COS 426
Final Project
Sara Dardik, Yael Stochel, Joseph Rubin, Noa Zarur

Penguin Game

I. Abstract

We present a fully three dimensional web game rendered in real time on your local computer. The player controls a penguin who slides along a snowy ramp; the penguin can move laterally and jump vertically in order to avoid obstacles, collect fish, and survive for as long as possible. The game is created in Three.js, and makes use of various computer graphical techniques that were acquired over the semester.

II. Introduction

A. Goal

1. Our project tried to implement a version of the Club Penguin sledding game, which is an endless runner game where a penguin sleds down a ski slope and avoids various obstacles, like rocks and logs.
2. Club Penguin was a popular gaming platform, in which users navigated through a variety of game options with penguin avatars. After being bought by Walt Disney, the site was closed down. One of the most popular and well-known games on the site was a penguin sledding game, the inspiration for our project. We hope that our project satisfies the nostalgia of Club Penguin enthusiasts and exposes a new audience to the pleasure of playing an Arctic-themed game of obstacle avoidance.

B. Previous Work

1. Other people have created similar infinite runner games, whereby a user that is moving on a ramp must avoid certain obstacles and possibly collect objects to increase a score. Games that we used as inspiration were Temple Run and Subway Surfers, as well as the Driver's Ed from the COS 426 Hall of Fame.
2. Previous approaches succeed when they employ randomization and variety on multiple levels to keep the game interesting. A successful approach used in both Temple Run and Driver's Ed is including a variety of hazards and rewards with different rules. For instance, the possibility of collecting coins to increase the score adds an element of actively seeking out rewards, rather than a mere game of avoidance. Similarly, using moving cars in Driver's Ed creates a hierarchy of "danger" with respect to various obstacles, forcing the user to think quickly about which obstacle is easier or harder to avoid. Successful games also randomize obstacle and reward positions and frequencies so that the user does not fall into a fixed pattern. Games may also include different levels of difficulty in order to tailor the experience to the individual user, and to encourage users to continue playing, as they aim to reach a higher level.

   On the other hand, games that lack diversity are less successful, since the user might grow bored. If the speed or the obstacles and rewards are fixed, the user might learn to master the game, and no longer feel the need to continue playing. Similarly, if the game is less visually appealing, users will be less likely to play.

III. Approach and Methodology

A. Forces

   1. In order to mimic the outcome of a penguin shifting right, left and jumping over various blockades we implemented gravity, friction, and acceleration. While there are various possible implementations for each, including the use of Verlet Integration which we did on Assignment 5, we chose implementation methods that would lead to the most clear and efficient code. For example, for acceleration we chose to use the semi-implicit Euler equation to mimic the speeding of the penguin as the keyboard was held down in the right or left direction. [4] We chose to implement forces in order for the penguin to slide with ease and to follow form with modern computer games. While implementing forces we also had to consider the boundaries. We considered both horizontal boundaries, in the way the penguin can slide to the left and right but never fall off the mountain, as well as vertical in allowing the penguin to jump high enough to avoid the obstacles but not reach the top of the screen.

B. Snowy Plane: The Endless Runner [9]
   1. In order to implement snow there were various images and lighting techniques that we used to produce the results. By loading a jpg we found of snow, loading the texture to the plane, and using directional light we were able to illuminate the scene. The texture is an RGB normal map, which tells the game how to light the snow in order to make it feel more detailed than a flat plane. In order to make the ramp feel infinite in length, the texture is wrapped along both of the plane's axes. This way, we were able to scale the snow texture exactly how we liked it without limiting how wide or long the plane could be.
   2. In order to mimic the snow moving down the hill we changed the offset of the texture and the normal map continuously as gameplay progressed.

C. Snowfall: To create snowfall, we generated spheres at random x and z values above the camera and applied gravity. In order to optimize the performance of the game, we tried to limit the number of snowflake spheres generated. To do so, we only generated snowflakes at the beginning of the simulation (when the timestamp is below a certain number). Then, once the snowflakes fell below the sight of the camera, we moved them back to the top at a new randomly generated position. However, even using recycled spheres was too slow for game play. See next steps for how we plan to fix this.

D. Penguin [12]
   1. In order to create the character for the scene we found a penguin gltf file. [
   2. The penguin responds to three controls. If you press the left arrow, the right arrow, or the space bar, the penguin will move left, move right, or jump respectively. Note that the penguin is subject to physical forces and boundary conditions, as explained above in "Forces". This functionality was implemented using event handlers, similar to the way we implemented a force applied to a cloth on a key press in Assignment 5.

E. Hazards and Rewards[7][10][11]][13][14]
   1. In order to mimic a snowy plane filled with obstacles moving toward the camera, we move all of the obstacles at the same speed. The objects are generated at a random point (i.e. a random x value, or a random spot from left to right from the perspective of the camera) along the horizon, and then all move at the same speed towards the penguin, which is fixed in the z direction. If the penguin collides with one of the hazardous objects, all of the objects stop moving and the game is over, mimicking a halt in the penguin's movement.
   2. In order to detect a collision between the penguin and a hazard, we find the bounding box of the penguin and the hazard and then call the THREE.Box3 function intersectsBox(). If the bounding boxes intersect, we set the scene state field of gameOver to true.

3. In order to optimize the performance of the game, we limited the number of hazardous objects generated. To do so, we only generated hazards at the beginning of the simulation (when the timestamp is below a certain number). Then, once the hazards move behind the sight of the camera, we move them back to a randomly generated position along the horizon.
4. Note that not all of the hazards have the same effect. Colliding with the rocks, logs, seals, and trees ends the game, but sliding on ice temporarily increases the penguin's speed (or more precisely, the speed of the hazards, rewards, and the ramp), which only makes the game more difficult to play. Similarly, seals can move from side to side, while rocks, logs, and trees are fixed along the x axis. As such, hazards are divided into three categories: regular hazards (including rocks, logs, and trees), moving hazards (including seals), and ice.
5. In addition to avoiding hazards, a user can also increase his/her score by directing the penguin to eat fish. Note that the fish reward obeys the same rules of random generation as the hazards, but the fish disappear and regenerate upon collision with the penguin, as well as after passing the camera. Though we downloaded gltf files for the rock, log, tree, and seal hazards, we created fish using geometries from three. For every fish, we generated a sphere, as the body of the fish, and a cone, as the tail of a fish and merged the two geometries together. We also created an array of bright colors and randomly selected a bright color for each fish, as well as adding a specular value, to ensure that the fish were easily distinguishable from the hazards.
6. There were a few possible options for the generation of recurring objects that eventually disappear (or are no longer in view of the camera), such as hazards and snowflakes. One option was to keep producing new objects and let them leave the view without collecting them. For hazardous objects, this would mean having all of the objects continue moving past the penguin forever. The second option was to keep producing new objects, but remove them from the scene once they're no longer visible. A third possibility is to only generate new objects for a fixed period of time at the beginning of the simulation, and then once those objects are no longer visible, reset their positions.
7. The second option has the advantage of hazard randomization, namely that the frequency of the hazards is random for any time interval of the game. However, this option still involves generating a lot of new objects, which is computationally expensive and noticeably slows down the simulation. The third option has less randomization. It chooses a random set of logs, rocks, and ice, but once this random set is chosen, it is reused, so the ratio of logs to rocks to ice remains the same for the duration of the game. However, the third option recycles the objects, making it more efficient.
8. The first possibility is the simplest, so we used this model just to get the hazards and the snow moving. Then, we used the third option (recycling hazards) as opposed to removing hazards and regenerating them. We did so because when we removed objects from the scene, it momentarily paused the simulation, leading to a somewhat glitchy effect. Recycling the hazards did not have this problem and thus looked smoother.

F. Ice: An Exploration in the Generation of Random Convex Polygons [1][2][3]

1. In order to create a range of ice patches without manually specifying each shape, we generated convex polygons using the Jarvis March/Gift Wrapping algorithm, which is one of many convex hull algorithms.
2. The Jarvis March algorithm goes through the following steps:
    a) Randomly generate a set of *n* points.
    b) Find the leftmost point in the set as the first vertex.

c) While the next selected vertex is not the leftmost point, iterate through the remaining points in the set to find the point that satisfies the following conditions: Imagine forming two line segments, one between the most recently selected vertex and the previously selected vertex, and one between the most recently selected vertex and the point which we're currently considering in the set of *n* points. (For the case of the first selected vertex, use a vertical line segment for the first segment.) Let the angle from the first line segment to the second be called *theta*. The point that forms the largest counterclockwise *theta* should be selected as the next vertex.

3. Note that once the next selected vertex is the leftmost point in the set, we have completed the convex hull, meaning that we have found all of the vertices of the convex polygon that surrounds all *n* points in the set.
4. The Jarvis March is considered to be a simpler and less efficient convex hull algorithm, because its runtime complexity is *nh*, where *n* is the number of points in the set and *h* is the number of vertices in the convex polygon. However, this simple algorithm performs quite well when *n* is small, which it is in our case of *n* = 20.

G. Terrain

1. The terrain in the background below the ramp is generated randomly, and is meant to look like a natural formation of mountains. To make this effect, we implemented a function for generating Perlin noise. [5] 2D Perlin noise is created by considering an imaginary grid of points which each have a random gradient vector associated with them. For any point, the value of the noise is equal to a weighted interpolation of the dot product between the gradient vector on each corner and the vector from the point to that corner.

To do the interpolation, we chose the smootherstep polynomial [6]

```
x * x * x * (x * (x * 6 - 15) + 10)
```

which is used to calculate a factor through which to mix two values. The four corner values are mixed pairwise until a final noise value is produced.

Generating a stable random gradient vector for each grid point requires a PRNG that can be seeded. Because JavaScript does not come with such a function, we created our own. The `gradientAt` function calculates a random gradient by summing the x and y coordinates of the point after multiplying each by a prime number (effectively calculating a hash), then feeding that value into a trig function (after being scaled by PI). The trig function (we chose cos) is used to generate values that are unpredictable. The value is then brought into the proper domain and its sin and cos become the random x and y values of the gradient vector.

H. Start and End Scene[8]

1. For the start scene there are many variations we can take: start the game when you press a button that says start, start the game when you press any key, start the button when you click on the screen with your mouse, and various other event handlers that let the game know it is time for the game to begin. The advantages of the first three mentioned is that users are used to playing games with these signalers making it more user friendly to follow suit with how others build games. We chose to go with pressing a button to proceed to the next page in order to be more clear and efficient. For the end scene we also showed the instructions for the game again in order to allow for clarity.

I. We implemented various features:

1. Scenes: We created three separate scenes for the start of the game, the game itself, and the end. The start scene includes instructions for how to play; the game scene includes all of the

features of the game described above, and the end scene appears once the penguin collides with the object and displays the user's final score.

2. Scene Graphics: We implemented a mountainous terrain generation, as well as a texture snowy plane.
3. Obstacles: For obstacles, we included seals, rocks, logs, and trees.
4. Physics: The obstacles movement corresponds to the movement of the snowy plane. The penguin is fixed along the z-axis, but can move from left to right as well as jump according to the laws of physics.
5. Point collection: We added fish that increase a player's game score when "eaten" by the penguin (i.e. upon collision).
6. Game intensity: The speed of the obstacles increases as time passes to make the game more difficult for advanced players.
7. Extra features: We included the music from the original Club Penguin game!

J. Features we did not implement:

1. Multiplayer: After analyzing various modern games we realized that most have shifted from being multiplayer through separate computers as opposed to splitting various keys on a single keyboard in order to allow multiple players. Given this trend, we chose to not implement a multiplayer feature in order to remain consistent with modern games.
2. Drop down menu with different penguins: Using the feedback we received from users we saw that this was a suggestion a user had. However, after collecting all the user feedback we deprioritized this feature. While changing penguin color can give the player more personality, game retention would be better addressed through further expanding the challenges of the game.
3. Turning through terrains: While testing our project, we noticed that some users were getting headaches and thought the amount of movement was becoming annoying. In order to improve the user experience, we decided to reduce the amount of non-essential movements in the game.
4. Having the logs embedded in the snow: Originally, we hoped to complete this feature to mimic the original Club Penguin layout. However, given our visual choices we noticed that the graphics looked better without covering the logs with snow.

K. Results

1. In order to measure success, we collected user feedback on how user friendly they believed the game was, retention rate, hours they would play the game, and visual appeal of the game.
   a) We received many positive comments about our game:
      (1) "Graphics are pretty good! They almost look like Minecraft, and that is one of the world's most popular games!"
      (2) "Very fun!"
      (3) "The instructions are quite clear…"
      (4) Overall users felt that the game was 8.5/10 for user friendly.
      (5) Overall users found the graphics 4.5/5 appealing.
      (6) Overall users mentioned they would play this game for around an hour.
   b) We also received many feedback points we were able to take action upon:
      (1) "Couldn't see the jumping."
      (2) "It gets so fast that it becomes unplayable after 3 minutes."

(3) When we first started surveying users, we found that the instructions were not clear enough in regards to the speeding up on the ice patches and that the up arrow on the keyboard corresponds to jumping over a given obstacle as opposed to pushing the penguin forward. In order to address these concerns, we edited the instructions to further clarify the goal and keys of the game. We also found that the speed was increasing to the point that made it very difficult to continue playing the game. In order to fix this issue, we created a max speed value.

c) Feedback points for next steps:

(1) “Has some mild lag issues, but this is probably due to the computer itself.”

(2) “I wish the penguin itself had more personality (ex: player can select different colored penguins).”

2. There are three experiments we executed:
   a) Feedback: We collected feedback from users in order to create prioritization of features.
   b) Scale: In order to accomplish our big goals, we started by playing around with a basic version of every feature we had and then gradually increased its complexity. For example, for the game scene we started off with a simple plane and then added more features, textures, and movements. Similarly, we initially modeled the penguin as a green cube and different hazards as different colored cubes, and tested the functionality of collision handling just based on cube collisions.
   c) Examples: We used games displayed in the COS 426 Hall of Fame as examples for how we should approach our game. We used these examples to plan out our goals and mission and experimented with using similar libraries.
3. We measured success by creating a game that is both fun to play and reminds people of the original Club Penguin game. Given all the feedback we received, we were able to analyze how fun our game was. Our results indicated that there was an overwhelming amount of positive feedback, and we addressed the limited negative feedback that we received. We also used graphics that were both visually thematic and penguin-themed. This leads us to believe that our project was successful.

IV. Discussion

A. Overall, and given the positive feedback, the approach we took to the design of our game was successful. By choosing to base our project on a game that exists within the specific world of Club Penguin, we ensured that our design included a consistent aesthetic that contributed to the appeal of our game. Since the original Club Penguin game was multiplayer, it relied on a competitive element for its appeal, as the users raced to be the first to the finish line. The most successful games must have a competitive aspect, as competition inclines the users to continue playing. Once we opted not to include a multiplayer option, we deemed it necessary to include an alternative method of tracking success for the user. We incorporated the collectible fish, which the Club Penguin game did not include, as a replacement for the multiplayer option. Furthermore, in increasing the speed of the game as time passes, we intended for the game to encourage replays, as returning users seek to avoid obstacles for longer. The specific approach of stabilizing the penguin, and moving the objects and the ramp towards the camera, was successful as well, as the effect correctly mimicked the penguin’s forward motion and simplified changes to the speed.

B. An alternative approach would have been to create our own objects, as this would have enabled a degree of sophistication in animating the moving objects. However, such an approach would have also demanded greater simplification, since the objects we included were complex. Instead, we prioritized realistic objects and maintained the aesthetic of a cartoon icy slope, complete with rocks, trees, and naturalistic animals. Considering the success of the game, we think we made the correct decision.

C. Since the final product includes many of the elements we set out to implement, follow-up work would simply involve embellishing the game. If we were to have more time, we would add character to the background by incorporating different landscapes, as one would see along a snowy slope. We would also add branching paths along the central slope, as they appear in Temple Run, so that the user can make turns. Lastly, follow-up work would include adding more three dimensionality to the game, by having the slope change angle over time perhaps. As we did not have time to sufficiently survey users for their thoughts on our final project, we would also collect and incorporate more of the feedback until the game.

D. Over the course of this project, we gained crucial experience in independently applying skills we had learned through other assignments this semester, as well as a deeper appreciation for the design choices that contribute to the creation of a successful game. In the process of creating the game, we learned that small details contribute substantially to the final results, and that incremental improvements, viewed altogether in a final project, form a cohesive, engaging product. We also gained an appreciation for collaboration, as every member of the group brought his or her perspective and preferences, contributing to the creation of a better, more appealing game. We learned from our frustrations as well, realizing that the limits of technology can sometimes be constraining, as when we realized that adding snow to the scene was slowing down the objects too much. Successful games must balance simplicity with engagement, ensuring that users are sufficiently drawn to the game, but that the rules and gameplay is simple enough to run smoothly.

V. Next Steps

A. Lives of the game: If you hit an obstacle, you lose a life. Based on which level you pick (easy, medium, hard), you will have an allocated number of lives. Once those lives run out then the game ends.

B. Logs spinning / tree falling: Additional obstacles with effects could add an element of entertainment and also include snowy features.

C. High score: A tracker that remembers the highest score that a player has achieved.

D. Sound effects: By connecting specific sounds to actions such as the sound of slipping on ice while getting the acceleration boosts from the ice patches will allow the scene to come to life.

E. Snow fall: Because generating even a small number of snowflakes is rather expensive and slows down gameplay, we will only use snowflakes as an effect at the beginning or the end of the game, but not during the game itself.

VI. Contributions

To accomplish this project, we divided the tasks amongst the four of us. Joseph worked on the mountainous terrain generation and the scoring system. Noa worked on creating the start and end scenes. Noa and Joseph worked on implementing the physics system. Sara and Yael worked on generating hazards, rewards, and snowfall, as well as adding the music. We all worked on debugging and adding finishing touches.

VII. Works Cited

[1] “Convex Hull Algorithm - Graham Scan and Jarvis March Tutorial.” YouTube, YouTube, 19 Apr. 2020, www.youtube.com/watch?v=B2AJoQSZf4M.

[2] “Convex Hull Algorithms.” *Wikipedia*, Wikimedia Foundation, 5 Nov. 2020, en.wikipedia.org/wiki/Convex_hull_algorithms.

[3] “Convex Hull: Set 1 (Jarvis's Algorithm or Wrapping).” *GeeksforGeeks*, 28 Mar. 2021, www.geeksforgeeks.org/convex-hull-set-1-jarviss-algorithm-or-wrapping/.

[4] “Integration Basics.” Gaffer On Games, 1 June 2004, gafferongames.com/post/integration_basics/.

[5] “Perlin Noise.” *Wikipedia*, Wikimedia Foundation, 15 Apr. 2021, en.wikipedia.org/wiki/Perlin_noise.

[6] “Smoothstep.” Wikipedia, Wikimedia Foundation, 15 Jan. 2021, en.wikipedia.org/wiki/Smoothstep.

[7]“How to Detect Collision between Two Objects in JavaScript with Three.js?” Stack Overflow, 1 Oct. 1963, stackoverflow.com/questions/28453895/how-to-detect-collision-between-two-objects-in-javascript-with-three-js

[8] https://github.com/dreamworld-426/dreamworld/blob/master/src/app.js

[9] https://meocloud.pt/link/a2c7ce38-9548-4f18-aac8-d79bcdadd59b/Snow_001_SD/

[10] Log: https://poly.google.com/view/dkRLlPSdgdR

[11] Tree: https://poly.google.com/view/269i7SjGmUt

[12] Penguin: https://poly.google.com/view/3t8nvyGamxW

[13] Rock: https://poly.google.com/view/62bVOJt7vHv

[14] Seal: https://poly.google.com/view/fudlK4rnsI-