## Homework # 3

Graphical Snake

Due by the date on the HW3 – Graphical Snake dropbox on MyCourses.

## Goals

Working with external APIs and pre-built DLLs. Creating middleware between two libraries.

## Overview

In HW2 – Gravity Snake, we make a small physics game using Box2D in which players try to move their avatar toward a target using physics. For this assignment, you will be taking the gameplay of HW2 and adding graphics to it using SFML. In order to accomplish this, and to help everyone start on even ground, a DLL (Dynamically Linked Library) has been provided in the HW3 starter code. This DLL contains functions that perform nearly all the HW2 - Gravity Snake behavior so that you do not have to write it yourself. Out of the box, the starter code is a functioning version of HW2 with some features (such as the scoring system) stripped out.

Your first task will be to convert the current game loop to work with the SFML while(window.isOpen()) loop. You will then need to create rectangles and circles to represent your walls, snake, and target using SFML. Once you have these shapes, you’ll need to devise a way to position them correctly within the window, and keep them positioned correctly as the snake and target move.

For the creative portion of this assignment, you will need to implement a way to score the game (this can be the same or different from your scoring system in HW2), and make some unique addition to make your game stand out. This could be custom graphics / textures, a menu, new gameplay elements, audio, etc.

## Details

In HW2 – Gravity Snake, you created the gameplay logic for a simple physics-based snake game. However, it was only playable through the console. For this assignment, you will be using the SFML library to display the gravity snake game graphically.

Starter code has been provided to you through MyCourses>Content>Homework>HW3 – Graphical Snake to make this assignment easier. The starter code provides a working version of HW2, although much of the logic is hidden away inside of a DLL. For details on how the DLL works, see below.

Using basic shapes (at a minimum), you will need to visually display all the gameplay elements of your Gravity Snake game. This means that all four walls that bound the world, the player’s avatar, and the target must all be visually displayed. You must draw the walls using some kind of sf::Shape. You cannot just use the edges of the window.

These visuals should line up well with the logic they are representing. This means that if your game has logically detected that the user is colliding with the target or a wall, it should look as if the two are colliding to the player.

The game should also display the controls to the user, as well as the score that they received after playing the game. These should be displayed using SFML, not by using printf() or cout to print to the console window.

Finally, you will be required to make your version of the game unique in some way. This could be by creating a graphical interface or menu, adding gameplay elements, creating a full physical body for the snake, adding audio, etc. The choice is up to you. Whatever you do MUST be well documented in comments at the top of your main .cpp file, or you will receive a deduction.

## Tips, Notes, and Resources

**The starter code must be run in x86 Debug mode!**

Working with Box2D and SFML

* Box2D and SFML use different coordinate systems. In Box2D, positive Y extends upwards, and negative Y extends downwards. In SFML, the top of the screen is Y=0, and positive Y extends downwards. Additionally, in SFML, the left edge of the screen is X=0, with positive X extending to the right.
  + You will likely need to create a function which can convert between Box2D and SFML coordinates every frame. Alternatively, you can attempt to use sf::view which can work like a camera. It is far more robust, but could also be overkill for this assignment.  
    <https://github.com/SFML/SFML/wiki/Tutorial:-Using-View>
  + Also note that in Box2D, the “position” of an object is read from its center. In SFML, the “position” of an object, by default, is its top right corner. Note that SFML shapes have a setOrigin() function which can move this point to elsewhere within the shape. For example, if I had a 10x10 square, I could move it’s origin to the center by calling square.setOrigin(5, 5) to offset the origin. [https://www.sfml-dev.org/documentation/2.5.1/classsf\_1\_1Transformable.php#a56c67bd80aae8418d13fb96c034d25ec](https://www.sfml-dev.org/documentation/2.5.1/classsf_1_1Transformable.php%23a56c67bd80aae8418d13fb96c034d25ec)
* You’ll need to refactor your game code so that it works within the while(window.isOpen()) loop. Note that you cannot put an additional while loop inside of this loop, because it will stop your game from rendering. Instead, use the loop to keep running your code and instead use something like an if() to check if your game is over.
  + Important: there is another while loop inside of the SFML { while(window.isOpen()) }. This handles window events, and you should not put your game logic inside of it. Leave it be.
* SFML, Box2D, and the DLL will create a number of unavoidable “false positive” memory leaks with the \_CrtDumpMemoryLeaks() function. As such, the function will be unreliable for this assignment. Instead, simply ensure that you are deleting anything allocated with the new keyword.

Using SFML

* To add text to your game, you will need a font file in your project. To get a font file, go to your computer's C drive (or whatever drive you have Windows installed on) and look for the "Windows" folder. Inside that folder you will find a "Fonts" folder. Find the font you want to use in that folder and copy it. Paste the .ttf file inside your project folder (aka the source folder, the place where your .vsxproj / .cpp / .h files are, not where the .sln is) and inside your project's Debug folder. The font files need to be in the Debug folder for your project to work. They need to be in the project folder so that they get sent through GitHub for when we grade them. From there, follow the SFML tutorial on text:

<https://www.sfml-dev.org/tutorials/2.5/graphics-text.php>

* This page has many tutorials about how to do most things in SFML: <https://www.sfml-dev.org/tutorials/2.5/>

## Working with the Starter Code

**Ensure you are running the starter code in x86!**

The provided starter code is a working version of HW2 – Gravity Snake. However, much of the functionality is obscured inside of something called a DLL (a Dynamically Linked Library). A DLL is a precompiled library of code that can be used by other programs. This means that you cannot see the actual implementations of the functions. We do this to maintain academic integrity, and to keep this assignment as streamlined as possible.

There is also a “Functions.h” and “Functions.cpp” that have been made for you. Within these, two functions currently exist. One for displaying to the console (which you will remove for this assignment), and one for handling input (which you may wish to modify for your special features / score system).

Through “#include <GravitySnake.h>” you get access to several functions provided by the DLL. It will be beneficial to know what sort of functions exist, and what they do. It’s also worth noting that you can actually open GravitySnake.h by putting your cursor inside of the include statement and pressing F12. This will give you full XML comment documentation for each function that exists. Similar information is documented below.

**DLL Functions:**

**void setTargetBounds(float minX, float maxX, float minY, float maxY)**  
This function takes in minimum and maximum positions for where the target can spawn. Later, when the target gets moved, it will be within these bounds. You should set this before calling initVariables().

**void initVariables(b2Vec2 gravity)**  
This function takes in a gravity vector, and sets up the physics world, target, and deltaTime variables. It should be called before any other functions except setTargetBounds().

**void releaseVariables()**This function frees all the memory allocated by the DLL, and should be the last thing called in your program.

**float updateWorldAndReturnDeltaTime(**)  
This function advances the physics world forward by the deltaTime timestep, and also returns the deltaTime in seconds as a float in case you need access to it.

**bool checkCollisionAndMoveTarget(float collisionDistance)**  
This function determines if the snake and target are close enough to “collide” based on the float parameter passed in. If the two are within the collisionDistance of one another, the function will randomly move the target and return true. Otherwise, it will return false.

**void applyForceToSnake(b2Vec2 force)**  
This function applies a force to the snakeBody’s center. Note that the function will scale the force by deltaTime, so you should not scale your forces beforehand.

**void createFloor(b2Vec2 center, float halfWidth, float halfHeight)  
void createCeiling(b2Vec2 center, float halfWidth, float halfHeight)  
void createRightWall(b2Vec2 center, float halfWidth, float halfHeight)  
void createLeftWall(b2Vec2 center, float halfWidth, float halfHeight)**These functions create the bounding boxes of the world. Keep in mind that the floats provided are halfWidth and halfHeight, and so the boxes will be twice as big as the numbers provided. For example, createFloor(b2Vec2(0.0f, -10.0f), 10.0f, 5.0f) would create a 20x10 box centered at (0, -10) giving us an “effective” floor of x=-5.

**b2Body\* createSnake(b2Vec2 center, float halfWidth, float halfHeight)**This function creates the dynamic snake body. Note that the halfWidth and halfHeight work in the same way as the functions above. This function returns a b2Body\* to the snakeBody so that you may reference it elsewhere in your code. You should not call delete on this pointer, as it will be deleted for you by the DLL.

**b2Vec2 getTargetPosition()**  
This function returns a copy of the target’s position.

## Rubric

|  |  |
| --- | --- |
| Objective | Maximum Points |
| Box2D and SFML properly linked to the project without any errors. | 2 |
| Makes use of logically implemented functions defined in the proper files. | 3 |
| Prompts user of how to play the game before the game begins. These instructions must be in the SFML window, not in the console. | 3 |
| Properly utilizes Box2D to calculate all the physics in the game, including gravity and bounding the game area with walls. | 2 |
| The game is playable, and can be ‘beaten’ by collecting two or more targets. | 2 |
| The game properly uses SFML to render all game elements as graphics to the screen, including text. | 4 |
| The graphical representation of the game matches the underlying logic well (aka you look like you are hitting a wall when Box2D blocks your movement). | 3 |
| The game has some kind of scoring system that is not solely based on the time taken to play (how you score the game is up to you). | 2 |
| You have added some unique feature to your game to set it apart from the other submissions (see above for suggestions). | 4 |
| Total Possible Points: | 25 |

|  |  |
| --- | --- |
| Possible Deductions | Points Off Total |
| Unique addition to the game is not well documented in the main .cpp file of the solution. | 4 |
| Code is not well commented / includes complex code that is not explained. (You don’t need to comment every line, but atleast tell us what each function does). | 4 |
| Code contains memory leaks.  Note: SFML, Box2D, and the DLL will create a number of unavoidable “false positive” memory leaks with the \_CrtDumpMemoryLeaks() function. As such, the function will be unreliable for this assignment. Instead, simply ensure that you are deleting anything allocated with the new keyword. | 4 |
| Code does not follow standard coding convensions (use of non-descriptive variable/function names, hardcoded values, etc). | 4 |