*Cinema*craft

Virtual Minecraft Presence Using OPERAcraft

Client	Ivica Ico Bukvic
Instructor	Edward A. Fox
Course	CS 4624 Multimedia, Hypertext, and Information Access Virginia Tech Blacksburg, VA 24061 Spring 2016
Members	Barnes, Brittany Godolja, Elsi Kiseleva, Marina

Table of Contents

User's Manual

Developer's Manual

Lessons Learned

List of Figures and Tables

Executive Summary

*Cinema*craft allows users to view their mirror image in the form of a Minecraft avatar in an interactive exhibit setting. Built off of OPERAcraft, a Minecraft modification created at Virginia Tech, *Cinema*craft uses Kinect motion-sensing devices that will track user movement and extract the spatial data associated with it. The processed data is then sent through a middleware, Pd-L2Ork, to the Minecraft code where it is translated into avatar movement to be displayed on the screen. The result is a realistic reflection of the user in the form of an avatar in the Minecraft world.

Within the display limitations presented by Minecraft, the avatar can replicate the user's skeletal and facial movements; movements involving minor extremities like hands or feet cannot be portrayed because Minecraft avatars do not have elbows, knees, ankles, or wrists. For the skeletal movements, three dimensional points are retrieved from the Kinect device that relate to specific joints of the user and are converted into three dimensional vectors. Using geometry, the angles of movement are found in each plane (X, Y, and Z) for each body region (arms, legs, etc.). The facial expressions are computed by mapping eyebrow and mouth movements within certain thresholds to specific facial expressions (mouth smiling, mouth frowning, eyebrows furrowed, etc.).

As it is sent to Minecraft, the movement data is intersected by Pd-L2Ork in order for it to be scaled and normalized. This minimizes error and allows for more fluid and natural avatar movements.

OPERAcraft itself is a Minecraft modification intended for K-12 students that allows users to create and perform a virtual opera in Minecraft. Upon completion, the Kinect user recognition modification created by this project is intended to be utilized by the main OPERAcraft system, which previously used keyboard commands to simulate minor avatar movements.

System Instructions

1. Ensure all hardware components, listed in Figure 4.1.2, are in present and functioning properly.
2. Connect the first and second monitors with a LAN connection so data can be exchanged.
3. If a third monitor is available (to be used as the primary display), share the display between the first and third monitors.
4. Run the Kinect skeletal tracking code on the first monitor along with the Minecraft code (if a third monitor is not available the first monitor will also be the primary display).
5. Run the Kinect facial tracking code on the second monitor.
6. Depending on the type of environment desired, set up the Minecraft game (i.e., open port and create a LAN world if needed).
7. The user should stand in front of the Kinect sensors (in order to be tracked) in an open and uncluttered environment.

1.0 Requirements

This project consists of three main components: Kinect sensors, middleware communication, and Minecraft code. Users will need to have two Kinect sensors connected to two computers, in order to track facial and skeletal movements, while simultaneously running the Minecraft code. Once this is setup, users can move freely within the sensor's range and control their Minecraft avatar with their own body movements.

1.1 Project Description

OPERAcraft is a custom Minecraft modification (mod) created at Virginia Tech (VT). It is used to bring together opera, movie production, and video games in real-time. Users are able to control their avatars with body movements consisting of facial and skeletal gestures, real-time camera feeds, subtitles, and, but not limited to, scene cues (2, 7).

Another software developed at Virginia Tech, Pd-L2Ork, serves as the middleware between the Kinect sensors and the OPERAcraft code (7). Pd-L2Ork allows for the distribution of production tasks (i.e. camera changes, subtitles, cues, etc.) and necessary data manipulation (7).

The project setup will consist of multiple screens and two Kinect sensors: one to track facial movements and one to track skeletal movements. The overall goal is to capture Kinect sensor feed data that tracks facial and skeletal body movements and translate it into Minecraft movement, resulting in the Minecraft avatar mirroring the user.

A prototype was presented in an interactive demonstration at South by Southwest (SXSW) in Austin, Texas in March 2016 and a later version at the Institute for Creativity Arts and Technology (ICAT) Day at VT.

1.2 Clients

The target audience for *Cinema*craft includes K-12 students who currently use OPERAcraft to design, create, perform, and watch fully produced virtual operas. The additional functionality of *Cinema*craft will allow them to fully explore their creativity and truly be a part of the entire experience since it is linked to their own personal body movements and expressions.

Additionally, another group of VT students working with FEET software will be using any applicable parts of the Kinect code to further their own project studying how Autism affects emotional comprehension from facial expressions. Particularly, adapting the code to register categorized emotions (e.g. happy, sad, angry, etc.) would aid in their study, since that's a large part of their current work.

Also included among the clients are the attendees of both demonstrations: SXSW and ICAT Day, in Austin, TX and Blacksburg, VA, respectively. These attendees will be able to participate by having their body and facial movements mirrored onto a Minecraft avatar that will be displayed on one of the setup screens. *Cinema*craft will engage users and allow them to imagine how it would feel to not only be a part of an orchestra production (OPERAcraft), but also to be immersed in a real-time, virtual reality environment.

1.3 Skills

Because of this project’s scope, team members will need to know how to use a variety of languages, software, and integrated development environments (IDEs). The various languages and IDEs used will be linked together by middleware software. Though team members are not expected to further develop the middleware they need to be familiar with how to utilize its functionality.

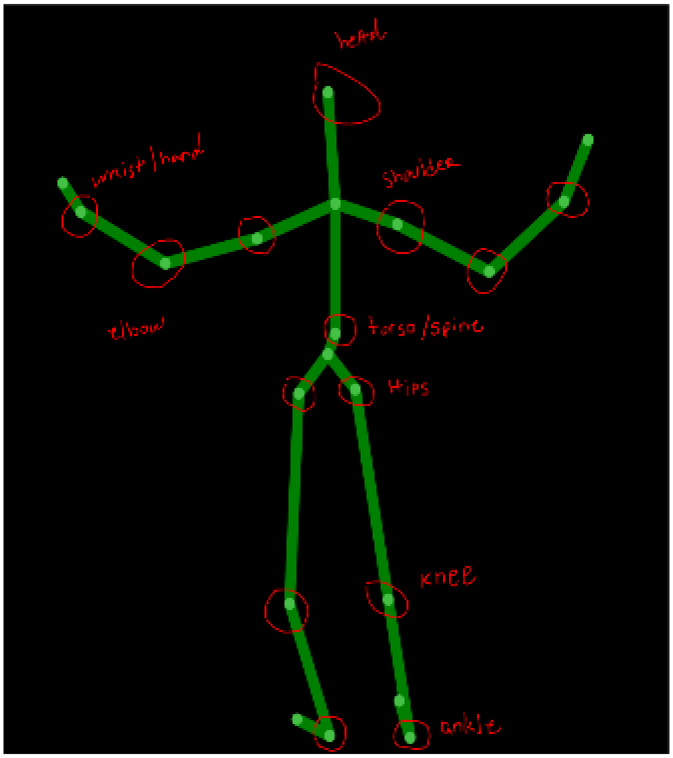


Figure 1.3.1A: Kinect Sensor Live Skeletal Point Feed. Points represent trackable joints, and are labeled to show joints of interest for this project.

1.3.1 Languages and Software

Languages include Java and C# using Eclipse and Visual Basic, respectively. Software includes Kinect SDK/API, Pd-L2Ork middleware, and various monitors and screens used during testing and demonstrations.

With respect to each language and software program, the project has been split up into three sections of work: obtaining and analyzing Kinect data, utilizing Pd-L2Ork middleware, and updating the current OPERAcraft code.

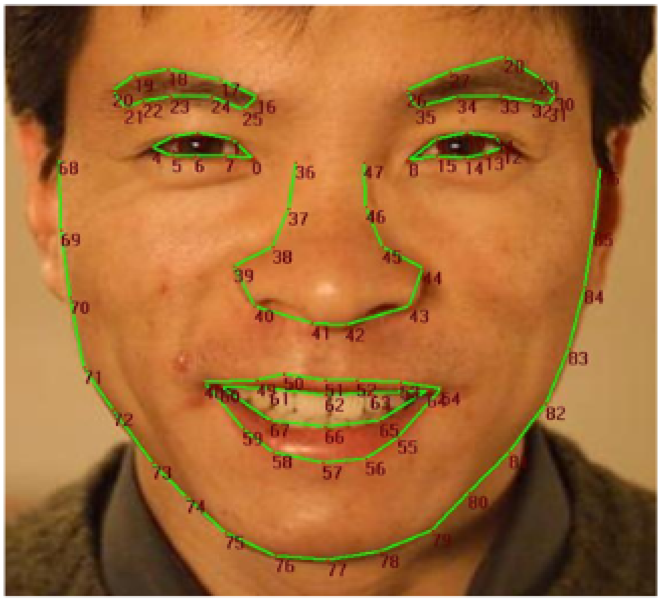


Figure 1.3.1B: Tracked Facial Points from Kinect Sensors. (Source: Microsoft, 2016). For our project we track the points 16, 18, 20 (left eyebrow); 26, 28, 30 (right eyebrow); and 48, 51, 54, 57 (mouth) (6).

First, Kinect sensor data of a user’s skeletal and facial points, Figure 1.3.1A and Figure 1.3.1B (Microsoft, 2016) respectively, will be captured, and subsequently processed and analyzed using C# code in the IDE, Visual Basic. The program is developed on the Kinect software development kit (SDK). This code will be used to properly format the body movements into an interpretable format, which will then be sent via UDP to the middleware.

The UDP packets must be formatted using pre-existing constraints in order to be correctly read by Pd-L2Ork and OPERAcraft. These constraints include delimiters, lines endings, avatar keywords, and specific position values. A more detailed account of the current and future constraints can be found in Table 1.1. Additionally, only one movement position can be specified on each line.

UDP Packet Constraints				
Delimiter(s)	Line Ending(s)	Avatar Movement Keywords	Movement Position Value(s)	Movement Position Types
whitespace	;\n	eyes	{0-3}	integer
		mouth	{0-4}	integer
		head	{(-90)-(90)} {(-180)-(180)}	float (xrot, yrot)
		rshoulder	{(y)}	float (y)
		lshoulder	{(y)}	float (y)
		larm	{(-180)-(180)} {(-180)-(180)}	float (xrot, zrot)
		rarm	{(-180)-(180)} {(-180)-(180)}	float (xrot, zrot)
		lleg	{(-90)-(90)} {(-90)-(90)}	float (xrot, zrot)
		rleg	{(-90)-(90)} {(-90)-(90)}	float (xrot, zrot)
		torso	{(x)} {(y)} {(-180)-(180)}	float(x, y, yrot)

Table 1.1: UDP Packet Constraints. Details possible delimiters, line endings, avatar movement keywords, movement position values, and movement position types. These UDP packets can only handle one movement position per line.

Second, the middleware, Pd-L2Ork, will be used to receive the UDP packets containing properly formatted Kinect sensor data (from the C# code) and relay it to the OPERAcraft/Minecraft code (in Java). As long as formatting is correct, this portion of the project is fully implemented. Pd-L2Ork cuts down unnecessary compile time when making additional changes. The user will not see any portion of this interaction. Figure 1.3.1C shows an example of P2-L2Ork being used to connect various user views in Minecraft for an OPERAcraft production.

Third, the existing OPERAcraft system will be updated to work with the additional functionality. The program will need to be updated to reflect additional facial and body movement commands. It will be used to render the various game views that users will see when participating in an OPERAcraft production. The user will never see any portion of any of the code and will only react with the compiled version of the game (depending on their role in the production), displayed in Figure 1.3.1D.

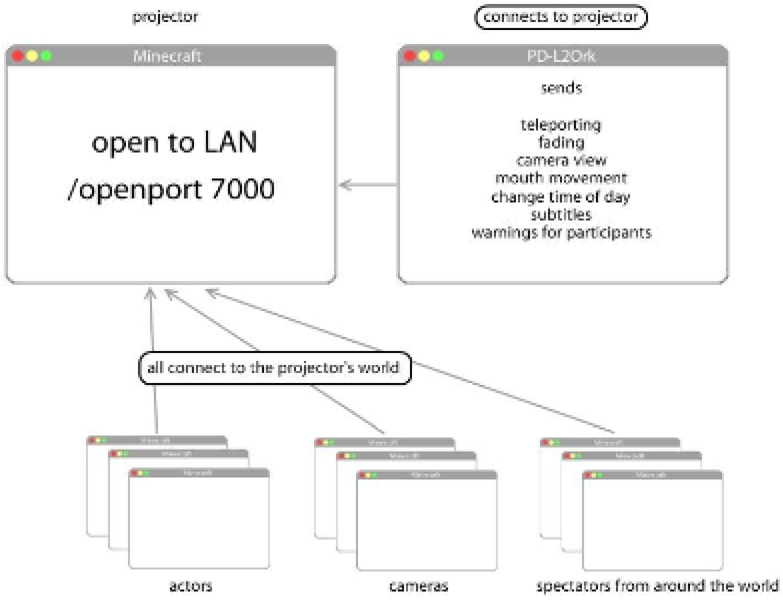


Figure 1.3.1C: Pd-L2Ork Connection Example for OPERAcraft (7).

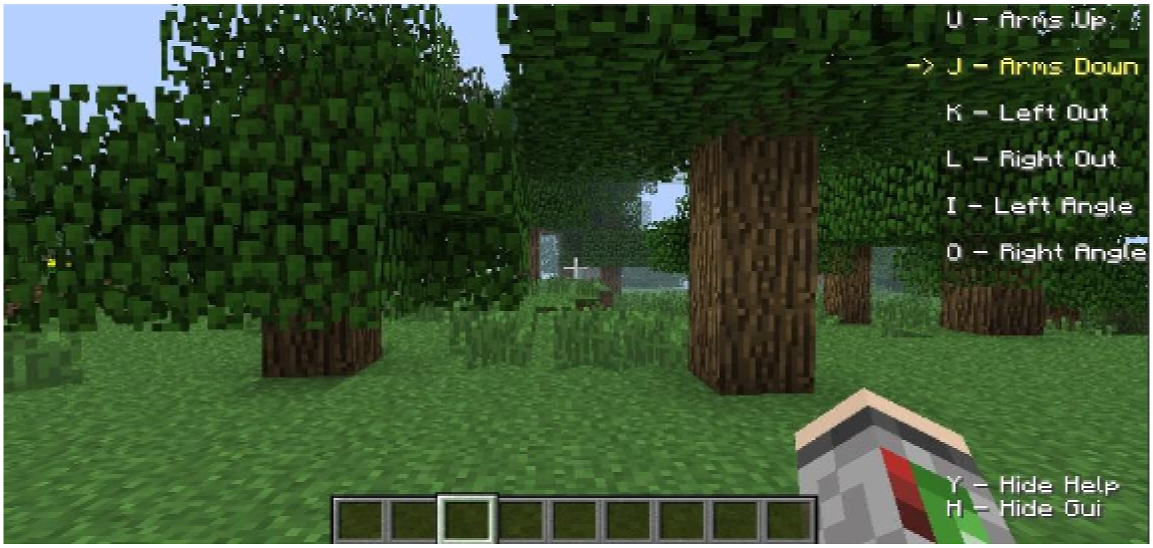


Figure 1.3.1D: Minecraft Gameplay as seen by the User(s). Kinect sensors will pick up on movements like the ones listed in the top right (without needing their respective keyboard controls) (7).

1.4 Developers

Point of contact:

- Ivica Ico Bukvic, VT Associate Professor for the School of Performing Arts (ico@vt.edu)

Additional developers:

- Barnes, Brittany, VT Undergraduate (blbarnes@vt.edu)
- Godolja, Elsi, VT Undergraduate (elsi5@vt.edu)
- Kiseleva, Marina, VT Undergraduate (marinaki@vt.edu)
- Narayanan, Siddharth, VT Graduate (nsiddh3@vt.edu)
- Youssef, Amira, VT Graduate (amira.youssef@gmail.com)

2.0 Software Design

There are three major components of this project: the existing OPERAcraft/Minecraft code in Java, C# code to process incoming data from Kinect software, and Pd-L2Ork which connects the first two components. The flow of these components is demonstrated in Figure 2.0. Individually these components use data structures to distribute information internally, as well as data structures that are used to distribute information among the different components.

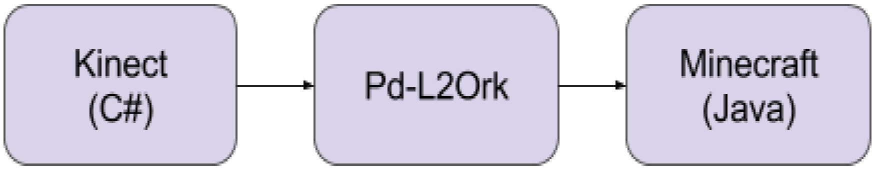


Figure 2.0: Project Components. This figure demonstrates the flow of data throughout the project. Data will be continuously streaming from the live Kinect feed into Pd-L2Ork where it will be sent to Minecraft in order to reflect the user's movements through an avatar.

2.1 Internal Software Data Structure

The Kinect software uses data structures such as point clouds, geometry processing, and computer vision (3). At 30 frames per second, the Kinect can generate point clouds which consist of a set of X, Y, Z coordinates for each visible point on an object. Computer vision utilizes concepts such as depth from focus (objects farther away tend to be blurrier), and depth from stereo (objects closeby get shifted to the side, when looked at from a different angle) (3). The Kinect then uses a decision tree to map depth images to body parts, and reconstructs the objects in front of the image sensors (5). The Kinect SDK makes available these structures to allow for retrieval of the information needed to send to the OPERAcraft software.

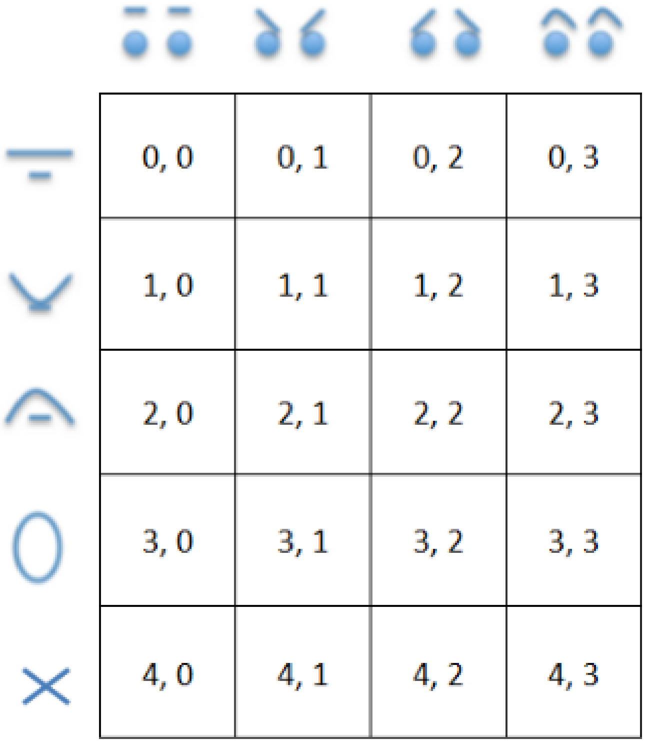
The OPERAcraft software is divided into several different components. The main components of an OPERAcraft production are the actors (controlled by different people from their local machines), cameras (to watch the actors, several cameras allows for different angles), projector (hosts the mod world, and can follow one camera at a time), and the spectators (connect into the game to observe the game from whatever angle). These components are then combined with Pd-L2Ork, which allows for the transportation of information in packets. Such packets can hold information for subtitles, mapping, teleporting, fading, changing camera views, etc. The actors, cameras, spectators, and Pd-L2Ork all connect to the projector.

2.2 Software Component Interaction

To transfer information between the major portions of the architecture, data structures from the aforementioned software were used. The Kinect gathers the information from the sensors in point clouds, and from these point clouds, the necessary points for the head, shoulders, arms, torso, eyes, and mouth are retrieved (3, 5). These points are then packaged in a certain format and their information is transferred via UDP packets. Consider this example of the information in proper format: "torso 0.1 0.3 -1.3 2.5;\n" which uses constraints detailed in Table 1.1. These packets are transferred through Pd-L2Ork from the Kinect to OPERAcraft. Upon receiving this information, the Minecraft code parses it and uses appropriate methods to make the avatar perform the respective mirrored movements.

Facial recognition is handled slightly differently than the skeletal recognition. While it follows the same packet formatting, OPERAcraft uses different avatar helmets for different facial expressions. These expressions will allow for four different mouth formations and four different eyebrow formations.

The four eyebrow and mouth formations were converted into a four by four matrix which represents all the possible eyebrow/mouth combinations. Twenty helmets for the avatar were designed based on this matrix, shown in Figure 2.2.



0, 0	0, 1	0, 2	0, 3
1, 0	1, 1	1, 2	1, 3
2, 0	2, 1	2, 2	2, 3
3, 0	3, 1	3, 2	3, 3
4, 0	4, 1	4, 2	4, 3

Figure 2.2: Eyebrow and Mouth Combination Matrix. This matrix displays the 20 possibilities for avatar helmets within Minecraft, based on the user's facial expressions. Eyebrow and mouth expressions are represented by column and row values, respectively.

2.3 Temporary Data Structure

The only temporary data created for this project are the UDP packets transferring information gathered from the Kinect directly to the middleware and then to Minecraft. As previously mentioned, the Kinect creates point clouds at 30 frames per second, so information is gathered quickly and often (3, 5). The UDP packets are created and sent through Pd-L2Ork to Minecraft from the moment OPERAcraft is started, all while new packets are being created, sent, and received until the system is shut down.

3.0 Implementation

The main components implemented were facial and skeletal tracking with Kinect sensors, processing and transforming the Kinect data into a usable format, transferring Kinect data (C# code) to the Minecraft code (Java) via Pd-L2Ork, editing the Minecraft code to allow for mirrored avatar movements based on the processed Kinect data, and creating the imagery needed to simulate the Minecraft environment and avatars.

This section will describe the nontechnical and technical project components, their implementation strategies, and the overall processes followed.

3.1 Interactive Components

The project had a variety of non-technical and technical interactive requirements to fulfill. Due to the unique nature of the project, and the fact it will be used in a variety of settings and by a plethora of users from different backgrounds and ages, the team had to be especially critical of the requirements to ensure maximum usability among users.

3.1.1 Nonfunctional Requirements

OPERAcraft, the foundational Java code utilized for the project, was originally designed to engage children in computing and musical concepts. Since *Cinema*craft will possibly be adopted back into OPERAcraft, the team had to make note of this particular young user-base. Furthermore, since the project will be displayed at conferences and events open to the public, the variety of users that might interact with the system in these particular venues had to be taken into consideration.

The following are the nonfunctional requirements determined for the diverse user-base:

1. As a user, I want to feel like I am a part of the system.
    a. The Minecraft avatar must portray the user as accurately as possible.
    b. The Minecraft avatar needs to be as customizable as possible (e.g. gender)

2. As a user, I want to feel engaged and creative using the product.
    a. The product should allow for maximum creativity and movement on the part of the user.
    b. In whichever odd angles and directions the user can imagine moving, the Minecraft avatar must be able to replicate (within system limitations).

3. As a user, I want to feel safe using the product.
    a. The space in which participants can move (in front of the Kinect sensors) must be marked off and safe.

4. As an administrator, I want to feel at ease having the product be used in a public setting, and ensure that no one can be harmed during use.

   a. We must properly mark the space in which users can occupy and instruct them accordingly.
   b. We must keep the area clear of obstacles (e.g., trash, furniture).

The demonstration setup was constructed with all of these requirements in mind. A sketch of the desired layout is shown in Figure 3.1.1. Components of this layout include a main table for the display monitors, moveable tables or shelves for each Kinect sensor (facial tracking is more accurate when the Kinect is closer, while skeletal tracking requires the Kinect to have more room), and a marked off area on the floor to designate the project's safe space.

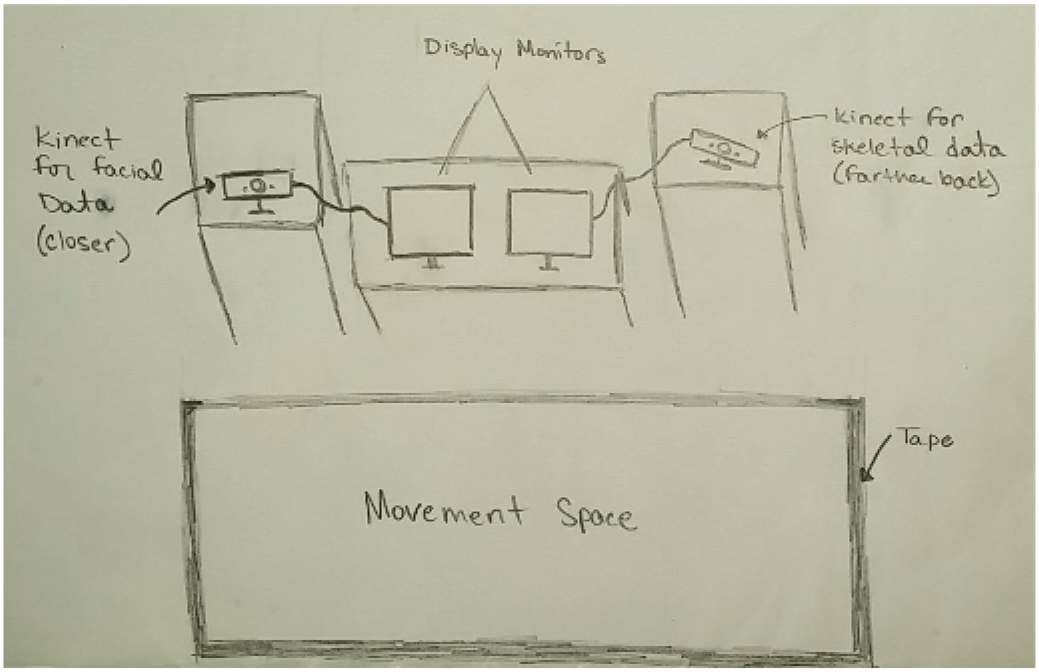

Display Monitors
Kinect for facial Data (closer)
kinect for skeletal data (farther back)
Tape
Movement Space

Figure 3.1.1: Interactive setup layout. The safe area is marked off with tape on the floor, the Kinect sensors each have their own moveable tables (or shelves), and the monitors are displayed on a main table in the center.

3.1.2 Functional Requirements

The main technical component of the system that users will interact with is the Minecraft display. The requirements of the external interface are:

1. The avatar will mirror the user's movements.
   a. The avatar's arms, legs, torso, head, mouth, and eyebrows will move as the user moves these respective parts of the body.
   b. The movements will be mirrored, not replicated, i.e. the avatar's left arm moves when the user's right arm moves.
   c. The avatar walks or jumps when the user walks or jumps, respectively. This is done by teleporting the avatar to the desired coordinate location.

2. The avatar will not unintentionally switch between users.

   a. The avatar will only track the facial and skeletal movements of the closest person in the safe space. This avoids displaying other movements.

3. The avatar will not mirror blocked body parts.
   a. If a previously visible portion of the user's body becomes hidden (i.e., blocked from or outside of the Kinect sensor's view), the avatar will maintain the last accurate values instead of mirroring potentially erroneous values.

3.2 Project Features

In order to connect the live Kinect facial and skeletal data to the Minecraft avatar, each portion of the project had to support the needed features. The features that are integral to the final product and intermediate demonstrations (i.e., SXSW, ICAT Day) are considered required features. These include the following, in descending order of importance:

1. Kinect sensors accurately tracks the desired facial points and skeletal joints.
   a. Facial points include the eyebrows and mouth (displayed in Figure 1.3.1A).
   b. Skeletal joints include wrists, elbows, shoulders, torso, hips, knees, and ankles (displayed in Figure 1.3.1B).

2. The facial and skeletal data is processed appropriately and the user's body movements are outputted in a compatible format.
   a. The format needs to match the Java code format used for Minecraft gameplay with respect to OPERAcraft constraints (detailed in Table 1.1).
   b. The data needs to be sent via UDP.

3. The Minecraft avatar mirrors the user's movements.
   a. Including arm, leg, head, and torso movements.
   b. The avatar teleports (appears to move to the desired coordinate location) as the user moves forward, backwards, or side to side.

4. The Minecraft avatar mirrors the user's facial expressions.
   a. Including mouth (e.g., smiling, frowning, open lips) and eyebrow movements (e.g., furrowed, anxious, elevated, or surprised eyebrows). The different combinations of these facial expressions are shown in Figure 2.2.

5. Developing a specific Minecraft avatar.
   a. Create the art for the facial expressions displayed in Figure 2.2.
   b. Create the art for the avatar's body (known as a skin).

In addition to these necessary features, there are also several features that were desirable, but not integral to the final product. Outlined below are the ideal features that are not required for the project to function, but would enhance future functionality if pursued.

1. Incorporating gender recognition by having female and male avatars for female and male users, respectively.
   a. Design female and male skins (avatar bodies) and faces.

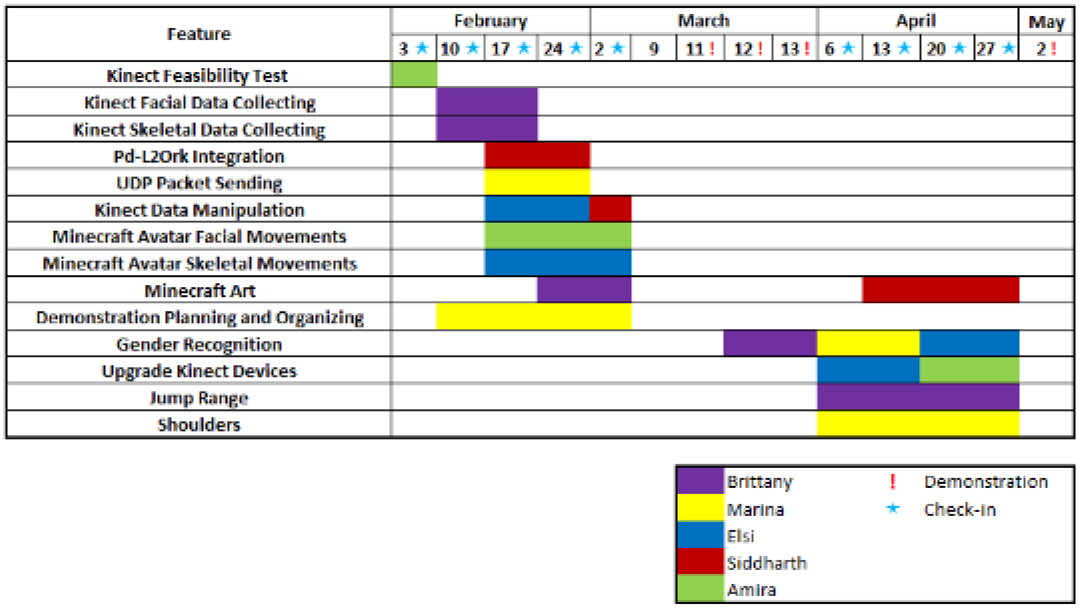
   b. Differentiate between males and females in the Kinect program or implement a switch within Pd-L2Ork.

2. Allowing multiple users to be mirrored at once.
   a. Program the Kinect code to track multiple faces and skeletons at once.
   b. Program the Minecraft code to project multiple avatars in the world simultaneously.

3. Designing a specific Minecraft world for demonstrations.
   a. Build all of the scenery within Minecraft.

3.2.1 Development Timeline

The development timeline for each feature is shown via a Gantt chart in Figure 3.2.1. Notable deadlines include the South by Southwest Expo (SXSW) from March 12 to 13, and ICAT Day on May 2.



Feature	February				March					April				May
	3 ★	10 ★	17 ★	24 ★	2 ★	9	11 !	12 !	13 !	6 ★	13 ★	20 ★	27 ★	2 !
Kinect Feasibility Test	🟩													
Kinect Facial Data Collecting		🟪	🟪											
Kinect Skeletal Data Collecting		🟪	🟪											
Pd-L2Ork Integration			🟥	🟥										
UDP Packet Sending			🟨	🟨										
Kinect Data Manipulation			🟦	🟦	🟥									
Minecraft Avatar Facial Movements			🟩	🟩	🟩									
Minecraft Avatar Skeletal Movements			🟦	🟦	🟦									
Minecraft Art				🟪	🟪						🟥	🟥	🟥	
Demonstration Planning and Organizing		🟨	🟨	🟨	🟨									
Gender Recognition								🟪	🟪	🟨	🟨	🟦	🟦	
Upgrade Kinect Devices										🟦	🟦	🟩	🟩	
Jump Range										🟪	🟪	🟪	🟪	
Shoulders										🟨	🟨	🟨	🟨	

🟪	Brittany	!	Demonstration
🟨	Marina	★	Check-in
🟦	Elsi		
🟥	Siddharth		
🟩	Amira		

Figure 3.2.1: Gantt Chart for Feature Implementation. The chart depicts the general timeline of implementation for this project. As seen above, some of the biggest challenges were manipulating the Kinect data and making it Minecraft-friendly. This involved computing the correct angles between ligaments and the body. The difficulty came when translating these angles from a certain dimensional space in order to make it match the coordinate system in Minecraft.

3.3 Technical Implementation

The technical implementation for this project was done in three main environments: Visual Basic for the C# code (Kinect data), Eclipse for the Java code (Minecraft), and Pd-L2Ork for the data transfer (UDP). The Kinect code implements the movement recognition and mathematical translation of movement into angles. Pd-L2Ork acts as the middleware by accepting data from the Kinect and making minor corrections to data errors (if any) before sending to the Minecraft code. The Minecraft code parses the messages retrieved from the Kinect, and sends the information to the respective parts of the code which handle player movement.

3.3.1 Minecraft

The existing OPERAcraft program was a Minecraft mod that allowed for avatar manipulation through a player's keyboard. The user could move the avatar left, right, forwards, and backward (using keys 'A', 'D', 'S', and 'W', respectively). The avatar had six possible arm positions (e.g., raised, lowered) which could also be manipulated by the player's keyboard (using keys 'U', 'J', 'K', 'L', 'I', 'O'). The user was able to control the speed of the movements by holding down the desired keys for shorter or longer periods of time.

The change that was to be implemented in the Minecraft Java code was to replace the keyboard control with Kinect sensory data control, allowing the user to manipulate the avatar with their own body and facial movements.

Development was done in six existing classes in order to complete the requirements. All of these classes are located in a folder titled 'net.minecraft.src'. A list of the altered classes includes:

- OpenSocketThread.java
- GuiNewChat.java
- ModelBiped.java
- GuiInGame.java, and
- OperaCraftPlayer.java
- Vars.java

The flow of information in these classes is as follows: OpenSocketThread.java accepts a connection and interprets messages coming in from the port once it is opened. Then the GuiNewChat.java calls existing methods to change the state of the player. For some calls, this class was altogether bypassed, and instead messages were sent directly to ModelBiped.java. The ModelBiped.java and OperaCraftPlayer.java are where the implementation of the arm rotations and leg rotations were made, and the GuiInGame.java is where avatar positions were controlled. Figure 3.3.1A depicts how the classes interact with one another, and Figure 3.3.1B provides a visual representation of this flow, which were used as a guide throughout implementation.

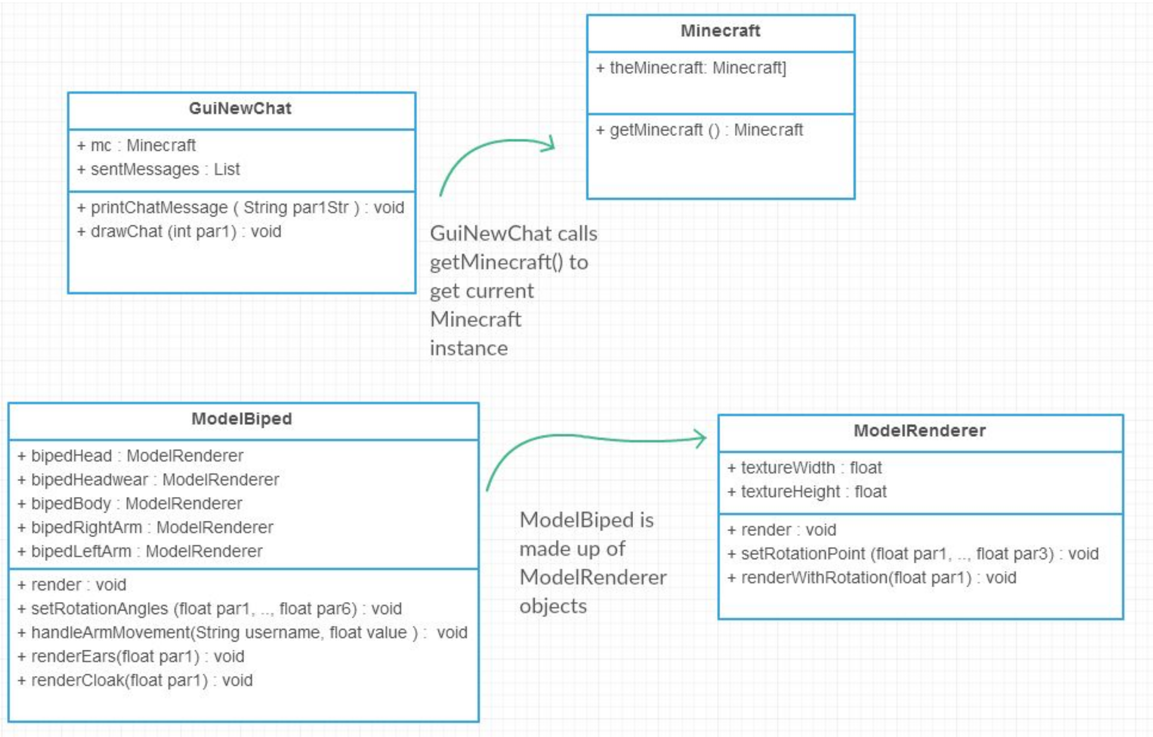

Minecraft
+ theMinecraft: Minecraft]
+ getMinecraft () : Minecraft
GuiNewChat
+ mc : Minecraft
+ sentMessages : List
+ printChatMessage (String par1Str) : void
+ drawChat (int par1) : void
GuiNewChat calls getMinecraft() to get current Minecraft instance
ModelBiped
+ bipedHead : ModelRenderer
+ bipedHeadwear : ModelRenderer
+ bipedBody : ModelRenderer
+ bipedRightArm : ModelRenderer
+ bipedLeftArm : ModelRenderer
+ render : void
+ setRotationAngles (float par1, .., float par6) : void
+ handleArmMovement(String username, float value) : void
+ renderEars(float par1) : void
+ renderCloak(float par1) : void
ModelBiped is made up of ModelRenderer objects
ModelRenderer
+ textureWidth : float
+ textureHeight : float
+ render : void
+ setRotationPoint (float par1, .., float par3) : void
+ renderWithRotation(float par1) : void

Figure 3.3.1A: Class Structures. Above represents the Class Structures for the Java OPERAcraft code component. OPERAcraft sends commands to the Minecraft avatars using the built-in chat client (handled by GuiNewChat). Also relevant are the ModelBiped and ModelRenderer classes, which convert commands and points into actual movement and appearance in Minecraft. The classes are slightly simplified to avoid listing the redundant fields and methods, but the main components are listed.

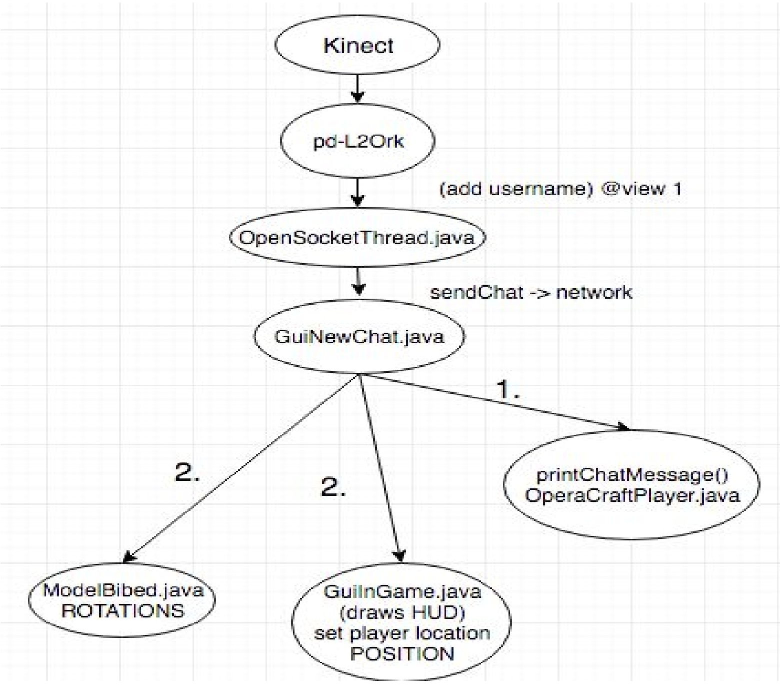


Figure 3.3.1B Project Information Dependencies. This is the initial flow chart of the information flow from the Kinect to Pd-L2Ork to the different Java classes of the Minecraft code. Emphasis is on the class dependencies in Java.

3.3.1.1 OpenSocketThread.java

This class is where Minecraft connects with Pd-L2Ork through a port (7000), and continuously retrieves messages in a specific format (i.e., "@karm username xLeftRoation yLeftRotation zLeftRotation xRightRotation yRightRotation zRightRotation "). An interpretMessage() method checks what the message is directed toward (i.e., @karm = kinect arm movement) and then it sends this parsed message to the Minecraft player using the sendChatMessage() method.

A command was created to handle the arm (@karm), torso (@tosro), legs (@legs), head (@head), eyebrow (@eyes), and mouth (@mouth) movements. There was an existing keyboard arm (@arm) command which was not altered.

3.3.1.2 GuiNewChat.java

The purpose of this class is to relay messages to the ModelBiped (avatar). It receives chat messages and parses them to call the appropriate method (e.g. changeOperaCraftArmState()) to manipulate the ModelBiped.

3.3.1.3 ModelBiped.java

This class is what constructs the Minecraft avatar. It is made up of ModelRenderer objects such as bipedHead, bipedHeadwear, bipedBody, bipedRightArm, etc. This class was essential in handling arm and leg rotations. Fortunately, there were existing hooks for X, Y, and Z rotations for each arm and each leg, which could handle values coming in from the Kinect. Additionally, this is where methods were called to handle body movements, including HandleArmMovements, HandleLegMovements, etc.

3.3.1.4 GuiInGame.java

This was the class that was modified to handle teleportation, based on the player's location in front of the Kinect. The changing position of the player and the redrawing of the environment was handled here. This was done by resetting the x, y, z, rotation yaw, and rotation pitch locations of the camera/projector position to manipulate the feed and create the illusion that the avatar is actually moving within the Minecraft world.

3.3.1.5 OperaCraftPlayer.java

This is the class that was modified in order to set the appropriate x, y, z rotations of the player's arms, legs, torso, head, and headwear to reflect the data tracked by the Kinect sensor.

3.3.2 Kinect

As this was the first interaction OPERAcraft had with Kinect, the Kinect program was built from the ground up, utilizing the Kinect for Windows SDK. Since two Kinects were used for the project, two programs were developed to control and monitor the skeletal and facial tracking programs separately.

3.3.2.1 Skeletal Tracking

The Kinect SDK was utilized to enable skeleton/body tracking. The skeletal tracking program collects the three dimensional locations of joints and landmark nodes on the body. An example of the skeletal tracking program in use can be seen in Figure 3.3.2.1.

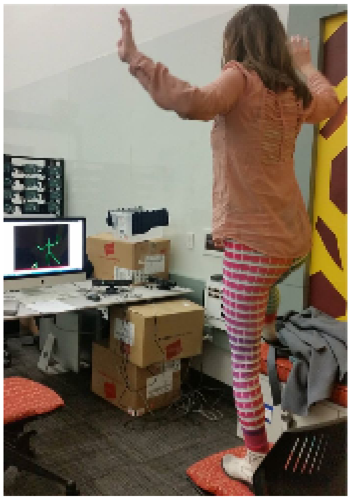


Figure 3.3.2.1: Live Skeletal Tracking. Above, a developer stands in front of the Kinect device and views her skeletal movements on the screen (the large green figure on screen). Note the smaller green figure also on the screen, these false positives occur when the Kinect picks up false data or nearby people. This was avoided by only tracking one person (the closest) at a time.

During implementation, specific useful joints were determined in order to convert the users actual movements into Minecraft avatar movements. These joints include the left and right wrists; left, right, and center shoulders; torso (spine); left, right, and center hips; and left and right ankles (depicted in Figure 1.3.1A). After storing this data, during each frame (each time the Kinect sends in a different feed view), the angles that each ligament had with respect to other parts of the body were computed.

Angles of movement were computed for the arms, legs, and neck, along the three planes of rotation (X, Y, and Z). First, the X, Y, and Z coordinates of each joint were stored and were used to create three dimensional vectors, using the existing Vector3D class. A series of angle formulas were then used to compute the angles between the different joints along each axis of rotation. For example, consider arms moving up and down along the Y axis, sideways along the X, and forward and backward along the Z. Further conversion was necessary to map these rotations to the existing rotation system in OPERAcraft.

Once these angles were determined, they were turned into commands to be sent via UDP through Pd-L2Ork (where any needed scaling occurs) to OPERAcraft where the avatar will move based on the calculated angles of rotation.

3.3.2.2 Facial Tracking

To obtain the facial data, the Microsoft Face Tracking SDK for Kinect for Windows (Face Tracking SDK) was utilized. The Face Tracking SDK tracks 2D points on the face, using three points for each eyebrow and four points for the mouth, as shown in Figure 1.3.1B.

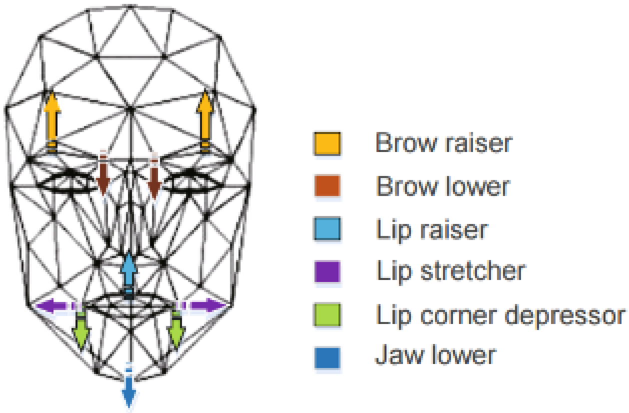


Figure 3.3.2.2: Visualization of tracked AUs. This shows the areas of 2D points that would affect various AUs to yield the delta values from a face's neutral position (4).

In addition to the 2D points, the SDK tracks animation units (AUs) which are subsets of the Candide3 model. These AUs are changes from the neutral shape of the face and are used to determine what expression the user is making. Each AU is expressed as a numeric weight between -1 and +1. The AUs used in the project are: Neutral Face, Upper Lip Raiser, Jaw Lowerer, Lip Stretcher, Brow Lowerer, Lip Corner Depression, and Outer Brow Raiser. A visualization of these AUs can be seen in Figure 3.3.2.2.

In order to detect specific facial expressions using these AUs, thresholds were set for specific expression values. In this way, five different mouths were detected: closed lips, open lips, puckered lips, frowning, and smiling. Eyebrow shapes were computed in similar fashion and divided into four categories: flat (neutral), furrowed (angry), raised (surprised), and slanted (anxious). By combining the different combinations of eyebrows and mouths, 20 different "helmets" were designed for the Minecraft avatar to display the maximum amount of information. Helmet examples are depicted in Figure 3.4.1B, and the matrix of all facial combinations can be seen in Figure 2.2.

3.3.3 Pd-L2Ork

Pd-L2Ork was utilized as the middleware in the project. Packets were sent from Kinect via UDP to a certain port, where they were received through Pd-L2Ork. The Pd-L2Ork program averages five such data packets from each command, thereby finding a more fluid position of motion. The corrected data is then sent to OPERAcraft. Before utilizing this averaging method the Minecraft avatars would often appear jumpy. Through a generalization of a group of data points, the avatar was able to move with more natural movement.

3.4 Minecraft Art

The Minecraft world is completely customizable, including the avatars and the world the avatars interact in. Specific avatar bodies (Figure 3.4.1A), facial expressions (Figure 3.4.1B), and world (in which the avatar will be depicted in), were designed for this project.

3.4.1 Avatar

An avatar, as shown in Figure 3.4.1A, consists of an initial skin and facial expression. Twenty helmets were developed in order to map to the twenty detectable facial expressions. These were created following Minecraft avatar structural guidelines.

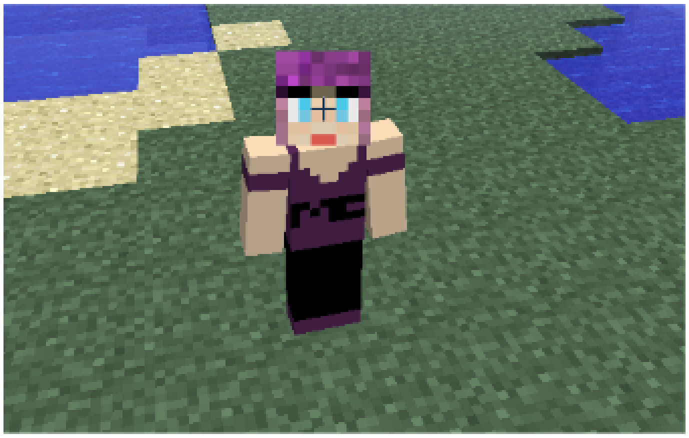


Figure 3.4.1A: Minecraft Avatar. An example of our avatar in one of the skins designed, wearing the initial neutral face.

The skin and helmets were created using Microsoft Paint. Each helmet is 16x16 pixels (stored in the mouths folder) and the skin for the body is 64x32 pixels (stored in the skins folder). The helmets are made up of six sides: back, top, bottom, left, right, and front (the face). Within OPERAcraft, the helmets are placed on the respective characters by using the naming convention “playerXsideY.png”, where X stands for the player number and Y is used only to differentiate between the facial expressions. For this reason, left, right, top, bottom, and back do not have a Y value. The skin follows the naming convention “username.png”. Examples of the front of these helmets can be seen in Figure 3.4.1B.

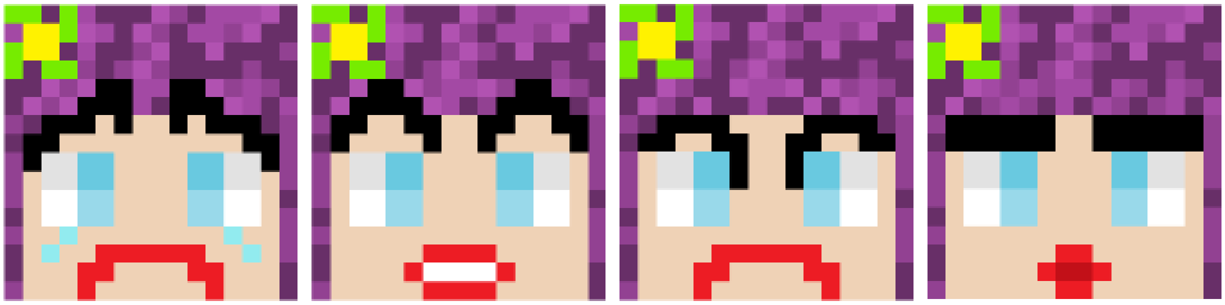


Figure 3.4.1B: Minecraft Facial Expression Helmets. These helmets were created to represent specific facial expressions detected by the Kinect. From left to right the helmets represent: sad (anxious eyebrows and a frown), surprised (surprised eyebrows and open lips), angry (furrowed eyebrows and a frown), and kiss face (neutral eyebrows and puckered lips).

3.4.2 World

A specific world was designed for the *Cinema*craft demonstrations. The world designed can be seen in Figure 4.2.

4.0 Prototype

The goal of the prototype was to combine the necessary software components with the respective hardware for data collection (of the user's facial and skeletal movements) and display (the mirrored movements of the user by a Minecraft avatar).

Below is a description of the software and hardware components utilized, the reasoning for their inclusion, and the overall functionality of the prototype.

4.1 Components

OPERAcraft utilizes both software and hardware components. The software components are what control the prototype's functionality, while the hardware components are used for data collection and presentation.

The various types of software and hardware utilized, and the reasons for their inclusion, are described below.

4.1.1 Software

The software components are what control the level of functionality for the prototype. They include:

- Pd-L2Ork

- OPERAcraft (Minecraft code)
    - Language: Java
    - Environment: Eclipse Luna/Mars (64 or 32 bit)

- Kinect Skeletal/Facial Programs
    - Language: C#
    - Environment: Visual Basic (64 or 32 bit)

Pd-L2Ork was chosen as middleware due to its portability. It allowed for rapid changes to the prototype without having to recompile (which can be costly at demonstrations) or search through the code to begin troubleshooting.

The system was built off of OPERAcraft, which was already developed in Java; therefore Java was used to develop the system further. Finally, the Kinect code was completed in C# in order to work best with the readily available Kinect SDK/API code, also available in C#.

The IDEs chosen to develop the project were not vital and were chosen due to preference. Additionally, the programming languages for the project could be altered, as long as the functionality is maintainable and the new language is compatible with the hardware being used to run the Kinect and/or Minecraft code.

4.1.2 Hardware

The hardware used for this prototype consists of the following items:

- Monitors/Computers (3)
    - Two monitors/computers were needed to connect the Kinect sensors in order to collect, process, and distribute user movement data.
    - One additional monitor is used for displaying the user's movements within Minecraft.
- Kinect Sensors (2)
    - Two sensors are needed for tracking skeletal and facial movements, respectively.
- Kinect Fisheye Adapter (1-2)
    - Used with the Kinect sensor tracking skeletal movements to alter movement area.
    - Sometimes paired with the Kinect facial tracking sensor when they are required to be close together to minimize them affecting each other's data. Does slightly skew the accuracy of the facial recognition when this is done.

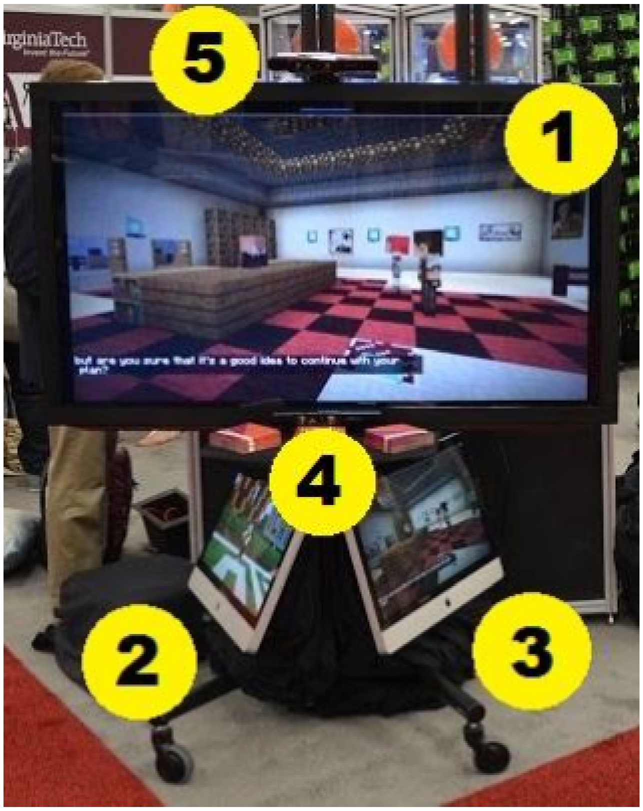

5
1
but are you sure that it's a good idea to continue with your plan?
4
2
3

Figure 4.1.2: Prototype Hardware Setup. As labeled, the prototype includes the primary display monitor (1), the computers for collecting data (2, 3), and the Kinect devices (4, 5). This is the SXSW 2016 display.

Three computers were used because multiple Kinects cannot share USB ports on the same hub, and the computers used for the prototype have all the USB connections on the same hub. Therefore the prototype requires two monitors to run, with an optional third monitor used solely as a larger display at demonstrations. One computer (the display monitor/computer) runs the Minecraft code, while the other two computers run skeletal and facial tracking programs separately. The Kinect data is sent through Pd-L2Ork to the Minecraft instance running on the primary computer, which is what is displayed to the users on the primary monitor. This setup can be seen in Figure 4.1.2. Without the optional third monitor, one of the Kinect-running computers also runs the Minecraft program.

The facial tracking Kinect is placed closer to the user than the skeletal tracker, to increase the accuracy of the facial data. The proximity was especially significant since the facial movements, which vary by only a few degrees, quickly become unrecognizable if the user is too far from the Kinect device. Conversely, the skeletal tracking Kinect was placed further away from the user in order to read in a wider range of positions. Allowing the user to move within a larger area was key since the prototype needed to be accessible to a wide audience of varying heights and mobility.

Further widening of the user's movement area was achieved through adding a fisheye camera attachment to the Kinect device tracking skeletal movement. This adaptation helped to avoid joint detection loss, such as when a taller individual would not fit into the device's camera frame and would begin to produce erroneous data for that untracked joint.

4.2 Functionality

A few key aspects still need to be refined, but the prototype has full functionality for the majority of the project's integral features.

These integral features are listed in Table 4.2, with their respective levels of functionality, as well as an indication of whether or not the functionality could be expanded. If the functionality of a feature is considered complete but could be expanded, then there is an additional feature that might be added once the base functionality of all features has been achieved.

Integral Feature	Base Level of Functionality	Does it need to be expanded?	Could it be expanded?
Skeletal: arm movement	complete	no	no
Skeletal: leg movement	complete	no	no
Skeletal: torso movement	complete	no	no
Skeletal: head movement	complete	no	yes
Facial: eyebrow movement	complete	no	yes
Facial: mouth movement	complete	no	yes
All movements are mirrored	complete	no	no
Minecraft: avatar art	complete	no	yes
Minecraft: world art	complete	no	yes

Table 4.2: Integral Feature Functionality Levels of the Prototype. This table details the various levels of functionality for the integral features of the prototype. A feature can be complete and still be expanded upon, if there are additional features that would like to be achieved once the base level of all features has been achieved.

Since the prototype was fully functional in all aspects except for facial expressions (not all faces detailed in Figure 2.2 can be detected yet), it was made available for public use during the South by Southwest Conference in Austin, TX in March 2016. In Figure 4.2, the prototype can be seen exhibiting the level of functionality indicated in Table 4.2 as Virginia Tech's president, Timothy Sands, has his skeletal and facial movements mirrored by the Minecraft avatar.

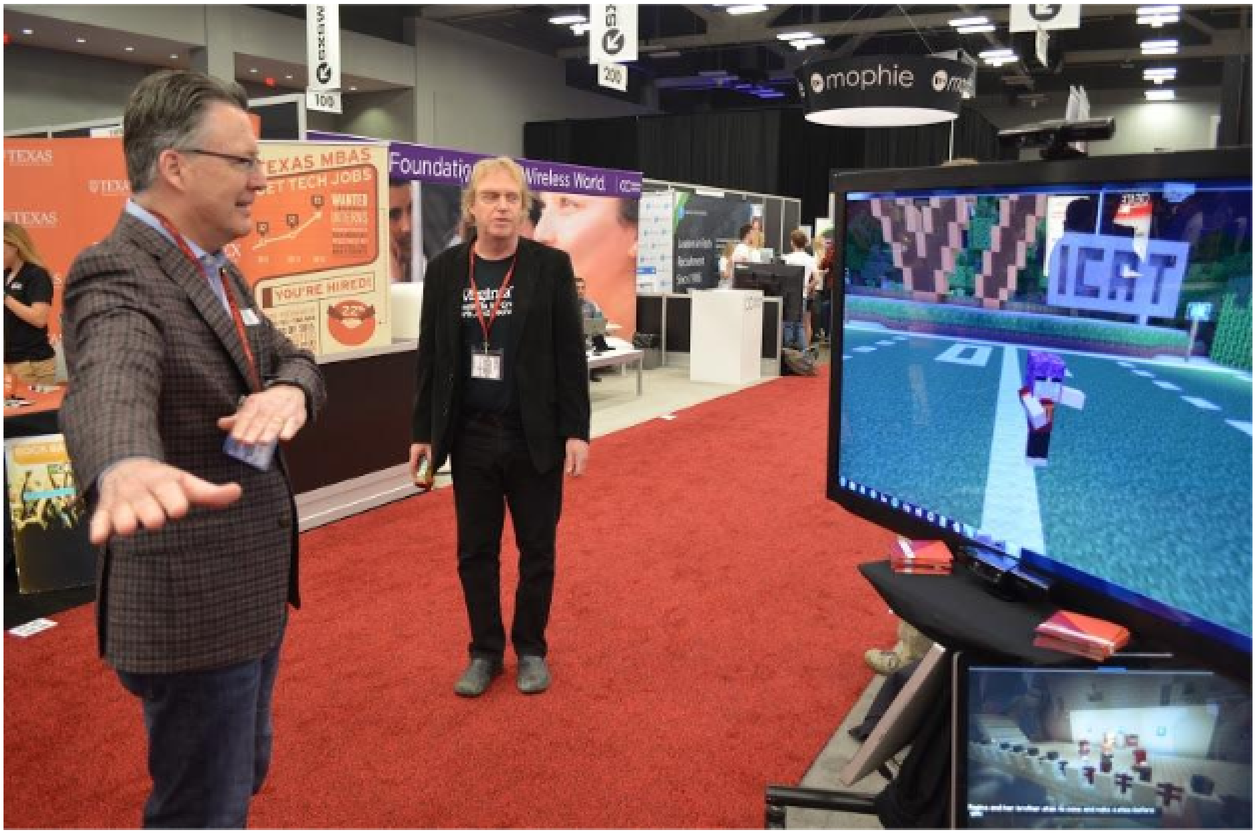


Figure 4.2: Prototype Functionality. Virginia Tech President, Timothy Sands, is shown using the prototype at SXSW 2016. The avatar is shown mirroring Sands' arm movements (i.e. Sands' right arm is straight out with his left arm across his chest, and the avatar's left arm is straight out with its right arm across her chest), as well as his facial movements (i.e. the avatar is smiling).

5.0 Debugging

After a change is made in the Kinect code the entire system should be tested to ensure it maintains the desired functionality. For a better understanding of how to test the system refer to section 7.0 Testing.

In the case that a malfunction or undesired effect occurs, Table 5.1 suggests the appropriate Java classes to reference; however, it is not guaranteed this is where the problem will be or where the fix should be implemented. But, assuming the Kinect code is correct and the Minecraft code needs to be updated to support the change, the portion of the code that most likely should be updated is listed in Table 5.1.

What is not functioning correctly?	**Handled in Java class:**
Messages/UDP packet reception	OpenSocketThread
Message parsing	GuiNewChat
Avatar rendering	ModelBiped
x, y, and z rotations of arms, legs, torso, and head	OperaCraftPlayer

Table 5.1: Debugging Functionality. If a change is made in the Kinect code causing one of the malfunctions or undesired effects listed above, then the respective class that most likely should be updated is listed.

If an update to any portion of the software or hardware components is available and will be integrated into the system, then all of the code will need to be properly updated. Typically, if there is an update with Kinect code, this will not affect the Minecraft code, and vice versa.

6.0 Future Work

The core functionality of *Cinema*craft is present within the prototype, however there are still potential improvements that could be implemented in the future; these are detailed in Figure 4.2.

6.1 Future Adaptations

Each adaptation is ranked using a priority system that ranges from high, to medium, to low. A high priority adaptation *will* be completed within the scope of the project, a medium adaptation *could* be completed within the scope of the project, and a low priority adaptation could be completed *outside* the scope of the project. Possible adaptations to the current prototype are described in detail below. Figure 3.2.1 provides a timeline for in-progress adaptations.

6.1.1 List of Changes

1. Adaptation: limit server (an offscreen avatar) distance from the actors (on screen avatars)
   Priority: high. Status: complete.

   At demonstrations the server needs to be further away from the actors in order to remain unseen. On the other hand, during a production, the server can be closer to the actors. It became apparent during the first SXSW demonstration that when the server is further away from the actors, the movement commands are not being sent or executed properly.

   The cause for this was the maximum distance to maintain full rendering clarity had been exceeded. Testing and analysis was performed to establish a threshold limit for the distance the server should be away from the actors.

2. Adaptation: upgrading current Kinect sensors to Kinect HD 2.0 sensors
   Priority: high. Status: in progress.

   The current version of the Kinect sensors require frequent use of fisheye lens attachments to increase scope and minimize interference. Upgrading to Kinect HD 2.0 sensors will allow for higher clarity (more accurate facial recognition especially) and a wider scope (necessary for full skeletal tracking).

   All of the facial tracking code has been successfully upgraded; however, the skeletal tracking code still needs to be upgraded.

3. Adaptation: facial movements and recognition expanded
   Priority: high. Status: complete.

   The base level functionality for facial recognition is detecting each of the twenty eyebrow and mouth combinations described in Figure 2.2. Currently, the prototype is only able to recognize neutral, smiling, and open mouth positions, as well as neutral and raised eyebrows; this only allows for six of the desired twenty possibilities.

Adaptation (2) was completed for facial tracking and this followed since the accuracy of recognition was increased.

4. Adaptation: shoulder movement for shrugging
   Priority: medium. Status: complete.

   Shoulder movement would be a subset of the arm movements. The current implementation for arm movements is considered complete, but only allows for the avatar to mirror the user's arm movement from the shoulder down.

   Movement range was expanded to also include vertical shoulder movements (y axis), in order to simulate a shrug.

5. Adaptation: avatar jump range expanded
   Priority: medium. Status: complete.

   With the current implementation, the avatar is able to mirror a user's jump. However, on screen the jump height, as seen in Figure 6.1.1, is not as high as desired. Therefore, the coordinate values for the avatar to jump to should be scaled to create a larger jump range.

   Jump values were scaled immediately after Kinect detection by determining if a jump occurred, then multiplying those coordinates by a predetermined factor.

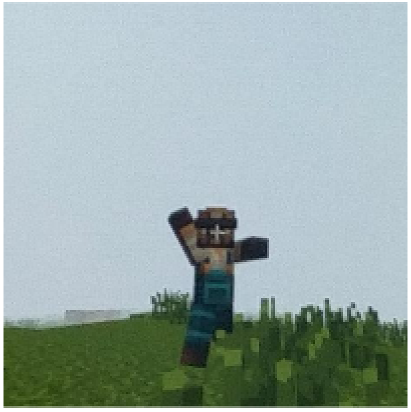


Figure 6.1.1: Avatar Jump Range. This is an example of how high the avatar can currently jump when mirroring a user's jump. Notice that the avatar is not jumping higher than the weeds, suggesting a need to scale the jump values in order to produce a larger jump range.

6. Adaptation: gender recognition
   Priority: low. Status: to be completed in future iterations.

   Gender recognition would need to be achieved in the Kinect code, most likely during facial detection. For the Minecraft code, instead of altering it to handle multiple avatar skins for the same player, there could constantly be two avatars (one female and one male) in the game, but only one on the screen at a time. As such, if the user is female, the female avatar will be on screen, then if the current user switches out with a male user, the camera would focus on the previously offscreen male avatar (who becomes the new actor) and the female avatar is now offscreen. This would only require adding another player to the game, and switching camera angles if a different gender is detected.

7. Adaptation: head movements expanded
   Priority: low. Status: to be completed in future iterations.

   The current implementation for head movements is considered complete, but only mirrors the user's head tilt and yaw. It would be expanded to also include purely forward (towards the Kinect sensor) and backward (away from the Kinect sensor) head movements (solely on the z axis).

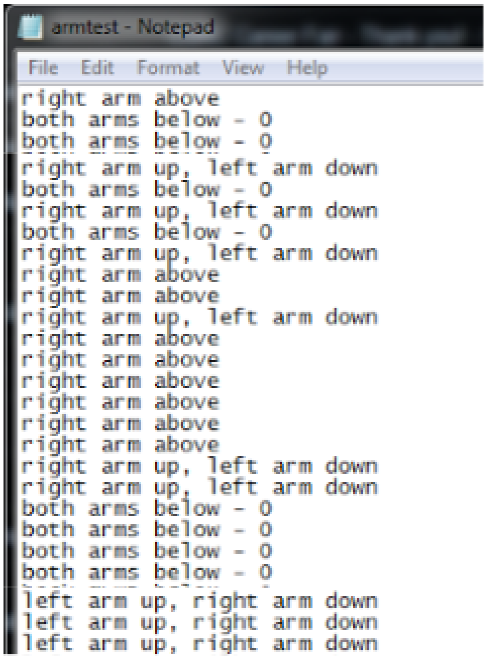
7.0 Testing

In order to ensure usability and reliability, the integration of facial and skeletal tracking into the existing OPERAcraft system was tested frequently throughout development. The usability tests consisted predominantly of the developers acting as users to the system and allowing the developers' movements to be tracked while writing coordinates or commands to the console; outputting commands to files; and/or visually analyzing the avatar's responses.

Security, stress, and load testing for the original OPERAcraft system was addressed in previous iterations of this project, so these criteria were not retested since the fundamental infrastructure remained unchanged.

For suggestions on what Java classes to check while debugging refer to section 5.0 Debugging.

7.1 Single-user Usability



```
right arm above
both arms below - 0
both arms below - 0
right arm up, left arm down
both arms below - 0
right arm up, left arm down
both arms below - 0
right arm up, left arm down
right arm above
right arm above
right arm up, left arm down
right arm above
right arm above
right arm above
right arm above
right arm above
right arm above
right arm up, left arm down
right arm up, left arm down
both arms below - 0
both arms below - 0
both arms below - 0
both arms below - 0
left arm up, right arm down
left arm up, right arm down
left arm up, right arm down
```

Figure 7.1A: Arm Test Gestures. This is an example of a typical file output used for testing gestures (specifically arms here) to ensure the code was functioning properly.

The main purpose of the system was to entertain and engage users in their representation in the Minecraft world. Therefore it was important to test how usable the entire system was for a single user, and how realistically the avatar could reflect that user.

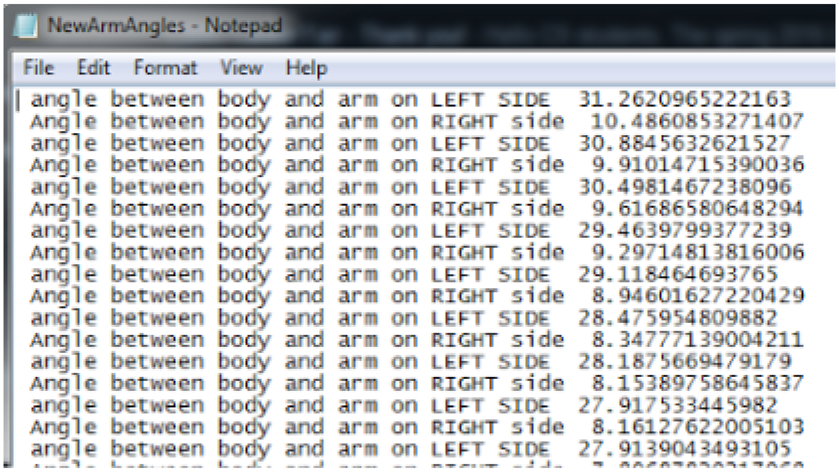
Testing occurred simultaneously with development. At each major change, the device was tested visually by running the system and mimicking user interaction. Developers would stand in front of the Kinect sensors which would track their movement. These movements were analyzed in two major ways

- outputting the joint or facial point coordinates, 3D vectors, and/or the detected position (e.g. x, y, or z coordinates), gestures (e.g., arms raised, jump), or facial expressions (e.g., angry smile, raised kiss), and

- visually analyzing the avatar's response to the user's movements

The console and file outputs were helpful when first developing the Kinect tracking code. It was used to determine the correct orientation of the X, Y, and Z coordinates and angles in both the Kinect and Minecraft coordinate spaces. The extremity coordinates and directions, needed to be reversed or mapped to another plane in order to achieve the desired mirroring and movement, were determined in this way. Figures 7.1A and 7.1B show examples of the types of file output worked with when testing arm gestures and angles (similar files were produced for other extremities).

Further, console output was initially used to test if the correct facial expressions were being recognized. Each of the twenty desired faces were made by a tester in front of the facial Kinect sensor (i.e. tester contorted face to achieve each expression), and checked that the output corresponded to each face correctly. For example, if the combination of raised eyebrows and frown were being tested, the console should output “raised frown”.



```
angle between body and arm on LEFT SIDE  31.2620965222163
Angle between body and arm on RIGHT side  10.4860853271407
angle between body and arm on LEFT SIDE  30.884563262152 7
Angle between body and arm on RIGHT side  9.91014715390036
angle between body and arm on LEFT SIDE  30.4981467238096
Angle between body and arm on RIGHT side  9.61686580648294
angle between body and arm on LEFT SIDE  29.4639799377239
Angle between body and arm on RIGHT side  9.29714813816006
angle between body and arm on LEFT SIDE  29.118464693765
Angle between body and arm on RIGHT side  8.94601627220429
angle between body and arm on LEFT SIDE  28.475954809882
Angle between body and arm on RIGHT side  8.34777139004211
angle between body and arm on LEFT SIDE  28.1875669479179
Angle between body and arm on RIGHT side  8.15389758645837
angle between body and arm on LEFT SIDE  27.917533445982
Angle between body and arm on RIGHT side  8.16127622005103
angle between body and arm on LEFT SIDE  27.9139043493105
```

Figure 7.1B: Arm Test Angles. This is an example of a typical file output used for testing angles (specifically for arms here) to ensure the code was functioning properly.

Once accurate console and file output was achieved for a particular feature, the team began visually testing the feature by analyzing the avatar’s movements (or lack thereof). This was done by opening an instance of the Minecraft game (username: ICAT01) and simulating body movements. Seeing the avatar on the screen was the most reliable way to test since it provides real-time, visual feedback. Testing in this manner shows exactly what a user would see and how the data was being interpreted. A demonstration of this testing setup is shown in Figure 7.1C.

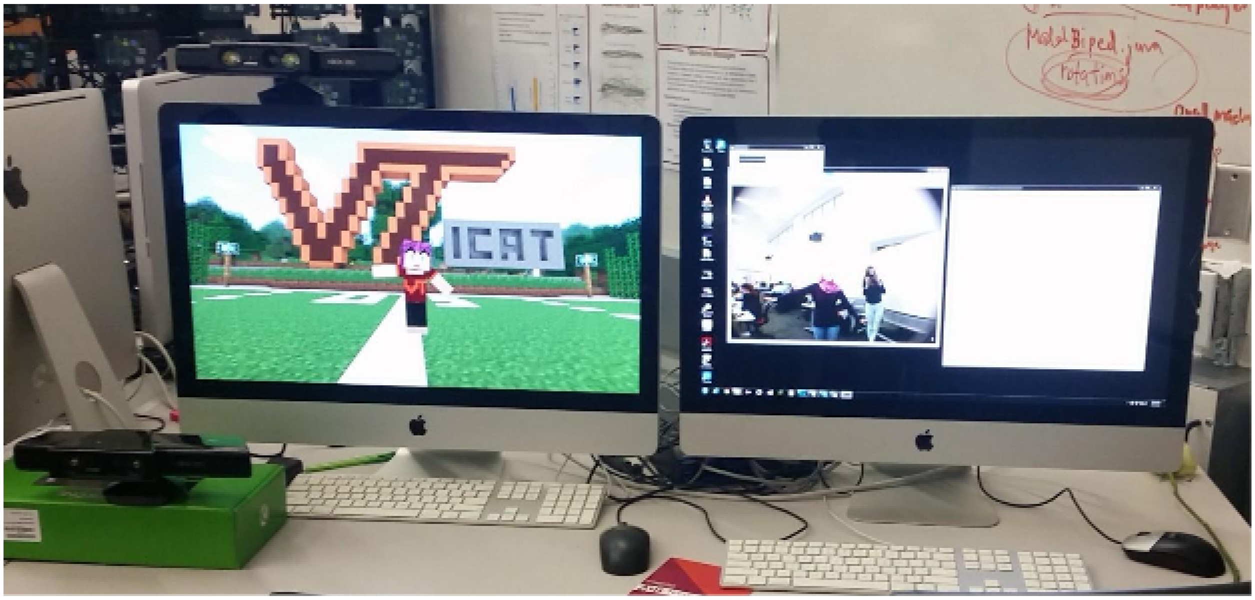


Figure 7.1C: Avatar Movement Testing. The most common form of testing was done by visually analyzing the avatar’s movements (or lack thereof) in response to the tracked movements. Above, a developer is shown (on the right) moving their arms to test how the avatar (on the left) is responding.

Additionally, reference 8 provides a video showing a member of the *Cinema*craft design team executing a visual test before the system's demonstration at SXSW. As can be seen in the video, these testing methods allowed for final verification of the integral features.

7.2 Multi-user (Server-Camera-Client) Usability

In addition to handling a single person using the system, the software should also be able to communicate with other users, which is a main part of the original OPERAcraft system. Therefore, it was important to test how the software functioned during these connections.

In order to test this, three instances of the Minecraft game need to be opened, one for the client (username: ICAT01), one for the camera (username: ICAT08), and one for the server (username: ICAT09). The client instance would need to start a game on the LAN network, and open a port for data retrieval using the OPERAcraft command "/openport". Once this is done, the other two instances can join the client's LAN instance game. This allows the camera and the server to view the client's avatar on their screen.

Multi-user capability was tested by

- making sure the movements being performed by the avatar representing the client were not causing the camera or the server's off-screen avatars (which are not visible to the main client) to move,
- validating that the client's avatar's movements are being rendered in full detail, and
- checking that manipulating the display screen camera angles of the camera and the server do not skew the client's display angles

Each of these were successfully validated during testing, meaning the system will work for multiple clients.

Acknowledgments

The completion of this project would not have been possible without the support and guidance of Dr. Ivica Ico Bukvic (from the School of Performing Arts, ICAT, Center for Human-Computer Interaction, Digital Interactive Sound and Intermedia Studio, and courtesy faculty of the Computer Science Department at Virginia Tech), Amira Youssef of the FEET project and the Electrical and Computer Engineering (ECE) Department, Siddharth Narayanan of the ECE Department, Dr. Lynn Abbott of the ECE Department, technicians at SXSW, Phyllis Newbill and the entire staff of ICAT, and the ICAT fellows who introduced us to the project and assisted with testing.

We would also like to extend a special thanks to Dr. Edward Fox and Yilong Jin for their feedback and assistance throughout the entire process.

References

1. Aldawud, Omar. 2011. Software Design Specification Template. Retrieved from http://www.cs.iit.edu/~oaldawud/CS487/project/software_design_specification.htm
2. ICAT. 2016. OPERAcraft. Retrieved from https://www.icat.vt.edu/funding/operacraft
3. Kim, Young Min. 2012. Kinect. Stanford. http://graphics.stanford.edu/courses/cs468-12-spring/LectureSlides/17_Kinect.pdf
4. Mao, Qi-rong, et al. Using Kinect for real-time emotion recognition via facial expressions. April 2015. Frontiers of Information Technology & Electronic Engineering. Vol. 16. Issue 4. pp 272-282. [Image] Retrieved February 2016.
5. McCormick, John. 2011. How Does the Kinect Work? Dickinson College Math/CS http://users.dickinson.edu/~jmac/selected-talks/kinect.pdf
6. Microsoft. 2016. Tracked Facial Points from Kinect Sensors. https://msdn.microsoft.com/en-us/library/jj130970.aspx
7. OPERAcraft. 2016. Pd-L2Ork Connection Example for OPERAcraft. [Image] Retrieved from http://disis.music.vt.edu/OPERAcraft/documentation.html
8. OPERAcraft testing. March 2016. Virginia Tech. [Video] Available at: https://drive.google.com/file/d/0B12jba-1Ut5jT2pPYTZoOExNWDg/view?usp=sharing