CO1301: Games Concepts
Lab 12 - Dot Product

1. Introduction

1. In this lab, you will be using dot product to to calculate whether one object (player) is within the field of view of another object (drone).

2. This worksheet introduces a number of new concepts. However, take it steady and you will have no problems with it.

3. The idea of this exercise is that we want to allow the player to sneak up behind the drone, and that is possible because the drone cannot see the player if the player is behind it.

2. Setting Up

1. Create a new TL-Engine project called "Lab12_DotProduct_Project".

2. Start by creating a kManual camera at (0, 10, -15). Then rotate the camera 20.0 about its local-X axis.

3. A set of media files are available on BlackBoard. You need to use the floor.x mesh to create a floor model, drone.x to create a drone model, and either sierra.x, casual_A.x or bikegirl.x to create a player model. Set up the models as follows:

   - Create the floor model at the origin
   - Create the drone model at (0, 1, 5) and **scale it down to 0.5 its original size (use the Scale() method)**.
   - Create the player model at (0,0,-5)

   When you run your program you should see player near the bottom of the screen and facing the drone; the drone should be positioned around the center of your screen and facing forward.

4. Implement player movement using W, A ,S and D keys, e.g.

```
if( myEngine->KeyHeld( Key_W ) )
{
    player->MoveLocalZ( kPlayerSpeed );
}
```

5. In the next section we can find out whether the drone can see the player or not by calculating a value that tells us whether the player is behind the drone or not. The value is calculated by taking the dot-product of the vector between the drone and the player, and the drone's facing vector (which tells us where the drone is facing).

3. Calculating and Displaying the Dot Product

1. The steps that need to be completed to find out whether the drone can see the player are:

   - Calculate the vector from the drone to the player.
   - Find out the facing vector of the drone.
   - Calculate the dot product between these two vectors.
   - Perform check:
     - If the dot product is greater than 0 then the angle between the two vectors is less that 90 degrees, if the dot product is less than 0 then the angle is greater that 90 degrees, and if the dot product is 0 then the angle is exactly 90 degrees.
     - Therefore if the dot product is less than 0 then the player is behind the drone and not visible to the drone.

2. To calculate the vector from the drone to the player, subtract the location of the drone from the location of the player. This should be implemented in the game-loop, and the x, y and z components of the vector should be stored in appropriately named float variables e.g. drone2PlayerVx, drone2PlayerVy and drone2PlayerVz.

3. To calculate the drone's facing vector, you have to carry out some work with the drone's matrix. We haven't covered matrices yet, so a function that will calculate the facing vector from a given model is provided below:

```
void facingVector( IModel *model, float &x, float &y, float &z)
{
    float matrix[16];
    model->GetMatrix( matrix );
```

```
    x = matrix[8];
    y = matrix[9];
    z = matrix[10];
}
```

Add the function to your code so you can call it to calculate the drone's facing vector.

4. Assuming that the model of the drone is called "drone" and we have three floating point variables ready to store the facing vector of the guard:

```
float facingVx;
float facingVy;
float facingVz;
```

You would then call the function like this:

```
facingVector( drone, facingVx, facingVy, facingVz );
```

Note that the function does not return any value, but is able to update the values of facingVx, facingVy and facingVz as they are passed by reference.

5. Now write a function that will calculate the dot product of two vectors V and W. The function should take in as parameters the componenets of V and W, and return their dot product (a float). The function declaration (or function prototype) looks like this:

```
float dotProduct( float Vx, float Vy, float Vz, float Wx, float Wy, float Wz );
```

Add this before your program's main function.

6. Below the closing brace of the main function, define the dotProduct function. You need to:
    - write the function declaration without the ending semi-colon
    - add curly brackets (braces) and place the body of the code within the curly brackets.

```
float dotProduct( float Vx, float Vy, float Vz, float Wx, float Wy, float Wz )
{
    // body of code here
}
```

7. The function prototype specifies that you need to supply the function with two vectors. The body of the function needs to calculate the dot product of the two vectors. The dot product is calculated using the following formula:

$v \cdot w = (v_x w_x + v_y w_y + v_z w_z)$

8. Assuming the facing vector of the drone be vector v and the vector from the drone to the player be vector w, use the dotProduct function to find v·w. Assign the result to a float variable named vDotW.

```
float vDotW = dotProduct( facingVx, facingVy, facingVz, drone2PlayerVx, drone2PlayerVy, drone2PlayerVz )
```

9. Finally apply the following test:
    - If vDotW > 0 then the angle is less that 90 degrees, if vDotW < 0 then the angle is greater that 90 degrees, and if vDotW == 0 then the angle is exactly 90 degrees.
    - Therefore if vDotW < 0 then the player is behind the drone and not visible to the drone.

10. Display the dot-product and visibility of the player on the screen:
    - At the top left of your screen, display the the dot product calculated at every frame in the format: "Dot-Product: 0".
    - At the top right corner of your screen, display the visibility status of the player. If the the dot product is less than 0, display "Visibility: Hidden" else display "Visibility: Visible".

11. Move your player around and note how the dot product and the visibility status change.

4. Extra (not advanced)

1. Update the code in your game-loop so that both the drone to player and drone facing vectors are normalised before the dot product is caluclated. Check out Lecture 6 and Lab 7 if you need a reminder on normalising vectors.

   Can you see a difference in the values of the dot-product displayed by the program?

2. Add code in your game-loop that rotates the drone about it's local y-axis. This means you will have to keep moving the player to avoid being seen by the drone.

3. Extend this program into a game; the game's objective is to deactivate the drone.

    - The player needs to touch the drone to deactivate it, therefore implement collision detection between the drone and the player (use sphere to sphere collision detection, you can reuse the function from last week's lab). Take the radius of the player's bounding sphere to be 0.2f, and that of the drone's bounding sphere to be 1.0f.
    - If the player collides with the drone, resolve the collision to make sure the player is not stuck inside the drone and display an appropriate victory message on the screen (centered vertiaclly and horizantally).
    - The player looses if seen by the drone, therefore if the player is within the drone's field of view a GameOver message should be displayed on the screen (centered vertiaclly and horizantally).
    - Right now the player is within the drone's field of view if the dot product is greater than 0. Is that realistic? Can you have a more realistic field of view?