Designing Apps for Intel® RealSense™ Technology

Introduction
When designing for emerging media technologies (such as gesture control), our goal as app developers is to make engaging experiences that are intuitive, familiar, and exciting to the user. When users first launch an application, the navigation design should be so intuitive that they start exploring the app’s features right away. In our most recent experiments with Intel® RealSense™ technology, we strived to make an application that a user could dive into while interspersing enough interesting gesture features to keep the user captivated. Instead of approaching Intel RealSense technology as replacing standard input, we focused on the strengths of natural gestures along with what the Intel RealSense Software Development Kit (SDK) offers in terms of unique features. Our first application, Space Between, focuses on hand and face tracking, while our second application branches out to include the SDK’s more unique features, including emotion detection and user segmentation. In our work, we learned several lessons we think developers might find useful, namely designing gestures for ease of use, matching gestures to the designed gameplay, developing interfaces that become familiar to the user, and creating menus that are easy to use and understand.

Designing Input for Intel® RealSense™ Technology

Figure 1: Space Between, developed to use the Intel® RealSense™ technology.
When designing our first Intel RealSense application, we began the design process thinking of the platform in mind. Instead of determining how we would port a gameplay style to gesture control, we thought of the unique interactions available through gesture control and what experiences we could develop around them. Since our development began with the Intel® Perceptual Computing SDK (the predecessor of Intel RealSense technology), we focused on 2D hand position and hand openness as the main user interactions, forming the core of our gameplay. With using only these two simple interactions, we wanted to give users a wide range of possible gameplay interactions. Most of the changes in gameplay interactions came from simply altering the orientation of the user’s hand, giving a different feel to the gestures—even though the measured values were the same.

The main application we developed with Intel RealSense technology is Space Between. Space Between is a game, developed in Unity®, in which the player explores the depths of the ocean by controlling different creatures [Fig. 1]. The game is broken down into multiple (sequential) minigames, each focusing on a different creature and input modality. Each gesture is used in a way that mimics the movement of the corresponding creature, driving the characters directly. These are often used one-to-one: the hand is oriented so that it aligns with the creature, with instant effect on the character’s movement, resulting in controls that were easy to understand and learn.

When designing these minigames, we knew we needed to start with gesture input in mind. From there, we iterated on each until they fit. After using Intel Perceptual Computing SDK’s hand tracking, face tracking, and voice recognition, we concluded that we were most excited with the hand tracking module. When transitioning to Intel RealSense SDK, we found that the strongest modules were the ones involving hand tracking, although the strength of the SDK lies in the sheer amount of modules available. The minigames all began with hand tracking as the primary control, and head tracking was used to alleviate problems with prolonged gestures (as discussed later).

Figure 2: Wave Motion in The Sunlight Zone stage - http://youtu.be/rMwMyy6O_o0?t=9s.
In our first minigame, the Sunlight Zone, the player controls a sea turtle in the profile view. The game’s design began with the idea of using a gesture that mimics holding your hand out a car window: fluidly moving your hand up and down in a wave movement [Fig. 2]. The turtle mimics the movement of the player’s hand, gaining speed with each wave completed. Originally, the only input was the y-position of the user’s hand in the viewport as a goal for the player’s character. After the prototyping stage, we were able to get a more accurate gesture using the hand angle. With this method, we could face the turtle to match the angler of the user’s hand, making the interaction feel more responsive. We were able to get the hand angle from the hand tracking module’s palm orientation by singling out an axis [Fig. 3].

```cpp
// handData Is an instance of PXCMHandData.IHand
PXCMPoint4Df32 handRotInte = handData.QueryPalmOrientation();
Quaternion handRotation = new Quaternion(handRotInte.x, handRotInte.y, handRotInte.z, handRotInte.w);

// Single out x-axis rotation (roll, as if rotating around the thumb)
float xRotation = handRotation.eulerAngles.x;
```

**Figure 3:** A code sample of singling out an axis from hand data.

This was an easy gesture to teach new players, but after play testing we noticed that the gesture proved tiring after less than a minute. From this we learned about “consumed endurance” [Fig. 4], which is the measure of the tiredness in arms while performing raised gestures. The problem with the gesture was that the elbow needed to be raised perpendicular to the body such that it could not support the rest of the arm, which turns out to be one of the most tiring gestures.

![Consumed Endurance Formula](http://blog.jhincapie.com/projects/consumed-endurance-ce-measuring-arm-fatigue-during-mid-air-interactions/)

**Consumed Endurance (CE)** = \( \frac{\text{Interaction Time}}{1236.5} \) \( \frac{1}{(F/F_{\text{max}} \times 100 - 15)^{0.618}} - 72.5 \)

**Figure 4:** Consumed Endurance Formula (source: Consumed Endurance (CE) – Measuring Arm Fatigue during Mid-Air Interactions from [http://blog.jhincapie.com/projects/consumed-endurance-ce-measuring-arm-fatigue-during-mid-air-interactions/]).

We still liked the wave movement for controlling the character, but in order to play comfortably (and for prolonged periods of time), users had to be able to rest their elbows. We added a cruising speed to the game, where the player’s character slows down and is controllable using hand angle exclusively. This allows the player to continue playing the game without feeling like they are penalized or required to perform the wave movement gesture for a long period of time.
Even with the addition of the hand angle to alleviate the tiring of the arm, players still needed time to recuperate before the next minigame, The Midnight Zone. To give them a breather, we added a minigame that didn’t use hand gestures. In The Twilight Zone, the player simply leans in any of the four directions to control the character, mimicking the movement of the player’s character (a whale). On the coding end, the tracking of these leaning movements comes from tracking the central head position, both in change of viewport x-position and depth.

**Designing Interfaces for Intel® RealSense™ Technology**

It didn’t take long for us to realize that designing gesture-based applications is not a straightforward process. For our demo version of Space Between we needed to include simple controls for minigame selection. The use case for these was as simple buttons: all we needed was a way to select an option and accept it. Our first gesture-based interface was a direct replacement for mouse control [Fig. 5]. Hand position was used for selection, while a push (and later, thumbs up) gesture was used for accepting with voice control for a backup. Although this was only a quick (and temporary) solution, we found that using the hand to make a menu selection using this method was difficult and tedious. Performing a gesture would often move the cursor location, requiring buttons to have large selection areas. Our iteration of this was splitting the viewport into thirds and only using the hand’s x-position for selection.

![Image](image.png)

*Figure 5: Our initial menu from the Space Between demo used for selecting minigames.*

Our next iteration was adding a left or right swipe gesture [Fig. 6] to switch between the games by rotating a radial menu. A push (or thumbs up) gesture would select the active menu. This proved better visually (by actually encouraging user interaction), along with reducing false positives and accidental selects. When designing interfaces for gesture control, we found it important to emphasize responsiveness through visual and audible feedback. This helped make up for the loss of tactile response.
When designing intuitive interfaces, we often pulled ideas from mobile interfaces over ones used for PC environments. Swiping and clicking are simple gestures that users are already familiar with, so we continued exploring ways to convert them to the gesture medium. One thing to note when using Intel RealSense SDK gestures is that swipes are a specific term where each hand moves in an explicit direction [Fig. 7]. But wave gestures (not to be confused with the wave movement we used in our first minigame) don’t have the direction defined. If you want to have either hand swipe in both directions, you need to follow the hand’s position and determine its velocity. The benefit of doing this is that as the user’s hand begins to be recognized as a swipe, the time and velocity of the swipe can be determined accurately. This allows you to add momentum to selections, similar to what users are accustomed to with mobile devices.

**Figure 7:** From Intel’s documentation, a swipe vs a wave gesture.
While these solutions work fine for navigating menus, we found that menus sometimes become unnecessary in our application altogether. When designing our game, we referenced *Journey* often. *Journey*, if you’re not familiar with it, is a gorgeous arthouse adventure game developed by thatgamecompany that focuses on minimalism to make the other elements of the game shine. The start screen consists of a desert backdrop and the words “Start: New Journey.” Menus are kept to an absolute minimum, and teaching the player the controls is done by using transparent controller animations [*Fig. 8*]. When designing the start screen for Space Between, we decided to skip the stage select entirely and instead focus on a play experience as the user’s first interaction. When the user’s hand becomes recognized, movements begin to swirl the air in front of them, forming wind gusts. As the user plays with the simple scene in front of them, the gusts of wind rock the boat, beginning the game experience. Instead of forcing a player to select a specific stage, each of the minigames is played in succession.

*Figure 8: A screen capture from the game Journey showing minimalist use of UI for instructions.*

When designing menus (or gameplay) that require gestures, adding graphical representations is important. This seems obvious, but it allows the user to quickly interact without having to learn every option. This is especially important when it’s not always possible to use an intuitive gesture for selecting menu options. When teaching the player how to perform our gameplay gestures, we kept the graphical representations as simple animated spritesheets [*Fig. 9*]. From these, the player is able to determine the orientation of the hand (or head), which hand to use (or for some, either), and the movement required to perform them. Since our game begins with no repercussions, having the player learn which actions the gestures operate did not pose an issue. We focused on an explorative approach for the game, which was emphasized through the progressively perilous stages. As the player learns gestures in earlier minigames, we use the same icons in later ones to keep the interactions familiar.
As users are not familiar with most of the interactions, feedback for actions is important. Gesture recognition is not perfect, so when input isn’t recognized the user needs to know. With our demo version of Space Between, these were obvious to the user, displayed at the top of the screen at all times [Fig. 10]. As hands, head, or some gestures were recognized, the appropriate icon would fade in. For our full application version, we decided to go with a more built-in approach. As a user’s input is no longer received, the creatures return to a default state. As an example, in the Sunlight Zone when the user’s hand isn’t recognized, the player-controlled sea turtle rotates back to swimming straight and switches its animation state. As another effect, all of our characters were designed so that when under player control they would glow a specific color. For the games that use cursors, we were able to fade the cursors in and out, along with having matching auditory cues, when input is received or lost.

Figure 9: An animated sprite sheet instruction for performing a wave movement in Space Between.

Figure 10: Visual feedback for detected hands and head in the demo version of Space Between.
When integrating complex menus, we found that integrating gestures as the primary control was not always necessary. If the use of the application allows it, using a mouse and keyboard for the most tedious elements—sliders and data entry—will be far less frustrating for the user. While gestures work well for state toggles and buttons, using position data that involves multiple axes can be difficult for the user to control. This can be remedied by using data entry that uses only one axis of movement once a grasp (openness or finger pinch) gesture is performed, but this doesn’t solve the root issue. Although gesture technology is getting much better, most users haven’t used it yet. If having standard input modalities isn’t a possibility for the primary input, the best solution is to make the menus large. Having a standard input modality as a fallback solution is not a bad option.

When deciding on gestures to control menus that can’t always be shown, selecting the gesture is extremely important. As mentioned earlier, though, many of these actions don’t yet have associated gestures or motions in the user’s knowledge base. As a case study, one of the most noticeable examples is a pause (or options) menu. Displaying a pause menu is important in most games and should be one of the quickest gestures for both the user to perform and the application to recognize. This presents multiple problems when it comes to design, though. Gestures from other known mediums (mouse and keyboard applications, tablets, mobile devices) don’t have anything in common. Keyboard games use the escape key, while smartphones tend to use swiping from the left edge of the screen (but even this isn’t a given). Usually this action has something to do with the top left corner, but even so many users often associate it with the close button from a desktop application, reaching to the top right corner. Using specific corners of the screen or swipe gestures doesn’t translate well, due to dropped tracking and accidental use, respectively. When it comes to Intel RealSense applications, Intel recommends using the “v sign” [Fig. 11] to bring up a main menu. The reasoning for this gesture seems to be ease of recognition and low probability to accidentally perform it. While this gesture isn’t intuitive or familiar to users, the answer may just be to rely on time to build the connection. In addition to implementing this gesture for the pause menu, we added multiple redundant systems. Losing tracking (the user’s hands are out of camera range) for a specified amount of time will pull up the menu—along with the familiar methods of the mouse and keyboard.

![Figure 11: The v sign gesture from Intel’s RealSense documentation, suggested for bringing up menus.](image)
Multiple Modules with Intel® RealSense™ Technology

When implementing multiple modules from the Intel RealSense SDK, you should consider more than ease of use and familiarity—performance is also important. When dealing with multiple modules, pausing and waiting to initialize modules is important. For *Space Between*, we switch active modules during scene changes so that the user doesn’t notice the hanging framerate or loss of tracking. Prior to the scene loading, we check if there are differences in required modules and, if so, run initialization. Switching active modules with Intel RealSense SDK is simple, completed by initializing new modules followed by calling the Intel RealSense SDK’s SenseManager function. Pausing modules in our application is used when we’re done using them (such as facial recognition) or when the user doesn’t have control over the application (such as turning off face tracking when a menu is displayed).

When dealing with the SDK modules—especially those that involve camera feeds—there’s a tradeoff between higher framerate and smoother data. When using AcquireFrame to gather new data, turning off waiting for all modules and adjusting the maximum wait time work to generally reduce stutter and increase frame rate, with the cost of losing some data if the wait time is dropped too low. Slower computers need to be given more time to process the frame’s data, while faster computers need less time. In Unity, this can be simplified so that faster gameplay settings (lower graphics) result in a longer allotted time to process data, with the opposite being true for the higher graphics settings. You can do this using Unity’s built-in QualitySettings [Fig. 12].

```csharp
// Our setup uses levels 0 (laptop), 1 (normal), 2 (high). These are selected by the user when running the application
int qualityLevel = QualitySettings.GetQualityLevel();
int acquireFrameWaitTime = qualityLevel == 0 ? 85 : (qualityLevel == 1 ? 60 : 50);

// SenseManager is an instance of PXOSENSEManager. Wait time is in milliseconds
ModuleStatus = SenseManager.AcquireFrame(False, acquireFrameWaitTime);
if (ModuleStatus == pxcmStatus.PXCM_STATUS_NO_ERROR) {
    // Do work here
}
```

**Figure 12:** A code sample showing RealSense running on the Unity thread with its wait time based on quality settings

Conclusion

Gesture technology is still new, so designing gesture-based applications will require more iteration than normal, though using a well-designed gesture application is ultimately worth it. Remember to keep the user’s knowledge in mind, borrowing from applications or mediums that the user is familiar with. Design applications that use minimal menus. And above all else, don’t be afraid to try something new—even if you have to change it later.
Future Enhancements/Developments for Space Between

Moving forward with Space Between, there is a lot that we have learned from the development of the demo and full version application that we will use to continue improving it. While there was much work done on making the gameplay as intuitive and easy to learn as possible, there are still things that can be done to improve it further. For example, in the demo we had visual feedback in the user interface for when a user’s hands and head are detected. In moving toward an even more minimalist user interface design, this was scrapped but we never got to include its replacement: visual feedback integrated into the characters and environment itself. Our idea was that instead of having a graphical user interface fixed to the top of the screen and showing at all times, we would light part of characters up to show that the user now had control of them. This solves the problem of informing the user that the system has recognized their input while keeping the game clean and focused on the environment.

Aside from Intel RealSense related features, there are other features that didn’t quite make the cut in the current version of Space Between. When designing the full version of the game, we did a lot of research on ocean life, especially life at great depths. One of the things that truly captivated us was the world of bioluminescence and the dependence creatures in the ocean have on it. We really wanted to bring this into our game because we felt it was necessary in telling the story of the ocean, but also because it was just so cool. In the current version of the game, you can see some of our attempts at integrating it into the environment – the points you collect are loose representations of it, sea anemones release it in The Midnight Zone, creatures release it on death in The Trenches. However these fall short of the full vision we had for bioluminescence in our game and don’t do justice to its beauty found in nature.

About the Author

Ryan Clark is one of the founders of Chronosapien Interactive, an Orlando based company. Chronosapien Interactive is a software development company that focuses on interactive media, specializing in emerging technologies. It is currently developing a demo for The Risen, its second application using Intel RealSense technology. You can follow them at chronosapien.reddit.com, or contact them at theoracle@chronosapien.com.