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

1. Introduction

1. This lab worksheet will help you write your first TL-Engine program.
2. This lab sheet involves a lot of reading as concepts and tools are introduced for the first time. Later lab sheets will tend to be shorter, with more work for you to do!
3. Finally, it is important that you **name the projects you create in each lab with the names suggested in the lab's sheet, and save them in a secure and accessible location (e.g. your UCLan provided cloud storage or Github)**. This is important because references will be made to projects from previous labs as we progress through the module.

2. Creating a new TL-Engine project

1. At this point you should either be working on a **Games Development** virtual machine, or on your personal machine with all the required software installed.
2. Click on the start menu and search for TL-Engine.
3. The TL-Engine template wizard app will appear, click on it to start the TL-Engine template wizard. You can also access this through 'All Programs'->'TL-Engine'->'Create New TL-Engine Project...'
4. In the name field, provide a name for your project, name your project **Lab1_Introduction_Project**.
5. In the location field, provide a location for your project to be saved.
6. The template wizard should by default have Visual Studio 2019 and "Automatically open new project" selected.
7. Click OK. The project will be created and started in Visual Studio 2019.
8. Once Visual Studio has opened your project, you should be able to see a "Solution Explorer" window to the right of your screen. This window contains the name of your project, with a tiny right facing icon by its left side, click on the icon. You should now be able to see your project structure.
9. Look for a file that has a name ending in '.cpp' (this is a C++ source file) and click to open it.
10. The file contains a template code generated by the TL-Engine, it does not do much, and you are going to add to it to make it more interesting. However, you can still run the project by pressing F5 or clicking the green 'play' button on the top menu.
11. The project will compile and run... 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

3. Using C++ Syntax

1. **C++** is a programming language that gives programmers high level control over system resources and memory. It is is one of the world's most popular programming languages and the language used to build most big console and PC games.

2. C++ programming will be taught to you as part of another module. However, this module will use the TL-Engine to introduce you to the features and concepts of C++ that are standard in the games development industry.

3. In CO1111 you used appinventor blocks to initialise variables for storing text, numbers, colors etc. You will also be using variables when programming in C++. However, working with varaibles is done a little differently in C++.

4. C++ has different variable types for storing different types of data (**int** for storing integers (whole numbers), **bool** for storing boolean values, **string** for storing text). The data type of a variable must be stated when declaring it to specify the type of data that the variable can store. Examples of variable declaration in C++ are shown below:

```
int length; // an integer variable named length is declared for storing whole numbers
string name; // a string variable named name is declared for storing text data
float speed; // a float variable named speed is declared for storing floating point - decim
bool check; // a boolean variable named check that can only take the values 'true' and 'fal
```

5. You can assign a value to a variable. Assignment is done using the equals symbol and can be done when or after declaring a variable.

```
int number = 10; // number is assigned a value of 10 when declared
bool check;   //check is first declared
check = true; // then check is assigned a value of true
```

   Note: Assignment is not the same as 'equals'!

6. The contents of a variable can be changed as shown in the examples below:

```
int i = 4;
i++; // i now has a value of 5, the ++ operatore increments a the value of a variable by 1

bool check = true;
check = false; // the value of check is changed to false

string name;
name = "Alex"; // the value of name is Alex
name = "";    // the value of name is changed to an empty string
```

7. While using the TL-Engine you will be using standard C++ variable types as well as custom variable types. An example of a variable declaration using a custom type is shown below:

```
I3DEngine* myEngine; // I3Dengine is a TL-Engine specific type for storing instances of a 3
```

   The type 'I3DEngine*' is not a C++ type; it is specific to the TL-Engine. Also note that 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, but specific to the TL-Engine.

8. 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 built-in C++ function in your own code when you want to perform a calculation or task – it saves you from 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 r
system( "pause" ); // system() is a function that issues a special system command to the cc
```

9. In the 'sqrt' example notice how the number 16 is 'passed' to the function (in brackets) i.e. given as input to the function. Also notice that the 'sqrt' function calculates a result – and this result is put into the variable 'SquareRoot'.

10. 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).

11. 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.

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

    ```
    I3DEngine* myEngine; // I3Dengine is a TL-Engine specific variable type
    myEngine->DrawScene(); // DrawScene is a function ("method") of myEngine
    ```

    In the above code, a variable named 'myEngine' is declared, then 'DrawScene()' is used on the variable (notice the use of the '->' symbol to indicate this usage). In plain English, the two lines of code state: "create a 3D engine and call it myEngine, then use myEngine to draw a scene". Functions like DrawScene() are called methods.

4. The TL-Engine Template Code

1. Your created project should be open in Visual Studio 2019 and the template code generated by the TL-Engine in the file with the extension '.cpp' should be visible on your screen... don't delete it – it is useful!

2. The template follows a structure that is common to game programs known as the "game code structure". The structure provides sections to setup, run and terminate games properly. These sections are *initialisation*, *game loop*, and *termination*; they are also shown in the code snippet below.

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

    //initialisation
    Load game objects from disk
    Set initial positions for game objects

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

    //termination
    Release resources
    Stop game engine
    End program
    ```

3. Now let us go over our template code. The first line contains a comment, followed by the inclusion of the TL-Engine file which 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.

    ```
    // Lab 1 - Introduction.cpp: A program using the TL-Engine
    #include  < TL-Engine.h > // TL-Engine include file and namespace
    using namespace tle;
    ```

4. In the next lines we see the 'main' function, it is where the execution of all C++ programs begin.

```
void main()
{
```

5. The main function begins with the initialisation section. Everything that is needed to get the game going is setup in this section. This includes initialising variables and loading resources (e.g. models and sounds).

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

   // Add default folder for meshes and other media
   myEngine->AddMediaFolder( "C:\\ProgramData\\TL-Engine\\Media" );

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

   In the code above, 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, meaning you will use the TL-Engine's features 'through' this variable.

   After declaring the 'myEngine' variable, it is initialised by using a function/method: 'New3DEngine(kTLX)''. In plain English that means "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 "myEngine->StartWindowed()" to say, "I want myEngine to start up in a window".

   The next line tells the engine to look in the folder provided (the folder chosen when TL-Engine was installed) for any files (e.g. 3D objects) that you load later on. Any resources you want to load to your game, should be loaded after this line of code.

6. Next comes the game loop section of the template code which executes in a loop until told to stop.

   ```
   // The main game loop, repeat until engine is stopped
   while (myEngine->IsRunning())
   {
       // Draw the scene
       myEngine->DrawScene();

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

   Currently the only code inisde this game loop is code that tells the 3D engine declared in the initialisation stage (myEngine) to draw the scene. Code can be added to the game loop to check for player input and update game objects in each iteration. The game loop's ability to perform several iterations every second and display updated *frames* creates the animation seen when playing games.

7. The final section, the termination section cleans up after the game, and releases all the memory resources used by the game. It contains a single line of code that deletes the 3D engine that you 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.

8. Now that we have some understanding of what the code template does, let us add code to make things more interesting.

5. Adding Your Own Code

1. In this section we will add code to the template code to declare new variables, load a mesh, create a model and a camera to display our scene.

2. Start by adding the code below in the initialisation section. Place it below `/**** Set up your scene here ****/`

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

   The above code declares two variables, a mesh variable and a model variable. Note how we use the same pointer syntax (*) as before when declaring a 3D engine variable, 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 a single mesh can be used to create several models in your scene (e.g. we may have a single tree mesh, but make a forest in our game using 20 models created from this mesh). Consider a mesh as a 'blueprint' for creating models.

3. Next, type the code below:

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

   The first line here uses the 3D engine (myEngine) to load a file and put it into our declared mesh variable (cubeMesh). Notice how we need to go 'through' the variable 'myEngine' again using "->". Think about this as asking myEngine 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, the code is saying: "Use myEngine to load a mesh called 'cube.x' and put it in the variable 'cubeMesh' ".

   In the second line, the mesh is used as a blueprint to create a model. Note that the syntax is similar to what we used to load the mesh using the 3D engine, but this time the mesh is used to create a model. In English, the code is saying: "Use the cubeMesh blueprint to create a model in the scene and put it in the variable 'cube' ".

4. The last piece of code we need to add to the initialisation section is the code below:

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

   The first line declares a variable named 'myCamera' that will be used hold a camera. The second line of code uses a similar syntax similar to what we have used when loading a mesh and creating a model. In English the code is saying: "Use myEngine to create a camera of type FPS and put it in the variable myCamera". FPS is a "first person shooter" camera that can be moved around with the mouse and cursor keys.

5. We are now done with the initialisation section and the game loop section already has code that draws the scene when the game is executed. Therefore when you run the project now, the 3D engine draws the scene using the viewpoint of our camera - the scene will be drawn into the window we opened earlier (in the initialisation section). We asked for a controllable FPS camera, so the image drawn will update and change as we move the camera around uisng keyboard or mouse.

6. Run the program (press the play icon or F5). You should now see a model and you can 'fly' around using the mouse and cursor keys.
7. When you run your project your FPS camera may be moving too fast, you can add a delay loop to slow it down. Add the code that follows inside the game loop immediately after `/**** Update your scene each frame here ****/` and run your project again.

```
for(int delay=0; delay < 1000000; delay++) {
    /* empty body */
}
```

8. Any changes or updates on game objects while the game is executing need to be performed in the game loop.

6. Practice Exercises

The sections that follow contain practice exercises on loading, positioning and movement; please attemp all these exercises. You should be able to complete the first three sections with little to no difficulty. If you finish these exercises in the lab, then move on to the advanced exercises by following the link at the end of this sheet, else try the advanced exercises in your time.

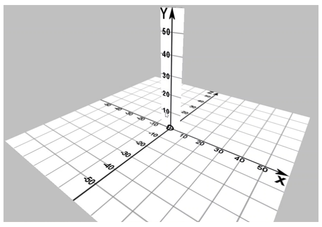
Loading

1. Here is a list of mesh filenames that can be loaded with the 'LoadMesh' function that we used in 4.3 (section 4, task 3).
    - "Cube.x"
    - "Torus.x"
    - "Sphere.x"
    - "Grid.x"
    - "TwoPence.x"
2. Our existing code can be updated to display a sphere instead of a cube by changing the name of the loaded mesh in 4.3 from 'Cube.x' to 'Sphere.x'.
3. Update the code to display the a torus using the 'Torus.x' mesh and run the program to make sure you have achived the desired result.
4. If we want to load more than one mesh at a time, we need to declare additional IMesh* variable(s) to put the new mesh in. The video hint below illustartes this by showing how a coin can be added and displayed in our existing scene.
5. Now display both the 'Cube.x' and 'Torus.x' in your program. Remeber, you will need to add new mesh and model variables (with sensible new names).

Positioning

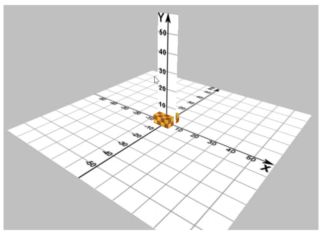
1. All the models we have created so far have been created in the same position in the scene. We can choose to create our models at different positions by using "Cartesian cordinates" to specify exactly where in the scene we want a model to be created.
2. It is easier to understand Cartesian coordinates by loading an example model.
3. Save your project, then create and open a new TL-Engine project.
4. Add code to load and display a grid using the "Grid.x" mesh. You can copy some of your code from the previous project.

5. When you run your project a grid will be displayed. The grid has lines that are marked with numbers, and each line represents a Cartesian cordsinate axis. The X axis runs horizontally from left (negative) to right (positive) of the grid; the Y axis runs vertically from down (negative) to up (positive), and the Z axis goes from the screen (negative) into the scene (positive); these are the three axes (plural form of axis, pronounced 'ack-sees') of the Caretsian cordinate system. At the center of the model is the origin, it is marked with 0 because at this point the values of X, Y and Z are all 0. The figure below shows the grid viewed from an elevated FPS camera:



6. Now add code to your project to display a cube so that your project displays both the grid and the cube. Run your project.

7. You will notice that the cube is created at the center of the grid, where the 0 marking is i.e. where x = 0, y = 0, and z = 0. This is because whenever the CreateModel method is used to create a model without specifying the intial position of the model, it is created at the origin of the scene. The figure below shows the position of the cube in the grid viewed from an elevated FPS camera:

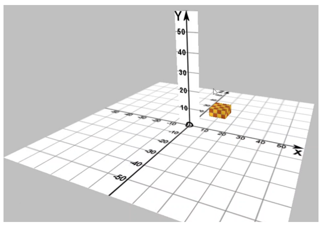


8. Now update your code by passing three values to set the initial position of the cube as shown below:

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

   This positions the model using 'Cartesian coordinates', with x = 10, y = 0 and z = 20. Run the project and see the new position of your cube.

   The figure below shows the position of the cube in the grid viewed from an elevated FPS camera:



9. Experiment with a few different x and z positions for the cube. Then try altering the y position. Keep all the values less than 50. Can you tell the length of each of the cube's sides?

10. From the scenes your program displayed and the scene snapshots shown above, can you tell the length of each of the cube's sides?

11. You should have noticed that the length of the sides of the cube is 10. This is clearer when you position the cube so that it sits on the floor of the grid and to the right of the origin using the code below:

```
cube = cubeMesh->CreateModel( 5, 5, -5 );
```

12. Add another cube to your program by declaring a new IModel* variable and putting a cube mesh model into it. Remember that you do not have to declare a new IMesh* variable and load a new cube mesh whenever you want to create an addional cube, you can create as many cube models as you want with the cube mesh you have loaded already (as long as you have the IModel* variables to put them in to).

13. See if you can stack the two cubes you have on top of each other.

14. Add and display a sphere in your program.

15. Now stack two cubes on the sphere.

16. Save your project at this point by selecting 'File->Save All'

Movement

1. Create a new TL-Engine project and name it **Lab1_Movement_Project**. Open the project.

2. Write code to load and display a grid and a cube (copy some of your code from the previous project).

3. In the game loop of our program, we can add code to update the scene e.g. 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.

4. 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'.

5. Now change the sign of the parameter passed to the rotation method i.e. change it from 0.05 to -0.05. Run your project and note the difference.

6. Hopefully you noticed a change in the cube's direction of rotation. The change occured because the direction of rotation/movement is determined by the sign of the parameter passed to the movement/rotation function.

7. 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 written as comments 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 );
   ```

8. Now write the appropriate code to rotate the cube around the other two axes.

9. Try to use the MoveX function to move the cube to the left.

10. Now change the code so that the cube moves to the right.

11. Save your project

Further Positioning

1. There are more functions available for positioning e.g. for setting the position of an object, and for getting the position of an object.

```
// 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 ); // e.g. cube2->SetZ( 7.5 );

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

// 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(); // e.g. zpos = sphere->GetZ();
```

   Note that the 'Get' functions calculate a float value (indicated by the word ‘float’ on the left), look carefully at the example usage.

2. Using the appropriate method from those provided above, update your code so that your camera is positioned at (0, 0, -50).

7. Advanced Exercises

Are you looking for extra exercises? Why don't you try creating mega structures using cubes and spheres.