Applications in Virtual Reality

Team: Bharat Jangir, Kumar Chandan, Vedant Raiththa

Department of Computer Science
Rochester Institute of Technology

VIRTUAL STAGE

Term Project Report

Organization

I. Introduction

II. Implementation

III. Results

IV. Conclusions / Future Work

V. References

VI. Bibliography

Abstract

The goal of this project is to create a virtual environment where the model/main-character (also called Avatar in Unity jargon) is controlled by Kinect; and update the Avatar's viewpoint as the human in front of the Kinect moves/interacts in physical space. All the clients (audience models) linked / connected to this Virtual space see the position/orientation and viewpoint changes of the Avatar immediately in real time.

We mainly used two different platforms - Unity 3D Pro and Windows Kinect SDK to realize this goal. There were other tools and libraries which we encountered on the way like Kinect SDK, Zigfu, Openni tools, etc. that helped gain insight about how each of these individual parts exist. We ended up using few of them in our final implementation while examining and learning from others. For the most part we met most of the items we initially proposed in our project. But as with any software development project, things can be further analyzed, documented and tested.

We started the project development and figured that it was a good idea to decouple the work related to Kinect and Unity into their independent modules. This simplified a lot of things for us down the road. It allowed for easier integration, convenient collaboration among the team members and a design that is closed for modification but open for extension. There were several checkpoints in the course where students were supposed to demo their work and provide a current status on the project. The decoupling made it easier to work on the two modules independently and without any hassle and dreaded code conflicts and merges. Finally the two parts were coupled back to complete the project. The Kinect module was responsible for tracking the human motion and sending the tracked movements and poses to Unity and updating the Unity Avatar accordingly and Vedant was supposed to create a virtual environment in Unity 3D Pro.

The goal was achieved though laced with many hurdles and surprises along the way. This paper talks in detail about the whole voyage in accomplishing this target. Different sections in the paper talk about the different phases involved in our project development. You can skip to the section of interest as they are exclusive to each other.

Index Terms

- Microsoft Kinect, Unity 3D, UDP Communication, Kinect Plugin, Zigfu, Networking, Virtual Stage, Windows SDK, ViewPoint Modification, Avatars, Virtual Space

System Overview

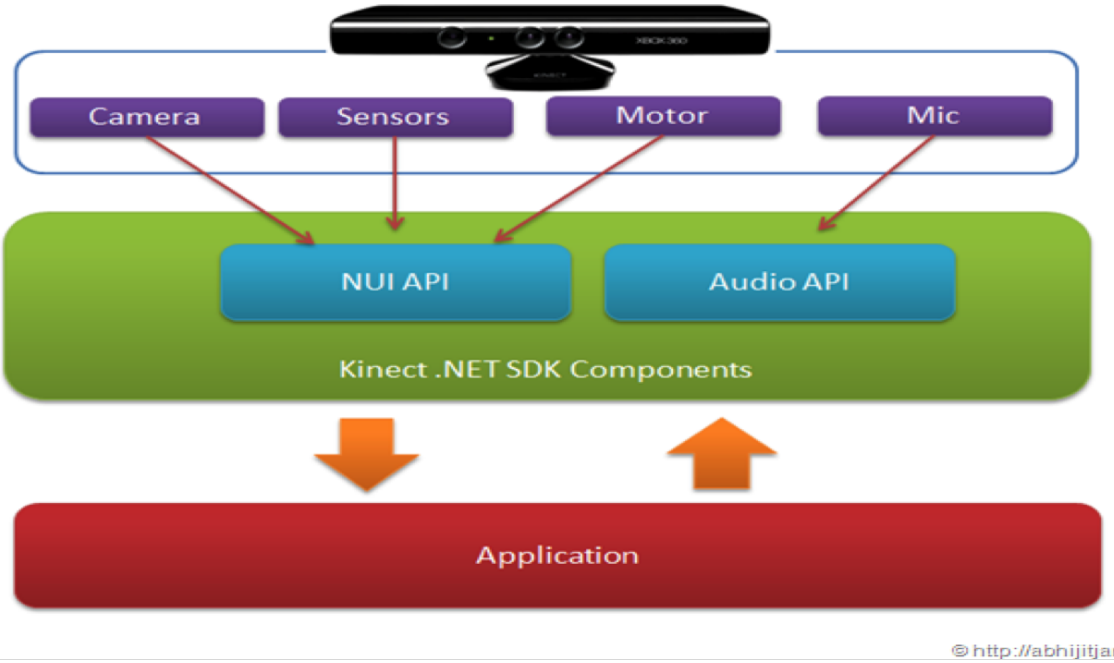


Image from: http://abhijitjana.net

This diagram above gives us the overview of the system. Kinect with its multiple modes of *motion* input gets all the data from its hardware and passes it to the API which standardizes the coding experience for manipulating the sensor information. The Kinect is essentially a webcam with a sensor to recognize the distance to an object for a more natural user interface using gestures and voice commands to get its input. The first generation of the kinect was released for the gaming console Xbox 360 in 2010. Later, in 2011 they released the software development toolkit for windows 7 and then in 2012, a kinect version of windows was released[1].

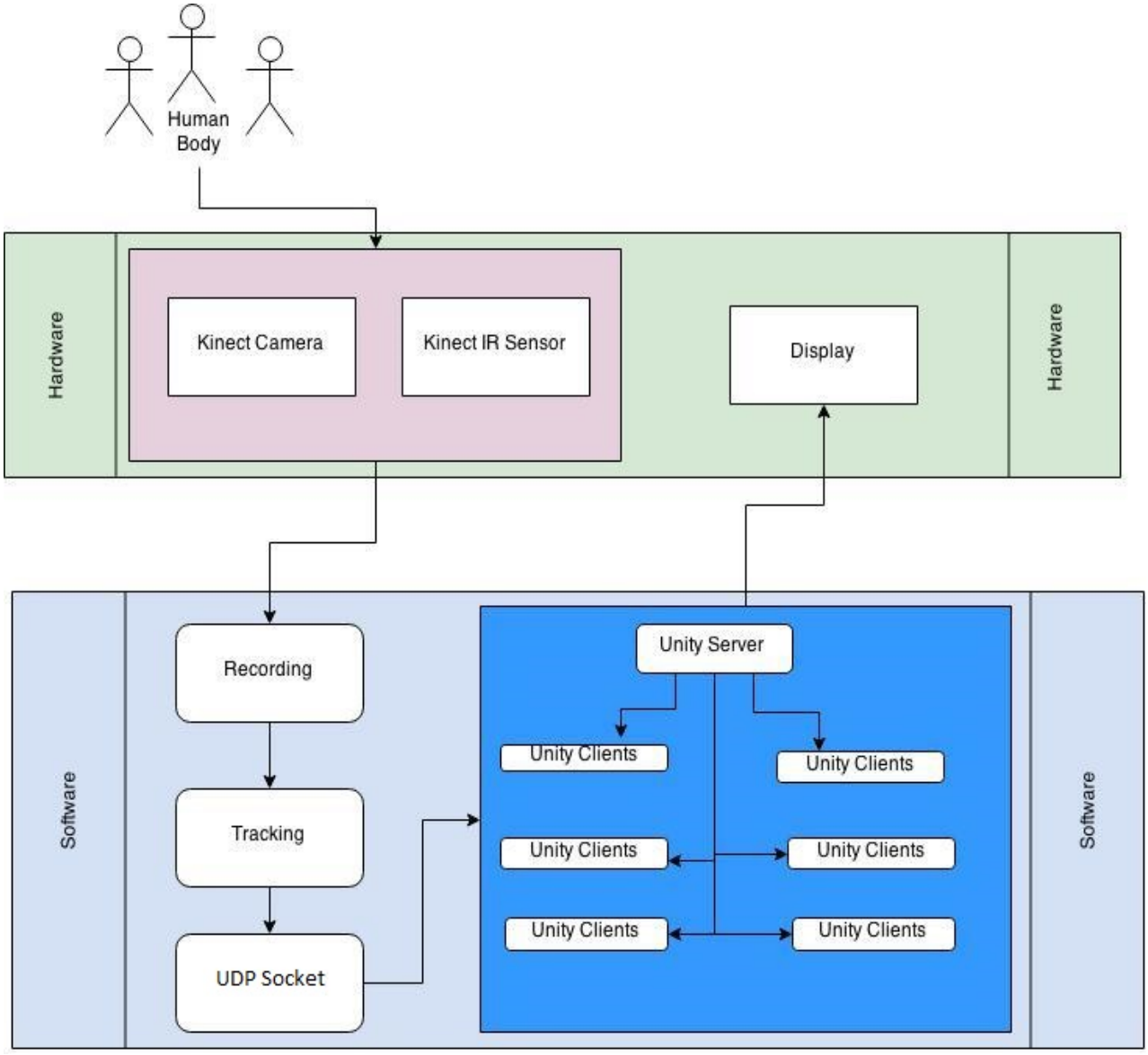
The Kinect .NET framework has been largely developed since its inception and recent launching of the Kinect 2.0. This framework is standard for Kinect for Windows as well as Kinect for Xbox. In this project we have used Kinect for Windows due to the limitation of the plug-in. The SDK allows the user to write in C#, Visual Basic .Net, or C++/CLI.

The Kinect SDK allows us to see various modes of input like depth information using the color gradients from white (near) to blue (far). The main advantage of using the Kinect for

windows is that it comes with raw sensor streams and innate ability to track skeleton data. The standard package also comes with good documentation for better and easier coding[2]. The setup mostly is pretty easy. Its all GUI with on screen prompts to do things. The SDK toolkit is pretty easy to use even for a novice user.

The Kinect hardware passes the information to the NUI API which is then passed to different applications. Therefore for any work with Kinect you have to import its libraries and can be coded natively in C#. We have used C# for this project inside Unity 3D Pro.

Architecture



The diagram above accurately depicts the working of the system. The kinect picks up lot of information but based on the requirement you can stream only what is required. For this project we are tracking the human skeleton data. We pick the human object and use transformation to modify the received data to behave The human body is first picked up by the Kinect camera and the infrared sensor inbuilt. This is recorded and then passed to the API. This is then picked up in the Unity 3D code where it is modified and transformed

to map it to the avatar. The output is the control of the avatars based on the input from the Kinect stream.

Although initially the plan was to have multiple kinect's to have a better accuracy it was later dropped. The kinect uses infrared, which if interferes with infrared of other working kinect, it causes it to create tremendous noise. This is a big limitation of the Kinect. There is a paper by Faraj Alhwarin et al. [3], *IR Stereo Kinect: Improving Depth Images by Combining Structured Light with IR Stereo*, that talks about how multiple kinects interefere with each other and results in distortion. We did some empricial analysis ourselves and found this to be true. They have discussed the simple yet powerful methods to overcome these limitations. This limitation will be discussed more in depth in section III.

The Unity 3D has one server and multiple clients. We tried to achieve a Observer Listener design pattern.The human skeleton data that is detected via Kinect is passed on to the API and the independent code that tracks this skeleton data. This information is then passed over network via UDP socket to the Unity 3D Pro. There needs to be transformation applied to the raw kinect data as the data obtained via this is real space (Kinect Space) and we need to change this information to act accordingly in Unity's Virtual Space. The transformation is then applied to the coordinates and which in turn modifies the position and orientation of the character in virtual space. There is one controlling server and there can be innumerable clients.

The main character is being controlled by the skeleton information obtained via movements and pose orientation of the controlling human being via Kinect. Multiple other characters (Clients) can connect via network to the server and participate in the Virtual Environment. Clients can login as main character or audience. There is only one main character and he is along controlled by the Kinect and the other clients (audience) are controlled via arrow keys. The clients are updated frequently and there is no perceivable lag in the update of the characters in the virtual space. The different clients can be logged on with different machines via network.

Inputs / Outputs and Essential Tools

Input

- Human beings in Kinect space
- Control of the character movements using Keyboard and mouse.

Output

- Movement of the main character (Avatar) and its viewpoint modification in the Virtual Environment based on Kinect tracking of the skeleton joints and keyboard input methods.
- Updating the rendered Virtual Space consistently on all the Unity Clients connected to the Unity Server.

Essential Tools:

Software Packages

- Unity 3D Pro
- Monodevelop IDE
- Visual Studio 12
- Kinect SDK 2.0
- Kinect Configuration Verifier
- .Net Framework

Drivers

- Windows Kinect Drivers
- Windows Toolkit Essentials

Hardware

- Kinect for Windows
- USB 3.0 for Kinect Windows
- Some network connection

Minimum System requirements to run this project

- 64-bit (x64) processor
- Physical dual-core 3.1 GHz (2 logical cores per physical) or faster processor
- USB 3.0 controller dedicated to the Kinect for Windows v2 sensor
- 4 GB of RAM
- Graphics card that supports DirectX 11
- Windows 8 or 8.1, or Windows Embedded 8

Implementation

Microsoft Kinect

Image Source: http://www.winbeta.org/sites/default/files/sp-kinect-img.jpg

The Kinect consists of an infrared, a RGB camera, a motorized tilt increasing its capability for more field of view and a multi-array microphone further enhancing the capability to accurately detect position. The inception of Microsoft Kinect has been created an assertive advancement in the recent years in the field of Computer Vision. With developed open-source communities like OpenCV there are so many readily available tools for Face Tracking, Mobile Tracking, Face Recognition, Augmented Reality, etc. There have been several human tracking algorithms and methods that existed before Kinect was introduced. Kinect, with its infrared technology, helped get better accuracy than a regular camera. Before starting our work we read substantial literature related to human recognition to fully understand the most optimal way to approach our problem.

The most common Human detection algorithm is probably by Dalal et al[5], “Histograms of oriented gradients for human detection”. The biggest advantage of this is that this is already implemented in OpenCV. This algorithm achieved nearly perfect result in the MIT pedestrian database[6]. This work been extensively worked on and is usually used as a benchmark in literature to compare the accuracy of their own work. The paper essentially talks about screening the depth information in the first pass, segmenting the human, and

then tracking (extracting from the frame) the human contour. This method is highly based on accurately detecting the human head although they plan to extending on other body parts in future. Therefore if the head is occluded for some reason, like wearing a hat, it will probably not detect the person as human. This method can usually be extended to other general objects too and can work with sensor information from other devices as well. Once combined with other body part detector, they can achieve pretty high accuracy.

The work was extended and improved by Munaro el al[7] by making few simple assumptions. They assumed that *standing people* are above the ground and are exactly perpendicular to it. They made clusters of these “standing people” and then fed it to the HOG detector. They were able to achieve higher accuracy with this method. The other problem dealing with partial bodies or occlusion of HOG detectors was improved by Wang el al[8] by combining it with a Local Binary Pattern (LBP). This outperformed the original algorithm.

Another problem with all the above work was that people can appear in multiple poses when seen from multiple perspective, making it very arduous for the classifier to train with all possibilities. Felzenszwalb et al[9] further enhanced the HOG detector by using the Pictorial Structures Framework[10][11] that shows different objects by its parts, ordered in a deformable layout. This system was based on mixtures of multiscale deformable parts. A latent SVM classifier allowed them to make models for people based on complete as well as occluded bodies.

To our surprise, the Kinect SDK bundle comes with Human tracking. Its detection is far more superior to all the other OpenCV implementations we came across. Therefore we decided to go with the standard Kinect SDK bundle for the Human Detection. There is a well organized guide on the microsoft website[12] with samples showing its capabilities. The API has all the basic components built in which defines each body as a different object. Each body object has a collection of joints making the whole skeleton. This, not only makes it easy to code with its object orientedness, but also is pretty intuitive to understand as the naming conventions of the joint objects are named according to standard use. For example left hand is used as HAND_LEFT, Right shoulder is used as SHOULDER_RIGHT, etc. Also starting the Kinect code is pretty simple.

The bodies tracked by the Kinect are oriented as if the observer was looking in the mirror. Therefore we work with all inverted attributes. Therefore you modify features of left to see effects on the right (will be displayed on right in the screen). Not all the 360 body joints are identified by the Kinect. You can see in the image below the names of the joints it identifies.

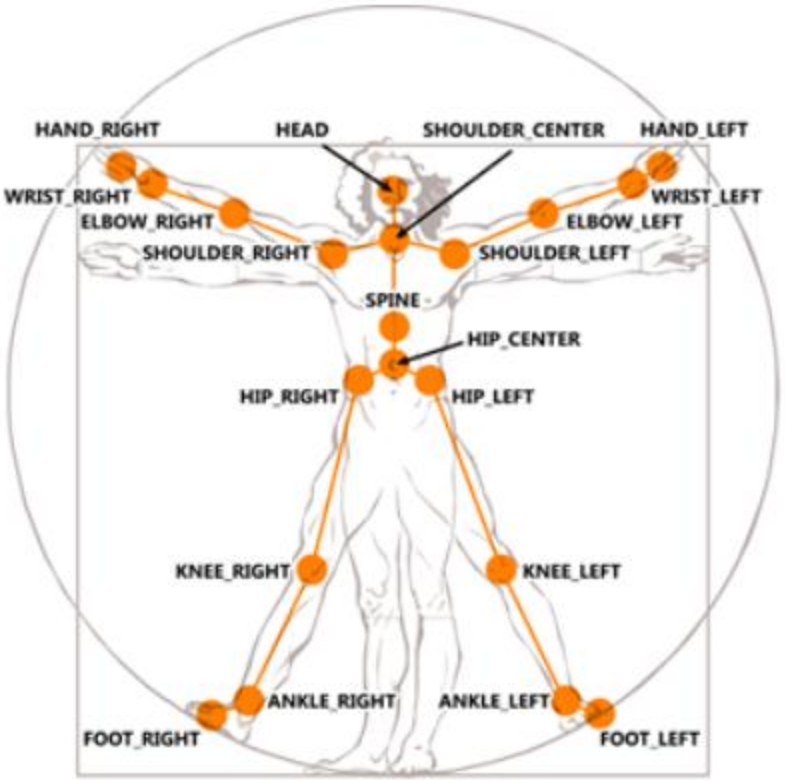


Image source: http://msdn.microsoft.com/en-us/magazine/jj159883.aspx

Once you have downloaded the SDK and are connected to Kinect, you can start coding.

There are some common things you have to set up while working with any Kinect. You can start by referencing to the kinect by this

```
KinectSensor sensor = KinectSensor.KinectSensors[0];
```

The ‘0’ indicates the first Kinect detected by the machine. If there are multiple Kinects these values will vary.

Next is to instantiate ColorStream component and SkeletonStream component. These are essential to Human Tracking as the ColorStream gets the RGB Video Stream required and the SkeletonStream provides us with the skeleton tracking ability.

```
sensor.ColorStream.Enable();
sensor.SkeletonStream.Enable();
```

Rest of the code is basic object oriented programming. One big benefit here is that you can create a window and see the real time output. This helps in debugging easily. And you can find the programming guide here http://msdn.microsoft.com/en-us/library/dn782037.aspx.

Unity 3D

Unity is a cross-platform game creation system developed by Unity Technologies, including a game engine and integrated development environment (IDE). It is used to develop video games for web sites, desktop platforms, consoles, and mobile devices. First announced only for Mac OS, at Apple's Worldwide Developers Conference in 2005, it has since been extended to target more than fifteen platforms.It is now the default software development kit (SDK) for the Nintendo Wii U.[13]
Unity also has a extensive documentation and tutorials available on its official website http://unity3d.com/.

Unity is notable for its ability to target games to multiple platforms. Within a project, developers have control over delivery to mobile devices, web browsers, desktops, and consoles. Supported platforms include BlackBerry 10, Windows Phone 8,Windows, OS X, Linux (mainly Ubuntu), Android, iOS, Unity Web Player (including Facebook), Adobe Flash, PlayStation 3, PlayStation 4, PlayStation Vita, Xbox 360, Xbox One, Wii U, and Wii. It includes an asset server and Nvidia's PhysX physics engine. Unity Web Player is a browser plugin that is supported in Windows and OS X only. Unity is the default software development kit (SDK) for Nintendo's Wii U video game console platform, with a free copy included by Nintendo with each Wii U developer license.
With an emphasis on portability, the graphics engine targets the following APIs: Direct3D on Windows and Xbox 360; OpenGL on Mac, Windows, and Linux; OpenGL ESon Android and iOS; and proprietary APIs on video game consoles. Unity allows specification of texture compression and resolution settings for each platform the game supports, and provides support for bump mapping, reflection mapping, parallax mapping, screen space ambient occlusion (SSAO), dynamic shadows using shadow maps, render-to-texture and full-screen post-processing effects. Unity's graphics engine's platform diversity can provide a shader with multiple variants and a declarative fallback specification, allowing Unity to detect the best variant for the current video hardware; and if none are compatible, fall back to an alternative shader that may sacrifice features for performance.

The game engine's scripting is built on Mono, the open-source implementation of the .NET Framework. Programmers can use UnityScript (a custom language withECMAScript-inspired syntax, referred to as JavaScript by the software), C#, or Boo (which has a Python-inspired syntax).

Integrating Kinect with Unity

The integration part was probably the main hurdle/challenge since we were trying to detect Kinect from inside Unity's framework initially. The most obvious way was to use a deprecated plugin included in Windows Kinect SDK 1.8 version but not in the latest version. We also examined official Windows Kinect libraries and various third party plugins like ZigFu (This is partly why Unity pro is need in our project, i.e. to allow working with 3rd party plugins), Openni and several projects on GIT to do the same.

It might have been lack of documentation and tutorials or our misunderstanding of certain components of the driver and plugin parts that we were unable to continue down this path. We were also unable to find a good working official demo / documentation from Microsoft / Unity on how to detect and access Kinect data from inside Unity's framework using C# script. People who have been working on doing something similar were either modifying the library DLLs or were using the paid version of ZigFu library and drivers. But since we did not find any substantial evidence that showed ZigFu or the more hackish DLL method worked, we decided to switch our focus on to communicating processed Kinect data to Unity Client using network communication, i.e. over a UDP socket to be more precise. The other option was to use inter-process communication or process to process pipeline. After investigating about these different ways and working on basic prototypes for each, we proceeded with Network communication since it seems more flexible and adaptable with the Clients-Server architecture of Unity and is also quite simpler to implement than the other approaches.

After being able to successfully detect movement and orientation of a Human body using Kinect in the WPF application and then processing the raw data and inferring various combination of movements (turn left, turn right, front, back, strafe left, strafe right and other diagonal movement combinations of front/back/left/right) of the Kinect actor, we worked on communicating and integrating these movement updates with the client which is logged in as the "Main Character" (Avatar) and then replicating these changes to the server and consequently all connected clients (audiences) to the server.

There were couple of challenges in inferring the movements precisely since we were not using the Kinect plugin, which provides APIs to easily detect and infer these movements and orientation changes. We started with investigation of the various Skeletal joints which the Kinect SDK Library provides access to. We initially chose Spine-Base Joint (available on Body.Joint object and is located near the Hip area) to detect movements as well as orientation. We used Spine-Base.Z for forward and backward movements and SpineBase.X to detect lateral movement, like Left/Right. Spine-Base seemed to work fine when orientation and movement changes were independent. But then we realized that it was not a good idea to use the same skeletal joint for both movement as well as orientation since it gave us poor performance and accuracy this

way. So we additionally used ShoulderRight Joint to better detect pose orientations (i.e when the user turned left or right).

The movement and pose updation of the Avatar on Unity is not perfect because it uses relative changes in positions and orientations instead of absolute ones. To further improve this and make the mapping between Kinect Actor and Avatar seem more seamless and natural, we could transform the absolute Kinect space coordinates of the tracked body sent to Unity into Unity's Virtual space (Coordinate system). This way we would just have to set the position of the Avatar's GameObject and we also won't have to keep track of two different skeletal joints (Spine-Base and ShoulderRight). This kind of absolute mapping would be more ideal and flexible.

On Unity's side, we worked on adding asynchronous C# script function callbacks to receive the processed Kinect data over a listening UDP socket. Once the turn/move direction is received on Unity's end, we update the Avatar on the client and the server using RPC calls. And because all the Unity Clients which are logged in as Audience, poll the main character's state frequently from the server, it is instantly available across all machines and thus the virtual space is synced across multiple machines, which can be remote or local.

There is scope to further generalize the Unity's Virtual Stage framework so as to make it easier for Kinect or as a matter of fact any other general interactive sensor like Oculus Rift, Leap Motion etc to easily integrate with the Virtual Stage framework setup inside Unity. One possibility is to add a C# Sensor-Interface which all Sensors implement (to be used inside Unity to receive sensor data) and a C# VirtualStage-Interface provided by Unity (which could be used by Sensors to send data seamlessly and in a decoupled manner)

Challenges

There were several challenges and limitations faced during the development of the project.

Kinect Plugin for Unity 3D

This was the biggest obstacle faced during the development of project. First problem we had that it would only work with the Unity 3D Pro (paid) version. This way we were limited to the machines having Unity 3D pro installed in them. Even after the plug-in is installed (you have to just import package for installation), we realized mid way that it only works with Windows version of the Kinect. Therefore we were limited to availability of one Windows version Kinect we had between the whole class to work with. We would have had more Kinects integrated, or at least attempted it but the resource too was limited. Even trying to find a proper download link proves to be a difficult task because it is managed by Microsoft website which keeps changing the download link and all the crawled searches of previous problems becomes futile.

There is no good documentation for this plug in. It comes with a readme file with few steps telling you to import files. There is no definite guide as to how the plugin affects the coding on either side. It does come with a sample project which, at least to a novice user, is not very intuitive. You need an existing project to run that sample. Also there are new versions of both Kinect SDK and Unity 3D Pro that come out asynchronously and this plugin is incompatible again. Therefore it needs a lot of factors in perfect harmony to work well and is very difficult to use. Hopefully this will improve over next few iterations of the versions coming out in the coming years.

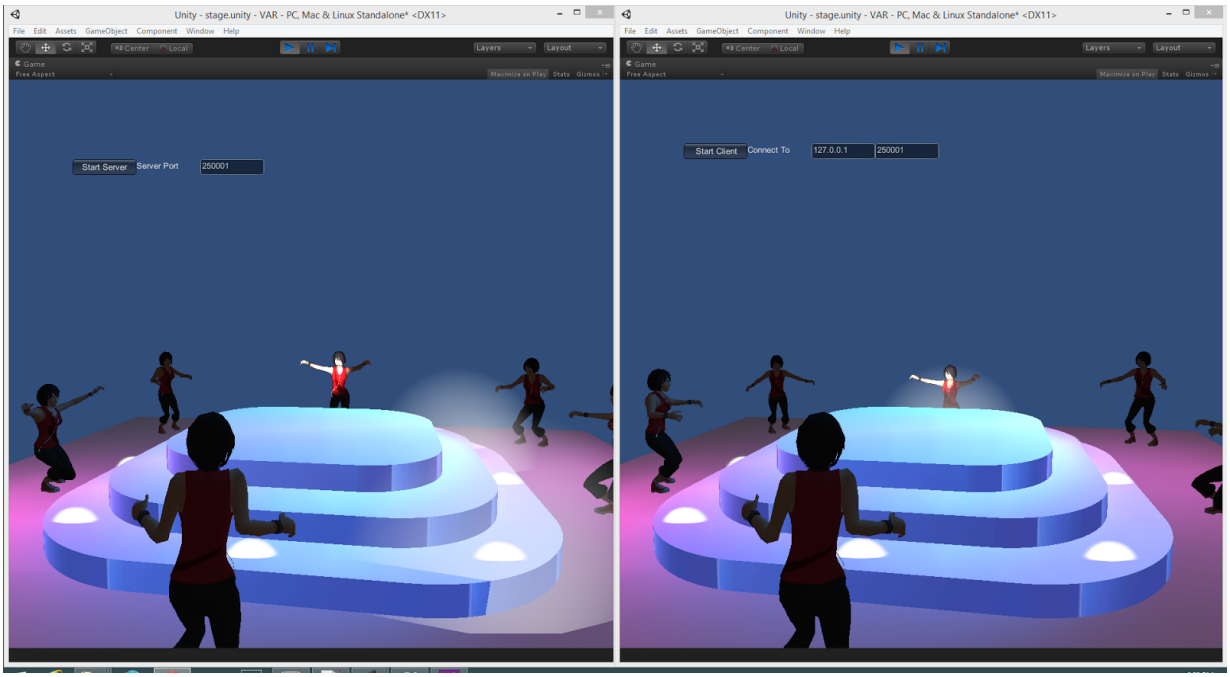
We eventually decided that it would be easy to just write a code and pass information over the network than try to make a broken underprepared tool work.

Kinect Infrared

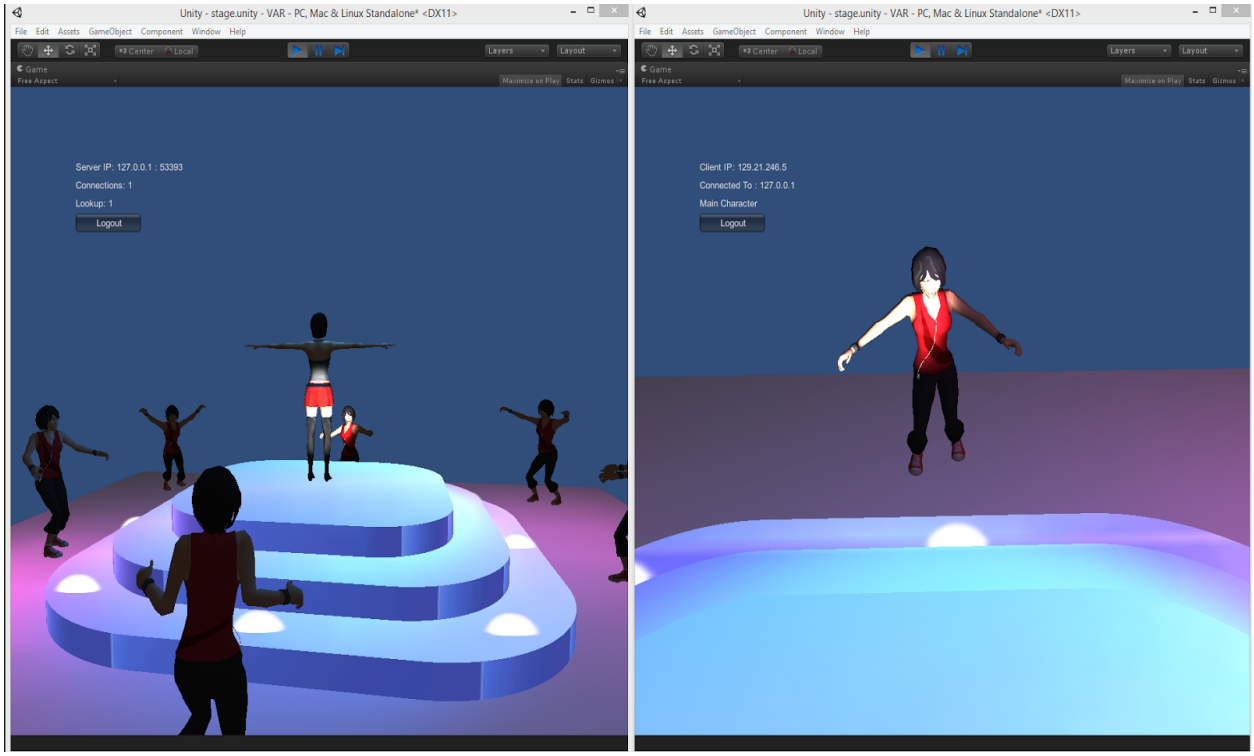
There are several limitations of Kinect with respect to range and field of view over the competitors but one major drawback that directly impacts this project is the Infrared. The infrared Kinects Infrared is far from the best. Faraj Alhwarin et al. [3] have discussed this in great detail. Lot of factors hinder the performance of the kinect infrared like sunlight (its blinding), rain, dust, lighting conditions, etc. There are several other sensor devices like the Hokuyo URG-04LX-UG01, XV-11 LIDAR, UTM-30LX, etc. that have superior specs but unfortunately they do not have standard libraries and open source community support.

Even with this limitations, the performance that can be achieved is industry standard. Therefore it was a natural choice and with all its API and community support it will keep growing substantially in the coming year ahead.

Results

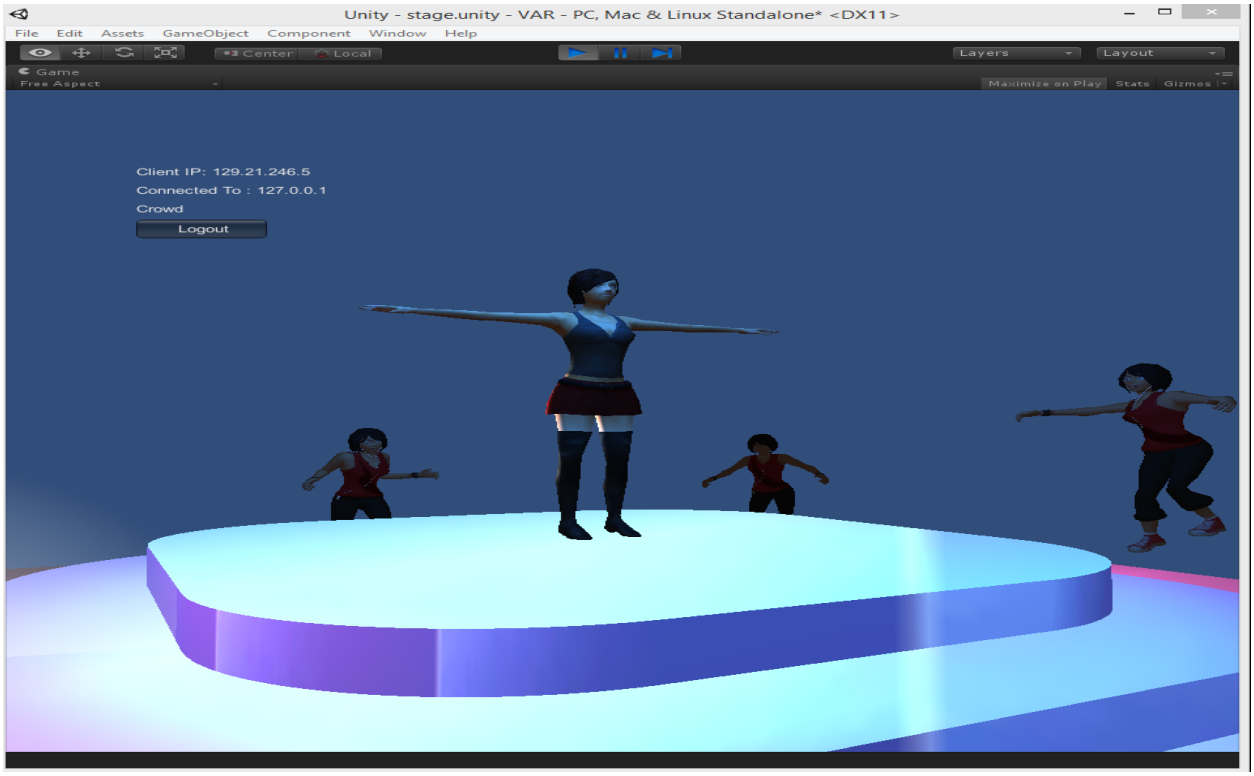


(Unity Client and Server Screens after launching respective Unity scenes)

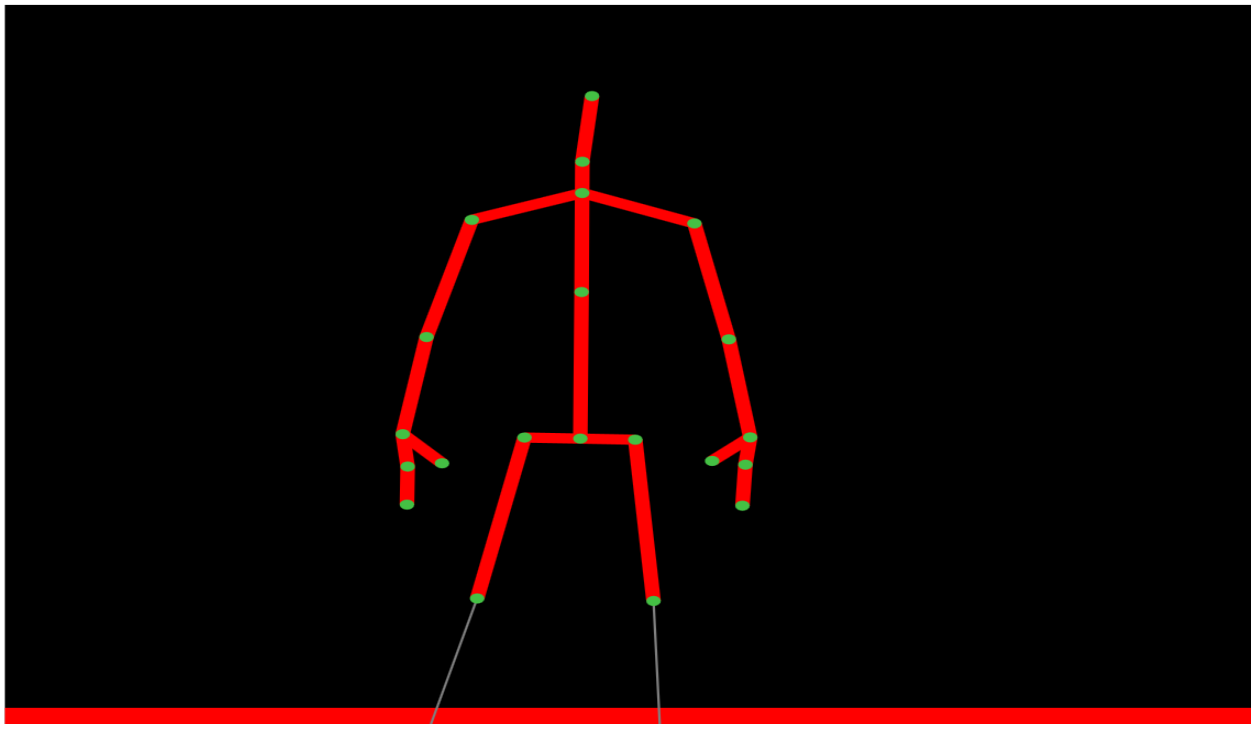


(Unity Server has started; Unity Client logged in as Main Character)

(One of the Unity Client screen presented with Login Options when “Start Client” button is clicked)



(One of the Unity clients logged in as a character in the audience)



(Kinect-WPF application sends body tracking data to client with main character)

Limitations

- New User Preference with respect to windows size are not respected. User needs to manually adjust the size
- Virtual Space does not follow all laws of physics.
- Models do not exhibit free body animations.
- Difficult to detect orientation of the Head or Neck joints for smooth viewpoint modification.
- Body motion in Kinect Space and Unity's Virtual Space isn't mapped very elegantly yet. Hence updating the Avatar with Kinect's tracked Skeleton data is little jerky and off at times. May need to transform Skeletal Joints data from Kinect Space to Unity's Space.
- Kinect Joint tracking capability degrades if the use turns by a significant amount to their side.
- Using multiple Kinects interfere with each other's fields of views.
- Responsiveness of the system is limited by Network's bandwidth and throughput.
- There can be interference due to objects in Kinect Space which may appear like Human body parts to Kinect.

Conclusion

A functional virtual stage has successfully been designed using the Unity Engine. The model is platform independent. The following goals were achieved:

- Designed a Virtual Stage using Unity Engine.
- Person Tracking (position, orientation) using Kinect and Windows Kinect SDK.
- Established relationship between physical and virtual spaces.
- Viewpoint modification for multiple screens (Unity Clients)
- Multiple users can share a virtual space.
- Faced several hurdles related especially to hardware drivers and plugins throughout the project and successfully worked around them as a team with Professor's guidance.
- A further developed version of the project can be used in various other disciplines / fields where collaborating in virtual space is required.

Team Contribution

Vedant :

- Designing the architecture of the Virtual Stage, constructing the virtual environment from scratch using shaders, particles and textures with Kumar.
- Organizing the physical space for virtual stage.
- Creating and Linking dummy models for testing.
- Creating a Server Client Network on which the system is based.
- Designed a generic framework for extending to multiple screens and increased field view.
- Seamless movement of characters in virtual space.

Kumar :

- Investigating Kinect related drivers and plugins (Investing about ZigFu 3rd party library and drivers for Kinect).
- Detecting various combinations of movements and orientations using raw Kinect data inside the Kinect WPF application.
- Investigating various mechanisms to communicate Kinect Data to Unity (Inter-Process communication, Process-Process pipeline,TCP/UDP socket communication etc)
- Communicating Kinect data (from Kinect-WPF application) to Unity Client over UDP socket asynchronously.
- Updating the movements and orientations of the Avatar in Unity using Kinect data by working with Unity C# scripts ClientScript.cs and ServerScript.cs.
- Worked on Unity Architecture of the Virtual Stage, especially the networking aspects of it.

Bharath:

- Investigating about Third party Unity plugins for direct integration of Kinect inside Unity. Also investing about novel ways to detect Kinect from inside Unity.
- Investigating about working with Multiple Kinects for better human pose detection.
- Gathering results and their analysis for better calibration of Kinect's working inside Unity.
- Worked on submodules related to Unity Server code and Kinect WPF pose orientation.
- Testing the System both on Unity's and Kinect's side.
- Finishing up most and major parts of the Project Report and code Documentation related to Unity and Kinect code.

Future Work

1. Give more control over the audience character to the user (Dancing, Cheering etc)
2. Design the Virtual Space to be more realistic.
3. Include sounds in the model and background score in Virtual stage.
4. Animate body parts (like arms, legs, hands etc) of Models to reflect Skeletal movements detected from Kinect.
5. Seamless and accurate mapping of character movements between Kinect space and Unity Space.
6. Better detection of pose/orientation of the user and research more about the use of multiple Kinects.
7. Integration with MoCap team and other projects.
8. May be develop a mini-game by working on a fork of the current project.
9. Provide an API from the Kinect WPF side of the application which makes it easy to fetch and work with our processed Kinect data.
10. Extend the Virtual Stage framework in unity to work with other sensors like Leap Motion, Oculus Rift etc which send data to the framework.
11. Add support for tracking more than one main character on Kinect-WPF Side and add functionality on Unity's side as well to control more than one main character.

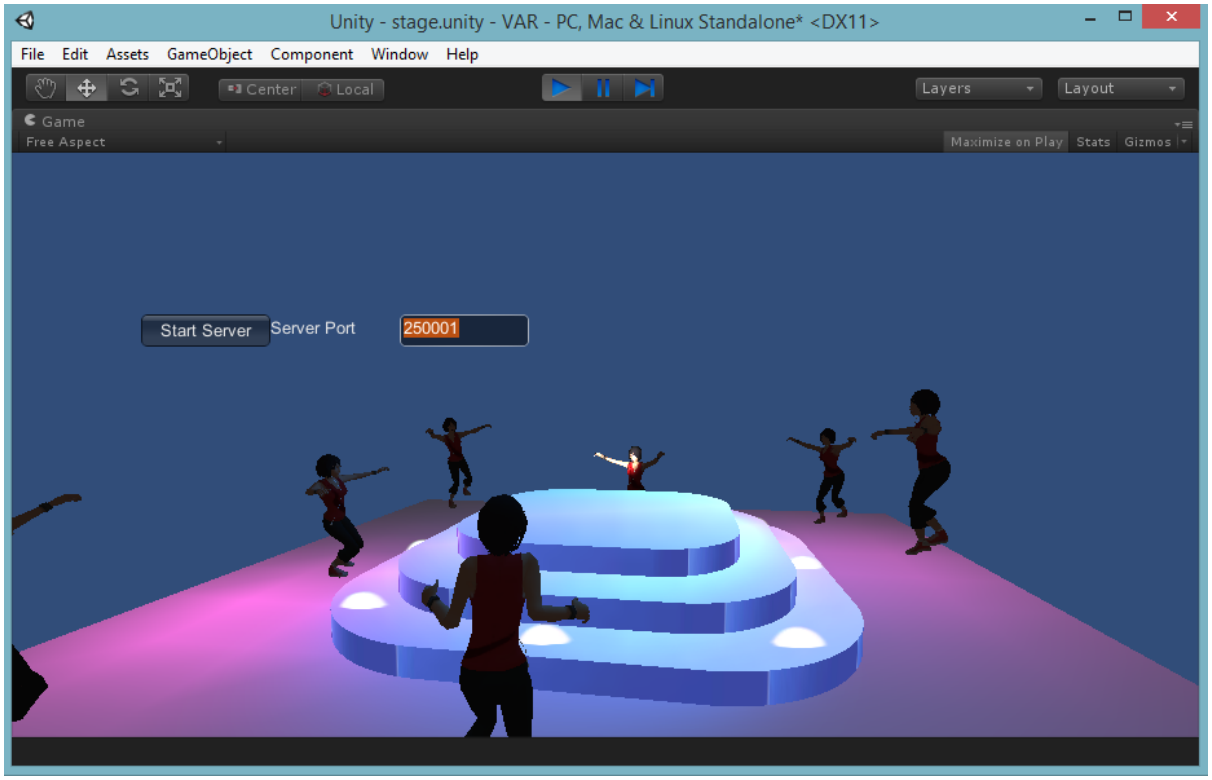
User Documentation

Setting up the workstation and Running the project:

For setting up the machines, please ensure that they meet the minimum requirements as stated in the Input/Output section

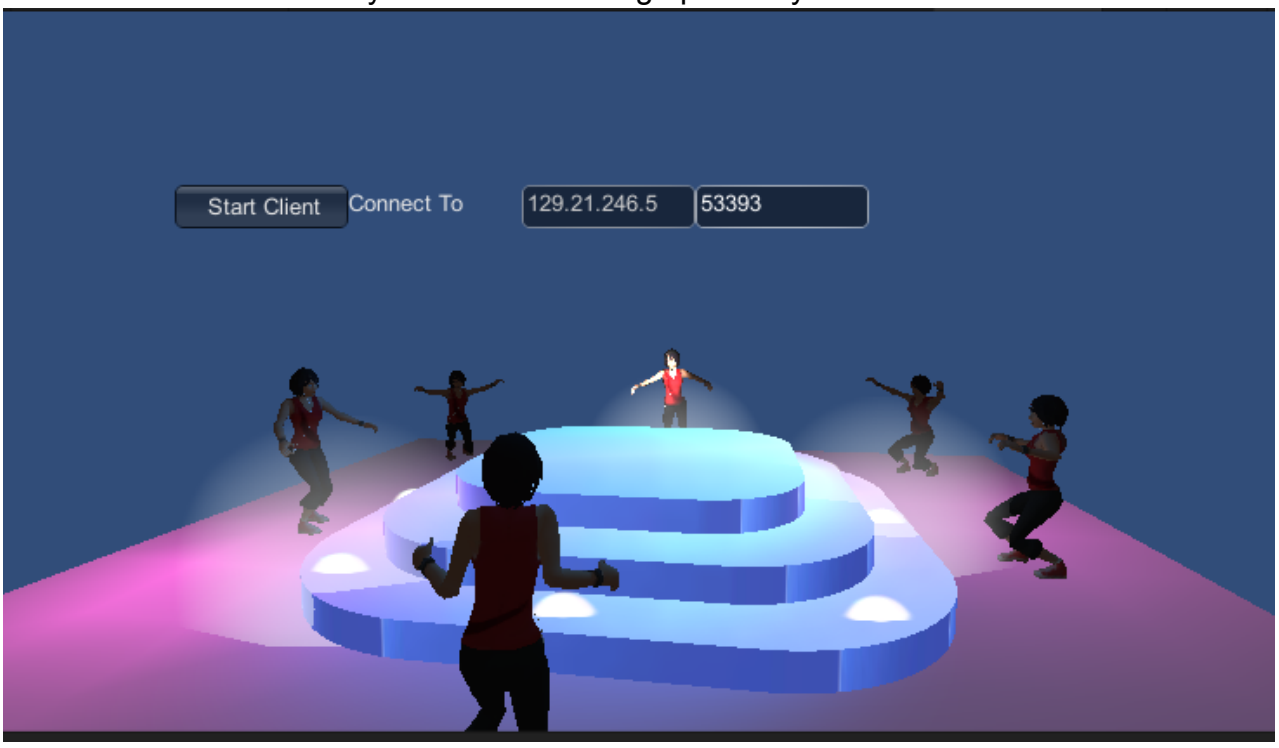
For End users:

1. OS specific Binaries are provided for Unity Server and Client instances along with this report. Users can run these binaries on the same machine or different machines since a facility to connect to the Server using its stated IP address is provided.
2. Launch the Server binary. This should bring up a Unity screen which looks as below:

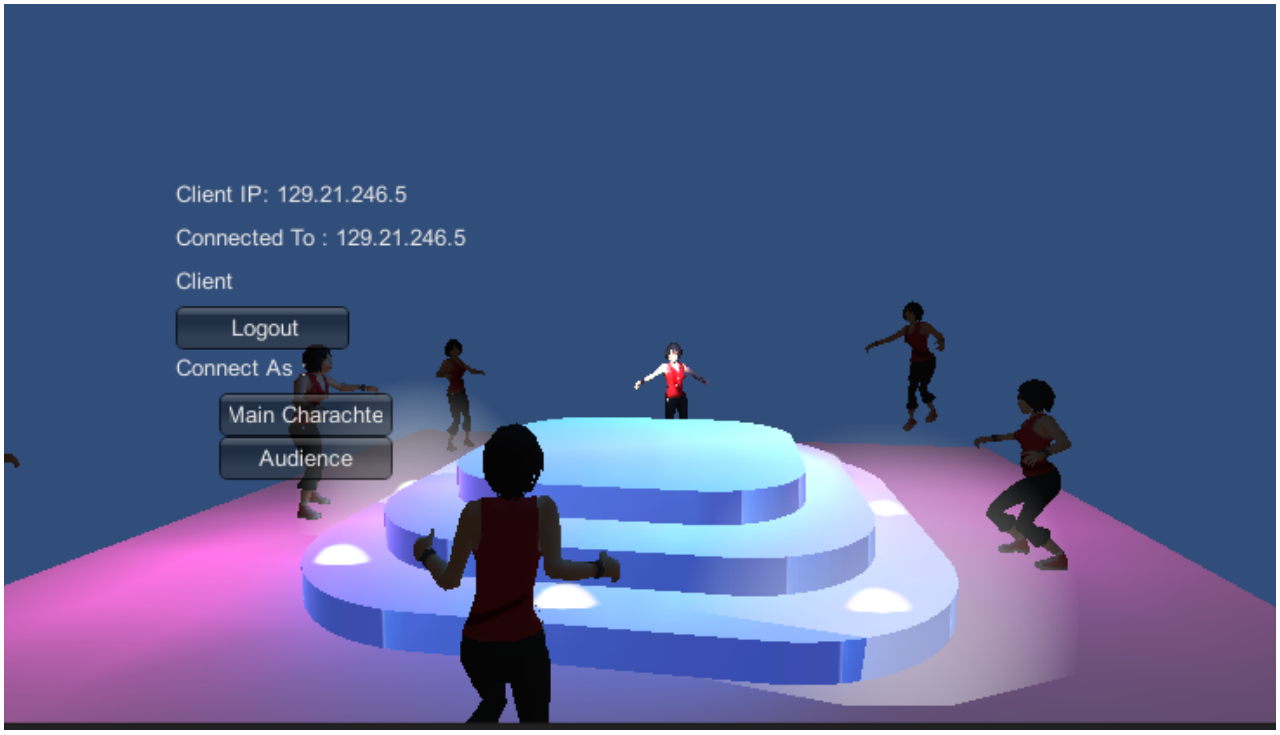


3. Click on the “Start Server” button to start the Unity Server to which all clients can connect by specifying its IP address (which can be seen in the screenshot below):

4. Launch the Client Binary and it should bring up a Unity scene as below:

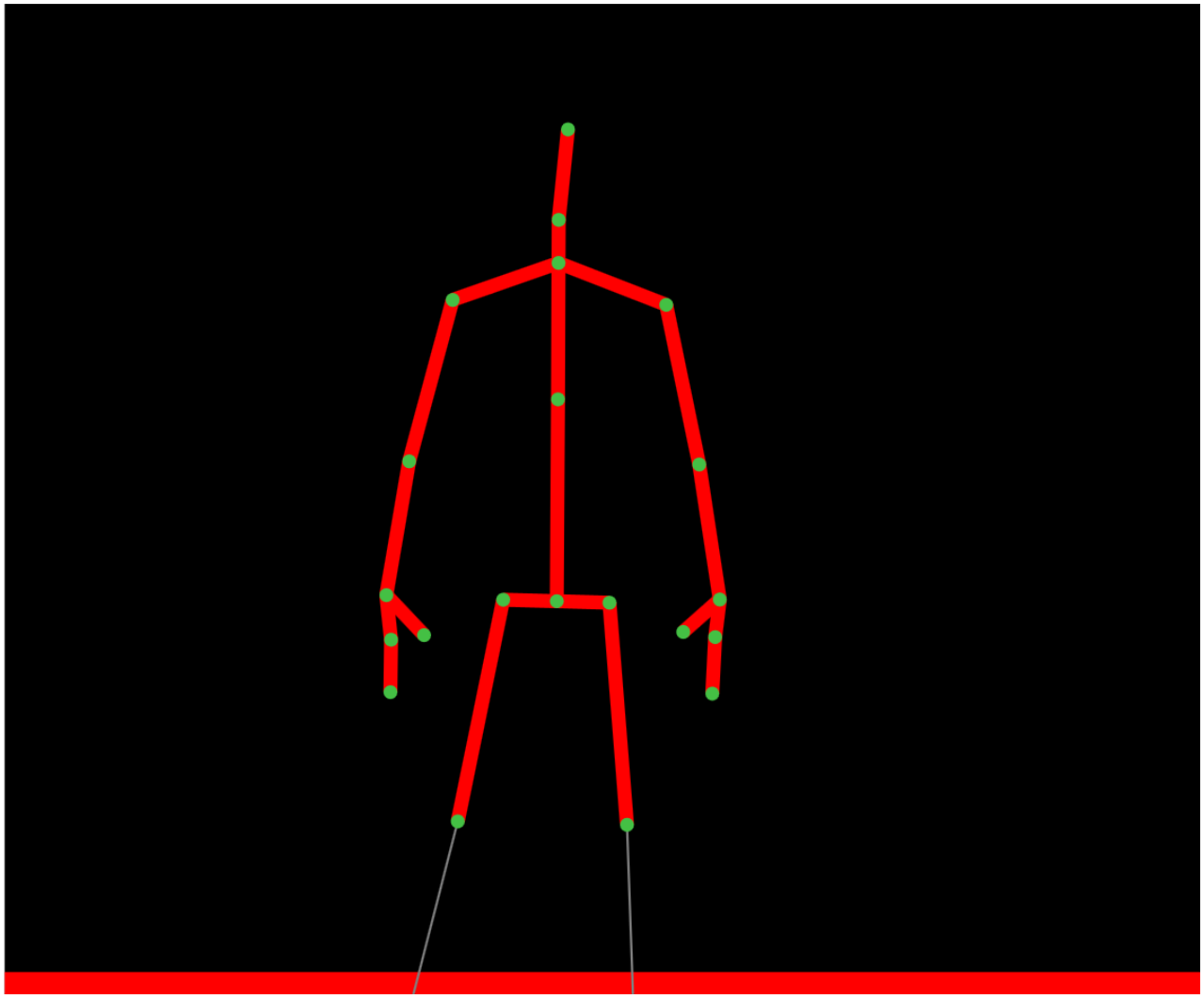


5. Click on Start Client after specifying the IP address and port of the Unity server which was started before. There are options to connect as the “Main Character” or the “Audience”



6. Connect as the Main Character and this should should present the viewpoint of the main character which is spawned in the center of the stage. The Unity Server instance should also see a new character spawn in the middle of the stage. Users can control the viewpoint and position using arrow keys (left, right, up, down) and by holding the shift along with the arrow keys to rotate the Avatar about its own Y-axis.

7. Now this setup is ready to work with the Kinect WPF application which sends character position and orientation data over the network. For controlling this new spawned Avatar (main character) with Kinect, launch the respective provided binary or the project in Visual Studio 2012. Launching it should present a screen as shown below which shows a tracked skeleton of the body in front of the Kinect.



For Developers and Testers of the System:

Assumes running the client and server Unity instances on the same machine. The Clients-Server Unity architecture will easily extend to multiple machines. Visual Studio's WPF application is also on the same machine.

1. The Computer should have the minimum requirements as stated in Input/Output section.
2. Start at least two Unity instances each pointing to two different copies of the original project. One will serve as the server and the other will serve as the client.
3. To configure one of the projects as the server, checkmark Server script in Inspector view of the Main Camera of the scene to launch this project scene with Server configuration. Make sure the Client script is unchecked.
4. To configure the other project as the client, checkmark Client Script in the inspector view of the Main Camera. Make sure the Server script is unchecked.

5. Then the respective projects can be launched by selecting the scene called ‘stage’ and hitting play button in Unity.
6. The two main components in the Unity project are ClientScript.cs and ServerScript.cs. ClientScript code is located in ClientScript.cs, it handles the client UI and listening on UDP port from Kinect WPF application and updating the main character’s position. ServerScript.cs code which has the UI and handles starting of the server. It also makes the RPC client calls to sync across clients. NetworkingScript.cs is responsible for spawning, removing of main character and audiences. It also does syncing across multiple clients. RotateCamera.cs is used in viewpoint modification.
7. Now moving to Kinect WPF application, it just has one main file which is MainWindow.xaml.cs. This file is an extension of the BodyBasics-WPF sample code included in the WIndows Kinect SDK. The changes we made to the code are between these comments: //VS-START *** and //VS-END. We have thoroughly documented this section in the code and it should be pretty straight forward for anyone who wishes to extend the application.

Links and Help:

- http://www.microsoft.com/en-us/kinectforwindows/
- http://unity3d.com/unity/download
- http://unity3d.com/learn
- http://kinectforwindows.codeplex.com/
- http://www.microsoft.com/en-us/kinectforwindows/develop/downloads-docs.aspx

Bibliography

[1] http://en.wikipedia.org/wiki/Kinect

[2] http://wiki.etc.cmu.edu/unity3d/index.php/Microsoft_Kinect_-_Microsoft_SDK

[3] http://www.cs.cornell.edu/~hema/rgbd-workshop-2014/papers/Alhwarin.pdf

[4] http://www.microsoft.com/en-us/kinectforwindows/develop/downloads-docs.aspx

[5] N. Dalal and B. Triggs, “Histograms of oriented gradients for human detection,” in Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on, vol. 1, pp. 886–893 vol. 1, 2005.

[6] “Mit pedestrian database.” http://cbcl.mit.edu/software-datasets/ PedestrianData.html. [Online; accessed 13-Sept-2014].

[7] M. Munaro, F. Basso, and E. Menegatti, “Tracking people within groups with rgb-d data,” in Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on, pp. 2101–2107, 2012.

[8] X. Wang, T. Han, and S. Yan, “An hog-lbp human detector with partial occlusion handling,” in Computer Vision, 2009 IEEE 12th International Conference on, pp. 32–39, 2009.

[9] P. Felzenszwalb, R. Girshick, D. McAllester, and D. Ramanan, “Object detection with discriminatively trained part-based models,” Pattern Analysis and Machine Intelligence, IEEE Transactions on, vol. 32, no. 9, pp. 1627–1645, 2010.

[10] P. Felzenszwalb and D. Huttenlocher, “Pictorial structures for object recognition,” International Journal of Computer Vision, vol. 61, no. 1, pp. 55–79, 2005.

[11] M. A. Fischler and R. Elschlager, “The representation and matching of pictorial structures,” Computers, IEEE Transactions on, vol. C-22, no. 1, pp. 67–92, 1973.

[12] http://msdn.microsoft.com/en-us/library/dn782037.aspx

[13] http://en.wikipedia.org/wiki/Unity_%28game_engine%29