Asteroids: Part 1 – The Player

Lesson Goals

1. Create the player ship from the game asteroids.
2. Learn how to use Unity’s input system package.
3. Learn how to make the player rotate and move.
4. Learn how to add acceleration to our player.
5. Learn how to make objects within the game world wrap around the screen.
6. Add some simple effects to make controlling the ship more satisfying.

Lesson Intro

1. In this lesson we are going to be making the classic arcade game **Asteroids**.
2. The goal of this lesson is to reinforce the vector math concepts we covered in the last lessons.
3. Instead of using rigidbodies we will be programming the physics of all the objects in the game.

Demo

1. By the end of this lesson, we will have something the looks like:  
   A white light in the sky

   Description automatically generated

GitHub Repository

1. For this project we are going to be creating a new GitHub repository. Go to GitHub and click the green **New** button to create a new repository.  
   ****
2. Set yourself as the owner and name the repository **Asteroids**.  
   A screenshot of a computer

   Description automatically generated
3. Set the repository to private:  
   **A black background with white text

   Description automatically generated**
4. **Be sure to add the Unity .gitignore file.  
   A screenshot of a computer

   Description automatically generated**If you do not, you will not be able to upload your project to GitHub.
5. Once all the options are set, click the green **Create Repository** button.  
   **A green rectangle with white text

   Description automatically generated**

Cloning the Repository

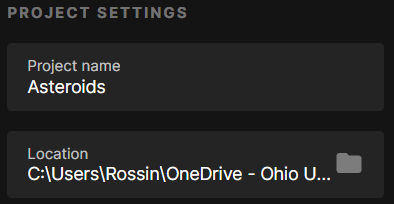
1. Open **GitHub Desktop** and clone the **Asteroids** repository that was just created. Go to **File > Clone Repository.**  
   **A screenshot of a computer program

   Description automatically generated**  
     
   Once selected, GitHub Desktop will return a list of all of your repositories. Select the Beetle Mania repository.  
     
     
   Lastly, choose a location that you want the repository to appear at. I recommend putting the repository on the desktop.  
   A black rectangular sign with white text

   Description automatically generated  
   Once set, click the blue **Clone** button.  
   
2. You should now see the Beetle Mania repository on your computer.  
   A yellow folder with a white paper in it

   Description automatically generated

Unity Project

1. Create a new 2D core Unity project. **Be sure it is being created in the local Asteroids repository.**   
   
2. Once the project is created, we just need to move the .gitignore file into the right place. First find the .gitignore file and then move it into the same folder the houses all the files for the project.  
   **A screenshot of a computer screen

   Description automatically generated**  
   **We need to make sure the .gitignore is in this spot for our project. If not, we will not be able to commit the project to GitHub. We will get a large file size error.**
3. With our project open change **aspect ratio** to **16:9**.   
     
   First switch over to the game tab.



Find the box labeled **Free Aspect**, click it and then from the drop down select **16:9 Aspect**.   
  
****

1. Be sure that **Toggle Gizmos** is active. In the top right-hand corner of Unity you should see the icon.  
     
     
   Clicking it will make the icon turn blue. This means you should be able to see game gizmos in the scene.  
   
2. Lastly select the **Main Camera** and change the **Background Color** property to **Black**.  
   A close-up of a computer screen

   Description automatically generated

Art

1. Create fold called art and drag in the sprites for the game.  
   A screenshot of a computer

   Description automatically generated
2. Click on one of the sprite sheets and set the **Sprite Mode** to **Multiple**.  
     
     
   Then set the **Filter Mode** to **Point**.  
   A black and white screen with white text

   Description automatically generated  
     
   Lastly, hit the apply button.  
     
     
   **Do this for each of the sprite sheets.**
3. Next, we need to slice up the sprite sheets into individual images. Click the **Asteroids\_Large** image and click the **Sprite Editor** button in the inspector.  
     
     
   In the sprite editor click the **Slice** button at the top left-hand corner. Set the **Type** to **Grid By Cell Size** and then the **Pixel Size** to **64** for both the X and Y.  
     
     
   Once done, hit the slice button.  
     
     
   Lastly just hit the **Apply** button in the sprite editor window.  
     
     
   You should now have separate sprites for the large asteroids.  
   A screenshot of a computer

   Description automatically generated
4. Repeat the steps above for the other sprites. However, we need to change the pixel size for the remaining two sprite sheets.  
     
   For the **Asteroids\_Small** sheet we set the pixel size to **32**.  
   A close up of a number

   Description automatically generated  
     
   For the **Ship** sheet we set the pixel size to **16**.  
   A close up of a number

   Description automatically generated

Player

1. Add in an **empty** object and call it **Player**.
2. Next drag in the first sprite of the player sprite sheet. This will add the ship sprite to the game. Make this sprite a **child** of the player object.  
     
     
   With the ship sprite select set the **Position** to **0** for the **X Y** and **Z** values. Then set the **Scale** to **2** for the **X** and **Y** values.  
   A screenshot of a calculator

   Description automatically generated
3. Lastly create a script called Player and then attach it to the player game object. **Be sure the script is not attached to the sprite.**A white square with green symbol

   Description automatically generated

Input System

1. Go to **Window > Package Manager**. This will open the **Package Manager** window.  
   A screenshot of a computer program

   Description automatically generated  
     
   **Packages** are a bundle of assets that can be added to our project. A package can contain anything from shaders to scripts to materials to anything in between. Typically, we will bring in a package to our project to add some functionality.
2. For this project we are going to add the **Input System** package. This package allows us to create input maps that store what buttons we will use for the inputs for our game.   
     
   To download this package, in the package manager window, at the top-left corner find the **Packages** drop down and change it to **Unity Registry**.  
   A screen shot of a computer

   Description automatically generated
3. Then in the **search bar** search the words **Input System**.  
     
     
   You should then see the input system package.  
   A screenshot of a computer

   Description automatically generated
4. Click the **Install** button at the top-right. Unity will then begin to download the package. This may take a few minutes.  
     
   After a short while, you will be prompted to restart Unity. Click **Yes, then click to save any changes.** The let Unity restart to properly install the package.
5. With the package now installed, right click somewhere within the **assets** folder and select **Input Actions** at the bottom of the drop down. Rename the new InputAction to **Player**.  
   A white paper with a blue lightning bolt

   Description automatically generated  
     
   Double click on the InputAction, this will open the Input Actions window.  
   A screenshot of a computer

   Description automatically generated
6. Click the **+** button where it says **Actions Maps**. Rename the new action **Player**.  
   A screenshot of a computer

   Description automatically generated  
     
   Create a new Action Map will also create a new action.  
   A screenshot of a computer

   Description automatically generated
7. Select the new action and rename it to **Move**. Then set the **Action Type** to **Value** and the **Control Type** to **Vector2**.  
   A screenshot of a computer

   Description automatically generated
8. Right click on where it says **<No Binding>** and select **Delete.** Then hit the **+** button to the right of **Move** and select **Add Up/Down/Left/Right composite**. Name the new binding **WASD**.  
   A screenshot of a computer

   Description automatically generated  
     
   Select the Up key and set the **Path** to **W [Keyboard]**. You will have to click on the empty path button and then type in the name of the keyboard key you wish to use.  
   A black screen with white text

   Description automatically generated  
     
   Repeat the process for the other keys.  
   A screenshot of a computer

   Description automatically generated
9. Once complete check the **Auto-Save** box on the top of the window. This will save any change we made to the player input.  
   Close out of the window once you are done.
10. Return to the **Player** game object and add the **PlayerInput** component. Drag the **Player InputAction** we create into the **Actions** slot on the component.  
      
    This effectively connects out InputActions to the player. So now the player will be able to receive input.

Reading Player Input

1. Open the player script and at the top add the library:  
   
2. Next add the variable:  
   A black background with white text

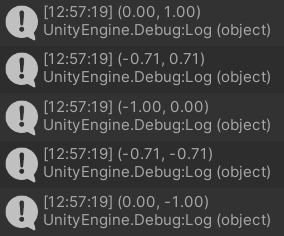
   Description automatically generated  
   **Input\_Vec** will store the input that we are receiving from the player input component.
3. Then add the function:  
   A black screen with white text

   Description automatically generated  
   This function will read in the input that the player input component is receiving. We know that the Move input action is of type Vector2 so we are storing it in the input vector we created.
4. Lastly in the **Update()** lets add the code:  
   A computer screen with white text

   Description automatically generated  
   We will just output what the input vector is each frame of our game.
5. Save the script and return to Unity. Go to the **Player Input** component. Set the **Behavior** to **Invoke Unity Events**.  
     
     
   Then find the **Events** drop down, click the drop down arrow and repeat the process for the player drop down. You should then see a box with the **Move** action that was created for the player.  
   A screenshot of a computer

   Description automatically generated  
     
   Click the **+** button at the bottom of the box. Drag the player from the **Hierarchy** and drop it into the empty slot.  
   A screen shot of a computer

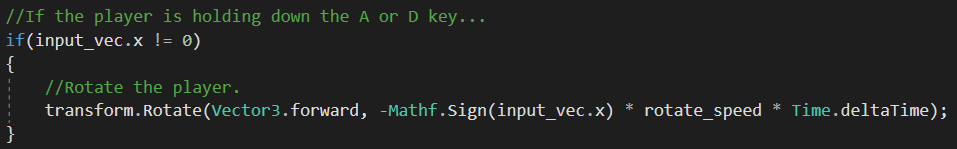
   Description automatically generated  
     
   Click the box labeled **No Function** and from the list select **Player > CaptureMoveInput**.  
   A screenshot of a computer

   Description automatically generated  
     
   In total, we just was allow the player input component to call the CaptureMoveInput() function on the player. So now whenever the player presses the WASD keys, the function will be called and the input of the player will be stored in the input\_vec.
6. If we run the game, we should see the player’s input being displayed to the console.  
     
   If the player is not pressing any keys, the X and Y values for our vector will be 0.  
     
   If the player presses just the A or D key the X value of the input vector will be -1 or 1. If the player presses just the W or S key the Y value of the input vector will be -1 or 1 as well.   
     
   However, if the player presses a combination of the A or D key and the W or S key the X and Y values will be between -0.71 and 0.71. This is because the input is being **normalized**. We will need to keep this in mind when we code our player’s movement.

Player Rotation

1. So now that we have our input setup, let’s get to work on making our player move. In asteroids, the player moves in the direction that the ship is pointed in. So, let’s first start by making our player rotate.
2. In the **Player** script let’s add the variable:  
   A black background with white text

   Description automatically generated  
   **Rotate\_Speed** is the amount in degrees that our ship will rotate by.
3. In the **Update()** function, remove the code that outputs the player’s input to the console. Let’s then write the code:  
     
   The transform’s **Rotate** method first asks what axis we will rotate around and then the amount of degrees to rotate. Our ship will rotate around the forward axis by the angle amount of rotate\_speed \* Time.deltaTime.  
     
   If we save the script and run the game, we will see that our ship will be rotating to the left.  
   A black screen with a white triangle

   Description automatically generated
4. Now we just need to make our ship rotate based on our player’s input. Back in the Player script lets edit our code to say:  
     
   We add an if statement that checks to see if the input vector’s x value is not equal to zero. From our debug information that we outputted earlier, we know that the only way the x value will be zero is if the player is not pressing the A or D key. So, we are just checking to see if the player is pressing one of those buttons.  
     
   We then added a few words of code to our Rotate function. By multiplying our angle of rotation by the sign of our input, we can now control what direction our ship will rotate in. We also invert the sign to make sure our ship will rotate in the correct direction.
5. If we save the script and run the game, the player should rotate based on what keys the player is pressing.  
   A white triangle in the middle of a black background

   Description automatically generated

Player Movement

1. Now that our player can rotate, let’s make it so they can actually move. In the **Player** script let’s add the variable:  
   A black background with white text

   Description automatically generated  
   **Max\_Speed** is the maximum speed that our player will be able to move.
2. We have our movement speed but what we need to know now the direction we want to make the player move in. Luckily for us, Unity is already keeping track of the direction the ship is facing.  
     
   An object’s transform has three properties that are used to tell the orientation of the object. These properties being **up**, **right** and **forward**. The image below shows our ships up, right and forward directions when the ship has not rotated.  
   A drawing of a pyramid with arrows

   Description automatically generated  
     
   However, if we go to rotate the ship along the forward axis like doing, look at what happens to each of the directions.  
   A drawing of arrows pointing to right and right

   Description automatically generated  
     
   Unsurprisingly the directions of the ship are rotated. But not that the direction that was used as the axis of rotation.  
     
   So what this means is that we can use the ship’s transform.up property to determine the direction that our ship should move in.
3. But what exactly is the transform.up property anyways? To show what it is, in the **Update()** function of the **Player** script lets write the line:  
     
     
   If we save the script and run the game. The first console message will be:  
   A black and white text

   Description automatically generated  
   This is the default Vector3 value that represents the up direction.  
     
   However if we go to rotate our ship, this value will change:  
   A screenshot of a black and white screen

   Description automatically generated  
   Things should hopefully start to look a little familiar. You should notice that the vector is being normalized and if you remember from the last lesson a normalized vector gives us the percentage of how far to move along the X, Y and Z axis.   
     
   So to answer the question, the transform.up property gives us a normalized vector that represents the up direction of an object.
4. Hopefully that wasn’t too confusing, let’s go ahead and put this into practice. If we head back into the **Player** script. Let’s delete the Debug.Log() line that we added in the update and replace it with the code:  
   A black screen with white text

   Description automatically generated  
   We first check to see if the player is pressing the W key. We know the player is pressing this key if the input\_vec’s y value is above zero.  
   If so, we move the player by adding to its position. The amount we add is the up direction of our ship multiplied by the distance to move (max\_speed \* Time.deltaTime).   
     
   To run through the math a little, if the distance to move is 5 and the up direction is (0,1,0), the ship will move 5 units along the Y axis. If the distance to move is still 5 but the up direction is (0.71, 0.71, 0), the ship will move 3.55 units along the X and Y axis.  
   **Remember that the numbers inside the normalized vector are percentages of how far the object will move.**
5. If we save the script and run the game, we should be able to have the player move in the direction the ship is pointed in. The ship will only move if the player is holding down the W key.

Acceleration

1. One problem that you may have noticed is that our ship can start and stop on a dime. This doesn’t really make it seem like the ship is in space. So, to fix this issue, lets add some acceleration to make our ship feel a little floatier.
2. In our **Player** script, lets add the following variables:  
   A black background with white text

   Description automatically generated  
   **Acceleration** will control how fast our ship will speed up.  
   **Velocity** will store what direction and how fast our ship moves.
3. In the **Update()** function, inside the if statement where we check to see if the player is holding the W key, lets change the code to be:  
   A screen shot of a computer program

   Description automatically generated  
   Instead of adding to our position we will add to the velocity. We will make sure to add acceleration in the direction of the ship’s up vector.  
   Then we need to make sure the player cannot travel faster that the move speed. To do this we limit the velocity by clamping its magnitude.
4. Lastly under the if statement lets write the code:  
   A black background with white text

   Description automatically generated  
   All this code does is move our player by the velocity vector.
5. If we save the script and run the game, the player will now have a sense of weight to them. The player will not be able to slow down since there is no friction in space.

Friction (Optional)

1. But since we are game developers, we do not need to adhere to the laws of the universe. We make our own rules. So let’s add friction to our ship.  
     
   First we just need to declare a variable:  
     
   **Friction** will control how fast our ship will slow down.
2. Then under the if statement where we are checking to see if the player is holding down the W key, lets write the else statement:  
   A black screen with white text

   Description automatically generated  
   To decelerate we use the Vector3’s MoveTowards method. This will make one Vector3 equal to another Vector3 by a rate. We are making the velocity vector move towards Vector3.zero by a rate of friction \* Time.deltaTime.
3. Saving the script and running the game we can see that the player will now slow down after a short while if no key was pressed after accelerating.

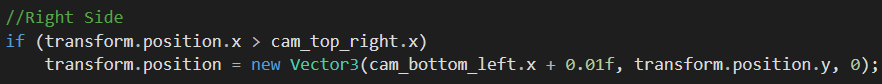
Screen Wrapping

1. Now that our player can properly move, we next need to keep them within the bounds of the level. Currently there is nothing stopping the player from leaving the view of the camera. To remedy this let’s add screen wrapping to our game.
2. In the **Player** script let’s add the variables:  
   A black screen with white text

   Description automatically generated  
   **Cam** will hold a reference to the camera that is in our game.  
   **Cam\_Bottom\_Left** will store the camera’s bottom left point in screen space.  
   **Cam\_Top\_Right** will store the camera’s bottom left point in screen space.
3. Then in the start function lets write the code:  
   **A screen shot of a computer code

   Description automatically generated**  
   First we get the camera’s top-right point in pixels. We don’t need to do this for the bottom-left point since the bottom left corner will be (0,0,0).  
   We then convert the bottom-left and top-right points to world space. We used this function before when we spawned dots in for our first game.  
   Lastly, we just out put what these two points are in world space.
4. Save the script and drag the Main Camera in to the cam slot on the player script component.  
   A screenshot of a computer

   Description automatically generated  
     
   If the run the game is ran,the console should output the messages:  
   **A blue and black box with white text

   Description automatically generated**  
     
   To check and see if these are accurate world space coordinates we can move our ship to both points. If we do, the ship should be on the edge of the camera which is what we want.
5. Back in the script in the **Update()** function, under the code to move our player, lets write the line:  
     
   Here we check to see if the player’s x value is greater than the camera’s right most edge. If so, we set the player’s x position to the left most side of the screen plus 0.01.  
     
   The 0.01 distance we add is just enough to nudge the player past the left most side of the screen. This is just a little precaution to avoid the player getting stuck teleporting between both sides of the screen.
6. If we save the script and run the game, we can see that the player will teleport to the opposite end of the screen when it hits the right most edge.
7. We can hop back in our scrip and add the remain code for each edge of the camera.  
   A screen shot of a computer program

   Description automatically generated
8. Saving the script and running the game, the player should now be able to wrap around the screen no matter what side they try to leave from.

Screen Wrapping Fix

1. There is one small issue with our screen wrap code. Currently the screen wrap code is checking if the player’s center position has left the screen, meaning the player doesn’t fully exit the screen before it is teleported. We want our player to fully exit the screen before the code teleports the player.
2. To fix this we are going to add a radius around our player. In the **Player** script add the variable:  
     
   **Wrap\_Radius** is the extra length the player will need to take into account to screen wrap;
3. To help visualize this lets add the **OnDrawGizmos()** function and write the code:  
   A screen shot of a computer

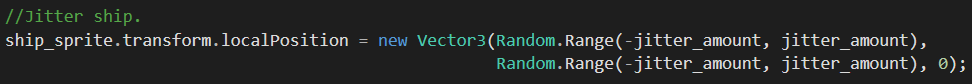
   Description automatically generated  
   The OnDrawGizmos() allows us to draw debug shapes for our game. These can be very helpful for when we need to visualize elements in our game.  
   We first set the color of our gizmos that we are going to draw to the color red.  
   We then draw the outline of a sphere. The sphere’s position is that of the player and it has a radius of wrap\_radius.
4. Save the script and let’s take a look at our ship. Be sure to have gizmos enabled.  
   A white triangle with a red circle in the middle

   Description automatically generated  
   The radius we add can be visualized as a circle our ship. This circle needs to fully exit the screen before the ship can be wrapped around to the other side.
5. Let’s go back to the **Player** script and take this new radius into account when wrapping the ship.  
   A computer screen shot of a program code

   Description automatically generated  
   All we are doing is offsetting the player’s position by the wrap radius.
6. If we save the script and run the game, the player will now fully move off screen before being teleported to the other side.

Ship Visuals

1. Lastly let’s add a few small details to our ship to make it feel better to play.  
     
   First we are going to make the ship jitter around when it is moving. In our **Player** script, lets add the variables:  
   A black background with white text

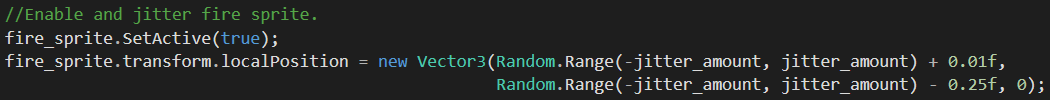
   Description automatically generated  
   **Ship\_Sprite** will hold a reference to the child sprite object in our player.  
   **Jitter\_Amount** is how much the ship sprite will jostle back and forth when moving.
2. Next in the **Update()** function, in the if statement for if the player is holding down the W key, lets add the line:  
     
   Here we are setting the ship sprite’s local position. **Local Position** refers to the object’s position relative to it’s parent object. Since the ship is a child of the player, we want it to move when the player does so, we use local position.  
   All we are doing is offsetting the local position of the ship by a random jitter amount.
3. Saving the script and returning to unity, we just need to give the player a reference to the sprite.  
   A screenshot of a computer

   Description automatically generated  
     
   Running the game we can see that ship will now jitter around when it moves.
4. Next let’s give the ship a fire sprite that will shoot out the back of it. First, drag in the second sprite of the ship’s sprite sheet into the screen.  
   A white triangle with a black background

   Description automatically generated with medium confidence  
     
   Make this new sprite object a child of the player and **rename** it to **Fire Sprite**.  
   A screenshot of a game

   Description automatically generated  
     
   Then set the position and the scale of the sprite to:  
   A screenshot of a black box with white text

   Description automatically generated  
     
   The sprite should now be sitting right behind the ship.  
   A white triangle with a triangle in the middle

   Description automatically generated
5. Back in the **Player** script lets add the variable:  
   
6. In the **Start()** function let’s write the code:  
     
   We are making the fire sprite not active at the start of the game. This effectively disables the object. A disabled object will not appear in the game.
7. Then in the **Update()** function, under the code where we are shaking the ship lets write the code:  
     
   All we are doing is activating the fire sprite so it will be seen in game.  
   We also jitter the fire around as well. We make sure to add on the ships offset position so the fire can stay right behind the ship.
8. Lastly, we add an else statement to the if statement we have been writing code in. We then write the code:  
   A screen shot of a computer

   Description automatically generated  
   The else statement will run when the player is not holding down the W key. If that is that case, we deactivate the fire so we can no longer see it.
9. Save the script and give the player a reference to its fire sprite.  
   A screenshot of a game menu

   Description automatically generated  
     
   If the game is ran now, we can see the fire behind the player. The fire will only appear when the player is pressing the W key.

GitHub

1. Push the project to the GitHub repository.