Ambient Edge Kinect Control Service

Cristina McLaughlin
EE 699: Dr. Darren Carlson
12/10/20

I. Introduction and Motivation

Smart spaces are physical environments enhanced by computation and networking technology to create a complex ecosystem that users can interact with. Devices within a space can range from small distance or noise sensors to objects that take voice commands to smart lighting solutions using projectors. The devices are coordinated to provide context-aware network services for the user. Smart spaces could improve personal and professional productivity; much like how smart home devices are gaining traction in helping home monitoring, security, entertainment, and in handling regular household tasks. Predictions suggest that by 2023 there will be 70.6 million smart homes [1]. Unfortunately, unlike smart homes and smart home devices, smart spaces are not one size fits all. They will be highly variant due to location, application, and user ability. For instance, a smart space used for entertainment will be designed differently from a space used by a workplace or a school. The main goal of designing a space is to meet the needs of its users and understand features necessary for each location.

Hospitals and healthcare settings have become the focus of our project. Implementing smart spaces in these areas could be highly beneficial to patients. On June 15, 2020, the FDA approved marketing for the first game-based therapeutic [2]. This is just the beginning of relating video game and smart space technology with active and preventive care.

Ambient Edge Overview

Ambient Edge is a networking framework designed to enable fast creation and expansion of smart space through real-time low-latency messaging. It addresses the problem of high variability between each smart space by employing plug-and-play architecture, allowing designers to pick and choose the perfect devices for their ecosystem. An Ambient Edge Smart Space is comprised of Edge Broker, Services, and Clients, all of which run on hosts throughout the space. Put simply—an Edge Service is software that performs a task while managing underlying hardware of a sensor or device while an Edge Client connects and consumes one or more Edge Services. The focus of this report was developing and testing an Edge Kinect Service and creating a Unity Integrated consumer to make use of the data.

II. Project Objectives

The following were the objectives for this project. The first task was to design a Kinect Service that was strongly based off the Azure Kinect Developer Kit API. The Kinect is a camera with sophisticated computer vision and AI sensors [3]. It contains a 1-MP depth sensor, 7-microphones for speech and sound capture, a 12-MP RGB video camera, an accelerometer, and a gyroscope. Since there are multiple data sources the Kinect can collect, this project focused on accessing IMU and body-tracking. The goal for the Kinect client was to have an Edge-Integrated Unity scene run while collecting and displaying the data from the service. The following User Story describes some of the smaller milestones for this project.

User Story

A user story is a description of an Edge Service from the perspective of a client. Suppose we have a Kinect within the smart space. How does it interact with users and Edge Clients? After some thought we decided it should have the following features:

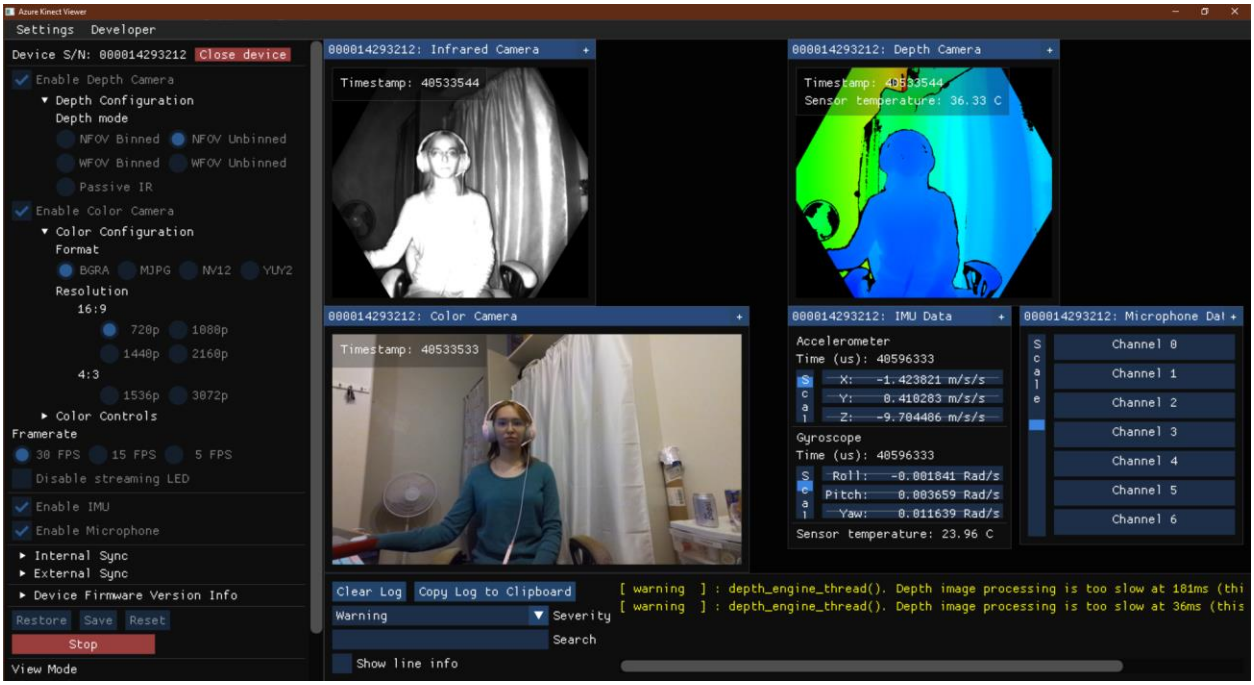
- The Kinect will collect and send out IMU and body tracking data

- The Kinect can only be configured by a smart space manager, not normal users since it is a room-wide service
- Multiple clients can make IMU and body tracking requests at the same time
- The camera will turn off when there are no current requests

III. Azure Kinect SDK Background Information

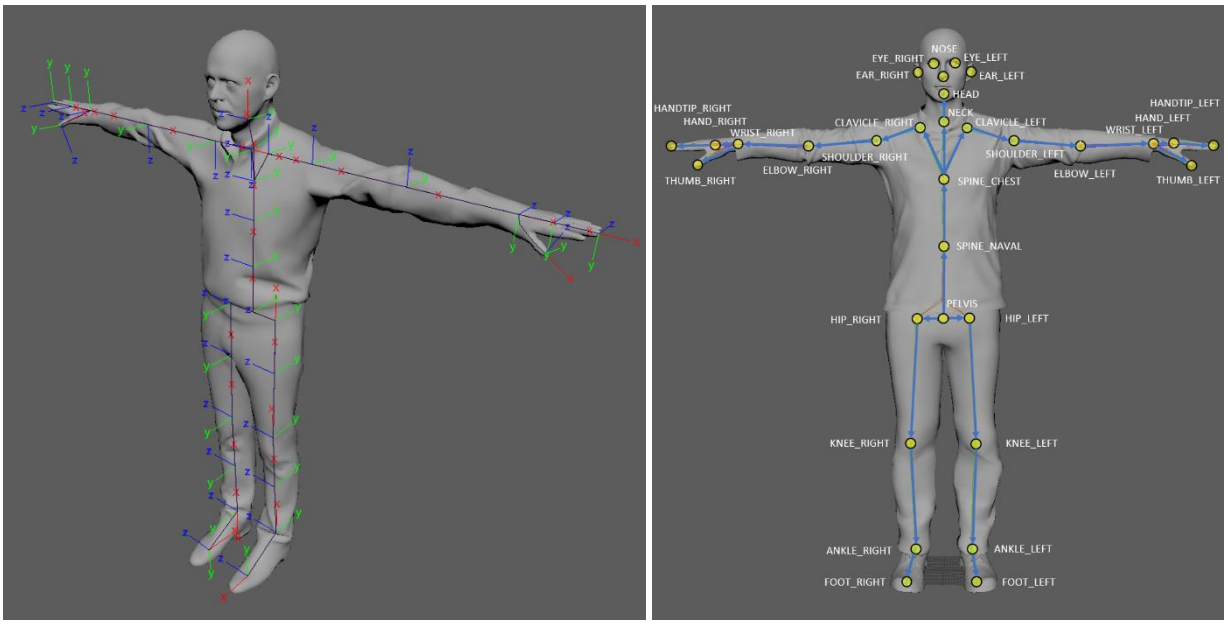
The Azure Kinect Sensor SDK API can be found here and the Body Tracking API can be found here. The SDK is primarily C API; however, the documentation also covers the C++ wrapper. This was the first problem encountered when beginning the Kinect service because Edge uses C# and .NET. However, there is a .NET NuGet wrapper available, which was used to complete the service [4]. Another .NET package was also found for the Body Tracking SDK [5]. There is no official documentation for the C# wrapper on the API page, so the C++ API was used as a reference instead to build the service functionality and .proto file.

The Azure Sensor SDK also has a Kinect Camera Viewer that can be downloaded to see the different functionalities shown in the figure below. This includes an infrared camera, depth camera, color camera, and IMU data. The left-hand panel also shows the different camera configurations possible. As stated above, these configurations are set to a default in the service and can later be changed by some service manager.



The Azure Body Tracking SDK documents the three key components of a body frame. Each frame contains a collection of body structs, a 2D body index map, and the input capture that generated the results. The Edge Kinect service focusses on collecting data from the body structs. Each struct contains Body ID, 32 joint positions, and joint orientations. Joint position and orientation are estimates relative to the global depth sensor frame for reference. The position is specified in millimeters and the orientation is expressed as a normalized quaternion. The estimates also have a confidence level of none, low, medium, and high; these levels indicate whether a joint is out of range, predicted, or within frame. The figures

below show the join axis orientation and joint hierarchy. Joint coordinates are also in axis orientations because it is widely used with commercial avatars, game engines, and rendering software. The joint data points were easily adapted to a .proto message. The 2D body index map is a map that segments each body instance from the background. The 2D array of pixels was more difficult to convert to a .proto message so it was left for future work.



IV. Development Process and Final Design

Software Design—Service Overview

This section will be discussing the development process, important function handlers, and problems encountered along the way. The first task was determining the Service's API based on the User Story discussed above. The Kinect Service API is shown below:

Verb	Path	Behavior
GET	.imu	Returns the current IMU data as a structured object including timestamp, XYZ gyroscope, XYZ accelerometer, and temperature
OBSERVE	imu.observers.{sub-id}	Allows an observer to connect and get a continuous stream of IMU data
GET	.skeleton	Returns the current skeleton data as a structured object with body ID, XYZ information on 32 different joints, and a confidence level per joint
OBSERVE	skeleton.observers.{sub-id}	Allows an observer to connect and get a continuous stream of bodytracking data
PUT	.config	Not fully implemented. Can be used to configure the camera when a manager uses the private configuration path

The Kinect Service uses the Google Protobuf serialization format to structure the data processed by the service. The IMUEvent message has four different fields shown below. This includes the timestamp of when the data was collected, gyroscopic data, accelerometer data, and the ambient temperature. The SkeletonEvent message has two fields: BodyID and Joint. The Joint data is composed of another message called JointID, which contains all 32 Joints specified in the Azure Kinect API. Each

Joint is described by a helper XYZ message that contains the X, Y, and Z coordinates of the Joint in space. The helper function also has Confidence Level which is an enum ranging from Not Applicable, None, Low, Medium, and High. This proto message was designed to match the confidence enum from the Kinect API discussed above.

```
// Message definition for IMU Events
message IMUEvent{
  string timestamp = 1;
  XYZ gryoscope_raw = 2;
  XYZ accelerometer_raw = 3;
  float temperature = 4;
}

//Message definition for Skeleton Events
message SkeletonEvent{
  int32 body_id = 1;
  JointID joint = 2;
}

message JointID{
  XYZ pelvis = 1;
        ...
  XYZ ear_right = 32;
}

// Helper for XYZ values
message XYZ {
  float x = 1;
  float y = 2;
  float z = 3;
  level Confidence = 4;

  enum level{
  option allow_alias = true;
    NOT_APPLICABLE = 0;
    NONE = 0;
    LOW = 1;
    MEDIUM = 2;
    HIGH = 3;
 }
}
```

```
private const string IMUDataType = "ambientedge.protocol.ext.kinect.imu.data:1.0.0";
private const string SkeletonDataType = "ambientedge.protocol.ext.kinect.skeleton.data:1.0.0";
```

Software Design—Kinect Service Code

The Command Line Setup region sets up the command line prompts and options. It follows the structure found in the Edge OSC Service example. It specifies getters and setters for clustername, servicename, and edgeurl which are variables used in creating the Edge connection.

The Main() region has remained unchanged from other services. It still configures logging, parses the command line arguments, and initializes the program. The updated services also utilize a running while-loop to keep the program thread alive if there are asynchronous threading issues.

In the Edge Vars and Kinect Vars region different private variables are instantiated. The reasonable default configuration for the Kinect is also set here.

Within the Init() function default IMU and skeleton messages are created. This was required in order to avoid the error "System.NullReferenceException: 'Object reference not set to an instance of an object.'" which occurs if the objects are not initialized. The service will try to fill the object message, but if it is not instantiated prior, it will throw an error. Since the XYZ data for each joint is also a .proto message, each joint has to be set to a new XYZ() object as well. For example, skeletonevent.Joint.Pelvis = new XYZ(); must be instantiated before populating the pelvis data. The was a major problem that I ran into while debugging because I did not realize each XYZ() message also had to be created. Setting up the instantiations was also a lot of repetition. In the future I would like to instantiate everything with a loop, but I am not sure how it would be done with each object.

After the Edge connection is opened, StartCamera() is called. StartCamera() is a function in the camera control region that opens the Kinect device, configures it, and then starts the IMU capture and body tracking capture. The StopCamera() function stops the Kinect camera and IMU collection and disposes of the camera properly so that the device is not overloaded.

In the Register Routes region, the different service routes discussed above are created. Since there are also two observation routes for IMU and skeleton, two separate resource managers are also set up. These managers deal with creating and removing observers for the data streams.

The final regions of the program are the IMU Resource Handlers and the Skeleton Resource Handlers. When calling GET for IMU data, the IMUCapture() function runs. This function calls the Kinect device to get an IMU sample, and then the sample is formatted to fill the protobuf IMUEvent message. This handler is very straightforward. The GET handler for body tracking was more difficult to create. The GetHandler() function calls the SkeletonCapture() function which configures the payload and returns the Edge response. Within SkeletonCapture(), PresentSkeletonTracking() is also called. The beginning of the function flushes the Edge SkeletonEvent to default in case it was previously populated with data. Then it gets a capture from the Kinect device and enqueues the capture in the body tracker function. Finally, within a large for-loop, the function gets all of the bodies within the capture and assigns a Body ID along with populating the Kinect joint data into the protobuf message format. This required using a large switch statement to check which joint was being read from the Kinect data and then properly assigning it to the correct protobuf joint. As discussed previously I am not sure if this could be done without as much repetition, but I would like to clean it up more if possible.

Software Design—Kinect Client Code

The development for the Kinect Client had multiple iterations leading up to a Unity Integrated Client. The first iteration was a simple console client that was used to test the service usage. The client tests GET and ASYNC GET requests for the IMU and skeleton data. It also sets up an observer for each resource for a certain amount of time, then closes the connection. In this client all of the joint data is printed to the console.

In the second iteration of the console Kinect Client, I wanted to test using the different body tracking data. This program sets up an observer to stream the body tracking data. Using the data, it will print to console if the user's right hand, left hand, or both hands are raised. This was a simple way to test if the body coordinates were working together correctly. To see if a hand was raised the program checks

if the Y coordinate of the hand joints are greater than the Y coordinate of the head. The test was simple but successful, and this type of gesture checking could be used for future Unity games.

The next step was to create a Unity client that utilized the Kinect data in a substantial way. Unity Integration was a brand-new concept, and it had a steep learning curve. We also discussed using a Unity Client very late in the semester, so understandably the package was also released late. Due to these factors, I did not have time to create the polished game that I originally envisioned.

The first Unity Client iteration involved modifying the example from Dr. Carlson to establish an Edge connection to my Kinect service, instead of the SenseHAT service. I changed the spinning cube to a sphere that would display the IMU and a cylinder to display the skeleton data. Originally the program was not updating the cylinder with data properly, however it turned out to be an issue of trying to access the IMU and skeleton data synchronously. This was not an issue in previous clients, so it will have to be furthered explored in the Unity setting.

The last Unity Client that was completed for this project was a scene containing a ball that moves with the players right hand movement. A skeleton observer is instantiated at the beginning of the program and runs for 30 seconds. Within the dispatcher enqueue there is logic to update a Vector3 instance on the sphere object with XYZ joint data from the right hand. Within the Sphere object there is the Vector3 instance and on Update() the instance's x and y coordinates are used to transform the sphere position. The coordinates are also transformed so that movements on screen mirror the player's movements in front of the camera. This client was time consuming because it was difficult to determine how to transform the sphere based off the Kinect coordinates. It required a lot of trial and error to get the motion fluid. I had to divide the Kinect coordinates by a factor of 100 so that the sphere would not immediately fly out of the scene. There is also a lot of inertia and acceleration that I was unsure how to fix. It was a good demonstration of feasibility though. This simple sphere could be expanded into a game where the user has to collect tokens by moving across the screen.

IV. Conclusion and Future Work

To conclude, the development of this Kinect Service encountered several setbacks due to the semester being online. This project will be developed further to have a full-fledged Unity game running based on the Edge service. The service code will also be cleaned up to remove the unnecessary repetition of instantiating the objects. The proto file will also be developed to include more parts of the Kinect API. Lastly, the final goal will be to implement image streaming over the Edge network so we can make use of the Kinect image captures.

References

[1] "Smart Home - United States | Statista Market Forecast", Statista, 2020. [Online]. Available: https://www.statista.com/outlook/279/109/smart-home/united-states.

[2] Office of the Commissioner, "FDA Permits Marketing of First Game-Based Digital Therapeutic to Improve Attention Function in Children with ADHD," 15-Jun-2020. [Online]. Available: https://www.fda.gov/news-events/press-announcements/fda-permits-marketing-first-game-based-digital-therapeutic-improve-attention-function-children-adhd. [Accessed: 18-Dec-2020].

[3] Microsoft, “Azure Kinect DK – Develop AI Models: Microsoft Azure,” – Develop AI Models | Microsoft Azure. [Online]. Available: https://azure.microsoft.com/en-us/services/kinect-dk/. [Accessed: 18-Dec-2020].

[4] Microsoft and Azure Kinect, “Microsoft.Azure.Kinect.Sensor 1.4.1,” NuGet Gallery | Microsoft.Azure.Kinect.Sensor 1.4.1. [Online]. Available: https://www.nuget.org/packages/Microsoft.Azure.Kinect.Sensor/. [Accessed: 18-Dec-2020].

[5] Microsoft and Azure Kinect, “Microsoft.Azure.Kinect.BodyTracking 1.0.1,” NuGet Gallery | Microsoft.Azure.Kinect.BodyTracking 1.0.1. [Online]. Available: https://www.nuget.org/packages/Microsoft.Azure.Kinect.BodyTracking/. [Accessed: 18-Dec-2020].

Appendix A. Detailed Procedure for Deployment and Running

Setup the Hardware

1. Setup the Kinect hardware by plugging the camera into a power source and the USB connector into the Edge service computer

Setup the .NET Application on Desktop (Windows)

1. Download the Service and Client projects on the desktop
2. Run `dotnet restore` on the Service program
3. Make sure all Unity packages are installed for Client

Final Service and Client Connection

1. Start up a STAN server session on desktop
2. Run the service on desktop using `dotnet run`
3. Run the client on Unity by selecting the scene and pressing play
4. Test the Kinect behavior by moving around in front of the camera