void SetZ( float );     //      etc.

// Set all axes at once (as CreateModel) e.g. myModel->SetPosition( 20, 0, 10 );
void SetPosition ( float X, float Y, float Z );

// The Get functions return the current X, Y or Z position of a model
float GetX(); // e.g. xpos = sphere->GetX();
float GetY(); // e.g. if (sphere->GetY() > 10)
float GetZ(); // etc.
```

Try to use these functions to:

1. Move a cube upwards until its Y reaches 30, then return it to its start position
2. Bounce a sphere left and right between X=-40 and X=40
3. Can you make the sphere roll while it bounces?

You will need to use your programming skills from The Four Week Challenge and Introduction to Programming for these exercises.