Motivation

Viewing and organizing a set of images on a 2D screen (laptop, phone, etc.) can be arduous and time-consuming because of the limited space, especially with a large set of images. In a 2D space, viewing the entire collection typically requires viewing a small number of images simultaneously, or else each image will be very small. As a solution, viewing and organizing a set of images in 3D space can be more efficient than in a 2D environment. Currently, we have not seen any use of 3D space for image organization, so we decided to create an environment for this purpose. One of the core components of our project includes the use of gaze data. Gaze data can be used to intelligently change the size of images, which we believe can increase the speed at which users can process images.

Environment and Devices

We initially decided to create the project in Babylon, but ended up using Playcanvas instead. We primarily switch to Playcanvas because of the ability for live collaboration within the different editors. In addition, Playcanvas' had their own version control as well as their own testing environment, which made it easy to manage our project without the need to use github or npm. In terms of testing devices, we used the HTC Vive, Quest, and WebXR Emulator. We mainly used the HTC Vive as the main VR headset for our project while the Quest and WebXR were used to test different implementations throughout the project. It is important to note that the three different devices seemed to produce different bugs. For example, while we ran the project on the HTC Vive, only one minor bug would appear, but when using the Quest, the project produces bugs on several different actions. This issue mainly occurred after the addition of transitions in our project.

Setup

In our interface for organizing images, the images are structured in a cylindrical grid, and the user is spawned in the center of it. On the slide, you can see a bird's-eye view of the grid. We wanted to make the cylinder more of an oblong sphere so that images would be facing inwards toward the user, but we ended up only changing the angle of images based on their height, but not their XZ positions. We implemented the grid with a 2D array of javascript objects, where each object has an attribute hasFile, a boolean, and file, a PlayCanvas entity. Throughout the development of our grid, we ended up attaching more attributes to the PlayCanvas file entity, such as w and h for its original dimensions, an isStack boolean, and a stackFiles array. All of the grid parameters, such as the grid's radius, height, and number of files on the perimeter are variables and can be changed to accommodate any number of images.

Features

- Zoom
- Hover Image
- Select Image
- Reset Image
- Snapping onto another Image
- Creating and Adding to a Stack
- Remove from Stack
- Repositioning
- Dynamic Image Movement
- Player Movement

Zoom
Zoom is one of the most important interactions in our application. The images on the grid will change in size depending on how directly the user is looking at them. So if a user is looking directly at an image, it will be large enough to examine in detail, whereas looking between images gives a view of several images simultaneously.

We implemented this with raycasting. Until this point, we had only used the raycasting of the sample project, which detects intersections between the ray and a mesh's axis-aligned bounding box. This did not work for us, as we wanted to raycast onto the cylinder, which is quite poorly estimated by a box. We had to switch to using the physics engine and its raycasting methods onto object colliders. However, with this implementation, we found that it only intersects from the outside of the mesh inwards, so we had to reverse the raycast to get it to work. We then used the intersection point on the cylinder and calculated the distance between it and all of the file objects. We had ideas for more efficient methods, including only calculating for the files on screen or perhaps calculating the distance for only the closest object and then propagating increasing values throughout the rest of the grid, but our naive implementation worked well enough. Once we had the distance, we converted it to a zoom level. At first, the conversion was linear, which looked a bit unnatural, so we mapped the values from 0 to π and fed it through a sine function, which created smoother results.

Hover
When interacting with images, it is an important factor to keep track of which object that the user is focusing on. In order to visibly display the images that the users are “focused on” , images that the controllers point to will have a yellow highlighted border. It is also important to note that each controller can interact with images at the same time, so each controller will produce a highlight when it is pointing at an image.

In order to implement a highlight, we created an bordered image in GIMP. We then created a plane object, using the image as its material and set the plane as a child of the hovered image to create a border. As long as a raycast from a controller hits an image, the parent of the highlight object will be updated.

Select
Two of the most basic and common interactions of our project is selecting and moving an image. Users can select an image using the controllers’ triggers and on select, a visual and audio cue will be initiated to indicate the completion of the action. A green border was added to the image in the same way that we did with hover, and a click sound was added when the image is properly selected. After selection, an image can be moved around on the grid as it follows a raycast from the controller.

Snapping Back in Place

While having an image selected, users have several actions that could be made. One of them includes putting the image back in place whether it was intentional or not. If you release a selected image in any invalid location, it will simply snap the image back in place. A transition of the image and audio cue is added to aid in this particular interaction, so that users do not lose their image, thinking it has mysteriously disappeared.

To implement the reset of an image, a simple check is conducted to see whether the location of the image at the time of release is within the list of locations of the valid actions. If the location is not a valid location, the image begins to move back to its original position (with a playcanvas function called tween).

Snapping onto an Image

We see categorizing images into a folder, or "stack" as we call it in our application, as one of the main tasks of organizing a set of images. To make this common action as easy as possible, we made it so that when a user drags an image within a certain radius of the center of another image, the dragged image will "snap" onto the other one as an indication that the user can create a stack. An audio cue and positional translation reinforce the interaction. A stack is made if the user lets go, or the image unsnaps if the user drags it far enough away.

This too is done by raycasting. While a user is dragging a file around, a raycast returns the intersection point on the grid. The theta and height of the point relative to the center of the cylinder are used directly to get the index of the nearest file. Then the distance between the dragged file and the closest file is calculated, and if the distance is less than a certain amount, the dragged file's position is set to that of the closest file as a visual cue. A boolean check helps play the "tick" audio cue just once when snapping, and not once every frame that the two files are snapped together.

Creating and Adding to Stacks

In our application, stacks represent directories in a file system. Were this app to interface with an actual file system, these would be folders in which to store images for the user. In our implementation, the stacks show up to 9 thumbnails of the images stored inside it, and an additional counter indicates the number of files it contains. To initially create a stack, the user must release an image after it snaps to another image.

When a stack is made, the recipient file entity in the grid is replaced with a new stack entity, which is given the attribute isStack, which, among other things, tells the zoom function to keep its dimensions square as opposed to searching for its dimensions based on its original image file. The stack entity is also given a stackFiles array, initialized with the two stored files.

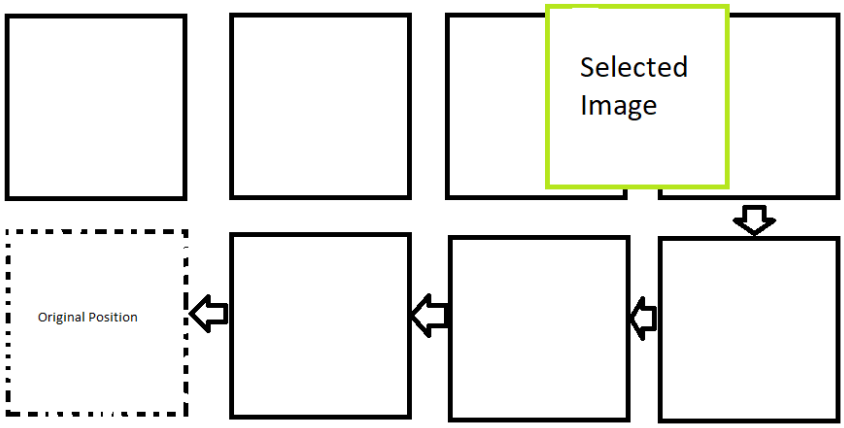
Remove from a Stack

Unstacking is the ability of the user to remove a file from a stack, either by repositioning it between two files or by dragging it directly into another stack. One decision we pondered is how to display a stack's files and how we should deal with the other files. As a solution, we decided to push the main grid into the background while viewing a stack, and this ended up being much

less jarring when we added animations. Also, it is important to note that when the second-to-last file is removed from a stack, the stack is removed, and the last image within the stack is placed into the grid.
Viewing a stack's contents involves populating a temporary grid object with the stack's contents and "scrolling" the grid to center the files. The scrolling is just two variables for offsets of the height and rotation of the grid. Removing from stacks was surprisingly tricky because by this point our code was suffering from a large number of states to keep track of, and we unfortunately had not vied for the well-structured state machine library.

Repositioning
One task that users may see helpful is moving images around on the grid. That being said, we have implemented a feature where users are able to select and move an image to a different spot by releasing the image between two other images. A hitbox will be highlighted green when you move a selected image between two images, and releasing an image within the green highlight will place the released image in its new location, shifting images around in the grid.

Each hitbox corresponds to an image position on the grid, and when re-positioning is triggered, the images will shift row-wise to the selected image's original position and then the images will shift column-wise of the new image position. A transition is applied to each image movement as it updates to its new position. An example is provided below.



Dynamic Movement of Images
It is important to note that during the process of creating/adding to stacks, images are removed from the grid. When an image is added to the grid, the images are coded to populate the empty spot in the grid (position of the image that was just added to a stack) by shifting the appropriate number of images row-wise and column-wise. This keeps images together and coherent without displaying random empty patches if multiple images are put into stacks.

Player Movement
The user can move along the XZ plane with the left controller pad and can rotate the camera with the right controller pad. Since the controller pads are also pressable buttons that are used in our interface, we made it so the directional input has to be above 40% before it moved the player, since it may be annoying to move accidentally when pressing the button.