HONG KONG INSTITUTE OF VOCATIONAL EDUCATION
Laboratory 4: Simple 2D Game – Phase III (Collision Detection)

Module Intended Learning Outcome (#2, #3):
On completion of the module, students are expected to be able to:
- develop 2D and 3D graphics programs for general gaming purposes;
- apply animation effects in developing 2D graphics and/or game programs;

Lesson Intended Learning Outcome:
On completion of this lab, students are expected to be able to:
- Create a simple program for Windows using MonoGame and Visual Studio.
- Compile, debug and execute a game program using MonoGame and Visual Studio.

In this lab, we will continue to create the simple 2D game, in which the player character will try to escape from the falling asteroids. This time, we will try to detect if the running man hits the asteroids. First, let us try the easier version: rectangle collision detection.

TASK 1 – Collision Detection – Rectangle

1. Add the ***CalcualteBoundingRectangle()*** method into your *Game1.cs* file. The method is given in your lecture notes.
2. Add a private method ***CheckCollision()*** to check if any rectangle collision occurs. The function should have the following header:

```
private bool CheckCollison()
```

3. In your ***CheckCollision()*** method, you should do the followings:
   i) Declare and initialize a ***Rectangle*** object ***personRect*** to store the bounding rectangle (translated to appropriate position) of your running man. Carefully select the width and height of your rectangle.
   ii) Use a *foreach* loop to check if the running man collides with each rock.
   iii) Inside the *foreach* loop, declare and initialize another ***Rectangle*** object ***rockRect*** to store the bounding rectangle (before translation and rotation) for each rock.
   iv) Declare and initialize a ***Matrix*** object ***transform*** that create the translation (center shifting and position reallocation) and rotation needed for the rock.
   v) Find the transformed bounding rectangle for your rock using ***CalculateBoundingRectangle()*** method.
   vi) Use ***Intersect()*** method in ***Rectangle*** class to check if the two rectangles collide each other.
4. Call ***CheckCollision()*** method in your ***Update()*** method. Stop all movement when a collision occurs.
5. Compile and Run your program. Observe the false collision between the game objects.

TASK 2 – Collision Detection – Pixel-Perfect

It seems not so reasonable for a hit to be counted for those false collisions. So, we need to have pixel-perfect collision checking.

1. Before you apply pixel perfect collision check, you need to grab the pixel data from the two images file first.
   - Inside both *Rock.cs* and *RunningMan.cs*, declare a public **Color[]** array variable ***data***.
   - After loading the images, create the array inside your LoadContent() function with the array size of (width x height) of your image. Call GetData() function to grab the pixel information:

```
data = new Color[texture.Width * texture.Height];
texture.GetData<Color>(data);
```

2. Add a private method ***PixelCollision()*** into your *Game1.cs* file. The function should have the following header:

```
private bool PixelCollision(
        Matrix transformA, Rectangle rectA, int widthA, ref Color[] dataA,
        Matrix transformB, int widthB, int heightB, ref Color[] dataB);
```

   Width of the whole sprite texture

   You may notice that the function header is slightly different from the method given (***IntersectPixels()***) in your lecture notes. A ***Rectangle*** object ***rectA***, which represents the rectangular region to be drawn on the screen (current frame in the animated sprite), is added so that we know which part of the sprite should be used for checking the collision.
3. The contents inside the ***PixelCollision()*** function is nearly the same as in the ***IntersectPixels()*** function in your lecture notes. The only thing that needs to be modified is the looping range of the nested for loop.
   - Outer for loop (***yA***) should loop through the range between ***rectA***'s top and bottom.
   - Inner for loop (***xA***) should loop through the range between ***rectA***'s left and right.
4. We won't call ***PixelCollision()*** in each and every case. ***PixelCollision()*** function is exhaustive and should only be called when we know a collision is likely to happen. Thus, we modify the ***CheckCollision()*** function to include the pixel-perfect collision detection after a rectangle collision occurs.
   - To call ***PixelCollision()*** in our ***CheckCollision()*** function, we should first construct the transform matrix for your running man – it consists of a translation matrix only.

```
Matrix personTransform =
        Matrix.CreateTranslation(new Vector3(rMan.position, 0.0f));
```

   - When a rectangle collision occurs, instead of just returning true, we need to check against ***PixelCollision()*** first.

```
if (personRect.Intersects(rockRect))
        if PixelCollision(...) return true;
```

     Of course, you need to fill in the parameters in calling ***PixelCollision()*** yourself.
   - If no rock in the array collides with person, return false for the result.
5. This time, we won't stop the movement while a collision occurs. Instead, tint your running man in red. You may need to add extra attributes in your running man class to finish the task.