The way of the motion based remote controller

involving bonjour services, udp communication, event injection, motion sensing, image generation, pixelfonts and cross platform development

by Milan Toth
http://milgra.com
milgra@milgra.com

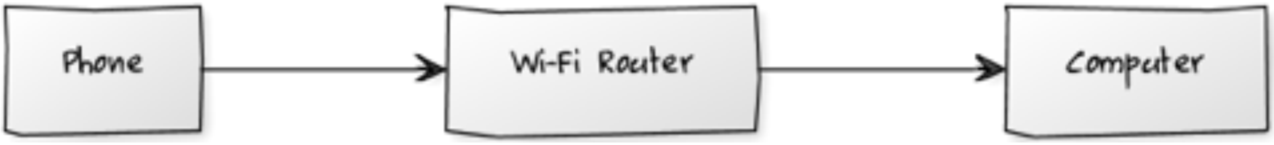
Table of contents

Preface

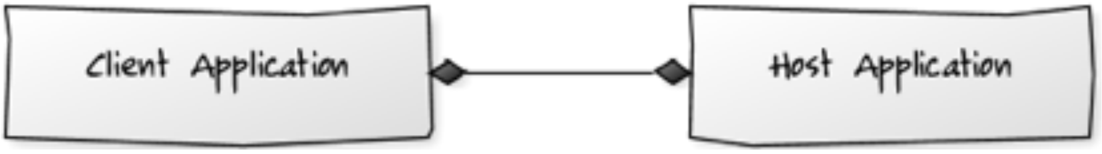
Last year I've tried to create a nintendo wiimote-simulator application for the iPhone, that controls first person shooters running on desktop machines. I put a lot of effort in it, i've emerged deeply in udp socket programming, created a simple udp communication protocol, learned how to use bonjour services, event inejction and accelerometer/gyroscope data, and I want to share this journey with you.

Designing the application

The application detects motion events on the phone, transfers these events to the desktop machine over a wi-fi router, and the host running on the desktop machines injects mouse events in the operation system's event flow based on motion events.

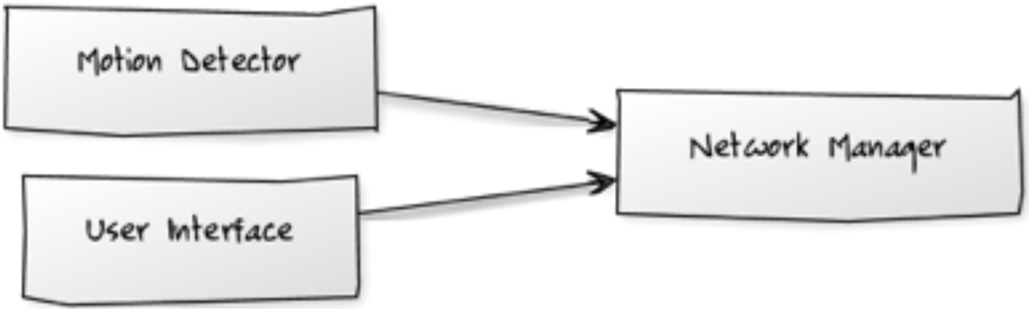


So two applications needed : one client application for the phone, and one host application for the desktop system.

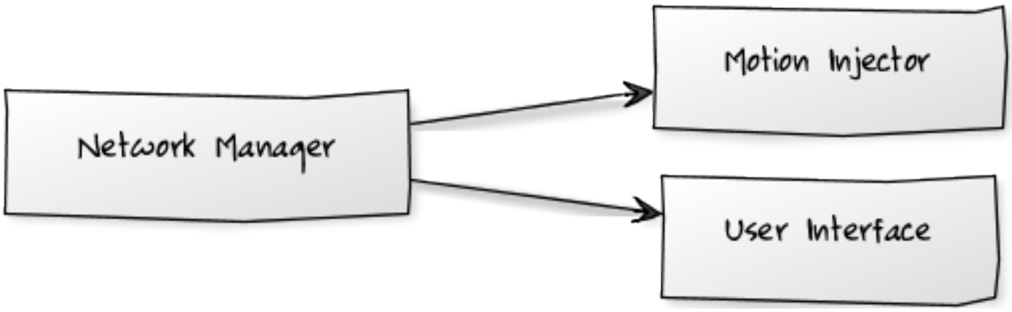


Let's dive in the application hierarchy. What modules do we need?

On the phone (client) side, the most important part is a Motion Detector that detects motion events. To send these events we need a Network Manager that finds the host, connects to the host and sends events continuously. We need a UI also that triggers connection and button events.



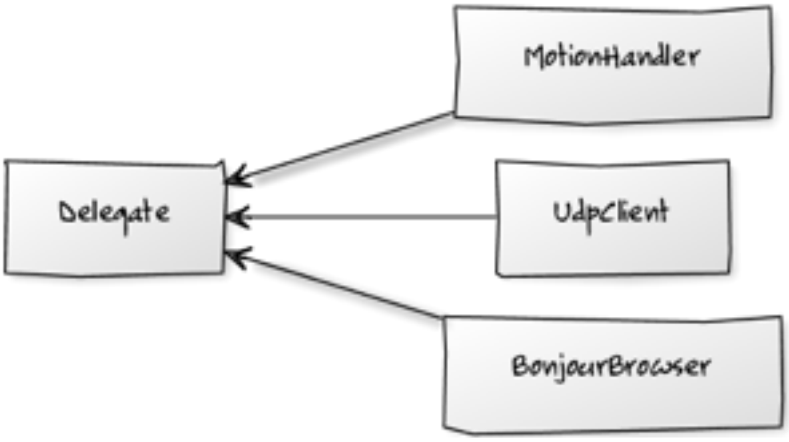
On the desktop (host) side, we need a "mirrored" client application, so there will be a Motion Injector that injects mouse events, a Network Manager that tells everyone that we are there, and accepts connections, and receives events. And the UI also needed for visual feedback.

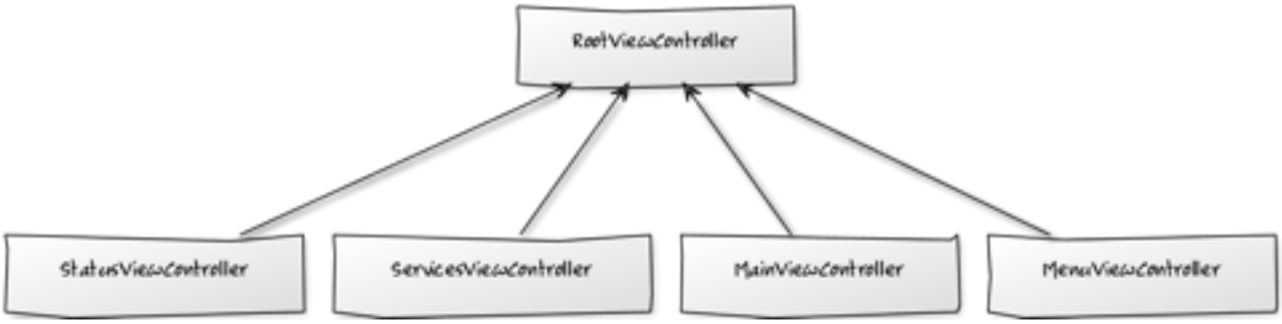


Diving deeper

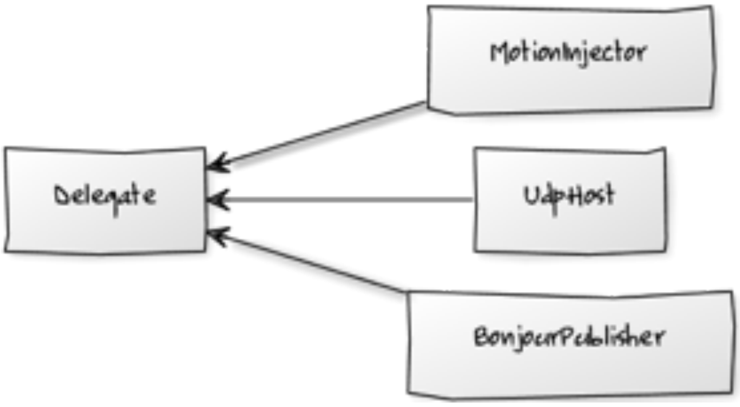
It seems simple now, but on Objective-C and system framework level it needs a little more abstraction. Let's see the client side.

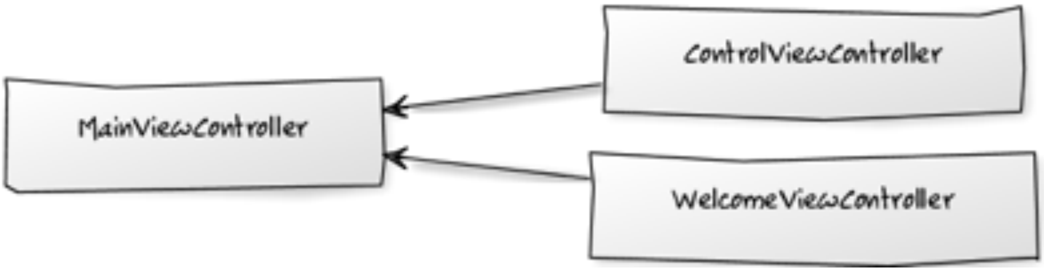
The mother of all will be the Delegate class, because it is mandatory for the UIApplication in iOS, it contains the entering point. There will be a Root View Controller, as a Navigation Controller, that will present the view controllers needed for the whole application : the Menu View Controller for the starting menu, the Main View Controller with the control buttons, the Services View Controller to select one host if there is more than one available, and a Status View Controller which shows if there is something went wrong.
State changes based on connection state will be complicated, so we need a StateHandler class, we need the MotionHandler class discussed in the previous part as Motion Manager, and the Network Manager Module will be UdpClient and BonjourBrowser classes.





On the host side there will be the Delegate, the MainViewController with two views : the Welcome view and the Control view. There will be the ButtonInjector and MotionInjector classes for event injection, and UdpHost on the other side of UdpClient, and a StateHandler also. A BonjourPublisher class is also needed.





But how do we connect all these instances? We create them right in the Delegate, and they will communicate with the delegate with the Event-Delegate model, and the Delegate dispatches the event to their places. I've created a general EventDelegate protocol for this, because it is generic and lightweight.

There are many event delegation paradigms out there, using the NSNotificationManager is the most common, but I don't like it because I cannot see behind it, and it is surely slower than a direct event-delegate call, and according to the documentation events can be lost, the other common paradigm in apple objective-c is creating a sharedDelegate on the app's delegate, and send everything directly in it, which is inelegant, and causes forward declarations, and specific event listeners/protocols need too much programming, although cannot be avoided in some cases.

The only thing left is figuring out the network communication. In case of a real-time controller, the effect should be immediate. What kind of internet protocols do we have to choose from? TCP and UDP. For real-time communication TCP is too slow, because there is a lot of forward-backward communication going on to ensure that every packet is arrived to the other side. But UDP is ideal for our quest, the only addition we should do in our communication layer is to create some high-priority packets, whose arrival to the other side is ensured.

So we stick with UDP communication. What API should we use for them? Since iOS and OS X are both unix based systems, and Objective-C is a superset of C, the best way is to use BSD sockets. (BSD sockets - serving you since 1982!!).

Starting to code

The client application will use iOS frameworks, the host application uses OS X frameworks. They have a lot in common, but they are not interchangeable.

We try to create as much cross platform code as possible. The less files/classes you have, the easier to understand and maintain your code, so we will use shared osx / ios classes where possible.

Create a Workspace for both projects
Create the iOS project
Create the OSX project

Creating the UDP socket

The most important part in both applications is the UDP communication. Let's create the UdpSocket class that will be used by both applications.
It should be able to open sockets on IPV4 and IPV6 networks, so the constructor should be

```
- ( id    ) initWithFamily  : ( uint ) theFamily
           withDelegate    : ( id   ) theDelegate
```

Retaining the delegate, storing the length of an ipv4 and a ipv6 socket address for later use :

```
delegate      = [ theDelegate retain ];

ipv4Length    = sizeof( struct sockaddr_in );
ipv6Length    = sizeof( struct sockaddr_in6 );
```

Storing socket family, creating the socket.

```
socketFamily = theFamily;
serverSocket = socket(  socketFamily ,
                        SOCK_DGRAM   ,
                        IPPROTO_UDP  );
```

If socket creation was succesful, we have to prepare local addresses to bind the socket to.

```
if ( serverSocket > -1 )
{

    if ( socketFamily == AF_INET6 )
    {
```

Trying to switch off IPV6 only feature to support IPV4 tunneling if possible.

```
        int option    = 0;
        int setStatus = setsockopt( serverSocket ,
                                    IPPROTO_IPV6 ,
                                    IPV6_V6ONLY  ,
                                    &option ,
                                    sizeof option );

        if ( setStatus == -1 ) NSLog( kUdpSocketTunnelError );
```

Create an IPv6 address on localhost, on an arbitrary port, and zero out other values.

```
        struct sockaddr_in6 serverAddress6;

        memset( &serverAddress6 , 0 , sizeof( serverAddress6 ) );

        serverAddress6.sin6_family = AF_INET6;
        serverAddress6.sin6_addr   = in6addr_any;
        serverAddress6.sin6_port   = htons(0);
```

Store address in a `struct sockaddr_storage,` which is a universal address storage for IPv4 and IPv6 addresses.

```
        serverAddress = *( ( struct sockaddr_storage* ) &serverAddress6 );

    }
    else
    {
```

Creating an IPv4 address on localhost, on an arbitrary port, zeroing out other values.

```
        struct sockaddr_in serverAddress4;

        memset( &serverAddress4 , 0 , sizeof( serverAddress4 ) );

        serverAddress4.sin_family       = AF_INET;
        serverAddress4.sin_addr.s_addr  = INADDR_ANY;
        serverAddress4.sin_port         = htons(0);
```

Storing address

```
        serverAddress = *( ( struct sockaddr_storage* ) &serverAddress4 );

    }
```

Binding address to socket, the OS will assign a free port.

```
    int bindStatus = bind( serverSocket ,
                           ( struct sockaddr* ) &serverAddress  ,
                           socketFamily == AF_INET6 ? ipv6Length : ipv4Length );
```

If binding was successful, getting port by overwriting present server address with binded address. `getsockname` needs proper `addressLength` for the given server address.

```
    if ( bindStatus != -1 )
    {

        socklen_t addressLength = socketFamily == AF_INET6 ? ipv6Length : ipv4Length;
        int addressStatus = getsockname( serverSocket ,
                                         ( struct sockaddr* ) &serverAddress ,
                                          &addressLength );
```

If getsockname was successful, getting port.

```
        if (  addressStatus != -1 )
        {

            if ( serverAddress.ss_family == AF_INET  ) port = htons( ( ( struct
sockaddr_in*  ) &serverAddress )->sin_port  );else
```

```
            if ( serverAddress.ss_family == AF_INET6 ) port = htons( ( ( struct
sockaddr_in6* ) &serverAddress )->sin6_port );
```

In other cases, log error messages.

```
        }
        else NSLog( kUdpSocketPortError );

    }
    else NSLog( kUdpSocketBindError );

}
else NSLog( kUdpSocketCreateError );
```

Let's create the listening method. It takes a packetsize parameter what tells the class how big packets to receive from the other peer. If the UdpClient is active we store the packet size, and detach a new thread. If you like Grand Central Dispatch, feel free to use it, the code was created before GCD.

```
- ( void ) listenWithPacketSize : ( uint ) theSize
{

    if ( !active )
    {

        packetSize = theSize;

        [ NSThread  detachNewThreadSelector    : @selector(read)
                    toTarget                   : self
                    withObject                 : nil ];

    }

}
```

Let's see the read function, which runs on a separate thread.

```
- ( void ) read
{
```

Activating UdpClient.

```
    active = YES;
```

Since recvfrom is a blocking function and blocks the thread until sufficient data comes, we can put it in an infinite while loop.

```
    while ( active )
    {
```

Preparing input buffer and input address, initializing `addressLength` because `recvfrom` has to know the length of the address format.

```
        char                    buffer[ packetSize ];
        struct sockaddr_storage address;
        socklen_t               addressLength = sizeof( address );
```

Zeroing out address.

```
        memset( &address , 0 , sizeof( address ) );
```

Receiving data through `serverSocket`, `recfrom` puts incoming data in `buffer`, it puts `packetSize` length of data in `buffer`, without flags, it puts sender's address into `address`, which has a length of `addressLength`.

```
        long sizeIn = recvfrom( serverSocket                              ,
                                buffer                                           ,
                                packetSize                                       ,
                                0                                                ,
                                ( struct sockaddr* ) &address   ,
                                &addressLength                            );
```

If End Of File received, close UdpClient.

```
        if ( sizeIn == -1 )
        {

            if ( !closed )
            {
                [ delegate readError ];
                [ self close ];
            }

        }
        else
        {
```

Count received bytes and packets, and dispatch event.

```
            bytesIn     += sizeIn;
            packetsIn   += 1;

            [ delegate  dataArrived : buffer
                        fromAddress : address
                        withLength  : addressLength ];

        }

    }

}
```

And let's see how will we send data to the other peer. The function gets the raw bytes, the size of the byte array, and the remote address to send the data.

```
- ( void )  sendBytes   : ( char* ) theBytes
            withSize    : ( uint  ) theSize
            toAddress   : ( struct sockaddr_storage ) theAddress
{
```

Sending data through `serverSocket`, `theBytes` is the outgoing data, `theSize` is its size, without flags, `theAddress` is the remote address, and the last parameter is the address length.

```
    long sizeOut = sendto( serverSocket                      ,
                           theBytes                          ,
                           theSize                           ,
                           0                                 ,
                           ( struct sockaddr*  ) &theAddress ,
                           socketFamily == AF_INET6 ? ipv6Length : ipv4Length );
```

Counting outgoing bytes and packets.

```
    bytesOut    += sizeOut;
    packetsOut  += 1;
```

If End Of File received, close UdpClient, else do nothing.

```
    if ( sizeOut == -1 )
    {

        [ delegate sendError ];
        [ self close ];

    }

    if ( !active )
    {

        NSLog( @"WARNING : socket is not listening" );

    }

}
```

The class delegate have to define three functions and implement the UdpSocketDelegate protocol.

```
@protocol UdpSocketDelegate <NSObject>


- ( void )  dataArrived : ( char*                    ) theData
            fromAddress : ( struct sockaddr_storage  ) theAddress
            withLength  : ( socklen_t                ) theLength;
- ( void )  readError;
- ( void )  sendError;


@end
```

On UdpSocket class we can build up our UdpClient and UdpHost classes. But first we have to define a communication protocol.

Defining the communication protocol

Packet length will be 25 bytes length, because rotation values need three double values, and a double is presented on 8 bytes. The first byte is always the packet type.

```
// packet format 1 ( type ) + ( 24 ) empty
// or            1 ( type ) + ( 3 * 8 ) rotation / motion
// or            1 ( tyoe ) + ( 1 ) button id + ( 1 ) button state + ( 22 ) empty

#define kProtocolPacketSize            25
#define kProtocolPingDelay             .5
#define kProtocolTimeOut               2

#define kProtocolTypePing              0x00
#define kProtocolTypePong              0x01
#define kProtocolTypeButton            0x02
#define kProtocolTypeRotation          0x03
#define kProtocolTypeDisconnect        0x04

#define kProtocolButtonA               0x00
#define kProtocolButtonB               0x01
#define kProtocolButtonC               0x02
#define kProtocolButtonD               0x03
#define kProtocolButtonE               0x04
#define kProtocolButtonF               0x05
#define kProtocolButtonG               0x06
#define kProtocolButtonH               0x07
#define kProtocolButtonI               0x08
#define kProtocolButtonJ               0x09
#define kProtocolButtonK               0x10
#define kProtocolButtonL               0x11
#define kProtocolButtonM               0x12
#define kProtocolButtonN               0x13
#define kProtocolButtonO               0x14
#define kProtocolButtonP               0x15

#define kProtocolButtonStateDown       0x01
#define kProtocolButtonStateUp         0x00
```

Creating the UdpClient

The constructor receives a delegate, and we set connected and reachable state to default.

```
- ( id ) initWithDelegate : ( id ) theDelegate

connected = NO;
reachable = NO;

delegate  = [ theDelegate retain ];
```

A prepareSocket method is needed to be able to speed up initialization process if we want to. We simply initialize the UdpSocket, and ignite listening.

```
- ( void ) prepareSocket
{
    if ( socket == nil )
    {

        socket = [ [ UdpSocket alloc ] initWithFamily : AF_INET6
                                       withDelegate   : self ];    // needs release

        [ socket listenWithPacketSize : kProtocolPacketSize ];

    }
}
```

Ping handling cometh.
startPing ignites a timer which sends ping messages repeatedly.

```
- ( void ) startPing
{

    [ self stopPing ];

    lastPong  = ( long ) [ [ NSDate date ] timeIntervalSince1970 ];
    pingTimer = [ NSTimer  scheduledTimerWithTimeInterval   : kProtocolPingDelay
                           target                           : self
                           selector                         : @selector(sendPing)
                           userInfo                         : nil
                           repeats                          : YES   ];

}
```

stopPing invalidates this timer.

```
- ( void ) stopPing
{

    if ( pingTimer != nil )
    {

        [ pingTimer invalidate ];
        pingTimer = nil;
    }

}
```

sendPing builds up a ping packet and sends it to the remote peer through the UdpSocket. It also checks for last pong, and if timeout happened, sends disconnect message. If connect does not happened yet, it sends a connection timeout message.

```
- ( void ) sendPing
{

    char packet[ kProtocolPacketSize ];

    packet[ 0 ] = kProtocolTypePing;

    [ socket    sendBytes : packet
                withSize  : kProtocolPacketSize
                toAddress : address ];


    long duration = ( long )[ [ NSDate date ] timeIntervalSince1970 ] -
                                                                  ( long ) lastPong;

    if ( duration > kProtocolTimeOut )
    {

        [ self stopPing ];

        if ( connected )
        {

            [ delegate  eventArrived : kConnectionDisconnect
                        fromInstance : self
                        withUserData : nil ];

        }
        else
        {

            [ delegate  eventArrived : kConnectionTimeout
                        fromInstance : self
                        withUserData : nil ];

        }

    }

}
```

Connecting and disconnecting the client.

```
- ( void ) connectToAddress : ( struct sockaddr_storage ) theAddress
{

    if ( !connected )
    {

        reachable = YES;

        [ AddressUtilities  convertAddress   : &theAddress
                            toAddress        : &address
                            toFamily         : AF_INET6 ];
        [ self prepareSocket ];
        [ self startPing ];
        [ self sendPing ];

    }

}
```

In case of closure sending a disconnection packet is a very elegant thing.

```
- ( void )  disconnect
{

    if ( connected )
    {

        reachable = NO;
        connected = NO;

        char packet[ kProtocolPacketSize ];

        packet[ 0 ] = kProtocolTypeDisconnect;

        [ self      stopPing ];
        [ socket    sendBytes : packet
                    withSize  : kProtocolPacketSize
                    toAddress : address ];
        [ socket    close ];
        [ socket    release ];

        socket = nil;

    }

}
```

We have to receive pong packages, and set lastPong timestamp.

```
- ( void )  dataArrived : ( char*                              ) theData
            fromAddress : ( struct sockaddr_storage ) theAddress
            withLength  : ( socklen_t                          ) theLength
{

    if ( reachable )
    {

        switch ( theData[ 0 ] )
        {

            case kProtocolTypePong :
            {

                if ( !connected )
                {
                    connected = YES;
                    [ delegate  eventArrived : kConnectionSuccess
                                fromInstance : self
                                withUserData : nil ];
                }
                lastPong = ( long ) [ [ NSDate date ] timeIntervalSince1970 ];
                break;
```

```
                        }
                }
        }
}
```

And now let's see the other side.

Creating the UdpHost

We start with receiving and retaining the delegate again, creating the socket, and then we start listening with the wanted packet size.

```
- ( id ) initWithDelegate : ( id ) theDelegate;
{

    self = [ super init ];

    if ( self )
    {

        delegate = [ theDelegate retain ];
        socket   = [ [ UdpSocket alloc ] initWithFamily      : AF_INET6
                                         withDelegate        : self ];

        if ( socket ) [ socket listenWithPacketSize : kProtocolPacketSize ];

    }

    return self;

}
```

Let's see the ping handling. In startCheckingPing we start a timer which checks for ping messages continuously.

```
- ( void ) startCheckingPing
{

    [ self stopCheckingPing ];

    checkTimer = [ NSTimer scheduledTimerWithTimeInterval  : kProtocolPingDelay
                            target                          : self
                            selector                        : @selector(checkPing)
                            userInfo                        : nil
                            repeats                         : YES   ];

    [ delegate  eventArrived : kUdpHostConnectionSuccess
                fromInstance : self
                withUserData : nil ];

}
```

checkPing checks for timeout.

```
- ( void ) checkPing
{

    long duration = ( long )[ [ NSDate date ] timeIntervalSince1970 ] -
                                                                   ( long ) lastPing;

    if ( duration > kProtocolTimeOut )
    {
        [ delegate  eventArrived : kUdpHostConnectionClosure
                    fromInstance : self
                    withUserData : nil ];

        [ self stopCheckingPing ];

        connected = NO;

    }

}
```

Pong message is our simplest packet, easy to assemble.

```
- ( void ) sendPong
{

    char packet[ kProtocolPacketSize ];

    packet[ 0 ] = kProtocolTypePong;

    [ socket    sendBytes : packet
                withSize  : kProtocolPacketSize
                toAddress : address ];

}
```

When data arrives, we have to sort packets by type. The method receives the data, the sender’s address and the address length.

```
- ( void )  dataArrived : ( char*                     ) pData
            fromAddress : ( struct sockaddr_storage   ) pAddress
            withLength  : ( socklen_t                 ) pLength
{

    switch ( pData[ 0 ] )
    {

        case kProtocolTypePing :
        {

            if ( !connected )
            {
```

First ping packet is the connection request, we accept the connection, and store the remote peer’s address.

```
                [ AddressUtilities  convertAddress  : &pAddress
                                    toAddress       : &address
                                    toFamily        : AF_INET6 ];

                connected = YES;
```

Start ping timeout checking.

```
                [ self  performSelectorOnMainThread   : @selector(startCheckingPing)
                        withObject                    : nil
                        waitUntilDone                 : NO ];

            }

            lastPing = ( long ) [ [ NSDate date ] timeIntervalSince1970 ];

            [ self sendPong ];

            break;

        }
        case kProtocolTypeDisconnect :
        {
```

In case if disconnection stop ping checking.

```
            [ delegate  eventArrived : kUdpHostConnectionClosure
                        fromInstance : self
                        withUserData : nil ];

            [ self stopCheckingPing ];

            connected = NO;

            break;

        }
        case kProtocolTypeButton :
        {
```

In case of button packet delegate event.

```
            [ delegate  eventArrived : kUdpHostButtonData
                        fromInstance : self
                        withUserData : pData + 1 ];

            break;

        }
        case kProtocolTypeRotation :
        {
```

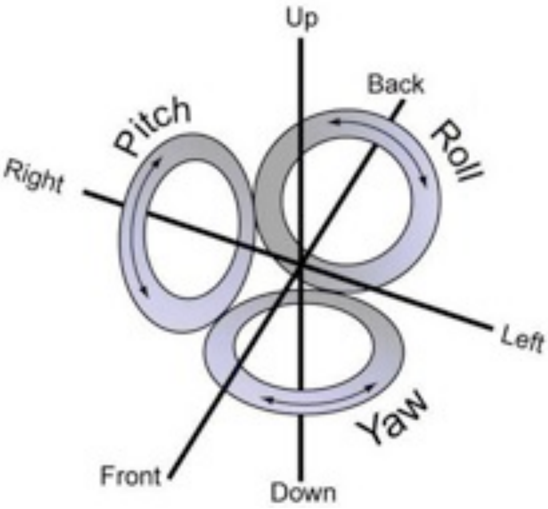
In case of rotation packet delegate event.

```
            [ delegate  eventArrived : kUdpHostRotationData
                        fromInstance : self
                        withUserData : pData + 1 ];

            break;

        }

    }

}
```

Motion Handling

Let's see how to capture motion events on the client (phone) side.

The iPhone has a built-in accelerometer and a gyroscope. The accelerometer is used for detect linear movement, the gyroscope is for detecting rotation movements.



Unfortunately none of them can give us absolute positions, just relative changes. We use the CoreMotion framework to receive motion events in iOS.

The constructor receives the MotionHandler’s delegate instance.

```
- ( id ) initWithDelegate : ( id ) theDelegate
{

    self = [ super init ];

    if ( self )
    {

        delegate = [ theDelegate retain ];
```

If CMMotionManager class exists in our OS version, get an operationqueue and init Motion Manager.

```
        if ( [ CMMotionManager class ] )
        {

            motionQueue     = [ [ NSOperationQueue  currentQueue ] retain   ];
            motionManager   = [ [ CMMotionManager   alloc        ] init     ];

        }

    }

    return self;

}
```

Start listening

```
- ( void ) start
{

    if ( [ CMMotionManager class ] )
    {

        if ( motionManager.gyroAvailable )
        {
```

Core Motion handler's have to be defined using blocks, so we create the motion handler block.

```
            CMDeviceMotionHandler motionHandler = ^( CMDeviceMotion*     motion ,
                                                      NSError*            error)
            {
```

Get roll, pitch and yaw values, and apply a simple low pass filter on them.

```
attitude.x = motionManager.deviceMotion.attitude.roll * kFilteringFactor +
             attitude.x * ( 1.0 - kFilteringFactor );
attitude.y = motionManager.deviceMotion.attitude.pitch * kFilteringFactor +
             attitude.y * ( 1.0 - kFilteringFactor );
```

In case of yaw rotation (rotation around the y axis), we have to check angle jumps (-180 to 180 degrees), and avoid filtering in case of jumping.

```
if ( fabs( motionManager.deviceMotion.attitude.yaw - attitude.z ) > M_PI )
{
      attitude.z = motionManager.deviceMotion.attitude.yaw;
}
else
{
      attitude.z = motionManager.deviceMotion.attitude.yaw * kFilteringFactor +
                   attitude.z * ( 1.0 - kFilteringFactor );
}
```

If we have all values, let's put them into 3*8 flat bytes, and send it to the delegate.

```
                char* flatRotation = malloc( sizeof( attitude ) );

                memcpy( flatRotation ,
                        &attitude ,
                        sizeof( attitude ) );

                [ delegate  eventArrived : kMotionHandlerRotationEvent
                            fromInstance : self
                            withUserData : flatRotation ];

                free( flatRotation );

            };
```

Finally, start listening with the previously defined block.

```
            [ motionManager setDeviceMotionUpdateInterval   : kUpdateInterval ];
            [ motionManager startDeviceMotionUpdatesToQueue : motionQueue
                            withHandler                     : motionHandler      ];

        }

    }

}
```

Motion Injection

Let's see how to inject motion events on the host (desktop) side.

The MotionInjector class is the heart of the application system. It decides what to do with the incoming rotation and motion data, it handles the screen center point. Since the motion sensors cannot provide absolute values, we cannot keep the cursor in the horizontal center of the screen, it will slowly move away from the center during usage, so i've added a center setter feature : if you roll your phone left or right heavily, it switches on "center setter" mode, and you can adjust the center with heavy roll, then simply change back to soft rolling.

We need a lot of helper points in our class. Let's see the constructor :

```
NSRect boundsRect   = [ [ NSScreen mainScreen ] frame ];
```

x and y borders are for storing the screen's (actual) dimensions, center is the actual center of the screen.

```
xBorders            = CGPointMake( boundsRect.origin.x ,
                                   boundsRect.origin.x + boundsRect.size.width );
yBorders            = CGPointMake( boundsRect.origin.y ,
                                   boundsRect.origin.y + boundsRect.size.height );
centerPoint         = CGPointMake( boundsRect.origin.x + boundsRect.size.width  / 2 ,
                                   boundsRect.origin.y + boundsRect.size.height / 2 );
```

actual and previous points are actual and previous cursor positions.

```
actualPoint         = CGPointMake( centerPoint.x ,
                                   centerPoint.y );
previousPoint       = CGPointMake( centerPoint.x ,
                                   centerPoint.y );
```

delta is the delta between the previous and actual position, screenPoint is the final (normalized) actual point.

```
deltaPoint          = CGPointMake( 0 ,
                                   0 );
screenPoint         = CGPointMake( centerPoint.x ,
                                   centerPoint.y );
```

delta angle is the difference between the actual yaw value and centerAngle.

```
deltaAngle          = 0;
```

center angle is the actual angle of the phone's yaw value which points to the center of the screen. It changes slowly, and continuous adjusment needed with heavy rolling of the phone.

```
centerAngle         = M_PI;
```

turning angle is what we add to deltaX when the phone is rolled.

```
turningAngle        = 0;
```

border angles are maximum and minimum possible yaw values, calculated by centerAngle + M_PI and centerAngle - M_PI.

```
upperBorderAngle    = 2 * M_PI;
lowerBorderAngle    = 0;
```

setup listener for screen dimension change

```
[ [ NSNotificationCenter defaultCenter ] addObserver : self
                                          selector    : @selector(screenInfoChange:)
                                          name        :
                                  NSApplicationDidChangeScreenParametersNotification
                                          object              : nil ];
```

let's see the heart of everything, the updateRotation method which processes the incoming motion data.

```
- ( void ) updateRotation : ( void* ) theData
{
```

get rotation struct for the raw char array (3*8 bytes for 3 double)

```
    memcpy( &actualRotation ,
            theData ,
            sizeof( Rotation ) );
```

we are working with 0 .. 2 * M_PI radians for easier calculations

```
    actualRotation.z += M_PI;
```

screen center angle correction when rolling heavily, checking roll values over 0.2

```
    if ( actualRotation.x < -0.2 ) centerAngle += ( actualRotation.x + 0.2 ) / 100; else
    if ( actualRotation.x >  0.2 ) centerAngle += ( actualRotation.x - 0.2 ) / 100;
```

normalize angle if overflow happened

```
    if ( centerAngle < 0 ) centerAngle += 2 * M_PI; else
    if ( centerAngle > 2 * M_PI ) centerAngle -= 2 * M_PI;
```

set border angles

```
    upperBorderAngle = centerAngle + M_PI;
    lowerBorderAngle = centerAngle - M_PI;
```

calculate yaw delta

```
    deltaAngle = centerAngle - actualRotation.z;
```

check angle overflow

```
    if ( actualRotation.z > upperBorderAngle ) deltaAngle = ( 2 * M_PI -
                                    actualRotation.z  ) + centerAngle; else
    if ( actualRotation.z < lowerBorderAngle ) deltaAngle = ( centerAngle - 2 * M_PI ) -
                                    actualRotation.z;
```

calculate actual cursor position based on center point and delta angle with sensitivity

```
    actualPoint.x = centerPoint.x + deltaAngle * ( kMotionInjectorBase +
                             kMotionInjectorLook * sensitivityRatio );
    actualPoint.y = centerPoint.y - actualRotation.y * ( kMotionInjectorBase +
                             kMotionInjectorLook * sensitivityRatio );
```

calculate position change delta

```
    deltaPoint.x = actualPoint.x - previousPoint.x;
    deltaPoint.y = actualPoint.y - previousPoint.y;
```

store actual point as previous

```
    previousPoint = actualPoint;
```

check turning threshold

```
    if ( actualRotation.x < -0.06 ) turningAngle = ( actualRotation.x + 0.06 ) *
kMotionInjectorTurn * sensitivityRatio; else
    if ( actualRotation.x >  0.06 ) turningAngle = ( actualRotation.x - 0.06 ) *
kMotionInjectorTurn * sensitivityRatio;
```

prepare screen point from actual point. Cannot use actual point because it will be modified below and we need its present state.

```
    screenPoint = CGPointMake( actualPoint.x , actualPoint.y );
```

screen border check

```
    if ( screenPoint.x < xBorders.x ) screenPoint.x = xBorders.x + 2; else
    if ( screenPoint.x > xBorders.y ) screenPoint.x = xBorders.y - 2;
    if ( screenPoint.y < yBorders.x ) screenPoint.y = yBorders.x + 2; else
    if ( screenPoint.y > yBorders.y ) screenPoint.y = yBorders.y - 2;
```

create mouse event

```
    CGEventRef mouseEvent = CGEventCreateMouseEvent( NULL          ,
                                                     eventType     ,
                                                     screenPoint   ,
                                                     0             );
```

set delta value for fps games

```
    CGEventSetIntegerValueField( mouseEvent ,
                                 kCGMouseEventDeltaX ,
                                 ( int64_t ) ( round( deltaPoint.x + turningAngle ) ) );

    CGEventSetIntegerValueField( mouseEvent ,
                                 kCGMouseEventDeltaY ,
                                 ( int64_t ) ( round( deltaPoint.y ) ) );
```

set default event flags

```
    CGEventSetFlags( mouseEvent , 256 );
```

post event

```
    CGEventPost( kCGHIDEventTap , mouseEvent );
```

cleanup

```
    CFRelease( mouseEvent  );

}
```

Button And Key Event Injection

Let's see how to inject mouse button and keyboard events on the host (desktop) side.

The host application must be able to inject mouse button and keyboard press events, so ButtonInjector class must have two separate functions.

Let's see the mouse event injector method.

```
- ( void )  injectMouseEvent  : ( CGEventType         ) theType
            withButton        : ( CGMouseButton       ) theButton
            withState         : ( uint                ) theState

{
```

Create mouse event with proper x and y value.

```
    CGEventRef              empty     = CGEventCreate( NULL );
    CGPoint                 point     = CGEventGetLocation( empty );
    CGEventRef              mouseEvent = CGEventCreateMouseEvent(   NULL    ,
                                                                  theType     ,
                                                                  point   ,
                                                                  theButton );
```

If pressed, adding proper clickstate and event pressure flags.

```
    if ( theState )
    {

        CGEventSetIntegerValueField     ( mouseEvent , kCGMouseEventClickState , 1 );
        CGEventSetDoubleValueField      ( mouseEvent , kCGMouseEventPressure   , 1 );

    }

    CGEventSetFlags ( mouseEvent , 256 );
```

Post and release event

```
    CGEventPost      ( kCGHIDEventTap ,
                       mouseEvent );
    CFRelease        ( mouseEvent );
    CFRelease        ( empty );

}
```

Let's see the keyboard event injection. It's much simpler, we create a plain keyboard event, and post it to the event flow.

```
- ( void ) injectKeyboardEvent : ( uint ) theCode
           withState                 : ( uint ) theState
{
    CGEventRef keyEvent = CGEventCreateKeyboardEvent (  NULL ,
                                                        theCode ,
                                                        theState );

    CGEventPost( kCGHIDEventTap ,
                 keyEvent );

    CFRelease( keyEvent );

}
```

Let’s see how to handle the incoming button packets.

```
- ( void ) updateButton : ( void* ) theData
{

    char* data = ( char* ) theData;
```

Get type and state

```
    char type  = data[ 0 ];
    char state = data[ 1 ];
```

In case of mouse button events, inject left/right up/down.

```
    switch ( type )
    {

        case kProtocolButtonA :
        {
            if ( state )
            {
                [ self  injectMouseEvent : kCGEventLeftMouseDown
                        withButton       : kCGMouseButtonLeft
                        withState        : state ];

                [ delegate eventArrived : kButtonInjectorEventLeftMouseDown
                            fromInstance : self
                            withUserData : nil ];

            }
            else
            {
                [ self  injectMouseEvent : kCGEventLeftMouseUp
                        withButton       : kCGMouseButtonLeft
                        withState        : state ];

                [ delegate eventArrived : kButtonInjectorEventLeftMouseUp
                            fromInstance : self
                            withUserData : nil ];

            }
            break;
        }
        case kProtocolButtonB :
        {
            if ( state )
            {
                [ self  injectMouseEvent : kCGEventRightMouseDown
                        withButton       : kCGMouseButtonRight
                        withState        : state ];
            }
            else
            {
                [ self  injectMouseEvent : kCGEventRightMouseUp
                        withButton       : kCGMouseButtonRight
                        withState        : state ];
            }
            break;
        }
```

In case of keyboard events, inject proper keycodes.

```
 case kProtocolButtonC :
      [ self injectKeyboardEvent : 123 withState : state ]; break; // LEFT ARROW
 case kProtocolButtonD :
      [ self injectKeyboardEvent : 124 withState : state ]; break; // RIGHT ARROW
 case kProtocolButtonE :
      [ self injectKeyboardEvent : 125 withState : state ]; break; // UP ARROW
 case kProtocolButtonF :
      [ self injectKeyboardEvent : 126 withState : state ]; break; // DOWN ARROW
 case kProtocolButtonG :
      [ self injectKeyboardEvent : 34  withState : state ]; break; // I
 case kProtocolButtonH :
      [ self injectKeyboardEvent : 38  withState : state ]; break; // J
 case kProtocolButtonI :
      [ self injectKeyboardEvent : 0   withState : state ]; break; // A
 case kProtocolButtonJ :
      [ self injectKeyboardEvent : 11  withState : state ]; break; // B
 case kProtocolButtonK :
      [ self injectKeyboardEvent : 8   withState : state ]; break; // C
 case kProtocolButtonL :
      [ self injectKeyboardEvent : 2   withState : state ]; break; // D
 case kProtocolButtonM :
      [ self injectKeyboardEvent : 14  withState : state ]; break; // E
 case kProtocolButtonN :
      [ self injectKeyboardEvent : 3   withState : state ]; break; // F
 case kProtocolButtonO :
      [ self injectKeyboardEvent : 5   withState : state ]; break; // G
 case kProtocolButtonP :
      [ self injectKeyboardEvent : 4   withState : state ]; break; // H

    }

}
```

Application State Handling

Let's see how to handle multiple application states with possible immediate interruption. The client application has 9 operating states with same or different UI states :

```
#define kStateMenu                      0
#define kStateBrowsing                  1
#define kStateSelecting                 2
#define kStateResolving                 3
#define kStateNoService                 4
#define kStateConnecting                5
#define kStateTimeout                   6
#define kStateDisconnected              7
#define kStateConnected                 8
```

So we need 9 state changer methods.

```
- ( void )    switchToMenuState;
- ( void )    switchToBrowsingState;
- ( void )    switchToSelectingState;
- ( void )    switchToResolvingState;
- ( void )    switchToNoHostState;
- ( void )    switchToConnectingState;
- ( void )    switchToTimeoutState;
- ( void )    switchToConnectedState;
- ( void )    switchToDisconnectedState;
-
```

The class has to contain all main modules to control them directly (

```
- ( id ) initWithConnection       : ( UdpClient*            ) pConnection
         withMotionHandler        : ( MotionHandler*        ) pHandler
         withBonjourBrowser       : ( BonjourBrowser*       ) pBrowser
         withRootViewController   : ( RootViewController*   ) pController;
```

And we have to check the actual state in every state changer method.

In `switchToMenuState`, in the first four states we have to stop the bonjour browser, in the next three states we have to disconnect also, and if we come from connected state, we have to stop motion handler, then can we store the new state and change the UI.

```
- ( void ) switchToMenuState
{

    switch ( applicationState )
    {

        case kStateBrowsing  :
        case kStateSelecting :
        case kStateResolving :
        case kStateNoService :
        {

            [ bonjourBrowser     stop ];
            break;
        }
        case kStateTimeout :
        case kStateConnecting :
        case kStateDisconnected :
        {

            [ bonjourBrowser     stop ];
            [ connection         disconnect ];

            break;
        }
        case kStateConnected :
        {

            [ motionHandler             stop ];
            [ connection         disconnect ];

            break;

        }

    }

    applicationState = kStateMenu;
    [ rootViewController openMenuView ];

}
```

The best example is switching to connecting state. It can happen only from resolving state, after bonjour browser resolved the detected host's IP address.

```
- ( void )   switchToConnectingState
{

    if ( applicationState == kStateResolving ||
         applicationState == kStateConnecting )
    {
```

Stopping bonjour browser, it is not needed any more.

```
        [ bonjourBrowser stop ];
```

If we have one or more addresses...

```
        if ( [ serviceAddresses count ] > 0 )
        {
```

change actual state, get address data.

```
            applicationState = kStateConnecting;

            NSData* rawdata = [ serviceAddresses objectAtIndex : 0 ];
            struct sockaddr_storage* clientAddress = ( struct sockaddr_storage* )
                                                                   [ rawdata bytes ];
            [ serviceAddresses   removeObjectAtIndex : 0 ];
```

disconnect connection if connected, and start connecting to the new address.

```
            [ connection         disconnect ];
            [ connection         connectToAddress    : *clientAddress ];

        }
        else
        {
```

If no more addresses left, switching to no service state and show status view.

```
            applicationState = kStateNoService;

            [ rootViewController setStatusLabel : @"NO HOSTS AVAILABLE"
                                 withRed              : YES ];
            [ rootViewController openStatusView ];

        }

    }

}
```

Building Up The User Interface

Let's see how to create shared UI code between OS X and iOS.

This is a little bit harder to do, because the other (model) classes are quite platform independent, but UIKit (iOS UI framework), and AppKit (OS X UI framework) differs. AppKit's base view class is NSView, UIKit's base view is UIView. They have a lot of methods with the same name, but they have opposing coordinate handling, UIKit is touch based, AppKit is mouse click based. So let's see how to create hybrid classes.

We have to define build target switches, it can be done in the build settings (not really transparent), or we can do it in the Prefix.pch (recommended, transparent).

So if we define iOS for build target in the iOS project

```
#define BUILD_TARGET_IOS 1
```

Then we can declare our view class with preprocessor macros :

```
#ifdef BUILD_TARGET_OSX

#import <AppKit/AppKit.h>
@interface ButtonView : NSView

#else

#import <UIKit/UIKit.h>
@interface ButtonView : UIView
```

One other problem is that NSView needs NSRect, UIView need CGRect in its constructor. How do we divide up our constructor then?

```
- ( id )    initWithId   : ( uint        ) theButtonId
            withLabel    : ( NSString*   ) theLabel
            withFrame    : ( CGRect      ) theRect
            withColors   : ( float*      ) theColors
            withDelegate : ( id          ) theDelegate
{

    expandedRect = CGRectInset( theRect , 7 , 7 );
    originalRect = CGRectInset( theRect , 2 , 2 );

    #ifdef BUILD_TARGET_OSX
    self = [ super initWithFrame : NSRectFromCGRect( originalRect ) ];
    #else
    self = [ super initWithFrame : originalRect ];
    #endif
```

Animation handling also differs. Let's see how to expand/shrink views in a shared code.

```
- ( void ) expand
{
    #ifdef BUILD_TARGET_OSX
```

AppKit uses animator.

```
    [ [ NSAnimationContext currentContext ] setDuration     : .05                ];
    [ [ self                 animator     ] setFrame  :
                                           NSRectFromCGRect( expandedRect ) ];

    #else
```

UIKit uses beginAnimations. And animation blocks in the latest versions.

```
    [ UIView beginAnimations            : @""
             context                    : nil                ];
    [ UIView setAnimationDuration       : .05                ];
    [ self     setFrame                 : expandedRect       ];
    [ UIView commitAnimations                                ];

    #endif

}
```

The last difficult topic is touch/mouse handling. On iOS, use touchesBegan

```
#ifdef BUILD_TARGET_IOS

- ( void )   touchesBegan   : ( NSSet*    ) theTouches
             withEvent      : ( UIEvent*  ) theEvent
{

    [ self.superview bringSubviewToFront : self ];
    [ self expand ];
```

Dispatch event.

```
    [ delegate  eventArrived : kSharedButtonViewEventPress
                fromInstance : self
                withUserData : &buttonId ];

}

#endif
```

On OS X, using mouseDown event

```
#ifdef BUILD_TARGET_OSX

- ( void ) mouseDown : ( NSEvent* ) theEvent
{

    [ self expand ];

}
```

Generating Pixel Fonts

Let's see how to create pixel fonts.
Pixel fonts used by remotion can have an arbitrary number of columns and six rows. They are described in strings, row per row, separated by space. Let's see the descriptor for letter "A".

```
000 111 101 111 101 101
```

for letter "M".

```
00000 11111 10101 10101 10101 10101
```

To optimize PixelFont generation, we need a pixel cache and size cache for every font generated, because letters are re-used heavily and in this case we don't have to re-calculate and re-generate sizes and arrays.

Let's see the class initializer where we create our arrays and fill up lettermap.

```
+ ( void ) initialize
{

    letterMap  = [ [ NSMutableDictionary alloc ] init ];
    sizeCache  = [ [ NSMutableDictionary alloc ] init ];
    pixelCache = [ [ NSMutableDictionary alloc ] init ];

    [ letterMap setObject : @"000 111 101 111 101 101"              forKey : @"A" ];
    [ letterMap setObject : @"010 111 101 111 101 101"              forKey : @"Á" ];
.
.
.
    [ letterMap setObject : @"0000 1111 1000 1000 1000 0000"        forKey : @"<" ];
    [ letterMap setObject : @"0000 1111 0001 0001 0001 0000"        forKey : @">" ];
    [ letterMap setObject : @"00 00 00 00 00 00"                     forKey : @" " ];

}
```

The pixel generator method receives the desired letter, the wanted grid and pixel size.

```
+ ( void ) generatePixels  : ( NSString* ) theLetter
           gridSize        : ( float      ) theGridSize
           pixelSize       : ( float      ) thePixelSize
{
```

Getting pixel map, and breaking it into rows

```
    NSString*       pixelMap      = [ letterMap objectForKey : theLetter ];
    NSArray*        pixelRows     = [ pixelMap componentsSeparatedByString : @" " ];
    NSMutableArray* pixels        = [ [ NSMutableArray alloc ] init ];
```

Looping through pixel rows.

```
for ( int rowIndex = 0 ;
          rowIndex < [ pixelRows count ] ;
          rowIndex ++ )
{
```

Getting actual row, and getting digits per column.

```
    NSString* actualRow = [ pixelRows objectAtIndex : rowIndex ];

    for ( int colIndex = 0 ;
              colIndex < [ actualRow length ] ;
              colIndex ++ )
    {

        if ( [ actualRow characterAtIndex : colIndex ] == *"1" )
        {
```

Create a rectangle based on pixel's row, column and grid size.

```
            Rectangle* rectangle = [ [ Rectangle alloc ] init ];

            [ rectangle setX        : colIndex * theGridSize ];
            [ rectangle setY        : rowIndex * theGridSize ];
            [ rectangle setWidth    : theGridSize ];
            [ rectangle setHeight   : theGridSize ];
```

Store rectangle.

```
            [ pixels addObject : rectangle ];

            [ rectangle release ];

        }

    }

}
```

Calculate full pixel character's dimensions

```
Rectangle* size = [ [ Rectangle alloc ] init ];

[ size setWidth        : [ [ pixelRows objectAtIndex : 0 ] length ] * thePixelSize ];
[ size setHeight       : [ pixelRows count ] * thePixelSize ];
```

Store pixel and size arrays in cache.

```
[ pixelCache    setObject : pixels
                forKey   : theLetter ];
[ sizeCache     setObject : size
                forKey   : theLetter ];

[ pixels        release ];
[ size          release ];
```

```
}
```

The letter generator method creates a letter instance, and gets the pixel array from the cache is exists. It also sets the letter's rectangle.

```
+ ( PixelFont* )    createLetter : ( NSString* ) theLetter
                    gridSize      : ( float       ) theGridSize
                    pixelSize    : ( float      ) thePixelSize;
{

    PixelFont*      letter        = [ [ PixelFont alloc ] init ];
```

Getting cached pixels.

```
    NSArray*        pixels        = [ pixelCache objectForKey : theLetter ];
    Rectangle*      rectangle     = [ sizeCache  objectForKey : theLetter ];
```

If not cached, force generation.

```
    if ( pixels == nil )
    {

        [ self  generatePixels    : theLetter
                gridSize          : theGridSize
                pixelSize         : thePixelSize ];

        pixels      = [ pixelCache objectForKey : theLetter ];
        rectangle   = [ sizeCache  objectForKey : theLetter ];

    }
```

And finally setup letter.

```
    Rectangle* letterRectangle = [ [ Rectangle alloc ] initWithRectangle : rectangle ];

    [ letter setPixels            : pixels ];
    [ letter setCharacter  : theLetter ];
    [ letter setRectangle  : letterRectangle ];

    [ letterRectangle release ];

    return letter;

}
```

Things to know

- properties make your code ugly
- blocks make code your ugly
- autorelease makes your memory management (and your code) ugly
- ARC keywords make your code ugly
- all of the above makes development faster and careless.
- if you are fast enough, you don't need them
- if you can't keep order in your code, you can't do it in your mind either
- keep it as objective-c 1.0 as possible!!!
- be a control freak
- handling objects/primitives as booleans in your statements make your code unreadable
- don't import anything besides the header in an implementation, other way sooner or later colliding imports will mess up your project
- if you create a c struct/variable, zero it immediately, it saves you a lot of headache
- inlineing makes your code ugly
- create every variable in a separate line, line overloading makes your code ugly
- avoid dot notation if possible, mixed obj-c and dot syntax makes the code unreadable
- getsockname and recvfrom needs proper address length values passed to them before they rewrite it with the result address lengths!

thanks for yUML for the awesome descriptive diagram drawer!!!