Homework 2 - Your First MonoGame… Game

**Due Sunday, February 9th by 11:59PM**

Your task is to create a small-scale MonoGame game. This will help you to become accustomed to many of the techniques you’ll need for future projects, as well as the MonoGame Framework in general.

This is a fairly simple game, with three possible states: **Menu**, **Game** & **GameOver**. During the game, the player will be able to move their character around the screen in any of 8 directions (Up, Down, Left, Right and the diagonals). If the player moves off the screen, they will wrap around to the opposite side. The goal is to grab as many collectibles as possible before time runs out. If you collect them all, you move to the next level (which will contain even more collectibles). If the player ever runs out of time, the game is over and can be played again by returning to the main menu.

Once your game is functioning the way it should, you will add one (or more) additional features to your game. Exactly what you add is up to you.

# Task Overview

This is a brief overview of the tasks you must complete for this assignment. Specifics are given in the corresponding sections later in this document. ***Read the entire document before starting to code!***

* Set up an overall **Finite State Machine** for the major game states
* Create a **GameObject class** with two child classes:
  + Player class and Collectible class
* Implement **overall game logic**, like collisions, screen wrapping, level progression, etc.
* Implement **drawing** of appropriate game objects.
* Add one (or more) **additional feature(s)**, such as enemies, high score save files, etc.

# Content

This game requires at minimum 2 images (Texture2D): one for the player’s appearance, and another for a collectible’s appearance. If you would like to “spruce up” the look of the game, feel free to add background images.

You may use any assets you like – a few great places to get free and royalty-free assets are:

* <https://opengameart.org/>
* <https://www.gameart2d.com/freebies.html>

You may create your own art, but be aware of 2 things: creating your own art takes time away from the time you have to code, and you won’t earn credit for making your own art as this is a programming class.

Lastly, use a SpriteFont for your game’s instructions.

Create fields for these three (or more) assets, and load them in the LoadContent() method.

# Finite State Machine

The game will be composed of three states which are used to drive a finite state machine.

## States

The three states are:

* **Menu** – Displays a title and the instructions for starting the game
* **Game** – Allows the user to play the game, and displays the current level, score & timer
* **GameOver** – Shows the final level and total score, and instructions for returning to the menu

Create a public enum to represent the three possible states. Add a variable of your just-created enumeration as a field in the Game1 class; this variable will hold the current state.

## Transitions

To move from the Menu to the Game state, or from the GameOver state back to the Menu state, the player should press the Enter key. Once in the game state, the game will continue until the timer reaches or goes below zero.

## Implementing the Finite State Machine

Changing this enum variable’s states – or implementing the finite state machine – will occur inside of the Game1 class’s Update() method. Depending on the current state, the Update() method will simulate the appropriate portion of the game and check for the transition conditions. (You’ll read more about this in a few pages.)

Your Draw() method should not change the state whatsoever, but it should check the current game state to determine what to draw.

# Class 1: Game Object

The main objects of the game will be a **player** and multiple **collectibles**. Both of these types of objects share similar traits, like a texture and a position on the screen. Start off by making a GameObject class, which will serve as a base class for the Player and Collectible classes.

You may need to add the following using statements to your custom classes:

using Microsoft.Xna.Framework;

using Microsoft.Xna.Framework.Graphics;

## Fields

The GameObject class will need fields to store its current **texture** and its **position**.

Store the position as a **Rectangle**. This allows you to specify both a position and a size in one variable (which can be used in place of a Vector2 for drawing), as well as simplifies collision code later.

## Properties

Set up relevant properties for fields. You may find it useful to create extra X and Y properties, which directly get and set the X and Y values of the Rectangle field. It’ll save you some time later on.

## Constructor

Make a parameterized constructor that receives all data that a GameObject will need.

## Methods

Create an override-able Draw method. Method declaration is below.

## virtual void Draw(SpriteBatch sb)

Use the SpriteBatch object to draw itself. Note: Just call the SpriteBatch’s Draw() method, NOT Begin() or End(). Assume that Begin() and End() are called elsewhere.

# Class 2: Player

The Player class inherits from GameObject.

## Fields & Properties

It should have two fields to keep track of the player’s score: levelScore and totalScore. LevelScore is the score of the current level, while totalScore will keep track of the player’s total score across multiple levels (the score since starting the game). The level score will be displayed and reset with each level of the game, and the total score will be displayed on the game over screen. Be sure to create public properties for these fields, and give them initial values in the constructor.

## Methods

No extra methods are necessary in the Player class.

# Class 3: Collectible class

The Collectible class inherits from GameObject.

## Fields & Properties

Since collectibles can be gathered by the player and will disappear when the player intersects with them, they will need a boolean field to determine if they are **active** or not (as well as a property). Collectibles should start out active by default.

## bool CheckCollision(GameObject check)

Returns a boolean signifying whether or not the provided GameObject parameter is intersecting with this Collectible. Since every GameObject has a Rectangle property, use the Rectangle’s Intersects() method to determine if the two objects are overlapping. You only want to do this intersection test if this collectible is currently active. If it’s not active, simply return false (since we don’t care if the player intersects with an inactive Collectible).

## override void Draw(SpriteBatch sb)

When the Collectible is drawn, you should check to see if it’s active. If so, call the base class’s Draw() method and pass in the appropriate SpriteBatch parameter. Nothing needs to be done if the collectible is inactive.

# The Game: Fields & Helper Methods

Now that you have your extra classes set up, you can focus on your main Game1 class.

## Fields

You’ll need a Player field that represents the player, as well as a List<> of collectibles. Add an int field to represent the current level the player is on, and a double field for the timer. To simplify checking for single key presses, add two KeyboardState fields: kbState and previousKbState.

## void NextLevel()

The NextLevel() method will set up each level the player encounters. It should do the following:

Increment the current level by one, and set the timer to an appropriate value (start with 10 seconds, but feel free to adjust).

Update the player by resetting their LevelScore property and centering them in the window. Use the following to get the size of the window:

* GraphicsDevice.Viewport.Width
* GraphicsDevice.Viewport.Height

Get the collectibles ready by first clearing the list of collectibles. Then calculate how many collectibles the current level should have (start with 5 collectibles in the first level, and add 3 more per level). Finally loop to create them all with random positions around the screen, adding them to the List as you go. None of the collectibles should be off-screen, and all the entirety of each collectible should be visible – no hanging-off-the-edge collectibles allowed!

## void ResetGame()

The ResetGame() method will be used each time the player transitions from the Menu state to the Game state. It will set up the initial values for the game: Set the current level to zero, reset the player’s TotalScore property and call NextLevel() to set up the first level of the game.

## void ScreenWrap(GameObject objToWrap)

ScreenWrap’s job is to keep an object on the screen at all times. If the object moves off one side of the screen, it should appear on the opposite side. You’ll need to check the object’s X and Y positions against the 4 edges of the screen, and adjust appropriately.

## bool SingleKeyPress(Keys key)

The SingleKeyPress() method will take a parameter representing the key to check (one of the “Keys” enum values). It should return trueif this is the **first frame** that the key was pressed and false otherwise. To do this, check if the key is down in the current keyboard state and up in the previous keyboard state.

## void MovePlayer()

The MovePlayer() method will change the player’s coordinates upon the directional key presses. Move the player by a fractional amount each frame that one of the directional keys are pressed. You will need to play around to find the amount you like best.

# Game1’s Default Methods

Finish up the Game1 class’s default methods.

## Initialize

The Initialize() method should initialize values that are not dependent on content loading. This would include the timer and current level fields. The initial state of the game (your enum variable) can also be initialized here.

## LoadContent

The LoadContent() method should create the player, create the list of collectibles (which will be populated in the NextLevel() method) and set the initial state of the game. Remember that you won’t be able to set either a player’s texture or a Collectible’s texture until they are loaded in using the generic Load method.

## UnloadContent

Nothing is required to occur inside of the UnloadContent() method. However, if you decide to add in file IO to save the state of your game, final saving should occur here.

## The Game Loop: Update

Update() is where your game logic takes place. You’ll need to set up your finite state machine here, and only update the current state of the game. Here is an overview of the states:

**Menu State**Check for a single press of the Enter key using SingleKeyPress()**.** Once found, change the state and call ResetGame() to set up the initial level.

**Game State**Save the old keyboard state in the prevKBState field and get the new one. Adjust the timer by subtracting the elapsed seconds since last frame (use gameTime.ElapsedGameTime.TotalSeconds).

Process input to move the player around the screen by calling MovePlayer() and using ScreenWrap() to keep the player visible. After the player moves, check all of the collectibles to see if the player has hit them. Update them appropriately, and reward the player with an appropriate number of points.

Also determine if the player has run out of time, or if they should progress to the next level.

**GameOver State**Don’t do anything until the user presses the Enter key once. Once they do, swap to the menu state.

## The Game Loop: Draw

Draw() will draw the current state of the game. You should Begin() your SpriteBatch before checking the state and End() the SpriteBatch afterwards.

**Menu State**Draw the title of your game (feel free to make one up) and the instructions for starting the game (which key to press) as text. Feel free to use additional images to “spice up” your menu for the user.

**Game State**Draw the collectibles and the player. Be sure to put some text on the screen to let the player know the current level, their current score for this level and the timer. Here’s an easy way to format the timer:

* String.Format(“{0:0.00}”, timerVariable)
* That will return the value of the variable as a string, with exactly two digits after the decimal point

**GameOver State**Draw the phrase “Game Over” on the screen, as well as the last level the player reached, the player’s overall total score and instructions for returning to the main menu (the key the user should press).

# Additional Feature(s)

You are required to implement at least one additional feature in your game; this makes your game unique! Exactly what you implement is up to you, but it should be obvious to the player (and grader!)

For instance: An enemy that immediately ends the game when hit is obvious. A hidden level that requires a cheat code to activate isn’t obvious, unless you tell the user about it somewhere.

Here are some potential ideas for your additional feature:

* Enemies that move in specific patterns around the screen
* Special collectables that “run away” from the player
* Power-up collectibles that extend the amount of time per level, increases a player’s score by a large amount, grants the player invincibility, etc. Be creative!
* A high score that persists after the program closes (probably in a binary file) and is displayed to the user somewhere in your game.

Since this additional feature is unique to your game, you MUST include a documentation file within your MonoGame project folder explaining what your additional feature is. Here’s an example of an acceptable documentation file:

Erin Cascioli  
1/26/2020  
  
My additional feature is the inclusion of two special collectibles. These collectibles inherit from the Collectible class into the class SpecialCollectible. These special collectibles are NOT included as part of the required collectibles to move to another level.   
The first special collectible is an extra time collectible: collecting this grants the user an extra 3 seconds to complete the level. The second special collectible is a score increase collectible: the player’s score increases by 100 points. Both of these special collectibles appear different than the “regular” collectibles. The time special looks like a clock, and the score special looks like an exclamation mark.