Journal of Mobile Multimedia, Vol. 8, No. 3 (2012) 225–240
© Rinton Press

CONDROID: USING AN ANDROID PHONE AS A 3D INPUT DEVICE FOR A MULTIUSER 3D DRAWING APPLICATION SETUP IN A COLLABORATIVE VIRTUAL ENVIRONMENT OVER THE WEB

DANISH CHOPRA DREW GLASS
University of Illinois at Urbana Champaign
dchopra2@illinois.com jaglass@illinois.edu

Received August 29, 2012
Revised November 20, 2012

3D collaborative virtual environments (CVE) are gaining popularity. One problem with 3D CVEs is lack of a natural 3D input device. Smartphones are gaining popularity for being used as a 3D input device for a variety of purposes. We present ConDroid, a system that uses a smartphone as a 3D input device in a 3D drawing application setup in collaborative virtual environment over the web with support for multiple users at the same time and on large or small displays. We tackle the problem of remote synchronization in CVEs using the concepts we have named as SUMD (synchronicity using minimal data) and remote semaphores with deadlock avoidance. For the CVE architecture, we use the basic idea of the Active replication model in which an atomic broadcast is used to deliver updates to all of the clients in order to keep them synchronized and use the SUMD and remote semaphores approach on top of it. Our work comprises of an Android application for 3D input, a 3D drawing windows application projected on a large (or small) display with remote collaboration capability, and a middleware server application. Our approach can be used by others to use smartphones as a 3D input device for computers as well design CVEs with quick and effective synchronization using the concepts of SUMD and remote semaphores.

Keywords: Collaborative virtual environment, 3D gesture recognition, Accelerometers, Android application, Semaphores, Remote synchronization, 3D drawing application
Communicated by: I. Khalil & G. Kotsis

1 Introduction

As technology expands the capabilities of communication and computing, 3D collaborative virtual environments (CVEs) are emerging as an interesting and challenging topic. 3D CVEs are becoming more and more popular in gaming, social, and learning spaces. They can be found in a wide range of places - from blockbuster video games to university courses. The proliferation of 3D CVEs exposes the inadequacy of traditional, sub-3D and single user, input devices in a collaborative 3D world.

Traditional input devices such as computer mice and keyboards lack the degrees of freedom to manipulate naturally virtual 3D objects. While traditional input devices are well suited for applications that operate in at most two dimensions, introducing the third dimension leaves them wanting. Attempts at mapping 2D input devices to a third dimension results in awkward interfaces that feel artificial to users and limit the range of possible interactions added by a third dimension. Some approaches at mapping traditional input devices to a 3D world require multiple input devices that lead to complex interaction schemas that differ greatly between applications. These schemas have steep

learning curves and usability issues. Over the past several years, companies introduced 3D input devices that overcome the obstacles imposed by traditional devices.

The current set of 3D input devices suffer from limitations and prove to be inconvenient to the user. 3D mice such as those from 3Dconnexion [1] are costly. 3D input devices designed by video game console manufacturers like the Wii remote control [14], Xbox Kinect [5], and PlayStation Move [2] provide a true 3D input space to users, but their expense prohibits general use, they are not widely available, and they are not portable. Smartphones although used less as 3D input devices provide a viable alternative.

Rapid technological advancement and the social adaptation to smartphones ushers forth an era where everyones pocket contains a versatile device that can interact with 3D CVEs in a natural manner. The addition of sensors such as accelerometers, gyroscopes, and magnetometers that can accurately and precisely locate a phones position in three dimensions have transformed smartphones into an ideal 3D input device. Hinckley *et al.* [7] recognized many different gestures made on smartphones to capture 3D inputs. Du *et al.* [3] developed Tilt and Touch that used smartphones to tilt viewport, scale viewport, and rotate objects in a virtual environment on a computer. Tang *et al.* [12] successfully developed a method to interact with 3D objects on smartphones using a client server architecture and later extended their work to a CVE on a smartphone.

Realizing a complete 3D CVE that incorporates smartphones as input devices still faces significant challenges. One area for improvement is recognizing complex gestures based on accelerometer values in real-time, especially when a remote server does much of the processing. A second area for improvement is synchronization of the CVE application in the presence of large delays due to transferring large 3D object data over the network. Synchronization is also an issue when many users want to perform simultaneously different operations on the same 3D object at the same time.

We propose Condroid which targets smartphones to be used as a 3D input device for real-time collaboration in 3D CVEs, because they receive little attention and require technologies that are just now emerging. Our system consists of an Android application and smartphone for 3D input, a 3D drawing application projected on a display with remote collaboration capability, and a middleware server application to support the remote collaboration. We propose two techniques to solve the problems of synchronization in 3D CVEs, Synchronicity Using Minimal Data (SUMD) and remote semaphores with deadlock avoidance.

Our approach is unique compared to current approaches because:

1. We are processing accelerometer data remotely, thus reducing the load on the mobile phone. Though there have been attempts to do this, we try to approach the problem in our own way.

2. We are using a remote server to maintain a state of all the activities happening across all the android applications and 3D drawing applications with minimal data possible using the SUMD approach.

3. We are introducing a concept of remote semaphores with deadlock avoidance for synchronizing remote data as well as techniques to avoid contention.

The paper is organized as follows:

1. Design section - In this section, we discuss the complete design of our system.

2. Recognizing gestures from raw accelerometer data - In this section, we discuss how we recognized gestures from raw accelerometer data.

3. Synchronicity Using Minimal Data - In this section, we discuss the concept of SUMD.

4. Remote semaphores and deadlock avoidance in them - In this section, we discuss the concept

of remote semaphores and deadlock avoidance in them.

5. Evaluation - In this section, we present the results of evaluating our system.
6. Related work - In this section, we list the related work most relevant to our research.

2 Design

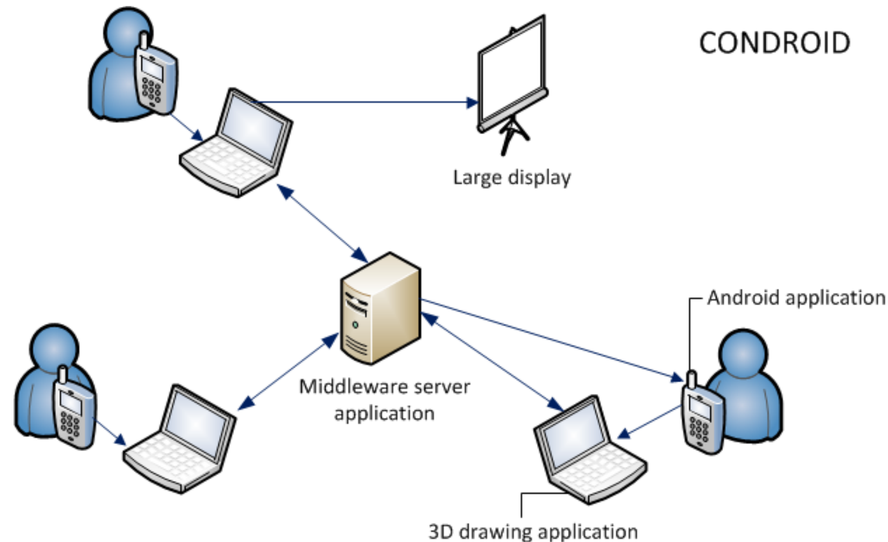

Fig. 1. System architecture showing a number of remote users using their Android applications to provide input to the 3D drawing application running on small and large screens in front of them. A middleware server co-ordinates the data sharing between the different instances of the 3D drawing application as well as between the Android application and the 3D drawing application.

Our design comprises of an Android application for 3D input, a 3D drawing windows application projected on a large (or small) display with remote collaboration capability, and a middleware server application (Figure 1). We have based our system on top of the basic idea behind the Active replication model of CVEs in which an atomic broadcast is used to deliver updates to all of the clients in order to keep them synchronized. Our system design is described as follows.

2.1 Android application

The Android application is responsible for providing inputs to the 3D drawing application. The Android application acts as a toolbox and a controlbox for the 3D drawing application. The capabilities of the Android application are:

1.1 The Android application connects to the 3D drawing application using a shared local WiFi connection. The Android application communicates with the 3D drawing application using UDP packets over this shared WiFi connection. Users are able to connect their smartphones to the 3D drawing application running on their computer by entering the local IP address of the computer into their Android application and the port number for the 3D drawing application (12345).

1.2 Users are able to draw 2D and 3D shapes on the 3D drawing application using simple on screen gestures (Figure 2) on their smartphones. We used the Android's internal gesture recognition classes (android.gesture) to recognize shape gestures. Once a gesture is made by the user, they can add the

associated shape to the 3D drawing application by clicking the ”Add” button located just below the gesture recognition area.

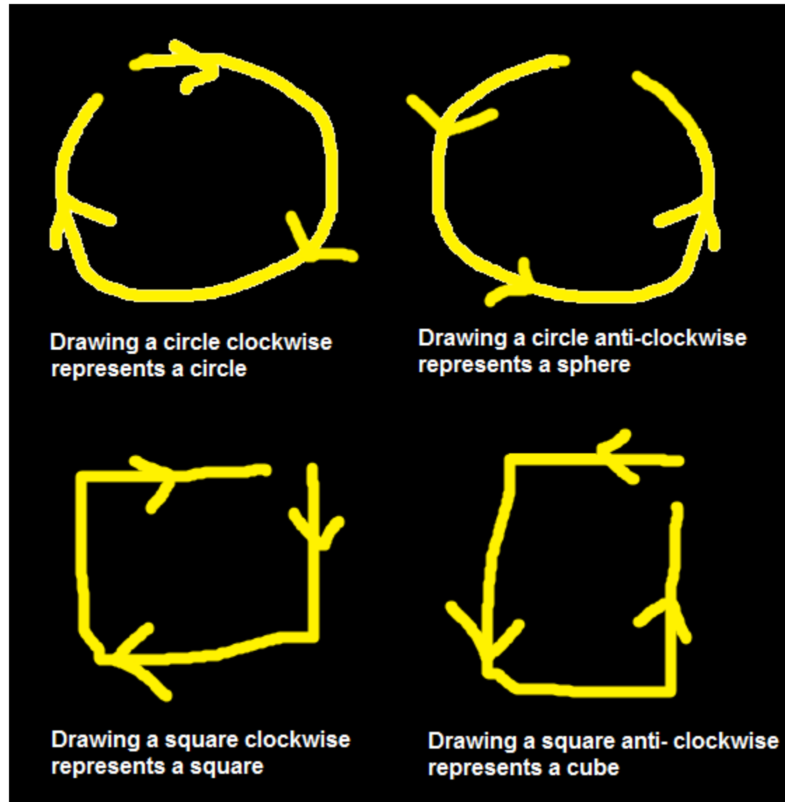

Fig. 2. Various gestures supported by the Android application to draw shapes on the 3D drawing applications

1.3 Users are able to perform basic 2D and 3D drawing operations such as rotate, move, resize and delete by selecting a shape from the shape list dropdown box and then clicking the specific operation button. The 3D drawing application fetches the shape list from the middleware server application. The rotate, move and resize operations are performed using specific smartphone gestures in the air (Figure 3). When the user clicks the ”rotate” or ”move” or ”resize” button on the Android application after selecting a shape, the Android application starts sending accelerometer data to the 3D drawing application till the user clicks the ”done” button. The Android SensorManager and Sensor libraries for the accelerometer generate the data. The accelerometer data consists of the integer values of the current orientation of the smartphone i.e. the X,Y and Z axis of the accelerometer as comma separated values. The system processes accelerometer data inside the 3D drawing application and not inside the Android application because:

1.3.1 The accelerometer values are processed by the 3D drawing application and not the android application as to not put undue load on the smartphone. The processing power, battery life and memory of the smartphone is limited as compared to a computer and so processing a large amount of data on the smartphone was less appealing to for the system design.

1.3.2 An important quality for the system is scalability. One part of scalability is supporting new

gestures (move/rotate/resize). This type of scalability is better achieved by processing the accelerometer data in the 3D drawing application and not by the smartphone.

1.3.3 It is an expectation these days that CVEs be very fast and responsive. This was only possible if the operations were done locally between the Android application and the 3D drawing application.

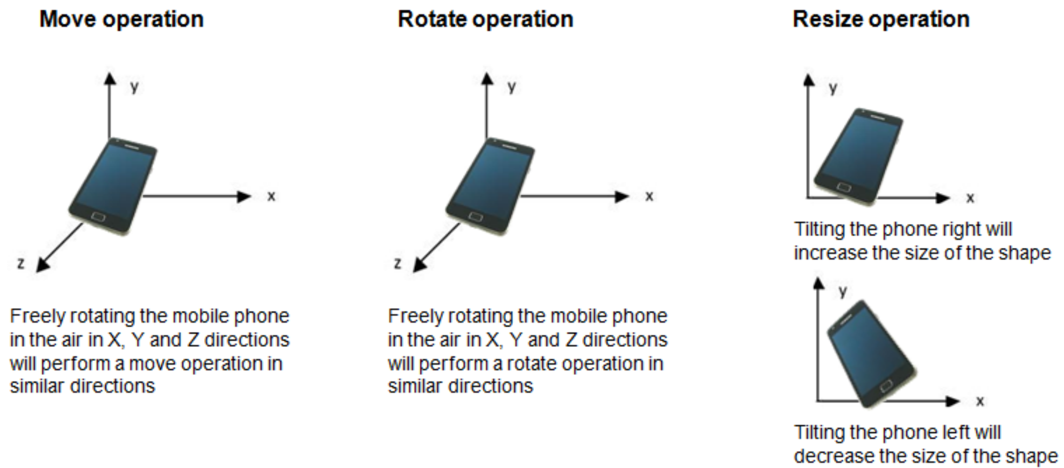

Fig. 3. Various gestures supported by the Android application to perform drawing operations on the shapes

1.4 Figure 4 shows the various commands sent by the Android application to the 3D drawing application when a user performs an action. The commands are queued by the 3D drawing application so that it wont miss the new command if it is busy doing a previous operation.

Operation	Command	Example usage
Add a new shape	add SHAPENAME COLOR	add circle red
Move a shape (is followed by sending accelerometer data)	move SHAPENAME	move circle1
Rotate a shape (is followed by sending accelerometer data)	rot SHAPENAME	rot circle1
Resize a shape (is followed by sending accelerometer data)	res SHAPENAME	res circle1
Delete a shape	del SHAPENAME	del circle1

Fig. 4. The commands used by the Android application to communicate with the 3D drawing application

2.2 3D drawing application

The 3D drawing application is a multiuser Windows based drawing application that supports basic 2D and 3D shapes and operations. To enable remote collaboration over a shared drawing activity, the 3D drawing application acts as a distributed application such that the multiple instances of this application

running at different places behave like a single shared instance. All the instances of this application are synchronized using the middleware server application. The Android application plays the role a toolbox and controlbox for the 3D drawing application. Any operation performed locally by a user is reflected globally. The 3D drawing application has the following capabilities:

2.1 Users are able to create a new drawing or load an existing drawing for a collaborative group activity by entering the unique drawing ID of the drawing activity.

2.2 Users are able to draw basic 2D and 3D shapes and perform basic operations on them such as transform, rotate, move and zoom-in/zoom-out. The 3D drawing application supports two modes of operation, the Android application mode in which the application is controlled by a smartphone running our Android application and the regular mouse mode in which the application is controlled by a mouse.

2.3 The 3D Drawing Application hosts a UDP listener thread that listens to any UDP packets received on the port number 12345. This is the port where the Android application sends the commands in form of UDP packets. Once a command is received, it is queued in the command queue. A separate dispatcher thread dequeues these commands from the command queue one by one and performs the associated operations. Once an operation is complete, the dispatcher thread alerts the server thread to notify other users about any changes. The dispatcher thread is also responsible for processing any new notifications it receives from the server thread (notifications from other users) and perform operations associated with these notifications locally. Another job of dispatcher thread is to notify the accelerometer thread to start receiving the accelerometer data after a move, rotate or resize command is received from the Android application. In such a case, the dispatcher thread goes to sleep until notified by the accelerometer thread that it has completed the operation. Henceforth, the dispatcher thread resumes its normal operation.

2.4 The 3D drawing application hosts a server thread which is responsible for sending any updates to the middleware server application as well as read any updates from it. The server thread listens to any new notifications from the dispatcher thread and updates other users about these notifications through the middleware server application. The server thread also continuously pings the middleware server application for any new notifications posted by other users. Once a new notification is received, it is passed on to the dispatcher thread to perform the associated operation locally.

2.5 The 3D drawing application hosts an accelerometer thread which is responsible for processing the accelerometer values after a move, rotate or resize command is received from the Android application and until the done command is received from the Android application. The dispatcher thread is responsible for notifying the accelerometer thread to start its function. The accelerometer thread, once activated, dequeues the accelerometer data from the command queue and performs the ”move” or ”rotate” or ”resize” operation according to the new data in real-time. Once the ”done” command is received from the Android application, the accelerometer threads wakes up the dispatcher thread and goes to sleep.

Figure 5 shows the various threads, their operations and interactions.

2.3 Middleware server application

The middleware server application is responsible for creation and maintenance of a global session to support a collaborative group activity over the web and for hosting global shared data to enable synchronization of the various instances of the 3D drawing application. We used a middleware server application rather than making one of the 3D drawing applications as a server for the following rea-

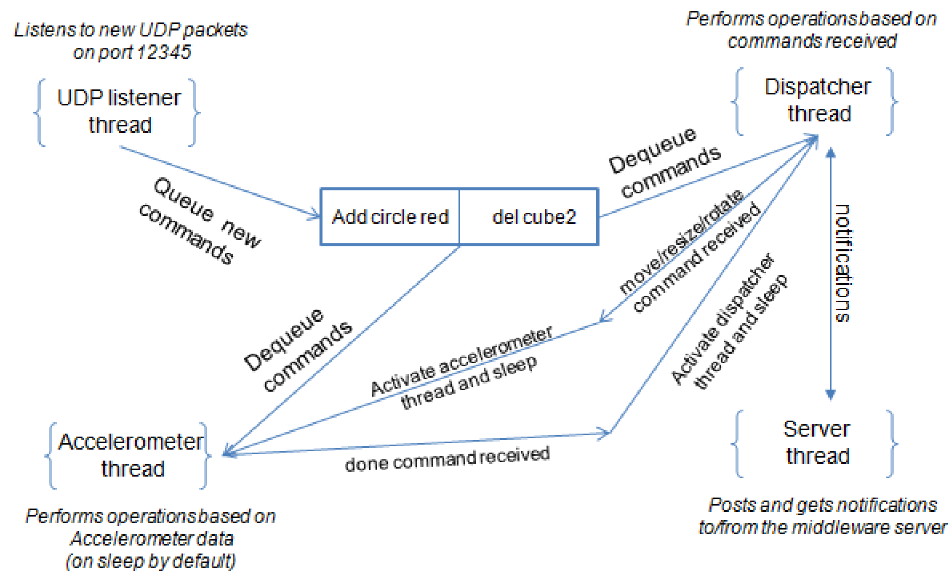

Fig. 5. Various threads, their operations and interactions in 3D drawing application

sons:

3.1 The system is not be dependent on any one user. This simplifies the logic for synchronization. In the system, users can leave the session at any time and the system will simply remove the user from the synchronization list.

3.2 The system is responsive and fast with a middleware server. Users that have a slow computer with a slow internet connection speed do not act like a bottleneck as they might in a distributed server system. Also, users' computers could suffer from power failures or loss of internet connectivity.

3.3 We wanted to save the sessions so that drawings could be loaded/resumed at a later time. It makes more sense to save this information at a central server rather than distribute it.

Moving on to the middleware server application, it maintains data in form of shared global objects. A shared global object is defined as a file/database entry on the server with the name as the name of the object and data as the data of the object. A global object is used to maintain data of 2D and 3D shapes as well as other important global shared data. A complete list of global objects are:

1. Usercount object to store number of users.
2. Userlist object to store list of all users (user ID's).
3. User object to store data of each user i.e. his ID, name and IP address of his computer.
4. Shape list object to store list of all shapes for the current session.
5. Shape object to store data of each shape with its shape id, 3D location(x,y,z co-ordinates) on the screen, size, color, angle and the current ticks (a long number) of the Coordinated Universal Time (UTC) clock in that order and in form of comma separated values.

Following are the various capabilities of the server application:

1. Create a new session with the given password. Command:
=createsessionid='unique session id'password='password'. Returns ack.

2. Create a new user, get the userid from the server and add it to the special 'usercount' object. Command:
cmd=createusersid='unique session id'name='name of the user'. Returns userid.

3. Create a new object and add it to the special 'list' object. Command:
cmd=createobjectsid='unique session id'oid='unique object id'addtolist=0:no,1:yes. Returns ack.

4. Delete an existing object and notify other users. Command:
cmd=deleteobjectsid='unique session id'oid='unique object id'uid='userid'. Returns ack.

5. Post new object data and notify other users about this new data. Command:
cmd=postsid='unique session id'oid='unique object id'data='data'ow=0:append new data to exisiting data, 1:overwrite exisiting datanotify=0:no,1:yes. Returns ack.

6. Get new object data and delete the data if accessed by all users. Command:
cmd=getsid='unique session id'oid='unique object id'ret='no. of characters to return or 0 for all characters'erase=0:do not erase old data,1:erase old data. Returns all or 'ret' characters separated by ' ' character.

7. Get notifications for the given user. Command:
cmd=getnotificationssid='unique session id'uid='user id'. Returns notifications.

8. Lock an object. If already locked, add requested user to lock array. Command:
cmd=locksid='unique session id'oid='unique object id'uid='userid'. Returns userid if success or returns the userid of the user having the lock.

9. Unlock an object and assign lock to the next user in the lock array. Command:
cmd=unlocksid='unique session id'oid='unique object id'uid='unique user id'. Returns 'unlocked' if no other users are in the queue or returns the userid of the next user in the queue that gets the lock.

10. Delete a user and all locks held by this user. Command:
cmd=deleteusersid='unique session id'uid='userid'name='name of the user'. Returns ack.

11. Delete session and all objects after all users have been deleted: Command:
cmd=deletesessionsid='uniquesessionid' password='password'. Returns list of all objects deleted.

We now explain how we recognize different gestures from the raw accelerometer data.

3 Recognizing gestures from raw accelerometer data

The accelerometer values returned by the Android OS are between -10 to + 10 for any given axis X,Y or Z. Negative values signifies positive X,Y or Z direction and positive values signifies negative X,Y or Z direction. As soon as the Android application starts sending these accelerometer values to the 3D drawing application after a move, rotate, or resize command, the accelerometer thread starts processing the accelerometer data and invokes the move, resize, or rotate function according the new accelerometer data. We describe here the workings of move, rotate and resize gestures.

3.1 Move gesture

To account for a movement in a given direction (positive or negative) for a given axis (X,Y or Z), we see if the average of last 5 values of accelerometer data for that direction (avgX, avgY or avgZ) was opposite to our given direction. If it was opossite, we move the shape by avgX*R positions in that direction. R is a factor dependent on the 3D drawing application screen size and is taken into account to account for a large change in shape movement at once. The bigger the screen size, more should be this factor. We propose that the value of R can be increased linearly by a magnitude of 1 with increase in the aspect ratio of the screen viz. R can be 2 for screens with aspect ratio of 800x600, 3

for 1024x768 and so on.

For example : For movement in positive Y direction, the average of last 5 values of accelerometer X should be negative.

To control the mouse pointer, we follow the same procedure and call the mouse move function with avgX and avgY as inputs instead of the shape move function.

3.2 Rotate gesture

To account for a rotation in a given direction (positive or negative) for a given axis (X, Y or Z), we calculate the net movement between any two initial and final points of settlement. Point of settlements are when the mobile phone is in a stable position and not moving. We save the initial accelerometer values of the phone in IX, IY and IZ. We then start taking the average of last 5 values of the accelerometer data for each direction till the final settlement point and convert these average values to an equivalent angle using the following formula:

$$\mathrm{ConvertedValue} = ((\mathrm{ValueToConvert} - \mathrm{a1})/(\mathrm{a2} - \mathrm{a1})) * (\mathrm{b2} - \mathrm{b1}) + \mathrm{b1};$$

Here, a1, a2 are the lower and upper bounds of the accelerometer values and b1, b2 are the lower and upper bounds of the shape angle i.e. a1 = -10, a2 = +10, b1 = -360, b2 = +360

So a value of -5 on the current X or Y or Z axis of the accelerometer (denoted by CX, CY or CZ) would turn out to be an angle -180 in that direction. We then subtract the absolute value of these current angles from the absolute value of the angles corresponding to the initial accelerometer values in that direction IX, IY and IZ. We call the new values as final values denoted by FX, FY or FZ.

FX = ABS(angle(IX)) - ABS(angle(CX))

FY = ABS(angle(IY)) - ABS(angle(CY))

FZ = ABS(angle(IZ)) - ABS(angle(CZ))

If the values CX, CY or CZ are negative, we rotate the selected shape in the positive direction and vice versa by a factor of the final values that we calculated FX, FY and FZ. We now replace IX, IY and IZ with FX, FY and FZ as the new initial values.

IX = FX

IY = FY

IZ = FZ

3.3 Resize gesture

For a resize gesture, we only take into account the average of the last 5 values of the current X parameter of the accelerometer data and ignore the Y and Z parameters. We do this, because the resize gesture is defined such that tilting the phone to right increases the size of the selected shape and tilting the phone to left decreases the size of the selected shape. Only one orientation parameter is sufficient to achieve this. We chose X because it was physically easier to manipulate the smartphone in this direction. If the average of the last 5 values of X are negative and less than -2, we increase the size of the shape by 1. If the average of the last 5 values of X are positive and greater than 2, we decrease the size of the shape by 1. We do not account for X values between -2 to 2 to ignore any small movements in the smartphone i.e. if the values of X are inside the bounds of -2 and 2, the resize gesture ignores them. This is because values inside the bounds of -2 and 2 account for small changes in the smartphone orientation with respect to the stable initial position of X=0.

We now explain the concept of SUMD.

4 Synchronicity using minimal data

While developing the 3D drawing application, one major challenge was to synchronize its various instances distributed over the web almost instantly and without consuming a large amount of network bandwidth. SUMD is a concept we introduce here that enables fast and instantaneous synchronization of the various instances of the 3D drawing application running on different computers across the web using a minimum amount of data. We anticipate that the approach can be applied to synchronize many different types of distributed applications setup in a collaborative virtual environment over the web.

There are alternative approaches to SUMD to achieve fast synchronization of the various 3D drawing application instances, but they do not provide as many advantages as SUMD. One approach is to have a single instance of the 3D drawing application running on a central server and all the users interact with this single instance. In such a case, the screens of the various users would have been an exact replica of the screen of this single instance. But the disadvantage of such an approach is that the system would have to receive commands from various users, process them in real-time, and perform an action. Then it broadcasts the screenshot (image) of the 3D drawing application every few hundred milliseconds back to the users. This would increase network traffic and server load and in turn cause delays in the synchronization process. Another alternative to achieve synchronization is to send each shape in a given instance of the 3D drawing application in a serialized form to other instances of the 3D drawing application through a central server application. This consumes a significant amount of bandwidth apart from having the overhead of serializing the shape before sending.

SUMD achieves fast synchronization by allowing the 3D drawing application to run locally on each of the users computer and synchronize these various instances using as little data as possible. SUMD is designed as follows:

1. Every drawing activity performed on a given instance of the 3D drawing application such as create a shape, move a shape etc. is posted to the server in form of a global notification. In other words, the ID of the updated shape is appended to the global notification object along with an access number whose initial value is equal to the number of users. Each user then reads these notifications from the global notification object and subtracts a 1 from the access number for the notification it just read. When the access number reads 0, we delete that particular notification from the queue. This approach assures that each user is up to date with all the notifications and never misses any of the notification. This makes the system robust.

2. Each notification posting is followed by updating the associated shape data on the server. In other words, the system updates the global shared shape object of the updated shape with the new data. So as soon as the user reads a notification, it knows which shape was updated and can then request the server for the new data of the updated shape. An advantage of this approach is that the users gets only the most important and relevant updates.

3. The system maintains the characteristics of each shape with as minimum data as possible. For example: To notify other users that Circle2 has a new location, we post c2 as a notification to the global notification object and update the Circle2 global shared object with new data: C2,1,3,0,2,r,32. The data means circle2 with X-axis 1*10=10, Y axis 3*10=30, Z axis 0*10=0, size 2, color red and angle 32. A point here to note is that before updating new data on the server, we divide X, Y and Z axis data by 10. After receiving new data from server, we multiply the X,Y and Z axis data with 10. This reduces the packet size that is sent to the server. On a long and continuous run of the system and with increase in number of users, this approach helps reduce the time to post and get new notifications to and from the server by a few hundred milliseconds.

We now explain the concept of remote semaphores and deadlock avoidance in them.

5 Remote semaphores and deadlock avoidance in them

With synchronization came the problem of inconsistency in the global shared data when multiple users were trying to update the same shape at the same time. The data was being lost and users were not able to see each others changes resulting in confusion and waste of efforts. One obvious way that we thought of solving this problem was to have a kind of locking mechanism that should protect these shared resources (shapes) from being accessed at the same time. The challenge that we faced was that who should get access to the requested resource first (shape in our case) when it is actually located on a different machine (middleware server application in our case) and being shared by a number of users having different time zones and different internet connection speeds and hence delays are inevitable. We solved this problem using the concept we introduce here, remote semaphores.

Remote semaphores are essentially a locking mechanism over global shared resources. The middleware server application provides 'lock' and 'unlock' commands to achieve locking and unlocking of global shared objects respectively. Whenever a user requests the middleware server application to lock a global shared object (shape), he is added to the lock array maintained by the middleware server application. The middleware server application maintains an internal lock array object for all the shapes. The lock array object contains the user id's of the users who requests a lock on a given shape. Locks are granted in the ascending order of the array index. In order to justify the users who demands the lock first but due to a different time zone, slow internet speed and network delays are unable to get the lock, we add an additional parameter to the lock request - the current time ticks of the Coordinated Universal Time (UTC) clock. Since UTC is the primary time standard by which the world regulates clocks and time, we used the ticks of its UTC clock to account for any differences in the time zones. On receiving the lock request, the server application adds the user id and the ticks (a long number) to the lock array and sorts the array according to ticks. Before granting a lock to the user at the starting index of the array, the server application waits for atleast 'n' seconds to determine if any other lock request is on its way. 'n' is calculated as follows:

$$\mathrm{n = x - (UTC.now.ticks - user.ticks)/10000}$$

This 'n' signifies the maximum relative delay in the system for any two given users and is calculated as the difference between 'x' and the time taken for the request of last user to reach to the server. We calculate the time taken in milliseconds for a request to reach to the server by subtracting the ticks in the user lock request from the current UTC clock ticks and dividing by 10000 (the number of ticks per milliseconds). And, 'x' is the maximum time in milliseconds that a request from the user with slowest internet connection speed, farthest distance, or bad network can take to reach the server application.

We calculate the value of 'x' dynamically for each session as follows:

1. When a user joins a session, we send several lock requests (10 in practise) to lock the special "dummy" object (maintained by the server application) and calculates the average time it takes for these requests to reach the server. This average time is saved on the server as a special "delay" object.

2. If a user joins a session and the average time for his requests to reach the server is greater than the value in the special "delay" object, we replace the value of the special "delay" object with that of this new user.

So we actually wait for 'n' milliseconds after getting the first lock request and before granting a lock to this request. This approach is fair to the slowest user in the system. On an average and after

several experiments, we noticed that the value of 'x' is usually between 0 and 5000 milliseconds. One disadvantage of this approach though is that if there is a very slow user in the system, the entire system suffers from large delays in getting locks. In our system testing, we used 'x'=0 because we wanted the system to be as fast as possible. The system worked perfectly well and slow users were ok with the first come first served approach to get locks.

To understand the concept of remote semaphores more deeply, let us consider a real life scenario. Assume that there are 3 users who are in different time zones and have different internet speeds and who are collaborating over a shared drawing activity using the 3D drawing application. The value of 'x' for these users is 100, 150 and 300 milliseconds. Also, let us assume that there are currently 3 shapes on the screens of these users : Circle1, Cube1 and Square1. User 1 wants to move Circle1 and User 2 wants to change the color of Circle1. User 2 has a slower internet connection than User 1. User 2 selects Circle1 and his request takes 150 milliseconds to reach to the server. User 1 selects Circle1 30 milliseconds after User 2 selects it, and his requests takes 100 milliseconds to reach the server. Now in the ideal case, User 2 should get the lock on Circle 1. But due to a slower internet connection, the lock request of User 2 reaches 20 milliseconds after the lock request of User 1. But thanks to the remote semaphores, User 2 still gets the lock. This is because the server waited for 300-100=200 milliseconds more after receiving the request of user 1.

Let us now address the issue of *deadlocks* in remote semaphores. There were situations when users who held the lock suddenly went offline due to a number of reasons such as loss of internet connectivity, power failures, switching to other application and then forgetting to switch back to 3D drawing application etc. This resulted in the exit button on the 3D drawing application not being pressed and hence the "delete user" function on the middleware server not being called either. In such a case, the middleware server application assumed that the user who went offline still held the lock and this resulted into permanent locking of that shape or in other words a deadlock. To address this issue, we propose two simple methods:

1. Status update In this approach, the middleware server application tries to contact the 3D drawing application after every few seconds to check its status. If there is no response, the middleware server application automatically deletes the user after a few unsuccessful attempts.

2. Automatic unlocking In this approach, the middleware server maintains a counter for each locked shape. The initial value of counter could be set according to application needs. We set it to 1 minute for the 3D drawing application, because 1 minute is sufficient for a user to hold a lock on a particular shape. After the counter expires, the lock is given to the next user in the lock array.

We now present the results of the evaluation of our system.

6 Evaluation

We evaluated our system with 10 users collaborating over a shared drawing activity. Among these users, 6 were located in the same city/surrounding area out of which 4 used the same internet connection. Out of the rest 4 users, 2 were located in a different state and 2 were located in a different country. Figure 6 shows the complete details of these users i.e. their locations, internet connection type, computer's configuration and smartphone type. We hosted our server application on a shared Linux hosting based server. The results are given below:

1. The average time taken to add, move, rotate, resize, or delete a shape from 3D drawing application via the Android application (local UDP packet transfer time) was between 30-35 milliseconds.

2. The average time taken by the 3D drawing application to post new notifications to the server

Location	Internet connection type	PC type	Smartphone
Urbana, IL, USA	DSL	UIUC Server: Intel Xeon/32 GB RAM	Used mouse
Urbana, IL, USA	DSL	Intel Core i7/4 GB RAM	Samsung Galaxy S2
Urbana, IL, USA	DSL	Intel Core i7/4 GB RAM	Samsung Galaxy S2
Urbana, IL, USA	DSL	Intel Core2Duo/2 GB RAM	Samsung Galaxy
Urbana, IL, USA	DSL	Intel Pentium dual core/1 GB RAM	Used mouse
Champaign, IL, USA	DSL	AMD Phenom II/3 GB RAM	Google Nexus
San Francisco, California, USA	Broadband	Intel Quadcore/3 GB RAM	HTC one
San Francisco, California, USA	Broadband	Intel Core2Duo/2 GB RAM	Motorola Droid Razr
Amritsar, Punjab, India	Broadband	Intel Pentium dual core/1 GB RAM	Samsung Galaxy S2
Amritsar, Punjab, India	Broadband	Intel Pentium 4/512 MB RAM	Used mouse

Fig. 6. Details of the users who participated in the system evaluation test. The red label signifies that the users used the same internet connection.

(upload time) was between 100-200 milliseconds for users located in USA and between 300-600 milliseconds for users located in India. This difference in times of users in these two countries was because the middleware server application was hosted in USA and because the internet connection speed of the users in India was not very good.

3. The average time taken by the 3D drawing application to get new notifications from the server (download time) was between 200-500 milliseconds for users located in USA and between 500-1000 milliseconds for users located in India. This difference was again because the middleware server application was hosted in USA and because the internet connection speed of the users in India was not very good.

4. We noticed that the upload time and download time was directly dependent on the internet connection speed. The users with a better internet connection speed had a smaller upload and download time and which is quite obvious (differed by a few hundred milliseconds). So the UIUC server computer having the fastest DSL connection of 1 Gbps had the lowest upload (80 milliseconds) and download time (100 milliseconds). And the internet connection speed had no effect on the local UDP packet transfer time and which is quite obvious as well.

5. We noticed that PC type had no effect on the upload time, download time and the local UDP packet transfer time because the data transfers between the system is very little due to SUMD and hence very little processing is required by the users computers.

6. We noticed that smartphone type also had no effect on the upload time, download time and the local UDP packet transfer time because of the similar reason as stated in 5.

7. We noticed that increasing the number of users had a direct effect on the upload times and download times and these times increased by a few milliseconds (10-30 on an average) with with

every new user. This is because the middleware server application being hosted on a shared hosting server was not able to handle multiple requests at the same time very effectively. We anticipate that the server application if hosted on a dedicated server will definitely increase the efficiency of the system in terms of handling more multiple requests at the same time and hence decreasing the average latency per user addition. The increase in number of users had of course no effect on local UDP packet transfer time because the smartphone is connected to the 3D drawing application via a local WiFi connection. The results are shown in form of a graph in figure 7.

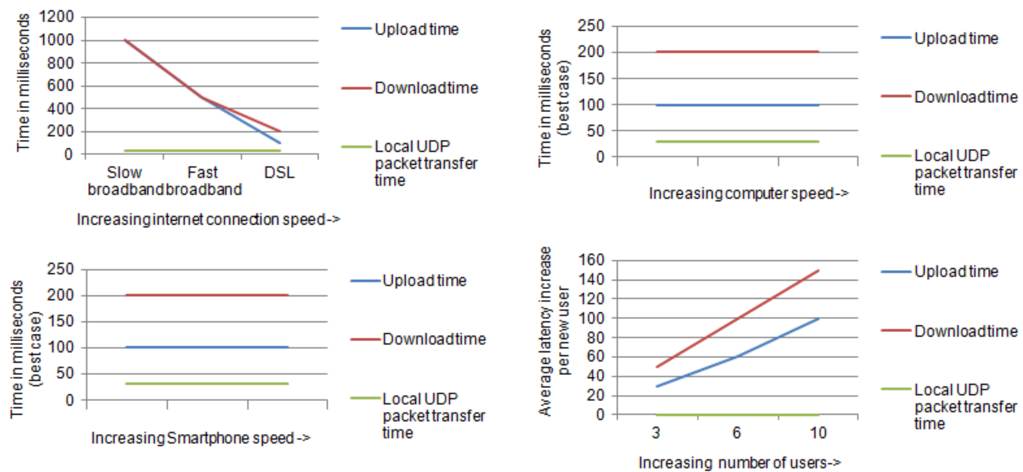

Fig. 7. Graphs showing relationship of upload time, download time and local UDP packet transfer time with internet connection speed, computer processor's speed and smartphone's processor speed respectively for 10 users using the system at the same time.

8. We did a load testing of our system in which all the 10 users quickly and continuosly added new shapes and moved/rotated/resized them one after the other. The system worked well and the users were able to see new shapes appearing on their screens as they were busy moving others. This shows that the SUMD and remote semaphores approach with deadlock avoidance works well even during high loads.

7 Related work

7.1 3D Input Recognition In Contemporary Smartphones

Niezen [9] and He [6] made use of accelerometer values to successfully recognize many different gestures made on smartphones and with good accuracy. Hinckley *et al.* [7] used these mobile touch and gestures to capture 3D inputs. Hutama *et al.* [8] developed a gesture recognition system to recognize English letters drawn in the air.

7.2 Manipulation Of Real And Virtual Environments Using Smartphone Gestures

MagicPhone and EasyPointer used accelerometer and gyroscope sensors to control appliances and facilitate physical pointing [15] [10]. Peters *et al.* [11] established smartphone gestures to control home and office fixtures including lights, window blinds, cooling and ventilation units. Tilt and Touch

allowed smartphones to tilt viewport, scale viewport, and rotate objects in a virtual environment on a computer [3].

7.3 Using Smartphones For Real-Time 3d Interaction On Large Displays

Finke *et al.* [4] and Yoon *et al.* [16] used smartphones manipulate 3D content on large displays.

7.4 Using Smartphones As A 3d Input Device For Real-Time Collaboration In CVEs

Tang *et al.* [13] successfully developed a method to interact with 3D objects on smartphones using a client server architecture and later extended the work to a CVE on a smartphone.

8 Conclusion

ConDroid is a system that uses an Android based smartphone as a 3D input controller for a 3D drawing application setup in a collaborative virtual environment over the web. The system design is an effective approach to enable use of smartphones as a 3D input device for a desktop application. Synchronization of multiple instances of this desktop application running on different computers connected over the internet through a middleware server application is facilitated by the concepts of SUMD (synchronicity using minimal data) and remote semaphores with deadlock avoidance. Our system was tested with 10 users located in different parts of the world and having different internet connection speeds, computing powers and smartphones. The results were promising and users were able to effectively control the 3D drawing application through their smartphones. They were able to see the changes made by other users almost instantly on their computer screens. Our approach is useful to those who wish to control desktop applications using their smartphones as well as those who wish to develop a CVE application with fast and effective synchronization mechanism.

References

1. 3Dconnexion: A 3D Mouse. `http://www.canalys.com/newsroom/smart-phones-overtake-client-pcs-2011`, 2008. [Online; accessed 18-February-2012].
2. J. Cashion, C. Wingrave, and J. J. LaViola Jr. Dense and dynamic 3d selection for game-based virtual environments. *Visualization and Computer Graphics, IEEE Transactions on*, 18(4):634 –642, april 2012.
3. Y. Du, H. Ren, G. Pan, and S. Li. Tilt touch: mobile phone for 3d interaction. In *Proceedings of the 13th international conference on Ubiquitous computing*, UbiComp '11, pages 485–486, New York, NY, USA, 2011. ACM.
4. M. Finke, N. Kaviani, I. Wang, V. Tsao, S. Fels, and R. Lea. Investigating distributed user interfaces across interactive large displays and mobile devices. In *Proceedings of the International Conference on Advanced Visual Interfaces*, pages 413–413. ACM, 2010.
5. V. Frati and D. Prattichizzo. Using kinect for hand tracking and rendering in wearable haptics. In *World Haptics Conference (WHC), 2011 IEEE*, pages 317–321. IEEE, 2011.
6. Z. He, L. Jin, L. Zhen, and J. Huang. Gesture recognition based on 3d accelerometer for cell phones interaction. In *Circuits and Systems, 2008. APCCAS 2008. IEEE Asia Pacific Conference on*, pages 217–220. IEEE, 2008.
7. K. Hinckley and H. Song. Sensor synaesthesia: touch in motion, and motion in touch. In *Proceedings of the 2011 annual conference on Human factors in computing systems*, CHI '11, pages 801–810, New York, NY, USA, 2011. ACM.
8. W. Hutama, P. Song, C.-W. Fu, and W. B. Goh. Distinguishing multiple smart-phone interactions on a multi-touch wall display using tilt correlation. In *Proceedings of the 2011 annual conference on Human factors in computing systems*, CHI '11, pages 3315–3318, New York, NY, USA, 2011. ACM.

9. G. Niezen and G. Hancke. Gesture recognition as ubiquitous input for mobile phones. In *DAP 2008: Proceedings of the Workshop on Devices that Alter Perception*, 2008.
10. G. Pan, H. Ren, W. Hua, Q. Zheng, and S. Li. Easypointer: what you pointing at is what you get. In *Proceedings of the 2011 annual conference extended abstracts on Human factors in computing systems*, CHI EA '11, pages 499–502, New York, NY, USA, 2011. ACM.
11. S. Peters, V. Loftness, and V. Hartkopf. The intuitive control of smart home and office environments. In *Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software*, ONWARD '11, pages 113–114, New York, NY, USA, 2011. ACM.
12. Z. Tang, O. Ozbek, and X. Guo. Real-time 3d interaction with deformable model on mobile devices. In *Proceedings of the 19th ACM international conference on Multimedia*, pages 1009–1012. ACM, 2011.
13. Z. Tang, G. Rong, X. Guo, and B. Prabhakaran. Streaming 3d shape deformations in collaborative virtual environment. In *Virtual Reality Conference (VR), 2010 IEEE*, pages 183–186. IEEE, 2010.
14. C. A. Wingrave, B. Williamson, P. D. Varcholik, J. Rose, A. Miller, E. Charbonneau, J. Bott, and J. J. LaViola Jr. The wiimote and beyond: Spatially convenient devices for 3d user interfaces. *Computer Graphics and Applications, IEEE*, 30(2):71 –85, march-april 2010.
15. J. Wu, G. Pan, D. Zhang, S. Li, and Z. Wu. Magicphone: pointing interacting. In *Proceedings of the 12th ACM international conference adjunct papers on Ubiquitous computing*, Ubicomp '10, pages 451–452, New York, NY, USA, 2010. ACM.
16. D. Yoon, J. Lee, K. Yeom, and J. Park. Mobiature: 3d model manipulation technique for large displays using mobile devices. In *Consumer Electronics (ICCE), 2011 IEEE International Conference on*, pages 495–496. IEEE, 2011.