Lab 5

##### Game States

#### Exercise 1

* Create a new working directory (folder) and start a new TL-Engine project.
* Create a camera at location ( 0, 0, -40 ). You don't need to move the camera at all today so I would actually suggest that you set camera type to kManual and not bother with any camera controls.
* Download the model and texture files from the website and unzip them into your working directory.
* Load and create a model using "Stars.x". This will be your skybox.
* The model you've just downloaded is called "particle.x". Load and create a model called "particleModel". Place it at the origin ( 0, 0, 0 ). Compile and run the program. If you've placed the camera correctly then you'll see a hexagonally shaped light in the middle of the screen. This model has been created using a technique called texture blending and this makes the model semi-transparent.
* Include a line of code within the main game loop to make the model rotate slowly around its local Z axis.

It is possible to remove models as well as create them. You use the mesh in order to create a model. In a similar fashion you use the mesh to remove a model. The following code uses a mesh to remove a model:

particleMesh->RemoveModel( particleModel );

The removal of a model means that it is completely destroyed. The model will disappear from view. It is no longer possible to use the model since it no longer exists and, indeed, if you tried (for example) to move the model after removal you would generate a real time error.

* Get rid of the rotate statement from your program if you've got one (otherwise the program will crash after the next addition).
* Include an *if block* within the main game loop so that when the Return key is **hit**, this removes the particle model. This should generate a real-time error and cause the program to crash. The code for the Return key is *Key\_Return*.

Although the model has been destroyed the IModel variable still exists. The IModel variable is a particular kind of variable called a pointer. A pointer variable is declared using a asterisk (\*). The pointer still exists but, with the removal of the model, it no longer has anything to point at. However, the IModel variable can be re-used. It can be used to store another newly created model. For example, the code below serves no real purpose but it does show this process of removal and reuse.

IModel\* particleModel = particleMesh->CreateModel( ); // declare variable

// and create model

particleMesh->RemoveModel( particleModel ); // remove model

particleModel = particleMesh->CreateModel( ); // create model

I want you now to modify the program so that when you hit the Return key if the model exists the model is removed. However, if the model does not exist then you need to create it. *Remember you do not need to load the mesh each time, merely use it to remove and create the model.*

* Declare an integer variable called particleState and set its value to 0. The variable should be declared **outside** the game loop.

The logic of the program is as follows:

* + If the Return key is hit and the particleState is 0, then the particle model is removed and the particleState is set to 1.
  + If the Return key is hit and the particleState is 1, then the particle model is created and the particleState is set to 0.

The code to implement this looks as follows:

if( myEngine->KeyHit( Key\_Return ) )

{

if( particleState == 0 )

{

particleMesh->RemoveModel( particleModel );

particleState = 1;

}

else

{

particleModel = particleMesh->CreateModel( );

particleState = 0;

}

}

In this fashion you can track the state of your program. In this particular example you are keeping track of whether or not the particle model exists. Furthermore, you are using the state of the particle to determine what action the program needs to take: whether to create the particle or to remove it. States are used extensively within game programming and throughout software engineering in general.

In this next exercise I want you to change the texture of the particle. The textures you have been provided with are all of different colours so changing the texture will cause the particle to change colour. Remember that in order to change the texture you use the SetSkin() method:

#### Exercise 2

* Reuse the integer variable called particleState. Set its value to 0.
* Everytime the Return key is hit the texture of the particle is changed to the next texture in sequence. When the last of the textures has been reached the first one should be re-used. In this fashion the program will cycle around the colours as the Return key is hit.
* Since there are 7 textures the value of particleState will range from 0 to 6.

Use a sequence of "else if" commands. Alternatively you could consider using a "switch" command.

In this session we will review the work we have covered so far with the TL-Engine. Below is a set of exercises that build up a simple prototype. Work your way through them, but don’t worry if you don’t finish in this session. Refer to your previous work and notes to remind you how to approach each question.

#### Matchbox Racer

This exercise builds up a simple car prototype - **don’t** start a new project for each question, keep adding to the program to reach a complete prototype by the end. The models you will need for this prototype are:

“Matchbox.x”, “Floor.x”, “Match.x”, “TwoPence.x” (and “Grid.x” for measuring)

Type the names carefully – if you spell them wrong, your program won’t work.

#### Instructions:

1. Start a new TL-Engine project. Start by loading and creating a ‘Matchbox’ model and an FPS camera.
2. Approximately what are the length, width and height of the matchbox model?

Length: \_\_\_\_\_\_\_\_\_\_, Width: \_\_\_\_\_\_\_\_\_\_, Height\_\_\_\_\_\_\_\_\_\_

(Hint: use the “Grid.x” model to measure)

1. Load and create a ‘Floor’ model and add keyboard controls to move the matchbox forwards and backwards on the floor.
2. Load and create a ‘Match’ model, set its parent to be the matchbox. If you do this correctly the match will follow the matchbox when it is moved with the keyboard controls.
3. Move the match to the front end of the matchbox – it will be the axle for the matchbox car. Then create another match model, parent it to the box again and move it to the back end. Your car will have two axles.
4. Load and create a ‘TwoPence’ model. Set its parent to one of the axles (not the matchbox) and position it to one end of the axle to act as a wheel. Repeat this with three more ‘TwoPence’ models so your car has all four wheels. Here is an image of what you should have created:



1. Now add code to rotate the axles when the forward and backward keys are pressed (this will rotate the wheels automatically if you parented them correctly). You can either guess or calculate the appropriate speed to rotate the wheels. (Hint: Use RotateLocalX)
2. Add two more controls to ‘steer’ the front axle left and right. Hint: what are the 'local' coordinates of the axle and what are the 'world' coordinates?
3. Use a **float** variable to keep track of the steering position. It should be initialised to 0 (centred steering) and should be updated when the steering controls are pressed: e.g. if you rotate the steering 0.1 degree right then add 0.1 to this value, if you steer 0.1 degree left, then subtract 0.1 from this value.
4. Set limits on the amount of steering allowed. If the steering variable becomes greater than 20 degrees then don’t allow any more right steering, if it becomes less than –20 degrees don’t allow more left steering.
5. Now make the car turn with the steering when it moves: when the car moves, it should rotate (in Y) by an amount based on the steering value. You cannot rotate by the steering value directly the steering will be too fast. Scale it down by multiplying by around 0.01 – you will need to experiment to get the best results.
6. *Optional:* Create a chase view camera and put a few other objects in the scene to complete this simple driving prototype.