DaVinky

Nadia Rodriguez, Dylan Tran, Michael Tran

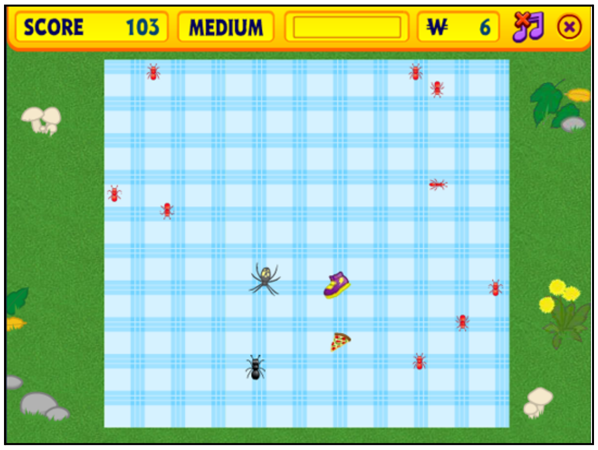
Abstract

DaVinky is a colorful and action-packed three.js game in which players control a cute monster named DaVinky. The objective of the game is to collect as many balls of paint as possible while avoiding various enemies that roam the game world. Players must use their agility and quick reflexes to evade enemy attacks and collect the paint balls. With vibrant graphics and addictive gameplay, Davinky is a thrilling and entertaining game for players of all ages.

Introduction

We based our ideation and implementation on ***Webkinz Ant Mania.***

The objective of Ant Mania is to earn as many points as possible by collecting pizza slices across the map. The game ends when the user runs into an enemy spider. The games get progressively harder as the user completes each round – more ants spawn, and they begin to move faster. The user can choose between three levels of difficulty – easy, medium, and hard.



We wanted to recreate a similar gameplay experience where the user navigates around a map to collect objects and avoid enemies. Our game gets increasingly more difficult – the more paintballs the user collects, the faster the enemies move. The game ends when the user reaches 0 points. The game features three difficult modes – easy (regular gameplay), medium (first-person), and hard (world rotation).

Stylistically, we wanted to model our game with an aesthetic similar to the Webkinz Ant Mania mini-game. The user plays from a third-person, birds eye view perspective. Score is tracked at the top of the user's window. The game features sounds for movements like eating pizza and running into enemies.

Methodology

User controls

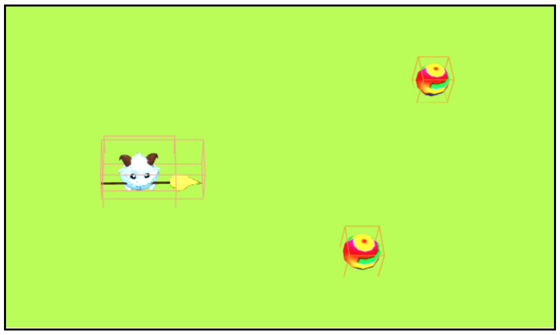
The user controls are simple – users can move up, down, left, and right using either the arrow keys or WASD keys. We use `EventHandlers` to detect keypress events; if the key pressed is

ArrowUp, ArrowDown, ArrowLeft or ArrowRight, or W, A, S, D, then the character moves in the corresponding direction by 0.1 (delta). Moving up and down modifies position on the x-axis, and moving left and right modifies position on the z-axis. On keypress, the character also rotates along its y-axis by a scale of PI to face the appropriate direction (i.e ArrowLeft turns Davinky to face left, etc.).

Pressing the spacebar enables the character to jump. We recreate a gravitational effect to simulate falling by updating the character's position along the y-axis by adding its current velocity along the y-axis. We then update the velocity on the y-axis by subtracting the coefficient of gravity. The plus side of this implementation is that for the goals for this game, it efficiently achieves its purpose without being too taxing on the game. Initially we were going to do a much more complex system for physics with acceleration, velocity vectors and more for more realistic movement, but we realized that it was going to either make the game much more laggy and create a lot of bugs with other parts of the game, so we opted for a more simpler implementation.

Collision detection

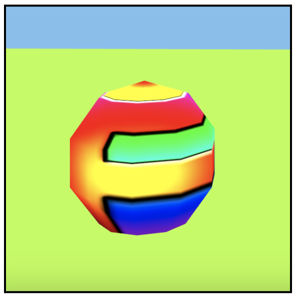
Our implementation utilizes the Three.js `Box3` class and `setFromObject` function to set bounding boxes around the character, enemies, and paintballs. We check to see whether or not a collision has occurred by calling the `intersectsBox()` function on two objects (i.e to see if the character's bounding box intersects with an enemy bounding box). If the character collides with an enemy or paintball, the object is removed from the scene and the user's score decrements or increments by 1, respectively.



Originally, we wanted to implement collision detection using raycasting, but we quickly found that this might not be the best approach for our game, because it would only detect collisions in the exact direction the ray was cast. We found the bounding box implementation to be more useful in our case, as we could then detect object collisions from all directions (from the front, behind, side, etc).

Texture mapping

We use the Three.js `TextureLoader` class to load in an online texture for the paintball meshes. We first create `SphereGeometry` objects to represent the shape of the ball. We initialize the material as a `MeshPhongObject` and set the texture from this URL: https://webglfundamentals.org/webgl/resources/f-texture.png. We



then pass in the sphere geometry and the material as arguments to initialize a new `Mesh` object for the paintballs.

Audio

We use the Three.js `AudioListener` class to load and render sound throughout gameplay. We store 15 different .wav audio files, including sounds for jumping, collecting paintballs, getting hit by enemies, and running into the edge of the scene map, in a `sounds` array which is passed through each of the relevant handlers (such as collision detection and player controls). When a certain action or event happens like jumping or collecting a paintball, the corresponding sound is referenced in the sounds array and played. The advantages of this implementation is that all of the sounds are stored in a single array and can be easily referenced in all parts of the code. Some of the disadvantages of this implementation is that if there were a large number of sounds, passing them to new functions constantly could become taxing on the performance and new sounds cannot be played until the current sound is done playing. For example, if the player were to get hit by an enemy and jump at nearly the exact same time, only one of the sounds would be played.

Scoring

We maintain a score tracker at the top of the user's window. The score is a reflection of the user's total points. At the beginning of the game, the player starts with 0 points. The player can either earn points by collecting paintballs, or lose points by running into enemies. The score is inherently tracked and stored as a part of the scene object. Points are incremented/decremented by 1.0 points, unless the user is playing in first-person perspective OR has adjusted the rotation speed to be greater than 1. In this case, the player is in first-person, all points incur a multiplier of 2. In the case of rotation speed being added, for each new point added, another point multiplied by the rotation speed is also added. The reason why rotation speed did not just incur another multiplier is to account for added rotation speeds between 0 and 1. For example, if the player had a rotation speed of .5 and collected a point, then the added point total would be .5 which is less than what the player would incur without rotation speed which doesn't make sense. With our score calculation the player would get 1.5 points.

Enemy Spawning and Behavior

Enemy spawning is reliant on the score of the player. The game technically does not start until the player collects their first paintball which then spawns the game's first enemy. This approach we believe is effective because it allows players to mess around with the controls and explore a bit before they have to start dodging enemies. We implemented a handler for enemy spawning which constantly checks the score of the game and n points the player has, n enemies will be present on the screen. We also made sure that enemies would only spawn near the edges of the island so we would not run into the case where an enemy would randomly spawn on top of the player. All the enemies spawned were stored in an enemies array and that was passed through the scene so any handler that worked with enemies could just call scene.enemies.

For the enemy behavior we made the enemies constantly move towards DaVinky. In order to do this we calculated normalized vectors pointing towards the player with each render of the animation and shifted the positions towards the player by a factor scaled by the amount of points the player had. This way the enemies speed up as the player collects more paintballs, progressively making the game more and more difficult. With each render of the new positions, we make the enemies face DaVinky at all times by using the .lookAt() function in the threeJS library.

In the case where enemies collide with the player, the enemy is then spliced from the enemy array and removed from the scene and decrementing the players point count. This method ensures that there is always a set amount of enemies on the screen based on the score which makes the difficulty of the game always scale consistently. Another plus side is that this is a very different approach to start the game. The game will essentially end or stop whenever the player's score goes back down to 0 and start again, otherwise making it such that players do not have to physically press play to start.

Camera Perspective
Initially, the starter code used orbital controls to always point the camera at (0,0,0) and allow for the mouse to rotate the angle. While this was nice to start out with, it was causing issues whenever we wanted to swap the perspective to first person and thus we decided to remove orbital controls and set the third person camera to be a static birds eye which in turn made the game a bit easier to control in third person as well.

For the first person angle implementation, we set the camera position to always be slightly above DaVinky with each render iteration. Setting it slightly above made it so that the model did not cover the players screen, and had a slightly better vantage point since DaVinky is a little bit short. We then need the screen to turn with DaVinky which proved to be a bit difficult. To do this we got DaVinky's world direction and created a quaternion to apply the rotation of the camera. However, at this point the camera rotation was almost instantaneous which felt very jarring as a player so we decided to use the .slerp() method to smooth this turning. One downside of this implementation, is that given how we implemented the controls, the camera turning can sometimes feel a bit strange in first person.

User Interface
The user loads into the scene as the game's main character (DaVinky) and enemy objects (Paper) were created on Blender and are loaded in as `gltf` files using the Three.js `GLTFLoader`. The interface is relatively simple with the player score in the top left corner and a simple GUI implemented using the dat.gui library in the top left. The options are to change the game's rotation (initially the starter code set this from -5 to 5, but we changed it to 0 to 5 in order to have proper scoring and clarity for game scoring. Additionally, there is a menu option marked FirstPerson and players can check the box at any point in playing the game and swap the perspective to first person. This implementation has a plus side of being able to change the settings during the game without having to reset one's progress. The way first person is triggered is that the checkbox is

linked to a boolean value and if true, the handleCameraAngle helper function will swap the way the camera is set. The way we implemented this is so that if players decided not to have first person anymore, the camera would automatically reset to its original position as soon as the box is unchecked in the GUI. The downside is that, while the camera is in third person, handleCameraAngle is called unnecessarily with each render; however, a different implementation would require us to reset the game which we felt was not very satisfying.

Results

We produced a fully functioning game with cohesive graphics and an enjoyable gameplay experience.

The first evaluative metric we used was functionality – did the game work as intended? We tested collision detection by making sure that the enemies and paintballs disappeared upon intersection, as well as making sure that the character could not move past the bounds of the scene. We tested audio functionality by making sure the appropriate sound file was played with its corresponding action (i.e hitting an enemy plays hurt1 audio). We tested user controls by making sure the character moves only when the arrow keys/WASD keys are pressed, and made sure it was positioned correctly upon movement. We also tested for the case where two arrow keys are pressed at the same time, in which case the character would move at a diagonal.

The second evaluative metric was playability – is the game intuitive and engaging? We initially were a bit worried that the game was a little too simple and potentially boring, but after a bit of playtesting among our team as well as friends outside of class, we realized that the game is quite fun even with the simple controls. We hypothesized that the simplicity of the visuals and goal of the game made it extremely easy for players to sort of disassociate and focus solely on doing a clear and easy task. Think of how games like doodle jump and flappy bird are extremely simple in gameplay, but somehow extremely successful! Another thing we noticed was that the game felt pretty fluid to play, the performance was almost an issue which we find is extremely important to enjoyment of a game. No matter how great a game looks, if it is extremely laggy, it is not an enjoyable experience.

Discussion

We learned that making a whole game from scratch is an extremely taxing and time consuming process. We initially came in very excited and eager to see how our game would turn out and were quickly met with a lot of unexpected walls and roadblocks that were a bit stressful to overcome. Overall, making this game really strengthened and tested our understanding of the concepts we learned in class which was quite rewarding being able to put that knowledge in practice to create a real game. We also learned that organization is very important in almost all aspects such as planning out and brainstorming the game, laying out how we want to approach creating the game, setting our priorities for what needs to be implemented first, and keeping internal code organized with helpful clarifying comments. Something that really helped us during the development process was sort of rotating roles as we got stuck during implementation of features. For example, Dylan and Nadia both made great contributions to unit collision detection. Dylan was able to set up a good base for sounds, but in the end Michael finished the implementation. Michael initially set up enemy spawning and movement, but Dylan was able to finish out despawning them. Had we stuck to our original roles or implementations that we started, they probably would have been extremely unfinished so having fresh eyes take a look at code is definitely very useful. It's also important to take breaks when to! A lot of the work was done together with all of us sitting next to each other discussing approaches and problem-solving, but a good amount of progress was also made asynchronously.

Conclusion

How effectively did we attain our goal?

When first going into development, we had a lot of stretch goals and nice to have functionality/aspects, butI think our main goal was to just have fun and challenge ourselves to have a memorable experience making a cute game. Needless to say we were definitely challenged and will certainly be remembering this experience. Throughout development we were very unsure of how things would turn out and what exactly we were looking for, but we think it is safe to say that we are quite happy with how the game turned out and have grown a bit attached to the main character in the first ever game that we made. We may not have been able to reach some of our stretch goals, but we achieved our MVP and our goal of learning and making a cute game was certainly fulfilled.

Existing issues

Currently, the way the user controls are implemented is such that the character moves with respect to the position of the camera when the scene is first rendered. This makes it especially difficult for the user to navigate across the map when playing in first-person since the controls were only made to change world position rather than shift the character based on the camera perspective.

Another potential issue is that there currently is no max score indicator so as players are trying to collect points and possibly lose points due to running into enemies, players can only view their current score and not the total score they had while playing the game. Additionally, the score is not very visually appealing since it is basically just plaintext on a while line.

Future iterations

In the future, in addition to addressing the issues mentioned earlier we would like to add HTML pages for a start screen, pause screen, and game over screen. Additionally we may overlay instructions on how to play the game or mute/pausing/restart functionality. We would also consider customizing the scene a bit more, possibly adding clouds in the sky or texture to the grass. If we had a lot of time to work on this game, adding animations and a system for DaVinky to fight the enemies would be nice. Also it would be really cool to see the game with more complex graphics and a better third person camera.

Contributions

Nadia Rodriguez

- Implemented user controls, collision detection, texture mapping, paintball spawn, reviewed pull requests.

Dylan Tran

- Implemented audio, collision detection, perspective change, enemy removal, map boundary detection.

Michael Tran

- Created Blender model of DaVinky and enemies, Implemented score calculation/tracking, HUD with perspective change functionality, enemy spawning.

Works Cited

Our code skeleton was provided by Reilly Bova ‘20.

Inspiration:

- *Ant Mania*, Webkinz, 2006: https://webkinz.fandom.com/wiki/Ant_Mania

Model:

- *Poro - League of Legends*, Riot Games: https://leagueoflegends.fandom.com/wiki/Poro

Audio:

- *Fall Guys*, Mediatonic, 2020: https://en.wikipedia.org/wiki/Fall_Guys

Other References:

- Three.js documentation: https://threejs.org/
- Audio implementation: https://github.com/harveyw24/Glider
- Advice and assistance: Yuting Yang