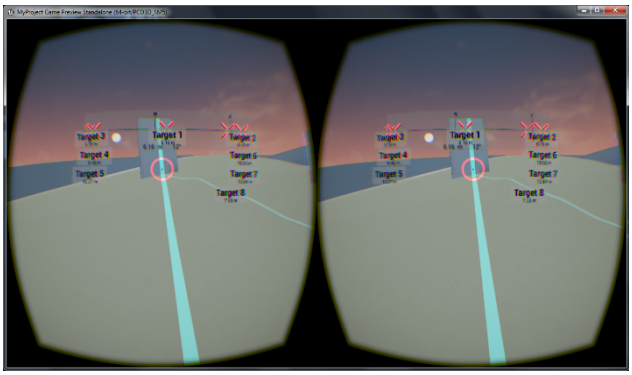
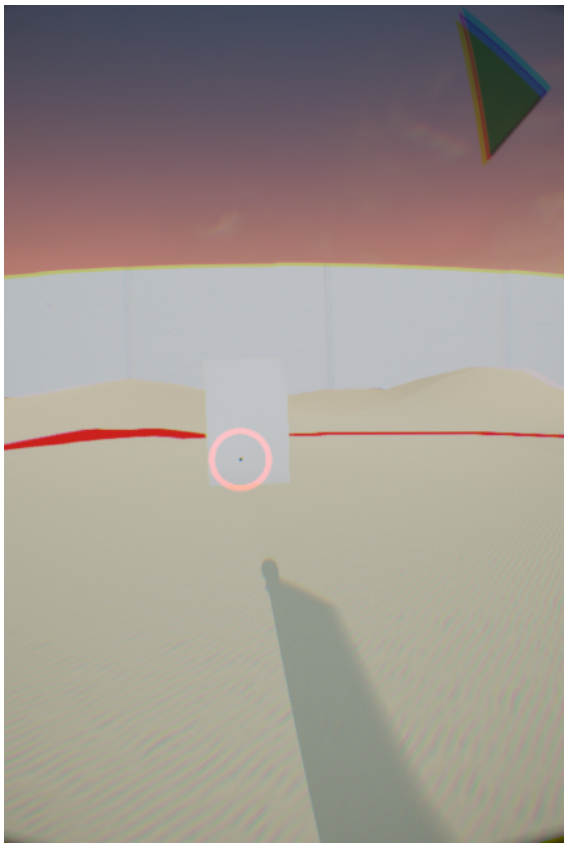
Oculus Rift / Unreal Engine 4 Simulation

Design



The UI is designed to be instantly intuitive to the user, by conforming to standards established either by reality or by video games; I operated on the assumption that most users would be young servicemen, and would therefore be relatively familiar with UI elements such as the **compass bar** at the top of the screen as well as the Targeting Highlights. These features are much less intuitive to users who have not played games before, so this is a possible weakness in the UI.



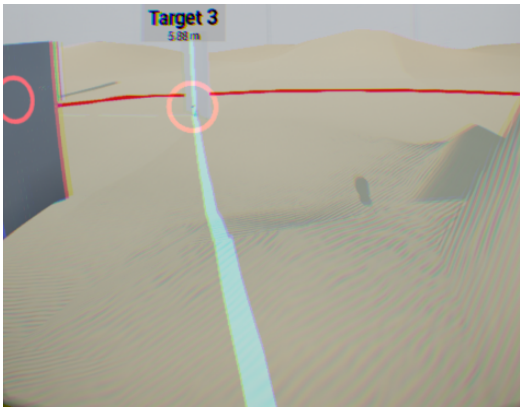
Arrows

The **arrows** on the side of the screen direct the user to a waypoint that is offscreen. Initially, the arrows indicated the location of an offscreen target, when there was only one active target in the world, but that has since been altered. Now, any target within the **range indicator** is considered active. The **arrows** are drawn to point the shortest distance the user would have to turn to point toward the waypoint. This will increase in scale when the waypoint is an extremely large angle in either direction. In addition, the **arrows** will indicate where in the user's vision the object will be once they turn to see it, helping them to orient themselves as quickly as possible.

Relevant Functions: Draw Arrows

Target Highlights

The red circles that are shown in the image are placed at the location of each target in the simulation to draw the user's attention to an important target. The **highlight** could be used to keep the user informed about the location of an object that the user is facing, but which may be obstructed by a building or other geometry, so that the user remains informed as to the objects position at all times.

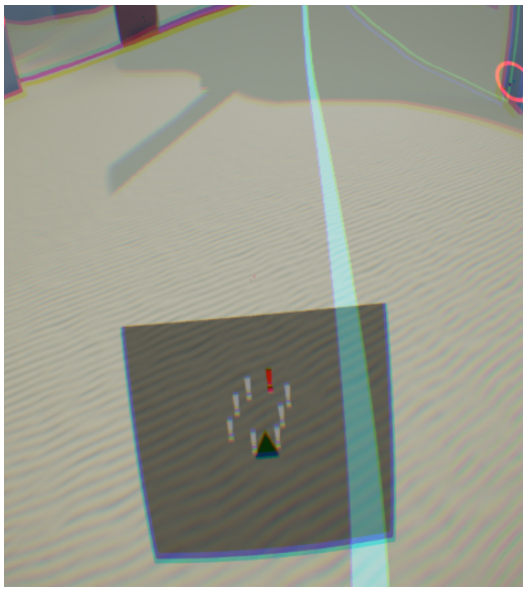


Waypoints

Waypoints are used to show the user a path they are meant to travel along. In the simulation, the user will be shown the path, projected onto the ground, of their current waypoint and the next waypoint, in a more faded and thinner line. This could be altered to show more waypoints, and these waypoints can be specified to be any point.

Unfortunately, the waypoints are drawn through the same process as the Range indicator. Thus, the same problems with topology are present. As a workaround, the waypoints would likely be placed on the map, as well as a marker placed on the Compass Bar directing the user toward the waypoint.

Relevant Functions: Ground Indicators



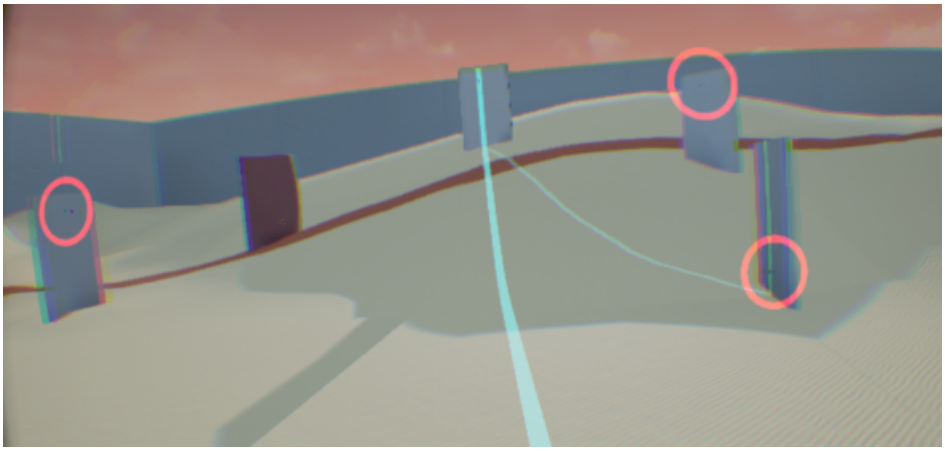
Map

This **map** appears when the user looks down at an angle >30 degrees. Once the **map** begins to appear, the **map** begins to fade in over the course of about 1/2 of a second. Once the user looks back up, the **map** fades out again over the course of 1/3 of a second. This allows the user to quickly look at a map without the use of their hands, and without obscuring the user's vision.

The downside to this method, as opposed to a game-style **minimap**, is that the **minimap** would be shown in a corner of the users vision at all times. This would allow constant awareness of the **map**, without the need for any movement at all on behalf of the user. However, there are disadvantages to the **minimap.** One problem is that **map** obstructs the vision of the user. Reducing the size of the **map** in order for it to obstruct less vision diminishes the minimap's usefulness, particularly when the user has the Compass Bar to inform the user of nearby targets.

Relevant Functions: Manage Minimap; Draw Targeting; Draw Distances; Place Enemy Indicators

Range Indicator



The **range indicator** highlights a ring around the user about half a meter thick, and 50 feet away from the user. This provides the user distance information regarding a point of interest without as much guesswork. In the simulation, the **range indicator** also determines how targets are acquired; anything within the ring will be given a **target highlight** and tracked on the **compass bar**.

One concern with the implementation of the **range indicator** is that it uses a shader. In a game, a shader is what gives a 3d object texture and color. This means that there isn't a true projection onto the terrain, but rather that the terrain, and all of the data of its topology, is used to determine if the point on the terrain should be highlighted red. Because of this, transition out of the simulation into an AR environment would require data regarding the precise topology of the surrounding area, and be able to draw a highlight over these points.

Relevant Functions: Ground Indicators; Check Range for Targets

Compass Bar

The **compass bar** is a common element in many open-world video games, with good examples being Skyrim and the Batman Arkham series. In these games, there are large numbers of points of interest in the world, and the **compass bar** helps inform the user of these points, without overloading them. The **compass bar** will not only show the direction the user is facing, but can also inform the user of what category an object belongs to, and where it is in relation to the user. In most cases, an object will only be shown on the **compass bar** when the user is within a certain range of the object, ensuring that only relevant objects are displayed to the user.

In this simulation, the **compass bar** has all of the features detailed above. However, In its current implementation, the **compass bar** does not show different types of objects, but doing so would be possible and relatively easy given the current implementation.The boxes of text below each indicator help to serve as a way for the user to be informed about the number of targets in a given location and their distance, without moving their head. Because the targets are acquired by distance, the system can automatically keep out irrelevant objects from being included, and a hard limit on the amount of objects shown on the **compass bar** can also be implemented.

Relevant Functions: Check Range for Targets; Draw Targeting; Position Compass Markers

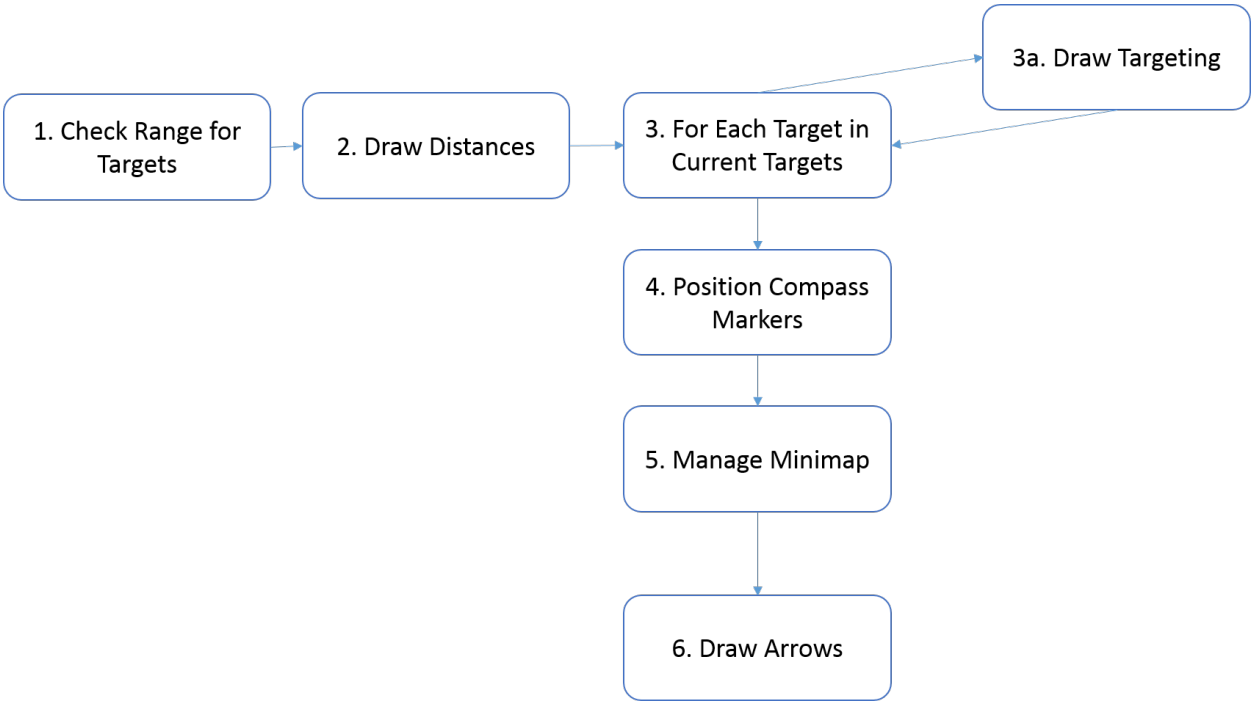
Implementation

Because of the nature of the project as a short term prototype, all "code" for this project is done in Epic's proprietary Blueprint visual coding system. This makes it near impossible to reuse outside of UE4, HOWEVER, the benefit to using Blueprint is that the code is much easier to understand at a glance. Each function is laid out as a graph, and can be commented easily. This makes it much simpler for a future developer to use this project's concepts for the UI without having to understand a coding language. Instead, they need only understand the basic math behind the process, and can follow the graph easily.

Flow Diagram

Event Tick

Called Every Frame



Major Functions

The functions detailed below are the primary functions used in managing the UI. There are additional convenience functions used in these functions, but most are very simple, and are just in place to simplify the flow of the blueprint visually.

Check Range for Targets

1. For Each possible Target
2. Set Boolean variable ‘is Target in Array’ to FALSE
    A. IF the target being checked is < 1000 cm from the user’s location
        I. For each enemy indicator on the compass marker
            i. IF the target enemy indicator is referring to the target being checked (confirmed by a string value representing the target’s unique name)
                1. Set ‘is target in Array’ to true
                2. Break out of the for each loop

II. IF ‘is Target in Array’ is FALSE

i. Add the target currently being checked to the array “current targets”, which contains all targets within 1000 cm of the player.

ii. Create an enemy indicator for the target in the array.

iii. Set the name property of the indicator to be the name of the target

iv. Add the indicator to the right panel

v. Set the position data of the indicator so that it lies on the compass bar, and is slightly offset to be properly 3D

vi. Add the indicator to the array of ‘Right Compass Enemies’ which contains all of the enemy indicators

vii. Create a compass detail, which will show the name of the target in the array below the Enemy Indicator.

viii. Set the detail’s scale so it fits properly

ix. Set the name variable of the detail to the name of the target in the array

x. Add the detail to the array of right compass details

xi. Add the detail to the right panel

xii. Set the position data similar to step 4, with larger offsets on the left and right margins, so that the detail appears to float in front of the indicator in the user’s perception.

xiii. Repeat steps 2-12, for the LEFT side of the UI

B. ELSE (i.e. if the target is > 1000 cm from the user’s location

I. For each enemy indicator in the array of compass enemies (either left or right, will not matter which)

i. IF the indicator’s name matches the current target in the array’s name

1. Set the ‘removal Index’ Integer to be the current index in the foreach loop.

II. IF removal Index has a valid index (the variable will have a default value of -1)

i. Remove all objects at removal Index, in the Right and Left arrays for both Compass Enemies and Compass Details from their parent (This can all be performed in one step, by connecting these arrays to a single node). This will not remove them from the array, but will instead remove them from their panel in the UI

ii. Remove the objects referenced in step 1 from their respective arrays.

iii. Remove the object at Removal Index from Current Targets.

Draw Distances

1. Get the map object from the user.
2. Call Place Enemy Indicators based on the user's location, and passing in the array of all targets
3. Call Write Enemy Distances, passing in the array of current targets, and the user's location.
4. Set the distance text property to be the length of the vector from the current waypoint's location to the user's location, converted from cm to m.
5. Set the heading text property to be the HMD yaw + the user's body yaw
6. Rotate the map user indicator, passing in the result from step 5. This keeps the map oriented to the user's rotation.

Place Enemy Indicators

1. For Each element of the array given to the function

A. Get the location of the target, and use that with the vector given to the function to get a new vector from the target to the given location.

B. Eliminate the Z (vertical) component from the new vector from step (a.) and pass the new 2D vector to Place Enemy Indicator, with the current index of the Loop

Place Enemy Indicator will then do the following steps

1. Scale down the 2D vector given to 7.5% it's original size, to fit on the map better.
2. Convert the shrunk vector from the world coordinate system to the screen coordinate system the map uses
3. The index given to the function will get an enemy Indicator from the array, and set that indicator's translation to the corrected position

Draw Targeting

This function is responsible for getting the location of a given target relative to the users position. It is used in conjunction with the targets to place the Target Markers on the Compass Bar.

The relative location is determined by getting a dot product of the vector from the user to the target and the vector representing the direction the user perceives as being their right. This dot product is then saved to an array of dot products based on the index of the target within another array of targets being tracked.

Manage Minimap

1. Gets the size of the user's view port, and finds the point in the middle of the screen horizontally, and 7/8ths of the way down the screen.
2. This point is then converted from a point on the user's screen into a point in the game world.
3. The location of the point above is then added to 80x the DIRECTION of that same location in relation to the user.
4. The map is placed at the point from step 3.
5. The map is rotated based on the orientation of the HMD, adding 90 degrees to both the pitch and yaw of the HMD, then being sent to the map. yaw is sent to yaw, but because of the different orientations of the map vs. the HMD, the pitch of the HMD is sent to the ROLL of the map.
6. IF the HMD's pitch is < -30 degrees (i.e. if they are looking down more than 30 degrees below level)
    A. The opacity of the map is increased by 3% every frame until it either reaches 100% opaque, or the user looks above -30 degrees again.
7. ELSE (i.e. if the user is looking ABOVE -30 degrees)
    A. The opacity of the map is decreased by 5% every frame until either it is at 0% opacity or the user is looking below -30 degrees again.

Position Compass Markers

1. Set the translation of all of the Tick markers (cardinal direction lines on the compass) based on the yaw of the HMD + the yaw of the user mapped from -360:360 to -800:800 as the x position. This will allow the tick marks to move as the user does, helping orient them, and showing the direction of the enemy indicators.
2. IF the compass bar is meant to be visible, go to (a.) below. Otherwise simply continue to step 3.
    A. Check if the tick markers are still on the compass bar. If the player turns their head enough so that, for example the marker for south is now off of the compass bar, it must be made invisible here. This is done for every tick marker, each having a different point at which they would become invisible.
3. For Each target in the ‘Current targets’ array. (Every target currently in range of the player)
    A. IF the target being checked is in front of the camera
        I. Find the position the target’s indicator should be placed on the bar. This is done by getting float stored in the array ‘Right Dot Products’ at the index of the target being checked. This value is mapped from -1:1 to -200:200, and set as the x position of the enemy indicator and target detail.
        II. The ‘slot’ of the target detail is now determined, by adding 200 to the x position, dividing by the width of the detail, and truncating the value. this is used as an index to access the array “detail slots”. The value at this index is incremented, and then

multiplied by the height of the target detail, and added to the base y position of the target detail, so that no detail will overlap another, but will instead be stacked vertically. This is the y position of the target detail

B. ELSE (i.e. if the target being checked is BEHIND the camera)

I) The scale and opacity of the enemy indicator is reduced, giving an idea of the target being different than just to the left or right of the user.

II) The element in ‘Right Dot Products’ at the index of the current target being checked is used to determine if the target is to the left of the player

III) IF the target is to the Left of the player

i. The leftmost(i.e. index of 0) slot in the ‘Detail Slots’ Array is incremented and used to determine the y position of the target detail

ii. The Target Detail and enemy indicator are placed at an x position of -200

IV)ELSE

i. The rightmost(i.e. index of array.length -1) slot in the ‘Detail Slots’ Array is incremented and used to determine the y position of the target detail

ii. The Target Detail and enemy indicator are placed at an x position of 200

Draw Arrows

1. IF the target passed into the function is visible (i.e. not hidden, not necessarily in view)
   A. Get the dot product of the target with the user's right vector. This will return a positive value if the target is to the right of the user, and a negative value if it is left of the user.
   B. IF the target's position can be projected onto the screen of the user, and have that position be within the bounds of the screen (i.e. if it is in view of the user)
      I. Don't draw the arrows. Hide them
   C. ELSE (i.e. if the target is NOT in view)
      I. Use the dot product from step (a.) to determine if the target is left or right of the user. this will determine which arrows should be used.
      II. Check the Z coordinate (vertical) of the impact point (where a vector beginning at the HMD and ending at whatever is the first solid object in the direction of the HMD's orientation), and the Z coordinate of the target. the difference between these two values will be the y position(vertical, in screen space) of the arrow on the UI.
      III. The dot product from steps (a.) and (i.) is reused again, this time taking the absolute value and clamping it between 0.1 and 0.7. This is then mapped to be between 0.05 and 0.2, and reversed so that a large dot product results in a small converted value, and vice versa. This is the Y scale of the arrow. The arrow will grow the farther away the user turns from the target, helping the user find the target.
      IV. Finally, the arrows are made visible

Leap Motion Post-Mortem

Initially, the Leap Motion hand tracker was used to select targets, and to dismiss or show parts of the UI selectively. However, the implementation of the Leap was not incredibly accurate, nor was it especially ground breaking. In addition, the Leap is not very feasible in the current AR environment, since it has a relatively limited capture area, and would the user would need to conform to this limitation, rather than the Leap conforming to the user. Finally, the leap relies heavily on a fully visible hand, so if the user was holding something in their hands, the device would become much more difficult to use.

However, implementing Leap support was very easy. Through the use of a 3rd party plugin, it became very easy to build in Gesture recognition as well. The problem did not lie in the use of the API, but seemingly in the actual recognition of gestures within the simulation, either because of the Engine, or the Leap SDK.

Tips

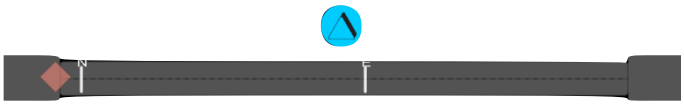
Make as many elements as possible 3D. The map, for example, is a 3D object, which gives it a sense of depth and realism.The compass bar, however, represents a different method of making objects 3D. Because it was difficult to make the bar follow the player properly, the bar was placed in a UI layer, meaning it is not treated as a 3D object. To simulate the 3D effect, each UI panel is shifted toward the center of the screen, horizontally. This gives each eye a slightly different perspective, the basis of 3D vision. In addition, the compass bar details are shifted again, making them appear closer to the player than the bar itself, heightening the effect of the 3D illusion. Doing this requires more work, and in many cases you must hard code the slight offsets necessary to make these objects seem 3 dimensional, but if necessary, this method can be used to great effect.

It is imperative to maintain a high framerate for both AR and VR. This is much harder in VR, because the simulation needs to render the entire world as well as the UI. In VR, either ensure you have a powerful GPU (Oculus recommends an Nvidia GTX 970 or above) or reduce graphical effects such as textures, lighting, and other graphics-intensive tasks to keep the framerate up. Oculus suggests that 90 fps is ideal, and anything below 60 fps is unacceptable. Below 60 fps, it is much easier to notice the slight disparity between the user's movements and the simulation's reaction, and the same would be the case in an AR environment as well. In AR, this will not be likely to induce motion sickness, but it could increase eyestrain.

Atheer Unity Demo

Introduction

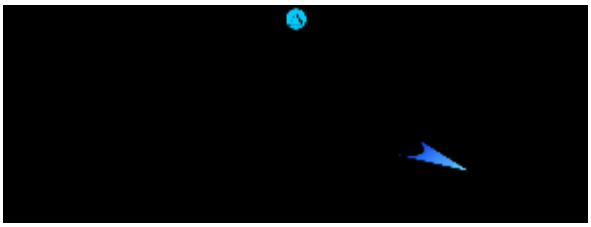
- Space Bar / One finger tap- Cycle different arrow types
- R key / Two finger tap - Reset Heading to 0°
- Q key / Three finger tap - Cycle target positions

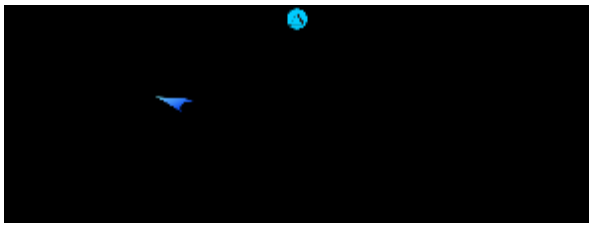


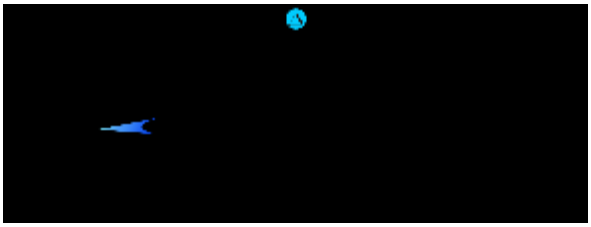
Design

Most of the design remains the same from the Oculus Prototype. However, there have been several changes, which are documented below.

Interrupt Arrow Indicator







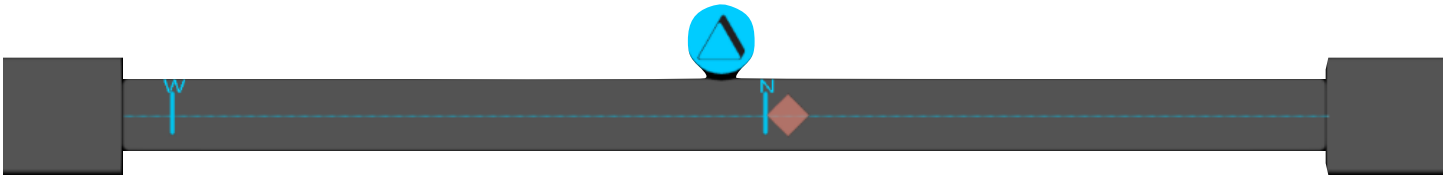
The **arrow indicator** from the previous prototype was initially used to show when the target was offscreen. Eventually, it was used only to show the direction to the current waypoint of the user. In this prototype, the **arrow** now points toward the **interrupt task** that will be given to the user.

In addition, the **arrow** is now always visible, and points in 3D space to the object, much more like a compass than the previous arrow. This was much more effective in indicating the direction to the target, as well as offering more clarity when demonstrating the target is directly behind the user. Making the arrow point in 3D space required additional changes to better give the user a sense of depth. The arrow was changed from a flat color to a gradient, and the shapes were made more complex. These changes give better feedback to the user as the arrow rotates. The previous simple triangle could be confusing, as users were often unable to see where it was pointing, and how it was rotated. Similarly, the arrow now bounces in the direction it is pointing, which both alerts the user to the arrow's presence and shows easily the direction of the target location.

Compass Bar

The **compass bar** has remained the same from the Oculus prototype in terms of the front 180°, but has been overhauled for displaying targets behind the user. Originally, the target indicator would slowly shrink and fade out of vision as the target approached being directly behind the user, but this is not ideal, because the user could become unaware of their key blind spot at their 6 o'clock. In the new

system, the target indicator will actually grow as it approaches the user's 6 o'clock. Additionally, even after the target is at an obtuse angle from the user, the compass will continue to give unique feedback to the user. While in the oculus prototype an indicator becomes locked to either side of the compass after it is at a >90° degree angle from the user, the unity prototype allows the indicator to continue to move along the compass as it scales up in size, though at a slower pace in order to ensure that the user can still see the indicator. This feedback should help make up for the compass bar's deficiencies in showing the user useful information of the full 360° around the user.



Tips

The Atheer HMD uses hand tracking to simulate a touch interface for the UI, but there are many problems with this. The space in which touch input is recognizable is very small, so it can be frustrating to interact with the Atheer. In addition, the accuracy of the recognition is lacking, so when forced to use touch input, large interaction elements are necessary. If possible, use the entire "screen" as the button.

When scripting functionality in Unity, it is often very useful to make variables public. public variables can be accessed from the Unity editor, and even altered while the program is running. This allows for rapid debugging, as well as for testing, by quickly changing the project quickly. For example, one could disable the arrow bouncing, and the compass, for a test and rebuild it, and then quickly re-enable them for a second test.

When Using the Atheer, the neutral color is black. That is, the closer a color is to black, the less visible it will be. This is the reason the background in all of these screenshots is black, as it makes the Atheer as transparent as possible. This was accomplished by going to the Main Camera, and in it's properties, setting the Skybox color to black. Then, open up the lighting settings in Window>Lighting, and setting the Skybox material to None. This removes the default skybox, and instead creates the flat black background used here.

Implementation

Though the design was relatively constant, moving from Unreal Engine to Unity required a complete rewrite of the logic behind the design elements. Unreal's visual Blueprint scripting made it relatively easy to move the logic into C#, which is used by Unity.

Scripts

- MoveTickMarker
- UI Manager
- Right Indicator Manager
- Tracking Enemy

MoveTickMarker

This script is placed on each Tick Marker on the compass bar, indicating the cardinal directions.

Start

1. Set defaultXPosition to be the current localPosition of the tick Marker. This will never be changed, so since each Tick Marker will have a different defaultXPosition, only this screen will be used
2. ZRotation is set to be the value of the y rotation in the Main Camera's eulerAngles. This is because in the Unreal engine, the Z-axis was the yaw, whereas in Unity, the Y-axis is the yaw.
3. The x value of localPosition of the Tick Marker is set to be the defaultXPosition - ZRotation * x, where x is a modifier to map the rotation properly on to the compass bar.

Update

1. IF the Main Camera's yaw is > 180, then the ZRotation is set to be the yaw - 360, otherwise ZRotation is set to be the yaw
2. The localPosition of the Tick Marker is set to be the defaultXPosition - ZRotation * x, where x is a modifier to map the rotation properly on to the compass bar.
3. IF the x value of the Tick Marker's localPosition is > 315
    A. The Tick Marker is disabled, making it invisible
    B. ELSE, the Tick Marker is enabled

UI Manager

This script is placed on the UI panel, and controls the spawning and visibility of the UI elements, as well as managing inputs.

Start

1. Get an array called tempTarg of all GameObjects with the tag "Target"
2. Create 2 arrays,one named TargArray and the other named CompassIndicators, each of the same length as the array from step 1
3. Create a 3rd array of all Sprites in the Arrows folder and subfolders within Arrows.
4. Set Arrow to be the GameObject with the "FrontIndicator" tag.
5. Iterate through the tempTarg array, doing the following each time:
    A. Get an array named target from the tempTarg array
    B. Create an Enemy Indicator that has it's "target" property set as target from step a
    C. Set the Enemy Indicator's parent as this.transform
    D. Set the Enemy Indicator's anchoredposition3D in RectTransform to be (0.0f,0.0f,2.5f), and it's localScale to be (1.0f,1.0f,1.0f) * 3.5f
    E. Set TargArray[i] to be the target from step a
    F. Set CompassIndicators[i] to be the EnemyIndicator
6. Setup an Array named Directions, with every element being a position you may want to quickly move the target to.
7. Set the mainTarget to be the front element in CompassIndicators
8. Set Arrow's mainTarget to be mainTarget, and it's dotProduct to be mainTarget's dotProduct

Update

1. IF the target is at the front index of the Directions array, make the arrow and compass indicator invisible. Otherwise, make sure that the arrow and compass indicator ARE visible

2. IF the space bar is just being pressed, or the screen has just been tapped
    A. Increment the index of the Array of Arrow Images, looping it around if it is greater than the length of the array.
    B. Set the sprite of the arrow to be the image at the current index of the Arrow Images array.
3. IF the R key has just been pressed, or the touch screen has detected a two finger tap
    A. Call the Main Camera's Reset function, setting the rotation of the camera back to 0°
4. IF the Q key has just been pressed, or the touch screen has detected a three finger tap
    A. Increment the index of Directions, wrapping it around if it becomes larger than the length of Directions
    B. Set the position of the Target to be the vector at the current index of Directions
5. Set the dot product of the Arrow to be the dot product of the mainTarget.
6. Find the Game Object tagged as "Network", and if it's UDPPosition is a value other than Vector3.Zero, then do the following
    A. Move the target to UDPPosition.
7. IF the boolean value of showCompasss differs from the current state of CompassIndicators[0].GetComponent<SpriteRenderer>().enabled
    A. Iterate through CompassIndicators and set the enabled state of each spriteRenderer to showCompass
    B. Iterate through every GameObject with the tag "Compass" and set the enabled state of it's image to showCompass
    C. Set the enabled state of the Text, SpriteRenderer, and MoveTickMarker Components to showCompass

---

Right Indicator Manager

This Script is placed onto the Arrow, and controls the location and rotation of the Arrow

Start

1. BaseZRot is set to be the z value of the gameObject's local euler angles.
2. Default position is set to be the objects anchoredPosition3D.

Update

1. Set angleToTarget to be the angle between the target and the MainCamera
2. IF the dotProduct is > 0, then flip the sign of the angle, so that it points the correct direction
3. IF rotationEnabled is true
    A. Rotate the Arrow in the Z axis to be baseZRot + angleToTarget
4. Set defaultPosition to be :
    A. X: -xModifier * sin(angleToTarget[converted to radians])
    B. Y: yModifier * cos(angleToTarget[converted to radians]) + yOffset
    C. Z: zModifier * cos(angleToTarget[converted to radians]) - 150
    D. The modifiers change the path the arrow will follow, , and the offsets change the center of the elliptical path
5. IF ArrowBounce is true
    A. Get a Vector from the anchor point of the Arrow (in this case 0,0,0, relative to the UI canvas) to the Arrow's center, and normalize it. This will represent the direction the arrow will bounce in

B. Get the vector to modify the defaultPosition with, by multiplying the Vector from step a. by 20 * sin(totalTime * 10)s
C. Set the arrow's anchoredPosition3D to be defaultPosition + the vector from step b.

Tracking Enemy

This script controls the Enemy Indicators that appear on the Compass, moving and scaling them as necessary

Start

1. Set the targetPosition to be a 2D vector, holding the x and y position, and discarding the Z (vertical) position
2. Set the playerPosition to be a 2D vector, holding the x and y position, and discarding the Z (vertical) position
3. Get the vector ToTarget, which is from the player to the target, and normalize it to get the direction vector to the target
4. Set dotProduct to be the dot product of ToTarget with the Right Vector of the player, which will represent how far to the left or right the target is from the player
5. Scale dotProduct up by a factor of 310 to fit it onto the compass Bar properly
6. Set the x position of the EnemyIndicator to be dotProduct
7. Set defaultScale to be the current scale of the indicator

Update

1. Set the targetPosition to be a 2D vector, holding the x and y position, and discarding the Z (vertical) position
2. Set the playerPosition to be a 2D vector, holding the x and y position, and discarding the Z (vertical) position
3. Get the vector ToTarget, which is from the player to the target, and normalize it to get the direction vector to the target
4. Set dotProduct to be the dot product of ToTarget with the Right Vector of the player, which will represent how far to the left or right the target is from the player
5. Scale dotProduct up by a factor of 310 to fit it onto the compass Bar properly
6. Set frontDot to be another dotProduct of toTarget and the Camera's Front Vector, to determine if the target is ahead of or behind the player
7. IF frontDot > 0 (i.e. if the target is ahead of the player)
    A. Set ahead to true
    B. Set the x Position of the Enemy Indicator to be dotProduct
    C. Set the scale of the Enemy Indicator to be defaultScale
8. Otherwise, if dotProduct > 0 (i.e. if the target is Behind the camera AND to the Right of the Camera)
    A. Set ahead to FALSE
    B. Set the xPosition of the Enemy Indicator to be the 310 + 40* |frontDot|
    C. Set the scale of the Enemy Indicator to be defaultScale * (1+|frontDot| *2/3)
9. Otherwise (i.e. if the target is behind the camera AND to the LEFT of the Camera)
    A. Set ahead to FALSE
    B. Set the xPosition of the Enemy Indicator to be the -310 - 40* |frontDot|
    C. Set the scale of the Enemy Indicator to be defaultScale * (1+|frontDot| *2/3)