**Cow**

*Moove*

Moove contains all code for cow behavior. Upon spawning, a cow’s variant (of Normal, Mint, Chocolate, Cash, Coconut, Devil, Golden, Lemon, Sickly, Soy, Strawberry, and Glitch) is chosen through random number generation, determining the cow’s points value if successfully abducted, the speed at which it runs across the screen, and the amount of lives lost if the cow successfully walks across the screen, escaping the player. The cows worth the most points (Golden and Cash) also move the quickest.

For cows to move, they have a variable canMove that defaults to true. This variable is checked each frame, and the cow can only move if canMove is true. When the cow is allowed to move, the value of its movement speed per second multiplied by Time.deltaTime is added to its current x position; Time.deltaTime returns the amount of time in seconds that has passed since the last frame update, so multiplying by Time.deltaTime ensures the cows’ movement is independent of framerate. When the tractor beam detects a collision with a cow, it calls the Moove function **stopMoving(),** which sets canMove to false. To determine whether a cow has left the screen, its x-position minus half the width of the sprite (to obtain the x coordinate of the sprite’s rear) is compared to 960, the coordinate of the right boundary of the screen. If the x-position is greater than 960, the cow calls the PlayerLife script **subtractLife(),** passing the cow’s own LIVES\_COST variable as a parameter. The cow then destroys itself to save memory.

Because the tractor beam may disappear while a cow is midair, the tractor beam may call the Moove function **fallDown()** on every cow it is abducting, which sets the cow’s Boolean isFalling to true and calculates the total distance the cow must fall. This is stored in the float deltaY, which is then used to calculate the cow’s falling speed.

*CowSpawn\_v2*

CowSpawn\_v2 is a small script that contains the logic for spawning the cows into the game. It contains two functions, the default **Start()** function and a **cowSpawn()** function. The **Start()** function calls the **cowSpawn()** function every .25 seconds. **cowSpawn()** uses a random number generation algorithm to decide whether to spawn a cow or not. It generates a random number between 0 and 100, and if the number is between 0 and 35 it spawns a cow by using the built-in **Instantiate()** function to create a copy of the cow prefab stored in the Resources folder.

**UFO**

*PlayerController*

This class contains two variables: UFO\_SPEED and startingHeight. The UFO\_SPEED float is in units pixels/second, which determines the speed at which the UFO moves across the screen when the player presses either movement key (‘A’ and ‘D’ for left and right movement, respectively). Each frame, the two keys’ states are checked, and the UFO’s x position is adjusted accordingly. To solve an issue with the UFO rising slightly with each collision with a cow, the UFO’s y-position is also set to startingHeight, which is set to the UFO’s height upon creation, each frame.

*UFOAbduction*

UFOAbduction is the script that creates the tractor beam and checks for collisions between the UFO and cows during every frame. To keep track of the tractor beam easily, a private GameObject tractorBeam is declared in the **Start()** function. When the player presses ‘S’ to send out the tractor beam, the method **abduct()** is called, which handles spawning in and adding every necessary component to the tractor beam. After creating a tractor beam GameObject and putting it on its own layer (to ensure collision checks can ignore it), each necessary script is added to the tractor beam, and it is given a collision box. The hitbox’s coordinates are set to be slightly less than the size of the tractor beam’s sprite so that the cows have to be fully inside of the beam to be abducted. When the tractor beam is destroyed in its own scripts, a call to **resetTractorBeam()** tells the UFOAbduction script to reset the tractorBeam GameObject to null; this is vital because a new tractor beam is only created if a check for (tractorBeam == null) returns true. The helper function **isAbducting()** simply returns the value of that Boolean to shorten the condition into a more readable form.

Inside the **Update()** function, collision checks using the CollisionCheck script determine whether the UFO is touching any cows. If any cows are found, a for-loop iterates through each cow and performs the following: a call to their **shrink()** method ensures they are destroyed, and a call to the PointsManagement script’s **addPoints()** method increases the player’s total points by the value of cow.

**Tractor Beam**

*BeamAnimator*

*BeamBehavior*

The BeamBehavior script is a long script that begins by declaring the following attributes:

* GameObject **parentUFO** to track the UFO's location
* List<GameObject> **cows** to store the cows being abducted by the tractor beam
* List<Vector2> **cowVels** to store the velocity of each cow
* bool **isExtended** that is true when the beam is fully extended and false otherwise
* bool **isRetracting** that is set to true when the beam is about to retract
* float **timeExtended** that keeps track of how long (in seconds) the beam has been extended
* const int **BEAM\_DURATION** (initialized to 2) that stores how long the beam should stay extended
* const int **BEAM\_SPEED** (initialized to 60) that stores the time in frames (60 per second) that the cows should take to be sucked up by the tractor beam.

After the class attributes have been declared, there are 9 functions:

* default:
  + **Start**()
  + **Update**()
* destructor functions:
  + **OnDestroy**()
* mutator functions:
  + **setParentUFO**()
  + **removeCow**()
  + **setExtended**()
  + **retractorBeam**()
* inspector functions:
  + **getExtended**() and **getRetracting**()

The **Start**() function initializes the script by assigning the UFO's location to **parentUFO** and instantiating Lists **cows** and **cowVels**. The bools **isExtended** and **isRetracting** are both set to false, and **timeExtended** is set to 0.

The **Update**() function begins by filling the **cows** with each cow that hits the tractor beam's hitbox. If either **isExtended** or **isRetracting** are true, a for-loop is called that cycles through each element of **cows**. For each cow in cows, the stopMoving() function (a function of their attached script Moove) is called, and a Vector3 **current** is declared and assigned the value of the cow's current position. Then the cow's change in position per frame is determined by multiplying the change in time (a single frame) by 235 (the value that gives the optimal speed), and the value is assigned to a float **deltaY**. The cow's position is increased vertically by adding **deltaY** to its vertical component, and the for-loop repeats those steps for each cow caught in the tractor beam. **timeExtended** is increased by the time elapsed since last execution of **Update**() (one frame). If **timeExtended** is greater than or equal to **BEAM\_DURATION**, **isExtended** is set to false and **isRetracting** is set to true.

The destructor function **OnDestroy**() is called after the tractor beam object has been destroyed. The cows that were still being pulled up when the beam was destroyed fall back to the ground.

The mutator function **setParentUFO**(GameObject newUFO) updates the **parentUFO** object with its parameter **newUFO**. This allows for tracking of the UFO's position.

The mutator function **removeCow**(GameObject cow) removes **cow** (the given parameter) from List **cows** and removes the corresponding cow object from the scene.

The inspector function **getExtended**() returns true if the tractor beam has already been extended completely and false otherwise.

The mutator function **setExtended**(bool ext) assigns the value of **ext** to **isExtended**.

The inspector function **getRetracting**() returns true if the tractor beam is ready to retract and false otherwise.

The mutator function **retractorBeam**() is called (in another script) when the user opts to retract the beam before the time stored in **BEAM\_DURATION** has elapsed. It sets **isExtended** and **isRetracting** to true, which tells the script to retract the beam.

**Clouds**

*CloudAI*

CloudAI contains integers for the cloud’s speed and scale and floats for its height and opacity. Each is determined during its creation using a C# Random object to generate random numbers within ranges. To obtain numbers only within range of x-y, the % operator is applied to the randomly generated number, with the difference between x-y as the divisor. The numbers are then assigned to their respective variables and applied to the different characteristics of the cloud. During each update, the cloud’s speed is added to its x-position, and the cloud despawns once the cloud exits the room (as determined by a check for x > 960, with 960 referring to the right-most x-position inside the frame).

*CloudSpawn*

CloudSpawn is a modified version of the CowSpawn\_v2 script, using the same logic to spawn clouds at least .25 seconds apart from each other, with a fifty-percent chance to spawn during each frame update after the minimum .25 seconds have passed. If the randomly-generated number between 0-100 is less than 50, a new Cloud GameObject is created from the Cloud prefab, containing the CloudAI script for all cloud-related behavior.

**Score**

*NewScore*

NewScore contains an integer playerScore where the player’s score is saved. It also contains a the gameobject pointsManager for easy access. A string that stores all the valid characters for a player’s name. The three characters of the name as integers. An integer that stores the current initial that is selected. It also contains 2 text objects and 2 integers for the size when an initial is selected and not selected. In the Start() function the first thing done is set the gameobject and the players score. It then sets the value of the 3 characters to A which is zero since they are stored as integers. It then sets the availableChars string to all the characters that are valid. It then sets the text displays for all 3 characters and the score. It then sets the text object of the text display to be the player score. It then checks if the 2 text sizes have integers set. If they aren’t set it sets the selected\_size to 128 and the unselected\_size to 108. It then sets the selected character to the first one and makes it bold and increases it’s size. In the Update() the first thing that is done is set up the a and d key to move the selected character. It then checks to see if a key was pressed this frame and if it was using a switch case to set the selected character bold and of a larger size and makes sure that the others aren’t bold and are smaller in size. It then checks to see if the w key was pressed if it was it subtracts 1 from the current stored integer value for that character. It then checks to see if the s key was pressed and does a similar thing but instead of subtracting 1 it adds 1. It then sets the text objects to be whatever character is in the spot of the available character string that corresponds to the given integer. Then it checks for the enter key to be pressed and when it is. It creates a new string called Name and adds the characters together. Then adds the new score to the pointsManager and swaps the scene to the scoreboard.

*PointsManagement*

PointsManagement contains an integer playerScore which is used to store the player’s current score as the game goes on. It also contains two arrays TopTenScores and TopTenNames which store the scores and names of the top ten scores. In the Start() function the first thing that occurs is that it makes sure that the pointsManager object isn’t destroyed when a round ends. It then sets the playerScore to 0 and declares both arrays to have 10 spots. It then checks to see if the current scene is the game scene and if it is sets a text object to be a text field. In the Update() function it checks to see if the current scene is the game scene and updates the score if it is. It then has a function addPoints(int points) which is used to add points to the playerScore. It also has a function getScore which returns playerScore. The function isTopTen() is used to see if the score is greater than one of the top ten scores if it is it returns true otherwise it returns false. The function loadScores() is used to load in the top ten scores from the file. It returns true if it was able to retrieve the scores. The sortScores() function is used to sort the TopTenScores and TopTenNames arrays. The function addNewScore(int newScore, string newName) is used to add a new score to the array. It removes the lowest score and replaces it with the new score and resorts the two arrays. The WriteScores() function is used to write the top ten score out to a text file. The createBlankScores() function is used to create the file that saves the highscores and creates all ten scoreboard spots and fills them with the default values 0 AAA. It also contains 2 functions getTopTenScores() and getTopTenNames() which returns the value of the two arrays TopTenNames and TopTenScores.

*ScoreboardManagement*

ScoreboardManagement contains two arrays TopTenScores and TopTenNames, and stores the text field for displaying the top ten chart. TopTenScores holds the integer value of the top ten scores while TopTenNames stores the three character long names as a string. In the Start() function it starts by checking if a Points Manager object already exists in the scene. If one is present it saves the scores and names to the TopTenScores and TopTenNames variables to be writing to the scoreboard. If one isn’t present it creates a new one and reads in the values from the highscores the file. Then if the object exists it destroys the object. Then creates a string to store the text used to display the highscore. It then uses a loop to adds each of the scores and the name associated with it to the string for all ten of the top scores. It then sets the text in the text field to the value of the string.

**Lives**

*PlayerLife*

PlayerLife contains the variable for the player’s current life and the sprites for each value of the player’s remaining life. In the Start() function a number of important variables are set. It sets the player’s current Lives remaining to the value of five and also set the variable for the display box onscreen used for the lives. In the Update() function The sprite associated with the onscreen display for the player’s lives is changed based on what the current value of currentlives is. It also continuously checks to see if the currentlives variable reaches zero and if it does checks if the player has achieved a new high score. If they have it moves the player to the new score scene otherwise it moves straight to the scoreboard. The final function in PlayerLife is the subtractLife(int lives) function. It takes an integer as a parameter and subtracts the value from the players remaining life.

**Menu**

*Button*

The Button class contains skeleton code to be used by each individual button. Each instance of the script has a Sprite object for the clicked and unclicked sprites, which are toggled by the mouse state. Upon clicking, the clicked sprite is set; the unclicked sprite is set whenever the mouse releases or leaves the button’s boundaries. Each version of the script also contains specific extra code in the **OnMouseUp()** function to execute the button’s purpose (i.e. the ‘Play’ button loads the gameCowPanic scene while the ‘Quit’ button closes the game).

**Miscellaneous**

*CollisionCheck*

This script contains one function: **checkCollision()**. It takes in a LayerMask as a parameter, allowing it to check for collisions only on certain layers. The function first determines the coordinates of the top-left and bottom-right corners of the GameObject’s hitbox. To do so, half the object’s height and width are added to or subtracted from the object’s center position. The values are stored in variables that are then passed as parameters to the Unity function **OperlapAreaAll(),** which takes in two coordinates over which it draws a box and checks for collisions; it returns an array of Collider2D objects, whose corresponding GameObjects are then added to a results array using a for-loop to iterate each object and retrieve the attached GameObject. The results array is then returned.