HONG KONG INSTITUTE OF VOCATIONAL EDUCATION

Laboratory 5: Simple 2D Game – Phase IV (User Input and Text Output)

Module Intended Learning Outcome (#2, #5):
On completion of the module, students are expected to be able to:

- develop 2D and 3D graphics programs for general gaming purposes;
- identify and apply appropriate user input methods for graphics and/or game programs

Lesson Intended Learning Outcome:
On completion of this lab, students are expected to be able to:

- Create a simple game with user interaction for Windows using MonoGame and Visual Studio.
- Compile, debug and execute a game program using MonoGame and Visual Studio.

In this lab, we will allow player to use keyboard to control your running man. Also, we will count the score and life to complete your first simple 2D game.

TASK 1 – Move Your Man

1. Modify the initialization part of the running man so that it won't move when the game starts (i.e., velocity.X = 0.0f).
2. Declare a public constant to store the speed of the man in the **RunningMan** class.
   ```
   public const float MAN_SPEED = 4.5f;
   ```
3. Add a private function ***UpdateInput()*** method into your *Game1.cs* file to handle the keyboard input. The function should have the following header:
   ```
   private void UpdateInput()
   ```
4. In your ***UpdateInput()*** function, you should do the followings:
   i) Get and store the **KeyboardState** information from your ***Keyboard*** object.
      ```
      KeyboardState keyState = Keyboard.GetState();
      ```
   ii) If "Escape" key is pressed in your ***keyState***, exit the program by calling **`Exit();`**
   iii) If "Left" key is pressed, set the X velocity of the running man to - **MAN_SPEED**.
   iv) If "Right" key is pressed, set the X velocity of the running man to **MAN_SPEED**.
   v) If "Up" key is pressed, set the X velocity of the running man to 0.0f.
5. Call ***UpdateInput()*** function in your ***Update()*** function.
6. Inside your **RunningMan** class, the ***Update ()*** function will be updated so that the frame number will only be updated if the velocity of the running man is not 0.0f.
   ```
   if (velocity.X != 0) { ... }
   ```
7. Further modify ***Update()*** function in the **RunningMan** class so that the running man will warp to the right when it comes to the left side of the screen.
8. Compile and run your program. Observe that the man will only face to right.
9. Add a ***SpriteEffects*** variable **direction** in your **RunningMan** class and modify your ***Draw()*** function to make the running man always run towards.
   ```
   // ADD THIS LINE OUTSIDE DRAW METHOD
   SpriteEffects direction = SpriteEffects.None;
   // ADD THE FOLLOWING LINES INSIDE DRAW METHOD, USE APPORPRIATE DRAW() FUNCTION
   if (velocity.X < 0) direction = SpriteEffects.FlipHorizontally;
   else if (velocity.X > 0) direction = SpriteEffects.None;
   ```

TASK 2 – Counting and Displaying your Score and Life

First, we start with counting your life and score.

1. Add two **int** variables to store the score and remaining life in **Game1.cs**. Initialize them to 0 and 5 respectively.
2. Add 1 to score when a rock hits the ground. In order to do this, you will need to let your game know that the rock has touched the ground. Add a public **boolean** variable in your **Rock** class so that it will become *true* when the rock hits the ground. In your ***Update()*** function in **Game1.cs**, update the score and reset the variable in your rock object to *false*.
3. Reduce life by 1 when collision occurs in your ***Update()*** function in **Game1.cs**.
4. Stop updating your game if the remaining life is 0.

Now, your program is counting the score and life. The next thing to do is to put the numbers on the screen in order to make it known to your player. In order to draw text on the screen, you need to use Sprite Font.

1. In your MonoGame Content Pipeline tool, select the **Content** folder. Click right mouse button to add a new item. Select **Sprite Font** in the dialog. Give it a reasonable name.
2. A spritefont file will be added under your Content folder in the Content Pipeline tool. Double-click it and use your text editor to open the file. The spritefont file is an XML file. You may like to change the default font and font size in this file.
3. After adding the **Sprite Font**, remember to save and rebuild your **Content** project.
4. In your *Game1.cs*, add a variable to store the sprite font in the module level.

```
SpriteFont font;
```

5. In the given ***LoadContent()*** method, initialize the font object.

```
font = Content.Load<SpriteFont>("MyFont");
```

6. You can draw your text message to the screen using ***DrawString()*** method in your ***SpriteBatch*** object with the specified sprite font.

```
spriteBatch.DrawString( font, // sprite font to be used
      "Score: " + score,                // text message
      new Vector2(20, GraphicsDevice.Viewport.Height - 30), // X-Y
position
      Color.White);                     // text color
```

7. One useful method in ***SpriteFont*** class is ***MeasureString()***, which will return the width and height (in number of pixels) of a given string print on the screen using the sprite font.

```
string message = "Life Remain: " + life;
spriteBatch.DrawString(font, message,
      new Vector2(
      GraphicsDevice.Viewport.Width - font.MeasureString(message).X -
20,
      GraphicsDevice.Viewport.Height - 30), Color.White);
```

   MeasureString() method returns a ***Vector2*** representing the width and height of the bounding rectangle of the text

8. Try adding a "Game Over" message near the center of the screen when life == 0.

TASK 3 – Collision Detection Delay

You may notice that the life of the running man often reduces to zero immediately when the man is hit. The reason is that we check for the collision in each and every frame. Each collision usually occurs in a time span of more than a few frames. All lives are then reduced.

One way to work around it is to reset the stone once the running man hits it. It is easy and you may try it yourself later.

Another way to work it through is to allow the player to be invulnerable for a short period of time after each collision. We can store the time when the first collision is detected. Then, we check the time each time when a collision occurs. If the time of next collision is very near to that of first collision, we just ignore the effect and let the player go. Once the grace period has passed, the collision affects the player's life again and a new time is stored for comparison.

1. Declare the following variables in the module level:

```
private double lastCollisionTime = 0;
const int LIFE_MISS_DELAY = 1000;
```

2. In each collision checking, add the following condition:

```
gameTime.TotalGameTime.TotalMilliseconds - lastCollisionTime >
LIFE_MISS_DELAY
```


The condition takes the difference of the current game time and the last collision time. If the difference is larger than the specified delay, the collision will take the effect.

3. Remember to update the last collision time each time when a life is reduced.