Opening Remarks

This is a submission to the Ubisoft NEXT 2024-2025 programming track. The game is called ParGolf. In order to run the project, open GameTest.vcxproj in Visual Studio and run the code.

Gameplay

The game is made to be played by two people or with one person playing two different players. Currently, there are 4 levels for the game.

There are two players for the game, player 1 and player 2. They will be on different "golf" courses. Once player 1 shoots, it will swap to player 2 and they will shoot. It will alternate between player 1 and player 2 for turns. One cycle of player 1 shoot and player 2 shoot is one turn. In order to complete a level, the amount of turns you took must be less or equal to the target amount of turns which you can view on the bottom of the screen.

This is a collaborative game. You can notice there are coloured blocks on the screens which may be blocking the whole. On the other players' screen, there should be a key of the same colour. If that player hits the key, the blocks of that colour on the other player's screen will disappear. Using this mechanic, the players should be able to complete the course in the required amount of turns.

Overall Layout

GameTest.cpp

This file is where all the main code is. In the Init() function, the SceneManager is made and all the Scenes and their GameObjects of the scenes. In the Update() function, the Scenes are updated and they are swapped if necessary. In Render(), the objects of the current scene are rendered.

Classes

SceneManager

This class is responsible for managing the scenes and keeping them together. It is a Singleton (since there should only be one and everything should have the permissions to request to swap scenes if required). It is used in the Update() function of GameTest.cpp to swap the scenes.

Camera

The Camera class is used to display the scene. It is currently used to position the GameObjects in the correct place to be displayed. In the future, it can be modified to follow one GameObject so that it is always in the center of the Scene displayed on the screen.

Mouse

Mouse is a Singleton and it keeps track of the mouse movements. It also keeps track of whether or not the mouse has been clicked on the current update. In order to check if the mouse has been clicked, it keeps track of the previous clicked state and if the mouse has been let go of a click, it will be counted as a click.

Scene
The Scene class represents one screen of the game. They are initialized in the Init() function of the GameTest class. The Scene class has a list of GameObjects to store the objects on the screen and a Camera to help with displaying the scene. The Scene class has some basic functions including addObject() to add a GameObject to the scene, renderObjects() to print out all the objects onto the screen and updateScene(). The renderObjects() command displays all the objects and ensures that they have the correct rendering for their type. The updateScene() function loosely follows the Template design pattern. Since there are two subclasses for Scene (DisplayScene and GameScene), it will customize the updateScene() function for both by using updateSubScene(). However, in the skeleton for updateScene(), the Scene updates the Mouse Singletonand checks for collisions between the GameObjects and updates their physics. It will also call update() on all the GameObjects.

- GameScene: The GameScene class is used to run the golf levels. During its construction, the GameScene reads from the corresponding .txt file and adds it to the Scenes' GameObjects. It also notes down a GameObject for the golfball and the hole the golf ball should go to if required. It also holds other information such as the redo button, the home button and the other Scene it is paired with (for the other player) along with which player it represents. For the updateSubScene() function, it updates the golfball based on its physics, checking for collision with the key for the boxes and the line movement. The GameScene also provides a resetScene() function to reset the scene if required.
- DisplayScene: This is any scene that is not a GameScene (scenes that just display objects such as the start screen or the menu screen). There are no special functions for this Scene.

GameObject
The GameObject class represents an object for the game. All the objects on a Scene are GameObjects. It keeps track of the sprite, the collisions, the physics, height, width, x, y, and the state of the GameObject. It provides a virtual update function which checks the physics and updates the object if it needs to. There are a couple subclasses for the GameObject.

- Button: This class is a GameObject to represent a button. It provides two states, one for when the mouse is hovering and one for when it is not. It checks if the mouse is hovering by checking if the circular collider of the mouse is touching the collider of the button. If it is clicked, it will send a request to swap to the Scene that it is linked to.
- Tile: This class is used to add the tiles on the screen. This version of the class is used to represent the basic tiles of the screen, mostly the walls and blank green tiles. It gets the tiles from Assets/golfcourse.png and it maps the letters to numbers (just subtracting the character so a = 0, b = 1 etc.) It also adds the collisions as lines based on the tile (stored beforehand). For the tiles that are not basic tiles there are two other classes.
    - LockTile + UnlockTile: These tiles represent the unlocked tiles + the key to them on the screen. Similar to the Tile class, they create the tiles based off of previously determined values. They are all given a collider using lines in the shape of a box. They loosely follow

the Observer pattern. When an UnlockTile is hit, it will notify its corresponding LockTiles and they will become inactive. They are linked through the GameScene.

- Line: This class is used to represent a line on the screen.
- Text: The text class is used to represent text on the screen.

Physics:

Physics is a component that can be added to GameObject. Currently the only physics available is for when a circle collides with a line. If a circle collides with a line, it will be reflected across the line perpendicular to the line. This is done by using matrix multiplication with edge cases for a 0 xcomponent or ycomponen. The values for the Physics class are stored as polar coordinates. Therefore, the xcomponent and ycomponent to be added is recalculated using the polar coordinates every time. The physics only updates Dynamic objects.

Collision:

The collision class creates collisions for the objects so that we can check if two objects are colliding. There are currently three types of collisions: box, circle and line. Do note that not all the collisions have been written.

The square and circle collision is first calculated by checking if the circle is in line with the edges. If it is, it must be touching it above or below. If that is not the case, then the only place where it could be there and still touching is if it is touching the corners. To check for this, you can draw a circle with the corner as a point and see if the center of the circle is in there to mimic the circle touching the area around the corner.

If it is two circles colliding, you can check to see if the addition of the radii is greater than the distance between the center points.

For checking a circle and a line, we first check the endpoints. The collision for this is done by checking the distance between the point and the center of the circle. Then check to see if it is greater or less than the radius. Next, there are two edge cases: a vertical and horizontal line. For this, the shortest path is to just go across or down respectively. Therefore, check for those points on the line, and if it exists, check the distance between that point on the line and the center of the circle. Other than that, we can calculate the line perpendicular to the original collision line. This will give the shortest distance from the center of the circle to the line. Using the perpendicular line, we can find the corresponding point on the line and calculate the distance from that point and the center of the circle. Then we can compare it to the radius value again.

The values that the collision object stores are the shape, the height, the width, and the x and y offset from the center of the object.