3D Immersive Environment
For Stroke Therapy

Conor Kaminer, Kevin LeBras, Jordan McCall, Tan Phan
Table of Contents

1. Introduction
   1.1 Abstract
   1.2 Research
   1.3 Motivation

2. Design
   2.1 Game
      2.1.1 Unity Basics
      2.1.2 Device Interfaces
      2.1.2.1 Kinect Interface
      2.1.2.2 Oculus Rift Interface
      2.1.2.3 Glove Interface
      2.1.3 Scenes and GameObjects’ Descriptions
      2.1.3.1 Carl GameObject Hierarchy
      2.1.3.2 Zigfu GameObject
      2.1.3.3 ShoulderROM Scene
      2.1.3.4 ReachBack Scene
      2.1.3.5 Login Scene
      2.1.3.6 SceneSwitch Scene
      2.1.3.7 CalibrateGlove Scene

2.2 Database
   2.2.1 Database Approach
   2.2.2 Database Schema Design
   2.2.3 Unity Interface Scripts
   2.2.4 Visual Graphing Interface

2.3 Glove Design
   2.3.1 Bend Sensors
   2.3.2 Accelerometer Circuitry and Communication
   2.3.3 Bluetooth
   2.3.4 PCB Design
   2.3.5 Glove Interface Program

3. Conclusions

4. References
1. Introduction:

1.1 Abstract

For our senior design project we created a system for gamifying stroke survivor therapy that provides a fully immersive 3D environment to the user in order to stimulate patient’s recovery. To do this we used a natural interaction/computer vision system, a virtual reality headset, and a glove controller. The natural interaction camera(s) act as a controller for the player’s in game body as well as a way to record data crucial to stroke therapy assessment. The virtual reality headset provides an eye into our virtual world. Lastly the glove will operate as a controller for the player’s in game hand in order to animate the hand and provide gesture inputs to the game, also data from the glove is recorded similarly to the computer vision system for stroke therapy assessment.

1.2 Research

There are a few recently developed prototypes similar to our system. Microsoft’s Kinect is a natural interaction product that has been shown to provide “accurate and robust” measurements for this type of clinical system (Kurillio, 2013, p. 6). Microsoft’s own research team is in the process of building their own system that uses the Kinect for Windows to aid in this process (Chang, 2014). Their prototype model tracks user gestures and adjusts the difficulty of rehab sessions based on their progress (Chang, 2014). They also have future plans of stroke patient interaction through social networking and also large scale data analysis of rehabilitation data trends (Chang, 2014). One reason behind the increased interest is the prospect that therapy may be cost effectively administered in patient’s own homes as an alternative to what is accepted by most professionals as the most effective treatment, Constraint-Induced Movement Therapy (CI). Ohio State University’s Wexner Medical Center research team have built a model for exactly this purpose. Their prototype comes in the form of a boat rowing game that utilizes a Kinect and two gloves; one to weigh down a patients nonaffected side, and the other to track movement by their affected side (Maggs, 2013).

1.3 System Overview

At the highest level our system can be broken up into two pieces; the glove hardware and the game software. Kevin LeBras and Andy Phan formed the sub-team that was in charge of designing the glove. Conor Kaminer and Jordan McCall formed the sub-team in charge of the software. The glove uses bend sensors to capture the motion of the glove wearer’s fingers and sends a corresponding linearized value for each bend sensor to the computer using Bluetooth. The software takes this data along with Kinect and Oculus Rift input and uses these devices to power a game in the Unity3D engine that has the player do exercises and games that are inspired by stroke therapy techniques. Meanwhile the software records data regarding the users movements from the Kinect and the glove into a database so that it can be read and graphed using a utility developed in Python that makes it easy to see and manipulate the data.
2. Design:

2.1 Game

2.1.1 Unity Basics

For the purposes of this report it is important to know a few things about how the Unity engine works. The basic building block of Unity is the GameObject. The GameObject by itself only contains one innate property that is it’s transform. This includes it’s position, rotation, and scale. In order to give a GameObject more properties like shape, physics, custom scripting, etc. you attach components to the object. There are many components already built into Unity like cameras, renderers, and light sources as well as components that can be imported like animations or models. The most important component for our purposes are scripts. Scripts, when attached to a GameObject, have access to that objects other components including it’s transform and functions or data from other scripts. This allows for scripts that access certain components to work on any GameObject with those components. Scripts have multiple innate functions but there are two that every script has, Start and Update. Start is called when the level is first loaded and Update is called every frame. Lastly a GameObject can have children and/or a parent. A GameObject can have any number of children but only one parent. The transform of a child has a position, rotation, and scale relative to it’s parent so any change in a GameObject’s transform is also applied to all of it’s children.
2.1.2 - Device Interfaces

In order to communicate with the external devices, Unity needs interfaces. All the devices (glove, Kinect, Oculus Rift) have their own interface.

2.1.2.1 - Kinect Interface

Our selected motion capture device tracks the movement of the user’s body. This allows users and stroke therapists to assess how well stroke patients can move their outer extremities. We are accomplishing this task using Microsoft’s Kinect for Windows. There are a few other 3D motion capture devices on the market, but the Kinect was the best fit for our project. It is currently the motion capture device with the most support for our purposes. The Kinect is also powered through Primesense 3D technology which is the leader in the motion capture industry.

Primesense, recently acquired by Apple, has also released their own motion capture device, but this sensor camera was out of the scope of our budget. In order to integrate the Kinect into our Unity project we are using a middleware created by the San Francisco startup, Zigfu. The Zigfu Development Kit (ZDK) for Unity3D is a wrapper that provides numerous tools for establishing communication between Microsoft’s Kinect for Windows and the Unity3D game engine. The ZDK works with both Mac and PC, and also OpenNI/NITE and Microsoft Kinect SDK. This gives us much needed flexibility as we develop our project. Specifically, the ZDK for Unity3D comes in the form of a downloadable package of all assets necessary for Kinect integration.

The Zigfu ZDK engine exists in our project in the form of a file ZigInput.dll located in the Assets folder of our project. There’s no way for us to expand and tinker with the library since it’s property of the company Zigfu. For this reason it is currently not possible to know the exact details on how the engine works.

Development can begin after importing the package to a Unity project and plugging in the Kinect. There is no formal documentation for the ZDK but there are a few sample scenes provided by Zigfu. These scenes are extremely useful so starting development from scratch can be avoided. We are using Zigfu’s basic method of body motion tracking. This method first involves creating a human body object to represent the user in the game. One of the body object hierarchies provided by Zigfu is shown in section 2.1.3. His name is Carl. The purpose of the hierarchy is to ensure that joints move together properly. For example when the Kinect detects a movement of the LeftShoulder object, all of its children will move accordingly. The Zigfu C# script “ZigSkeleton.cs” performs the adjustment of Carl’s components once every frame by sampling joint position data from the Kinect.

2.1.2.2 - Oculus Rift Interface

The Unity interface for the Oculus Rift is provided by the Oculus company in the Rift SDK. It provides all the scripts necessary to make a properly functioning in-game camera that appears correctly in the Oculus headset. It even provides a pre-made Unity object hierarchy (shown below) that is setup to be a plug and play camera.
The interface for controlling this camera is provided by a script called OVRDevice.cs. It includes wrapper functions for all exported C++ functions, as well as helper functions that use the stored Oculus variables to help set up camera behavior. OVRDevice.cs is attached to the OVRCameraController GameObject shown in the hierarchy above. This script only needs to be instantiated once in the whole scene. The next script is OVRCameraController.cs. This component is also attached to the OVRCameraController GameObject and allows for easy handling of the lower level cameras. All camera control from Unity is done through this component. The two camera’s, CameraLeft and CameraRight, have identical components to each other. The first component is the Unity camera. The camera renders an image based on the direction it is facing. These objects also have a couple scripts. The OVRCamera.cs script handles reading the Rift tracker and positioning the camera position and rotation. It also is responsible for properly rendering the final output using data from the last component required, the OVRLensCorrection.cs script. OVRLensCorrection.cs contains the variables required to set material properties for the lens correction image effect. This is a screen warp effect that makes the camera view not rectangular so that when the image passes through the lens inside the Oculus Rift it appears correctly as if the image is projected from real life sources.

2.1.2.3 - Glove Interface

Raw data from the glove interface program described later in this paper is stored in a text file. In order to use the data from the text file to animate our hand we create an interface class called GloveReader. This class is contained in a C# script (GloveReader.cs) but it is not a Unity Engine component. It is a C# object that can be instantiated inside any Unity Engine component script. The interface for accessing the glove data is very simple. The initialize function creates a new GloveReader object and initializes all the gesture booleans (namely the left hand grab boolean and the right hand grab boolean), the calibrated finger data and the database access class. GloveReader.getValues returns an array of all the raw data from the glove sensors. There is a list of accessor functions (these are functions not variables) for each data point in the array that describes what that member of the array is. These functions return the index of the described data point.

GloveReader.RH_IndexFinger    -- Bend Sensors
GloveReader.RH_MiddleFinger
GloveReader.RH_RingFinger
GloveReader.RH_PinkyFinger
GloveReader.RH_AccX           -- Accelerometer data
GloveReader.RH_AccY
GloveReader.RH_AccZ
GloveReader.LH_IndexFinger    -- Bend Sensors
GloveReader.LH_MiddleFinger  
GloveReader.LH_RingFinger  
GloveReader.LH_PinkyFinger  
GloveReader.LH_AccX  
GloveReader.LH_AccY  
GloveReader.LH_AccZ  
GloveReader.LH_GripIndex  
GloveReader.LH_GripMiddle

In order to check if the user is making a gesture you must first call the 
GloveReader.UpdateGestures function that updates the gesture boolean variables. Then check 
the gesture boolean variables GloveReader.RightIsGrab or GloveReader.LeftIsGrab to check if 
the right or left hand is making the grab gesture respectively. This describes how most objects 
interact with the glove data. There are some intermediary scripts for scaling the data and 
applying it to the fingers on the player model that will be described later on in this paper.

2.1.3 - Scenes and GameObjects' Descriptions

There are multiple scenes in our game and each one has it’s own GameObject 
hierarchy. Some GameObject hierarchies are used in multiple scenes and some are specific to 
each scene. One GameObject hierarchy used in multiple scenes is the “Carl” character model.

2.1.3.1 - Carl GameObject Hierarchy

Our main in-game character is named “Carl” and he exists as a hierarchy of Unity game 
objects. Each object within Carl’s hierarchy represents a piece of his appearance. At the 
highest level is the object “Carl”. This object serves as a container for the rest Carl’s complete 
skeletal structure. Shown below is a level directly below the containing “Carl” object. 
Appearance specific objects (hair, glasses, shirt, shoes, etc.) are included here. The root node 
of Carl’s skeletal structure is also defined here as “Hips”. Underneath the Hips object is a tree 
representing all the movable joints in Carl’s body.

Fig. 3 High Level Carl
Attached to the “Carl” containing object is a script named “ZigSkeleton.cs”. This Zigfu provided script is present in all of our interactive scenes. The script uses the Zigfu ZDK to update joint positions and rotations based on values collected from the Kinect sensor. This is
done by adjusting the joint’s Transform component. ZigSkeleton begins by instantiating an array of all transforms present in Carl’s hierarchy. When the Kinect senses a movement by a user an event is triggered. This event then evokes the ZigSkeleton function UpdateRotation() which moves the corresponding joint accordingly.

Carl’s hierarchy differs slightly depending on which scene is running. The ReachBack scene integrates the Oculus Rift and requires a few extra game objects. The first of these extras is an OVRCameraController object. This object is attached as a child of Carl’s head object.

There are two Oculus specific scripts attached to the OVRCameraController Object, OVRCameraController.cs and OVRDevice.cs. These scripts hold and position the Oculus on the head of Carl.

The second extra feature required for the ReachBack scene is a second container for the Carl hierarchy. This object exists at the same object level as the original Carl container. This container appears as a capsule surrounding Carl in the editor. It is needed to stabilize the Oculus on Carl.
Our ShoulderROM scene does not require any of these added features as it does not utilize the Oculus Rift. Other than these Oculus specific components the only difference between Carl's hierarchy in our different scenes is the scripts that are attached to Carl. The different scripts will be explained in the next sections regarding specific scenes.

2.1.3.2 - Zigfu GameObject

Our ShoulderROM scene and ReachBack scene make use of the same Zigfu prefab object. This “Zigfu” object exists at the highest hierarchy level along with Carl. The Zigfu object is used to host a variety of Kinect related scripts.

<table>
<thead>
<tr>
<th>Script Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zig.cs</td>
<td>Kinect configuration script for choosing Kinect input type, smoothing parameters, mirroring, and more</td>
</tr>
<tr>
<td>Script Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ZigDepthViewer.cs</td>
<td>Displays a Kinect depth visualization in the bottom right corner of the screen</td>
</tr>
<tr>
<td>ZigUsersRadar.cs</td>
<td>Displays a tracked user radar in the top right corner of the screen</td>
</tr>
<tr>
<td>ZigEngageSingleUser.cs</td>
<td>Launches events for users found and users lost</td>
</tr>
<tr>
<td>KinectSpecific.cs</td>
<td>Displays configurable Kinect options (sensor elevation) in the top left corner of screen</td>
</tr>
</tbody>
</table>

Fig. 7 ZigFu Script Descriptions

2.1.3.3 - ShoulderROM Scene

Our ShoulderROM scene is used to measure various maximum and minimum shoulder angles. It is also used to assess arm strength. The purpose of the scene is to generate accurate Goniometric measurements using the Kinect sensor as an alternative to manual protraction methods.

The object hierarchy of this scene contains a Carl prefab, floor, main camera, light source, and Zigfu prefab.

![ShoulderROM Hierarchy](image)

A special component is attached to Carl in this scene, ShoulderReader.cs. This script controls the exercise flow. There are 8 separate exercises in the scene. There are 2 range of motion exercises (Left and Right Arm) and four strength exercises.

<table>
<thead>
<tr>
<th>Exercise</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max/Min Right Arm Shoulder Abduction Angle</td>
<td>User is asked to raise right arm as high and as low as they can</td>
</tr>
<tr>
<td>Max/Min Left Shoulder Abduction Angle</td>
<td>User is asked to raise left arm as high and as low as they can</td>
</tr>
<tr>
<td>Right Arm Side Strength</td>
<td>User is asked to hold their right arm out at their side as long as possible (30 sec max)</td>
</tr>
<tr>
<td>Exercise Description</td>
<td></td>
</tr>
<tr>
<td>--------------------------------------</td>
<td></td>
</tr>
<tr>
<td><strong>Left Arm Side Strength</strong></td>
<td></td>
</tr>
<tr>
<td>User is asked to hold their left arm out at their side as long as possible (30 sec max)</td>
<td></td>
</tr>
<tr>
<td><strong>Right Arm Front Strength</strong></td>
<td></td>
</tr>
<tr>
<td>User is asked to hold their right arm at their front as long as possible (30 sec max)</td>
<td></td>
</tr>
<tr>
<td><strong>Left Arm Front Strength</strong></td>
<td></td>
</tr>
<tr>
<td>User is asked to hold their left arm at their front as long as possible (30 sec max)</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 9 Exercise Descriptions

All of the above exercises write to the database as soon as a transition is made to the next exercise. These writes are executed inside the Unity OnGUI() function whenever the “Next” button is pressed.

The ShoulderReader script also takes advantage of the JointSampler class defined in JointSampler.cs. This class is used to store time series joint positions and rotations. The ShoulderReader script creates an instance of the JointSampler. The JointSampler function SampleAllJoints() is then called 10 times every second. The function takes a snap-shot of all joint rotations and positions before storing them into the associated user’s time series database tables. The sample rate can be configured inside the editor through the option “capture_rate”. We have not tested at a more frequent rate than mentioned above. It’s unclear what effect this will have on the database.

2.1.3.4 - ReachBack Scene

Our First exercise scene tasks the user with moving four cylinders placed around him/her into a box in front of the user. Two of the cylinders are on a shelf above the user and the other two are on a counter behind the user. The scene records user position data continuously and the time it takes the user to place the objects in the box. This data is stored using the Unity database interface described in the database section of this paper.
Pictured above is how the scene appears in the Unity editor with the GameObject hierarchy tree located on the left. The two most simple objects are “Area Light” and “Floor”. The “Area Light” is a inherent Unity GameObject that creates light in an area with tuners for range and intensity. The “Floor” is made from an inherent GameObject called cube (it is called cube but it can be any rectangular prism). This cube is scaled to be planar along the X and Z axes in the level. Also we added a texture to it’s renderer to make it appear more realistic. In order to create the “Counter” the “Box” and the “Shelf” we do similar steps. These are all cube GameObjects scaled and textured to appear the way they do. The “Box” is actually made up of five cubes, four for the walls and one for the bottom. Lastly we have the cylinders which are made from an inherent Unity GameObject called cylinder. They are once again scaled and textured to fit our purposes.
In order to make the player able to grab the cylinders we have to create a way for the player's virtual hand to communicate with the cylinders as well as the glove. To do this we create an object called GrabObject and attach it to Carl's hands. This object is essentially an invisible rectangle that fits around Carl's hand that we use to detect collisions. To do this there is a script component attached to the GrabObject called GrabLogic.cs. Inside this script we use the UnityEngine's innate function OnCollisionEnter. This is a function that gets called whenever the object this script is attached to enters a collision. In our version of this function we check if the object we collided with is a cylinder. Also we check if the player is actually making the grab gesture using the GloveReader interface. If these are both true then the script updates the transform of the associated cylinder in the Update function with the position data of the hand. When the player let's go of the object, the script stops updating the position of the cylinder.

The last part of the scene is recording how long it takes the user to drop all the cylinders in the box. To detect that the cylinder's are actually in the box we use a script component on both the cylinder objects and the bottom of the box. The cylinders’ script component is called CylinderLogic.cs. This script has a very simple purpose. When it collides with the bottom of the box, send a message to the bottom of the box GameObject (called “bottom”) that the cylinder is there. In order for the “Bottom” object to receive this message it has a script component of it’s own called CylinderCatch.cs. CylinderCatch.cs keeps track of time during the scene and keeps track of how many cylinders have been caught. When the number of cylinders reaches 4, the time stops updating and gets recorded to the database.

2.1.3.5 - Login Scene
The Login scene is driven completely by the script `DatabaseSetup.cs`. It begins by checking to see if a UserProfiles table exists in the database. If it does not or the database doesn’t exist, all tables will automatically be regenerated. This is done by calling the access function `GenerateTables()`. The scene then drops into its `OnGUI()` update function which is executed up to several times per frame. Users have the option to login, create a new user, or delete a user. There are also configurable options to force the glove calibration scene (although calibration data already exists), and to reset any existing times series data. All of these database accesses are completed by opening the database, calling an access function, and closing the database.

### 2.1.3.6 - SceneSwitch Scene

The SceneSwitch scene was created as an alternative to loading levels using an in-scene menu. We originally created a menu within all of our scenes. This menu could be activated by pressing the ESC key. Once the menu was present you could select any scene to switch to. We encountered a bug with this approach that caused the Kinect to stop functioning after any scene transition.

Instead of loading a menu this way the project now loads the scene "SceneSwitch" when the ESC key is pressed in any of our scenes. From within this separate scene we can successfully load any desired scene with zero Kinect problems. This is the most basic of our project’s scenes.

The scene consists of a GUITexture component that serves as the background for the scene. There is also a `MainCamera` game object that contains the script `LevelSelect.cs`. This script uses Unity `OnGUI()` function to display a list of buttons, each one indicating the desired scene to jump to.

### 2.1.3.7 - CalibrateGlove Scene

The CalibrateGlove scene is used in recording data at specific points on the gloves bend sensor range in order to match the sensors input with the game’s hand animation. The user is presented an image of a hand that they are supposed to mimic, and then press "Space Bar" to
move on to the next image. At the end all the values are stored to the database with the associated User’s profile.

![Fig. 13 CalibrateGlove scene in Unity editor](image)

The scene is comprised of four images of a hand, one for each step of the calibration sequence, a camera that travels over these images, and a text object that is used for displaying instructions. The most important piece of this scene is a script component attached to the camera called RecordAndMoveCalibrate.cs. This script is responsible for moving the camera, updating the instructions text, and storing the calibration data into the database. While this sounds like quite a lot, it is rather simple. The script waits for the user to press space bar. When this event occurs it records the values from the glove sensors by reading from the GloveReader, moves the camera to the next image by changing the transform of the camera, and changes the text by directly accessing the text object. The text reads “Copy the hand position shown in the image. Finger Extension: <percent extension>%%”. When the camera reaches the last picture and space bar is pressed, the data is stored into the database and the SceneSwitch level is loaded.

### 2.2 Database

For our project it is essential that we have an effective way to store and track data from our users. This requires the ability to identify and collect important data points that can be presented meaningfully at a later time. There are numerous steps involved in achieving this goal such as database selection, schema design, and database access.

#### 2.2.1 - Database Approach

There are three common database strategies that can be implemented with the Unity Interface. Two involve instantiating a remote database (outside of the Unity application), and using various methods to communicate with them. The third uses a pre-made library and can be built completely inside of a Unity3D project.
The first of the two remote strategies uses Microsoft’s SQL Server along with a C# database access library native to Unity. This method is appealing because a library to execute database queries on a remote server (inside Unity) already exists. The down side is that Microsoft’s SQL Server software costs $50 (as of 3/13/14). This is the reason we are not using this approach.

The second remote approach is open to the majority of servers types. This method involves sending HTTP requests from a Unity program to the remote server which in turn runs appropriate server-side scripts to execute the desired queries. The use of HTTP requests is recommended by most Unity developers as it ensures that backend processes are not being executed by the client. For that reason this is the most secure database implementation. To avoid constructing a database access library from scratch, web frameworks such as Django are often used in support. Unfortunately we did not have the resources (time and money) to implement this strategy. This approach requires instantiating a web server and building an access interface on both the server-side and client-side (Unity).

We are using the third and final database option for our project. Our project utilizes a SQLite dynamic link library (DLL) as its database engine. This database comes in the form of a single file SQLite.dll which can be placed anywhere inside our projects Assets folder. The DLL can be downloaded directly from the SQLite web page https://sqlite.org/download.html. Once this file is inside the project’s “Assets” folder, it can be used with MonoDevelop’s Mono.Data.Sqlite library to execute queries which read and write from an automatically generated text file. We are also taking advantage of a database access class, dbAccess, provided by Unity’s community wiki page at http://wiki.unity3d.com/index.php/SQLite. This class has great functions for executing specific database queries using the DLL. Although the DLL approach is not ideal, it takes away much of the complexity involved with creating database access methods. For this reason we are able to focus on our constantly changing database schema. A flow chart of our database set-up is shown in the figure below.
The most important table we maintain is UserProfiles. This table contains administrative data for our users such as their name. This may later come to hold information such as the user’s physical attributes and/or condition. The most important field is our user’s identification number or “p_id”. This value is used as a key to access data from other tables that pertain to the user. This allows us to easily hold on to one value (the identification number) throughout the project scenes and use it whenever a database access is necessary for the active user.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>p_id</td>
<td>Unique user id</td>
</tr>
<tr>
<td>created_dtm</td>
<td>Profile creation datetime</td>
</tr>
<tr>
<td>first_name</td>
<td>User first name</td>
</tr>
<tr>
<td>last_name</td>
<td>User last name</td>
</tr>
</tbody>
</table>
shoulder_rom_rot | Time series rotations data table name from Shoulder ROM scene
--- | ---
shoulder_rom_pos | Time series positional data table name for Shoulder ROM scene
cyl_reach_rot | Time series rotational data table name for cylinder reach scene
cyl_reach_pos | Time series positional data table name for cylinder reach scene
cyl_reach_glv | Time series glove data table name for cylinder reach scene

*Fig. 15 UserProfiles Table*

The LeftCalibration and RightCalibration tables hold user specific glove calibration data. The values in this table are used as a part of our scaling process in order to make the hands' opening and closing game animation as realistic as possible. Each calibration table holds a glove reading for each finger when the hand is completely open, and when it is completely closed.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>user_id</td>
<td>Foreign Key references UserProfiles (p_id)</td>
</tr>
<tr>
<td>index_0</td>
<td>Index finger sensor open hand</td>
</tr>
<tr>
<td>index_90</td>
<td>Index finger sensor closed fist</td>
</tr>
<tr>
<td>middle_0</td>
<td>Middle finger sensor open hand</td>
</tr>
<tr>
<td>middle_90</td>
<td>Middle finger sensor closed fist</td>
</tr>
<tr>
<td>ring_0</td>
<td>Ring finger sensor open hand</td>
</tr>
<tr>
<td>ring_90</td>
<td>Ring finger sensor closed fist</td>
</tr>
<tr>
<td>pinky_0</td>
<td>Pinky finger sensor open hand</td>
</tr>
<tr>
<td>pinky_90</td>
<td>Pinky finger sensor closed fist</td>
</tr>
<tr>
<td>knuckle_0</td>
<td>Knuckle sensor open hand</td>
</tr>
<tr>
<td>knuckle_90</td>
<td>Knuckle sensor closed fist</td>
</tr>
</tbody>
</table>
The results table contains fields for various high scores and measurements collected during sessions. Similar to the calibration data, user rows in this table are referenced using the \( p_id \) field defined in the UserProfiles table.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>user_id</td>
<td>Foreign Key references UserProfiles (( p_id ))</td>
</tr>
<tr>
<td>reachback_count</td>
<td>How many cylinders successfully dropped in bin (ReachBack scene)</td>
</tr>
<tr>
<td>reachback_time</td>
<td>Time it took to complete ReachBack scene</td>
</tr>
<tr>
<td>l_shoulder_rom_max</td>
<td>Maximum left shoulder abduction angle</td>
</tr>
<tr>
<td>l_shoulder_rom_min</td>
<td>Minimum left shoulder abduction angle</td>
</tr>
<tr>
<td>r_shoulder_rom_max</td>
<td>Maximum right shoulder abduction angle</td>
</tr>
<tr>
<td>r_shoulder_rom_min</td>
<td>Minimum right shoulder abduction angle</td>
</tr>
<tr>
<td>l_tpose_time</td>
<td>Left arm side extension hold time</td>
</tr>
<tr>
<td>r_tpose_time</td>
<td>Right arm side extension hold time</td>
</tr>
<tr>
<td>l_arm_front_time</td>
<td>Left arm front extension hold time</td>
</tr>
<tr>
<td>r_arm_front_time</td>
<td>Right arm front extension hold time</td>
</tr>
</tbody>
</table>

Lastly we have our time series tables. The purpose of our time time series tables is to collect real time joint rotational and positional data at a variable frequency. After a session is complete, we expect to use these time series tables as another way of assessing user performance. By using this data in a time controlled graph we are able to re-watch any user’s session.

The time series tables have the potential to become very large, very fast. By default we sample 132 joint values 10x/sec. For this reason we designed the tables in a way such to avoid DELETEs and individual row SELECTs and INSERTs. Each user that uses the system has their own set of time series tables with a unique name. The user’s primary key id is appended to table base names to construct the unique table name. The table implementations only support complete table operations. In order to reset a user’s time series data, the tables are destroyed.
and regenerated. To access the data, a query of the entire table is executed. To insert a row, a complete array of values for each table field is given.

<table>
<thead>
<tr>
<th>Table Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReachBackRot_&lt;p_id&gt;</td>
<td>Joint rotation data for ReachBack scene</td>
</tr>
<tr>
<td>ReachBackPos_&lt;p_id&gt;</td>
<td>Joint position data for ReachBack scene</td>
</tr>
<tr>
<td>ReachBackGlv_&lt;p_id&gt;</td>
<td>Scaled glove values for ReachBack scene</td>
</tr>
<tr>
<td>ShoulderROMRot_&lt;p_id&gt;</td>
<td>Joint rotation data for ShoulderROM scene</td>
</tr>
<tr>
<td>ShoulderROMPos_&lt;p_id&gt;</td>
<td>Joint position data for ShoulderROM scene</td>
</tr>
</tbody>
</table>

Fig. 18 Time Series Tables

2.2.3 - Database Interface

The heart of our database implementation exists in the form of two files. A SQLite.dll file downloaded from the SQLite web page [https://sqlite.org/download.html](https://sqlite.org/download.html) is located in our project main Assets folder. The database access class “dbAccess” is written in the file “dbAccess.js” and is located in the Plugins folder in the directory “Database”. We originally downloaded “dbAccess.js” from the Unity community wiki page at [http://wiki.unity3d.com/index.php.SQLite](http://wiki.unity3d.com/index.php.SQLite). We then added to the class to turn it into a completely self containing access class for our project. The following lines are present throughout our code base:

```javascript
    db_control.OpenDB();
    db_control.<Function>(args);
    db_control.CloseDB();
```

Wherever a database access is necessary, a “dbAccess” instance (db_control) is opened, a query is executed, and the instance closed. We do this to ensure that database instances do not linger open. If a single database table is queried by two different instances, an SQLite exception will be thrown and the database compromised.

The database access class “dbAccess” also serves as a variable reference. The name of the database file, “RehabStats.sqdb”, is defined here. The class also contains all scene names, database table names, table fields, table default values, and table constraints. The database table structural information is not immediately accessible through a dbAccess instance but is used to generate/regenerate all the database tables.
The database operates through a IDbConnection interface which is implemented by Microsoft .NET framework data providers which access relational databases. This interface exists in the Microsoft System.Data namespace. When a process attempts to open the database, an IDbConnection instance is created with a call to SqliteConnection() given the database file name. Processes can execute queries by calling any of the class’ predefined access functions. They can also generate complete query strings and execute at a higher level. In either case, the query is executed the same way. A complete query string is created and assigned as the “CommandText” to an IDbCommand object. The IDbCommand object’s ExecuteReader() function is then called which returns an IDataReader object. The IDataReader is then read record by record into an array and returned to the caller.

Every scene in our project touches the database at least once whether it involves reading, writing, or both. Our goal was to keep as much of the database complexity inside the access class as possible. This way the higher level scripts can focus on the scene appearance itself.

2.2.4 - Visual Graphing Interface

There are two python scripts that are used for graphing the data recorded during the exercises in this database. The first, “readDB.py”, provides an interface into the database that allows us to easily retrieve all the data we are looking for without having to write out SQL queries. The second, “dbGUI.py”, describes a wxPython application that has a graphical interface for selecting and graphing exercise data. In order for these scripts to work there are list of dependencies beyond just Python 2.7 required: pyparsing, matplotlib, numpy, python-dateutil, pytz, six, and wxPython. These are all python packages.

The readDB.py script provides an interface to get necessary data from the database without having to specify SQL queries manually. The following is the interface description.

readDB.dbReader() returns a new dbReader object. This object can be used to access data using the interface described here.

dbReader.readNames() returns a list of strings of all the users names in the database along with the user ID number. The format is <id> <first name> <last name> for example "1 Bob Builder".

dbReader.readExerciseTables(id) takes a user id (obtained in the string from dbReader.readNames()) and returns a list of time series table names for each exercise associated with the given user ID.

dbReader.readTimeTables(table_name) takes a string representing the desired time series data table (a list of these are retrieved in the readExerciseTables function) and returns a
list with all the datapoints from this time series data table. These data points are XYZ locations of each joint or rotation values of each joint.

The dbGUI.py script describes a GUI for selecting what user data to graph in the graphing utility. This script should be executed directly in order to run the graphing GUI. In the future this can be compiled into an executable if necessary. This program is an event driven interface. The user interacts with menus and this causes function calls that progress the GUI to a different state. Below is are images of how it appears.

![Fig. 19 User select screen.](image)

The program starts by calling a function called InitUI(). This function creates the wx panel object that we put our GUI widgets into and populates it with the initial widgets. These widgets are as follows:

- Dropdown menu of all the User Names in the database (calls the OnSelect function when clicked).
- A text box that displays the selected user.
- A button that causes the GUI to progress to the next screen (it calls the SelectDataScreen function when pressed).

OnSelect(event)
Updates the static text box to the currently selected User Name from the initial dropdown menu.

SelectDataScreen(event) is a function that gets called when the button declared in InitUI() is pressed. It takes the user id from the User Name string and uses it to get a list of names for all the time series data tables. These tables are specific to each user and exercise combination. These names are used to populate another dropdown menu for selecting what exercise data you want to look at. This dropdown menu calls the OnSelectTable function when activated.

![Fig. 20 Data select and graph screen.](image)

OnSelectTable(event) gets a list of time series data points from the data base for use in graphing. Also a slider widget is created for moving through the graph. When the slider is moved it calls the OnSliderScroll function.

OnSliderScroll(event) is a function that is responsible for actually populating and drawing the data plot. It takes the list of data points, turns them into lists of x, y, and z coordinates respectively, and then plots those points in the graph against each other. Also this function updates a text label that says the value in seconds into the exercise the current plot represents.

2.4 The Glove

The glove was an integral part of our project, because we wanted the user to be able to interact with objects in the virtual world. The kinect does not have the resolution to map a user's hand at a distance. During the design process of our gloves we made key choices. The first decision we made was to use bend sensors as a way to map finger positions. Another decision we made was to make the glove fingerless so that it could fit on wider range of hand sizes. The third decision we made was to make it communicate wirelessly through bluetooth, so that the
user could move freely. And the last decision we made was to add an accelerometer to track the hand orientation.

![Fig. 21 The glove.](image)

Making the gloves fingerless was a choice we made after the first couple glove prototypes. We noticed that even on our team the glove would not fit everyone, which was a bad sign if we wanted it to be worn by a range of different people. To make the glove fingerless, however, we needed a way to brace the bend sensor to the top of the finger. What we came up with was a series of velcro rings that slip onto the bend sensors and hold them on the finger. The finger slip allow for the sensor to move freely along the finger while holding it down.

2.3.1 - Bend Sensors

To gather data from the users hand, we used the 2.2” flex sensor, SEN - 10264. This flex sensor behaves like a variable resistor. As the sensor is bent, the resistance across the sensor increases. We tested these sensors to observe if the resistance and angle of bend had a linear relationship. Using a goniometer and a multimeter we were able to measure the resistance of these bend sensors in 15 degree increments. The figures below illustrate these relationships.
When placed on a glove, as shown above, the sensor will be at minimum resistance when the user has an open fist and maximum resistance when the hand is in a fist. To generate a voltage from the bend sensors, we used a non-inverting amplifier circuit. The op-amp we used was the MCP6004. This is a general purpose op-amp. Our glove operates at low voltages and at low frequencies and that is why the MCP6004 is a suitable option. The circuit uses a the variable resistance of the flex sensor and a 100 kΩ resistor. We chose 100 kΩ because the bend sensor features a resistance that increases as the sensor ages. The resistive ink does not return to the original placement and the resistance goes up. Having a large enough resistance covers the entire range of the sensor and allows for the sensor to age and still work properly. The inputs to the non-inverting amplifier are 3.3 volts and 1.65 volts. The 1.65 volts comes from the 3.3 volts rail put through a voltage divider and buffered. This voltage is buffered to avoid loading down the divider circuit.
This circuit is simpler than previous models of the glove, however, this is because of the inclusion of another bend sensor across the lower knuckle of the middle finger. Previously, we used 4.5” sensors to represent the entire finger for the index, middle, and ring fingers. We decided that using the extra sensor allows use to sacrifice output range in exchange for more concrete information. In Unity, the extra bend sensor provides the movement of the lower knuckle of all the fingers. The rest of the finger is bent using the individual bend sensors on each finger. The additional data allows Unity to display more animations compared to previous methods of the glove. Giving the user more control was the reason for adding another sensor, but in order to preserve space we had to reduce the amount of op-amps needed on the PCB.

2.3.2 - Accelerometer Circuitry and Communication

The accelerometer we used on our glove was the MPU-9150, which is an accelerometer, gyroscope, and magnetometer. We, however, were only interested in the accelerometer readings. The MPU-9150 has a three axis 16 bit accelerometer, which refreshes at a 50 hz rate. We communicate with the MPU-9150 with a I2C interface from the MSP430 microcontroller. The circuitry for this link can be seen in the figure below.
The circuit above depicts the two wire interface between the microcontroller and the accelerometer. The resistors in this circuit are pull up resistor which serve to pull the voltage of the communication lines up when they are not being used.

I2c is a communication protocol which uses two wires to communicate between master and slave devices. One of the two wire is the data line and the other is the clock line. On a single i2c line there can be multiple slave devices but there can only be one master. In our case there is only one slave, the accelerometer, and one master, the micro-controller. To begin communication using in i2c we have the master send out the 7-bit slave address on the i2c lines along with a 1-bit read or write bit. The corresponding slave device will respond with an acknowledge pulse and then the master will write the register address of interest. The steps after this depend on whether you are writing or reading from the slave. The following charts describe the read and write protocols.

**Single-Byte Write Sequence**

<table>
<thead>
<tr>
<th>Master</th>
<th>S</th>
<th>AD+W</th>
<th>RA</th>
<th>DATA</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slave</td>
<td></td>
<td>ACK</td>
<td>ACK</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Burst Write Sequence**

<table>
<thead>
<tr>
<th>Master</th>
<th>S</th>
<th>AD+W</th>
<th>RA</th>
<th>DATA</th>
<th>DATA</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slave</td>
<td></td>
<td>ACK</td>
<td>ACK</td>
<td>ACK</td>
<td>ACK</td>
<td></td>
</tr>
</tbody>
</table>
Each of the three accelerometer data points is stored as a 2’s compliment number split up between two registers. So when we receive the data we have to reconstruct it on the micro. The first step is to retrieve both half of the accelerometer data from the accelerometer. We then convert the data from chars to 16 bit unsigned ints. After the two have been converted we shift the upper 8 bits to 8 bits to the right and add it to the lower 8 bits. to get the 16 bit 2’s compliment value of the data.

2.3.3 - Bluetooth

To get the glove data from the glove to the computer we used a bluetooth connection. We used bluetooth because it allowed us to replace our physical serial wires with a bluetooth connection with no code or major wiring alterations. The chip we used to send the data via bluetooth was the HC-06, which is a bluetooth device that can both send and receive data over a serial link with any bluetooth dongle.

The data from the glove is sent to the HC-06 through the UART on the MSP430. It communicates at baud rate of 9600 and connects to a serial port on the computer automatically, when the dongle is present. with this particular device we get a range of up to 100 ft indoors, which is far more than we will ever need with such a device.

2.3.4 - Printed Circuit Board

To put all of this together, we used a printed circuit board. The MSP430 takes input from a 16 channel multiplexer, we used the HC 4067 chip. The mux takes inputs from the bend sensors and is used in case there is any additional sensors that want to be added to the glove. With the microcontroller having a limited amount of pins, a multiplexer was needed. The PCB is powered using a 9 volt battery that is put through a 3.3 volt linear regulator. A switch was added to the board in order to easily turn the glove on and off. Additional bypass capacitors were added throughout the design to protect the circuit.
2.3.5 - Glove interface program

To interact with the glove we needed the data to be streamed into Unity. To do this we created our own application, which is responsible for receiving data streamed from the micro controller on the glove and packaging the data in a text file for Unity. The advantages of using an independent program to interface with the glove is that it gives us a layer of protection from the glove. If in the rare occurrence the glove stop transmitting data, transmits to quickly, or transmits the data incorrectly Unity will not be affected and will continue running normally.

The application is a window that allows the user to select the gloves com port, open and close communications, view the raw serial stream coming in, and view the unpackaged and organized data. The top box labeled raw data, allows the user to see the data streaming in from the glove. The second box down displays any status or error messages provided by the glove or the application. On the side we have three buttons labeled "INIT", "OPEN", and "CLOSE". The first button "INIT" checks to see if the selected com port is available, if it is not the application will write the error in the error box and if it is it will write that the com port is available in the error box. The "OPEN" button, opens the serial port, which begins the data receiving process. The "CLOSE" button end the communication with the serial port. There is also a serial port scrolling menu, which allows the user to select their desired serial port. Below the error box we have a few text boxes dedicate to displaying the most currently received data from the glove.
In order to send the data over a serial COM we design a simple method to package the data. The first thing we do to the data is encapsulate the data into a string. This is done by converting the data to base 10 digit by digit and storing each digit as a single byte. After we create the string we add a letter to the front of the data packet. This letter determines what the data is for. For example, “1034” becomes “A1034” so that we know that this number is associated with the first fingers bend sensor reading. When the data is received on the other side of the link we sort the incoming data based of the letter received prior to the number. The advantage to sending the data as a string as opposed to a more compact integer is that we can determine what each byte is for. If we send the data over the serial line as an integer it would be very difficult to determine a unique way to organize the data and signal a new byte is coming. Once the data is received we decode into numbers and place them in a text file, one data-point per line.

3. Conclusions

Our project shows a lot of promise for use in the field of stroke second scene serves as a proof of concept. The scene will have patients attempt to move their arms to desired angles while the We would like to thank our senior design project mentors Paul Naud and Mirceau Teoderescu. We would also like to thank Sri Kurniawan that they would enjoy in order to help them focus on the game system measures the actual angle of their joint bends. Currently, this Kurniawan for funding our project and providing resources in the stroke therapy community, and distract them from the fact that they are doing therapeutic is done by hand by the doctor. therapy. The eventual goal would be for a patient to be able to play exercises. This
has been shown to be very effective. The work we started will be continued by others at UCSC once we leave. Together, the glove and Kinect can take measurements from the user while they are performing a test. We pushed for simple, practical movements that are relevant to everyday life. Our Unity scenes reflect this. In stroke therapy, we have to observe how a patient's body has to compensate in other parts of the body while performing certain movements. The scenes ask the user to demonstrate basic tasks and observes how they complete them. The data that is stored into our database can be used to observe multiple joints on the human body throughout the entire testing period. The glove also works to get as much data from the users hand, somewhere the Kinect lacks in resolution. The glove uses five flex sensors and an accelerometer in order to measure how well a person can perform hand and wrist movements. We focused on making a few scenes as smooth as possible but there is a lot more someone can do to add to or improve the system. Altogether, we have created a strong foundation for an 3D stroke therapy environment.

4. References


