University of Central Lancashire
UCLan

Computing Taster Day

Session 1: Games Concepts (60-75 minutes)

This session will introduce students to UCLan's in-house 'TL engine' (Teaching and Learning Game Engine). The environment is special, as it allows you to learn the basics of game development in a learning environment. You will use our TL engine to develop a 2D game, writing and executing C++ code where you will begin to understand some core game-algorithms. This session will introduce students to software engineering concepts with a game development focus.

Instructions:

Your tutor will introduce you to the TL Engine and the template code via a short demo.

This worksheet will then guide you through various challenges, to help you achieve the following:

1. Explore and understand template code,

2. Adjust the code to achieve new outcomes,

3. Write your own code to improve functionality.

If you cannot access the template code on your allocated PC, following these instructions:

You may access the resources for today's session from the following SharePoint location:

Taster Day Activity

Start by clicking the file bearing extension .sln (solution). 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.

Look for a file that has a name ending in '.cpp' (this is a C++ source file) and click to open it.

The project will compile and run using the TL Engine. A console window will appear and tell you what is happening, then a second blank grey window appears. This second window is a 3D engine showing an empty scene.

Finally, should you get stuck, there is a short introduction to C++ and the TL Engine at the end of this worksheet.

We hope that you enjoy the activity and your taster day at UCLan!

Note: A big thanks to Dr Zubair and Dr Mitchell for the design of assets and their work on UCLan Games modules.

Challenge 1: Adding a 'test' camera to the Game Scene

Observe the following lines of code within the 'initialisation' section of the template code:

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

ICamera* myCamera;
myCamera = myEngine->CreateCamera(kManual);
myCamera->SetPosition(0, 0, -130);
```

The first line declares a variable named 'myCamera' that will be used hold a camera. In English, the code is saying: "Use myEngine to create a camera of type 'Manual' and put it in the variable myCamera". The camera is initially positioned in a static position (approximately 130 units 'above' the scrolling platforms of the game scene).

1. Start by adding the code below to the bottom of the 'game loop' section (i.e., Place it above *"}//end loop"*) .

```
//basic 'test' camera controls
	cameraPan = abs(myCamera->GetZ()) / 1000;

	if (myEngine->KeyHeld(Key_X1)) {
		myCamera->MoveX(cameraPan);
	}
	else if (myEngine->KeyHeld(Key_X2)) {
		myCamera->MoveX(-cameraPan);
	}//end if

	if (myEngine->KeyHeld(Key_Y1)) {
		myCamera->MoveY(cameraPan);
	}
	else if (myEngine->KeyHeld(Key_Y2)) {
		myCamera->MoveY(-cameraPan);
	}//end if

	if (myEngine->KeyHeld(Key_Z1)) {
		myCamera->MoveZ(cameraPan);
	}
	else if (myEngine->KeyHeld(Key_Z2)) {
		myCamera->MoveZ(-cameraPan);
	}//end if

	if (myEngine->KeyHit(Key_Escape)) {
		myCamera->SetPosition(0, 0, -130);
	}//end if
```

2. Test your code to operate the camera (F5). The code above should generate a few errors (this is expected).

The code template uses placeholders (X1, X2, Y1, Y2, Z1, Z2) to identify where the required key 'identifiers' are expected to be defined (e.g., The 'up' arrow key has the key identifier 'Key_Up').

For this example, we will use keys associated with a traditional keyboard input for games (known as 'WASD'). This means that we use the 'W' key to tell the camera to move to the 'left'. The key identifier for this is simply 'Key_W'.

3. Add the correct key identifiers to the above code to add the following controls to the camera: - Move left; - Move right; - Move up; - Move down; - Zoom in; - Zoom out. Save your work (using the shortcut 'CTRL+S').

4. Note the final 'if' statement listens for a key press (rather than key 'hold'). This 'Escape' key allows the camera to 'snap back' to the default position defined above (once testing is complete). Experiment with the different methods *'KeyHit* and *'KeyHeld'* for your camera operation. Which do you prefer?

Challenge 2: Add some enemies and test for collisions.

1. Add the following code to the bottom of the 'initialisation' section (see above). Consider what this code does.

```
	//add enemy objects
	IMesh* eneMesh = myEngine->LoadMesh("enemyCube.x");
	IModel* enemyCube1 = eneMesh->CreateModel();
	IModel* enemyCube2 = eneMesh->CreateModel();

	//attach enemies to background
	enemyCube1->AttachToParent(background1);
	enemyCube2->AttachToParent(background1);
	enemyCube1->SetLocalPosition(0, 0, 0);
	enemyCube2->SetLocalPosition(0, 0, -40);
```

There are two errors in the above code, which we will now correct:

2. Both enemy 'cubes' are currently visible on the screen at the same time. Enemies should instead be positioned in 'sequence' so that only one enemy cube is visible to the player at any one time). How may this be corrected?

3. The enemies are positioned incorrectly within the scene. Enemy 1 appears to 'sink' within the background, while Enemy 2 appears to 'float' above the platforms (hence is out of reach from the player).

Study the composition of the code and identify which parameter within 'SetLocalPosition' (0, 0, 0) corresponds to the X ('horizontal'), Y, ('vertical'), and Z ('height') axis. Correct these values for both enemy cubes so that the enemies spawn 'on top' of platforms (i.e. the same position as 'playerCube' in respective 'Top' and 'Bottom' states).

Hint: cubes measure 20 units in height, platforms measure 20 units, backgrounds measure 120 units (in height);

4. Add the following code to the game loop:

```
//COLLISION DETECTION
	float x1, y1, z1, x2, y2, z2;

	x1 = playerCube->GetX() - enemyCube1->GetX();
	y1 = playerCube->GetY() - enemyCube1->GetY();
	z1 = playerCube->GetZ() - enemyCube1->GetZ();

	x2 = playerCube->GetX() - enemyCube2->GetX();
	y2 = playerCube->GetY() - enemyCube2->GetY();
	z2 = playerCube->GetZ() - enemyCube2->GetZ();

	float collisionDist1 = sqrt(x1 * x1 + y1 * y1 + z1 * z1);
	float collisionDist2 = sqrt(x2 * x2 + y2 * y2 + z2 * z2);

	if (collisionDist2 < 20 || collisionDist1 < 20)
	{
		displayText = "Game Over";
kGameSpeed = 0;
playerAlive = false;
	}//end if
```

The code implements 'sphere to sphere collision detection'. The code uses spheres of radius 20 as bounding volumes for playerCube and the enemies. The code calculates the distance between each enemy's and playerCube's bounding spheres. If either of the distances is less than 20, then there is a collision. Twenty is the sum of the radius of the enemy and playerCube bounding sphere. This code is required to implement the 'difficulty' in avoiding the enemies.

Challenge 3: Increase the game speed.

1. Study the 'initialisation' section of the code. Can you identify which variable controls the game speed? This directly affects the game's difficulty (slower speed = easier play). Experiment with different values for this floating point (decimal) number. Try the following: -0.05, -0.1, -1.0 (good luck!). Why must this value be a negative number?

Now you have the 'default' positions for the enemy cubes, we may further increase the game's difficulty by randomising their 'spawn' state (either 'Top' or 'Bottom', with no predictable pattern).

2. Identify the code functionality for 'scrolling background and game difficulty' (movement of background objects to simulate player movement). Add the following line of code below the 'SetPosition()' method call for 'background1':

```
rSpawn = (rand() % 80) - 40;
```

Can you describe the function of this line of code? Can you identify the range of numbers which is being sampled?

3. Noting that 'rSpawn' now contains the calculated height ('Y co-ordinate') for the enemyCube spawn, correct the corresponding 'SetLocalPosition' for 'enemyCube1' value to reflect this output (Hint: insert and modify the following line of code directly below that inserted above. Remember to replace '[X], [Y] and [Z]' with the correct co-ordinates.

```
enemyCube1->SetLocalPosition([X], [Y], [Z]);
```

4. We will now adjust the game speed when the player successfully 'dodges' an enemy. The trigger for this event is the point in time when the enemy cube is 'recycled' within the code (look for the line of code which repositions the object to the immediate 'right' of the camera). Add the following line of code below the 'SetLocalPosition()' method.

```
kGameSpeed += -0.01;
```

You have now implemented the correct functionality for 'enemyCube1'. Repeat this process for 'enemyCube2'.

Your 'scrolling platforms' conditional statement code should now look like this:

```
if (background1->GetX() <= -260) {
	background1->SetPosition(260, 0, 10);
	rSpawn = (rand() % 80) - 40;
	enemyCube1->SetLocalPosition(0, rSpawn, -10);
	kGameSpeed += -0.01;
}
else if (background2->GetX() <= -260) {
	background2->SetPosition(260, 0, 10);
	rSpawn = (rand() % 80) - 40;
	enemyCube2->SetLocalPosition(0, rSpawn, -10);
	kGameSpeed += -0.01;
}//end if
```

5. Test your game with F5. How many enemy cubes can you avoid? Change the value of 'kGameSpeed' to adjust the incremental game difficulty to suit. Consider and set an appropriate value for the initialisation of 'game speed'.

Challenge 4: Change orientation of Game Scene and UI.

We will now adjust the game scene so that it resembles a more traditional 'endless platform' game (like Temple Run). This will require that the platforms scroll from the top to the bottom of the screen (hence the player will move 'left and right'). There are several adjustments which must be made to the code to achieve this, although this adjustment is essentially simply swapping the 'X' and 'Y' axis for any object or definition which references a position.

Part 1: The required changes are as follows:

Initialisation section:

1. Rotate the background objects 90 degrees around the Z axis (should occur before the enemies are added!). (Hint: you should use background[x].RotateLocalZ() for this, where [x] is the background to be rotated).
2. Change the initial position of the 'background2' object to appear above background 1 (not on the right). (Hint: this should occur when the model is created via 'CreateModel()', i.e., swap x and y co-ordinate here).

Game loop (player movement):

3. Change the method call for checking playerCube position to reference the X co-ordinate ('GetY' to 'GetX').
4. Change the method call for moving playerCube to reference the X-axis ('MoveY' to 'MoveX').

Game loop (scrolling background):

5. Change method call for moving the background objects to reference the Y-axis ('MoveX' to 'MoveY')
6. Change method call for checking background position ('GetX' to 'GetY') Hint: there are 2 instances of this.
7. Finally, change the 'recycle' position of background objects from 'right of screen' to 'top of screen'. (Hint: adjust the 'x, y, z' values of the 'SetPosition' call for each background. There are again two instances.

Note that you do not need to change any of the parameters for the positioning of the 'enemy' cube objects. Why do you think this is? (Hint: Remember that we adjusted 'enemyCube' to become a child of the 'background' object).

Part 2: Finally, we will implement a score function. Notice that the template contains the following lines of code:

```
//library for drawing text
#include <sstream>
using namespace std;

string displayText = "Hello";
IFont* myFont = myEngine->LoadFont("Arial", 48);

//display dummy UI text
stringstream outText;
outText << displayText;
myFont->Draw(outText.str(), 0, 0, kWhite);
```

This code allows us to define a very simple user interface (UI) to communicate the game state to the player. We have used this so far to alert the player to a collision (using the UI text 'Game Over', and by disabling the keyboard input).

1. Define a new 'integer' variable named 'score' within the 'initialisation' section and assign the value '0';
2. Identify at which point within the game loop the player should score a point (and hence increment score). (Hint: this should ideally be at the same point where the game's difficulty is increased).
3. Write the code to increase the score by 1. Consider the correct operator for this.
4. Finally, set 'displayText' to the value of score, so that the score will be presented via the UI

Hint: you will need to convert score to a 'string' type using the 'to_string()' method. Your code to assign the correct value to 'displayText' must look something like the following, although you must consider where to place this line...

```
displayText = "Score: " + to_string(score);
```

Congratulations, you have completed the activity. Consider how you may allow the player to restart the game without exiting the application (Hint: consider using a 'key hit' to reset the score and re-enable scroll/ the player control). This should ideally be presented as a conditional 'if' statement, which should occur once every game loop).

Optional: Some light reading to help you understand the TL-Engine and Template code.

Intro to C++

C++ is a programming language that gives programmers high level control over system resources and memory. It is one of the world's most popular programming languages and the language used to build most big console and PC games. This activity will use the TL-Engine to introduce you to the features and concepts of C++ that are standard in the games development industry.

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 - decimal numbers

bool check; // a boolean variable named check that can only take the values 'true' and 'false'

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

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 3D engine*

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 '*'. Further reading on both custom types and pointers can be found online. For now, just remember that 'I3DEngine*' is a type just like an int or a float, but specific to the TL-Engine.

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 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) 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'.

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; some don't. 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.

The TL-Engine Template Code

Open the example project in Visual Studio 2020 and open the file with the extension '.cpp' using Visual Studio's "Solution Explorer" to the right of the screen. The template code 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.

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. These opening lines must begin all programs using the TL-Engine, you do not need to understand their detail, just make sure they are included.

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*'. 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. Any resources you want to load to your game, should be loaded after this line of code. All of the meshes (compositions) and textures (visual skins) required for our activity are already loaded for you.

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

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

The game loop's ability to perform several iterations every second and display updated frames creates the animation seen when playing games.

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 do not 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, and so this line is important.

Additional media folder (should you reach the end of the worksheet).

End.