“Ski Survival”

Evan Nygard

15-112 term project (spring 2021)

Project description

This is a survival game heavily based on the 1991 Microsoft game “SkiFree” by Chris Pirih. The player controls a single skier and attempts to avoid various obstacles and non-player characters (NPCs) as the skier skis down the slope. Points are principally gained through survival, but the pace of the game and the saturation of obstacles on the map both increase the longer the player survives. The game ends when the player crashes or is caught by an NPC.

Competitive analysis

As previously stated, this project was based on “SkiFree” by Chris Pirih, and there are many similarities both in visual design and in gameplay. Both SkiFree and Ski Survival control the skier in basically the same way, both involve the goal of avoiding trees and rocks, both are viewed from a bird’s eye view, and both contain monster NPCs that attempt to catch you and eat you. Ski Survival also contains visual similarities to the MSDOS game “Ski or Die”.

There are, however, multiple differences between Ski Survival and SkiFree. Ski Survival contains a different selection of obstacles than SkiFree. Both contain trees, rocks, and slalom poles, but Ski Survival, while not including other skiers and random dogs (included in SkiFree), also involves jumping over rivers and avoiding falling logs (which are not included in SkiFree). Additionally, successfully avoiding NPCs is part and parcel of playing Ski Survival, whereas in SkiFree the NPCs only appear at the end and are much more difficult to avoid. Finally, SkiFree has a finish line and doesn’t appear to get any more difficult over time, whereas Ski Survival does increase in difficulty over time and has no finish line.

Structural plan

The project will consist of three main files (more may be added if necessary):

- term_project.py: contains the cmu_112_graphics functions necessary to launch and run the app. Much of the main work will be done in the timerFired() function, which calls a separate function generate() to create random terrain, and which checks for collisions between the player and obstacles.
- tp_objects.py: All obstacles, as well as the player, are implemented using OOP. tp_objects.py will define a superclass Obstacle(object) in order to define the basic functionality of each obstacle. Each individual obstacle will then be subclassed to implement its individual functions. Functions include moving the obstacle, drawing it on

canvas (if possible), checking for collisions, and (in the case of NPCs) detecting the player, calculating an ideal interception point, and chasing the player down.
    - Note: the Player object is a special case, and is not subclassed from Obstacle. Although the Player object can also check for collisions and draw itself on canvas, its movement is implemented differently than other obstacles. In addition, it has unique functions such as jumping. Therefore, it gets its own class.
- tp_collisiondata.py: Different obstacles are implemented differently. Some are implemented as a rectangle, some as a half circle, and some as an algebraic y = mx + b type formula. Therefore, getting location information and calculating collisions cannot be implemented the same way for each object, hence the CollisionData class. CollisionData provides an extra layer of abstraction; it takes in general information about location and can calculate whether a collision has occurred based on this information. The class contains functions to calculate whether a collision has occurred between two different obstacles or between the obstacle and the player. (If there is supposed to be an amount of whitespace between one object and another, these calculations may return different results.)

Algorithmic plan

Two sections of the project stand out as more difficult than the rest: the NPC class's interception algorithm (IA) and the collision testing algorithms.

- In order to determine the optimal path to interception, the IA will utilize the Law of Cosines ($C^2 = A^2 + B^2 - 2AB\cos C$), where A is the vector between the NPC and player and B is the player's motion vector. C is the vector that provides the optimal interception direction. The idea for using the Law of Cosines, as well as the specific mathematical operations involved, were taken from this site.
- In order to test for collisions, all of the relevant data will be collected into a CollisionData object in each obstacle's getCollisionField() method. CollisionData takes in data such as the object's height, width, leftmost x-coordinate, algebraic y = mx + b function for calculating topmost y-coordinate at any x, whether the obstacle can be jumped over, and whether the obstacle requires whitespace around it where no other obstacles can be generated (a "buffer").
    - The meat of the collision testing work is performed in the _collidedWithObjectOrBuffer(self, objData, buffer) method, which is NOT meant to be called by any function other than CollisionData methods. This function works as follows:
        - If the test involves two rectangular objects, calculate whether the two rectangles plus their buffers overlap at any point. This test is relatively simple and takes O(1) time.

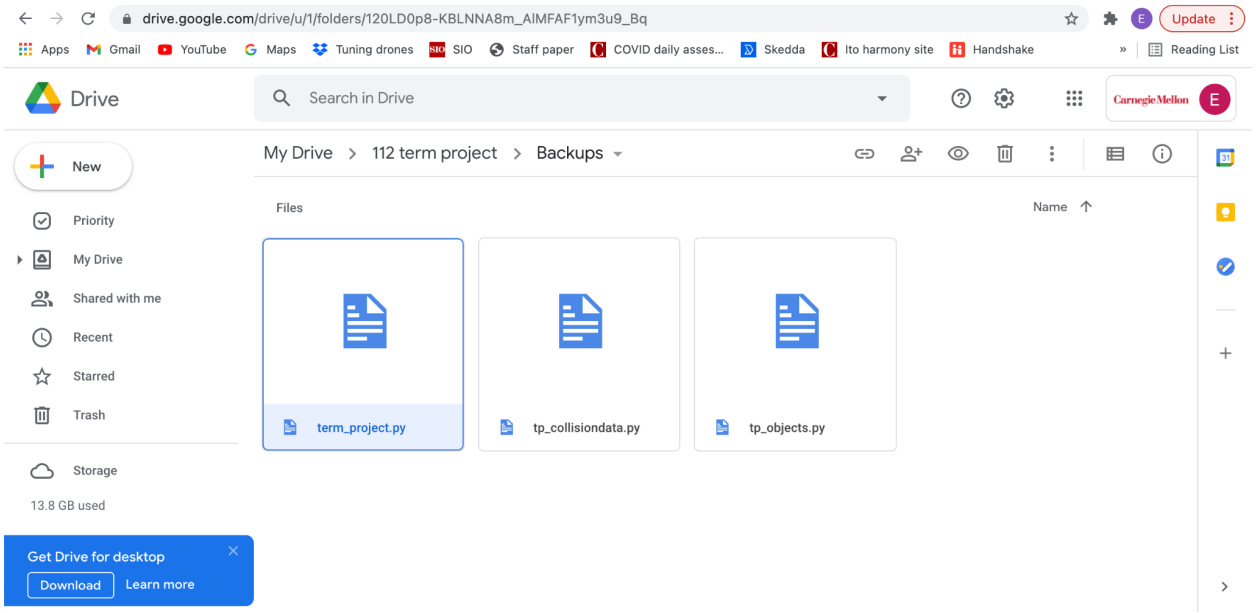
- If the test involves one rectangular object and one algebraically-defined object (e.g. a River), loop through each tiny rectangle composing the algebraically-defined element and perform the rectangular test for overlap on each of those rectangles. This takes slightly longer, but for each algebraically defined object there are a finite number of rectangles that are drawn (perhaps 1000), so this technically also takes O(1).
- If the test involves two algebraically-defined obstacles that span the entire width of the app, use algebra to figure out the intersections between the two lines. (So with two y = mx + b equations, set the equations equal to each other and solve for x). If there is no intersection that occurs on the screen, there is no collision. This should take O(1).
- In any other case, loop through the tiny rectangles composing each of the two objects and test for collision. This takes O(N^2). Ideally, this scenario should be optimized to be more efficient between MVP and TP3.

Timeline Plan

April 27: Finish the CollisionData class. Implement the FallingLog and SlalomPole classes and implement proper game messaging (this should be quick if CollisionData is completed). Finish restructuring the Rock class. Begin implementing the River class.
April 28: finish implementing the River class.
April 29: begin pulling in visualizations.
April 30: finish implementing the visualizations.
May 1: make final edits for MVP.
By May 3: optimize the NPC IA. Default version of the game must be flawlessly implemented. All extra files must be ready to go. If there's time, have the high score functionality implemented by now.
May 4: Submit the term project by the afternoon.
May 5: (Likely out of commission due to second covid vaccine)

Version control plan

All code files are being stored in a dedicated folder on Google Drive.



Module list

No external modules are required for MVP (besides CMU 112 Graphics).

TP2 UPDATE

Basic gameplay remains the same. A few implementation updates are as follows:

- Removed the River class; replaced with SimpleRiver (a river that simply travels straight across the screen rather than using an algebraic formula for generation)
- Rather than using a SlalomPoleSet class that generates multiple poles at once, slalom poles are now represented by a single SlalomPole class and generated one at a time.
    - Poles are red or blue and must be passed by on a specific side (within a short distance) to get extra points.
    - There is no “buffer” of extra whitespace around these poles.
- Added a Snowdrift class. If the player enters a snowdrift, their speed is reduced dramatically.
- Rocks are drawn as half circles.
- The collision algorithms were updated in order to give the illusion of 3D gameplay. For example, a collision with a Tree will only register if the player collides with the bottom half of the trunk, rather than the top half.
    - This is because the game is played from a birds-eye view -- from above, what might look like a collision with the top half of a tree trunk would not be a collision in real life.

    - This is a general principle that was applied with all obstacles in the game.

TP3 UPDATE

- Added a home screen and an instructions screen
- Added two extra modes in addition to the Standard mode: Werewolf Chase and Slalom
    - In Werewolf Chase, the only obstacles generated are Trees and Werewolves, with Werewolves being the most likely option. The goal is the same - gain maximum points by surviving as long as possible.
    - Slalom is a timed round lasting sixty seconds. The only way to gain points is by skirting right next to a pole in the proper direction. The goal is to gain as many points as possible before time runs out. The only obstacles are Trees and SlalomPoles, with SlalomPoles being the most likely option.
- A game over message now displays on the screen after game over.
- A crash image is now included for the player. If the player crashes, the image will switch to this crash image.
- Werewolves are represented as images as well, and appear differently when they are stationary vs. chasing the player.