```
#include "ClientController.h"
#include "GameController.h"
#include "ClientNetworkController.h"
#include "GameView.h"
#include "StatusView.h"
#include "ScoreView.h"
#include "Vector.h"
#include "InputController.h"
#include "TestMessage.h"
#include <iostream>

#include "LocalPlayer.h"

using namespace blaster;
using namespace std;

ClientController::ClientController( void )
{
    video_bpp = 0;
    videoflags = SDL_SWSURFACE | SDL_OPENGL;

    screenWidth = 640;
    screenHeight = 480;

    fullscreen = false;

    running = true;

    setupSDL(  );
    setupSDLNet(  );
    setupGL(  );

    View *gameView = new GameView( NULL );
    View *statusView = new StatusView( gameView );
    View *scoreView = new ScoreView( statusView );

    topView = scoreView;
    topView->init(  );

}

ClientController::~ClientController( void )
{
    delete topView;

    SDLNet_Quit(  );
    SDL_Quit(  );
}


ClientController *ClientController::instance = NULL;

ClientController *ClientController::getInstance( void )
{
    if ( instance == NULL )
    {
        instance = new ClientController(  );
    }

    return instance;
}

// turn this on if you want to run the client without a server
```



```
#define NOSERVER

// turn this on to create another player
#define FAKEPLAYER

void ClientController::run( void )
{
    // Hard-code the server location for now
    Uint16 port = 1337;

    std::string address = "192.168.0.41";

    long launchtime = SDL_GetTicks(  );
    long oldtime = 0;
    long newtime = 0;

    float time = 0.0f;
    float dt = 0.0f;

    InputController *input;
#ifndef NOSERVER
    ClientNetworkController *network =
        ClientNetworkController::getInstance(  );

    network->connect( address, port );
#endif

    // Get player id from server at some point
    Player * player = new LocalPlayer(  );

    player->setID( 42 );

    setLocalPlayerID( 42 );
    GameController::getInstance(  )->addPlayer( player );

#ifdef FAKEPLAYER
    Player *p2 = new Player(  );

    p2->setID( 1337 );
    GameController::getInstance(  )->addPlayer( p2 );
#endif

    while ( running )
    {
        newtime = SDL_GetTicks(  ) - launchtime;
        dt = ( float ) ( newtime - oldtime ) * 0.001f;
        time = ( float ) newtime *0.001f;

        oldtime = newtime;

        input = InputController::getInstance(  );
        input->think(  );

#ifndef NOSERVER
        network = ClientNetworkController::getInstance(  );
        network->think( time, dt );

        network->UDPSendMessage( new TestMessage( time ) );
#endif

        // Move this elsewhere?
        if ( input->quit )
        {
```

```
            quit(  );
        }
        if ( input->fullscreen )
        {
            setFullscreen( !fullscreen );
            setupGL(  );
            topView->init(  );
        }

        // Update the game world
        GameController::getInstance(  )->think( time, dt );

        // Draw the screen
        topView->drawFrame(  );

#ifndef NOSERVER
        // Send any messages that we've queued during the frame
        network->flush(  );
#endif
    }

    SDL_ShowCursor( 1 );
}


void ClientController::setupSDL(  )
{
    video_bpp = 0;
    videoflags = SDL_SWSURFACE | SDL_OPENGL;

    if ( SDL_Init( SDL_INIT_VIDEO | SDL_INIT_TIMER ) < 0 )
    {
        fprintf( stderr, "Couldn't initialize SDL: %s\n", SDL_GetError(  ) );
        exit( 1 );
    }
    SDL_GL_SetAttribute( SDL_GL_RED_SIZE, 8 );
    SDL_GL_SetAttribute( SDL_GL_GREEN_SIZE, 8 );
    SDL_GL_SetAttribute( SDL_GL_BLUE_SIZE, 8 );
    SDL_GL_SetAttribute( SDL_GL_DEPTH_SIZE, 16 );
    SDL_GL_SetAttribute( SDL_GL_DOUBLEBUFFER, 1 );

    SDL_ShowCursor( 0 );

    setFullscreen( fullscreen );
}

void ClientController::setFullscreen( bool value, bool alreadyFailed )
{
    screen =
        SDL_SetVideoMode( screenWidth, screenHeight, video_bpp,
                          videoflags | ( value ? SDL_FULLSCREEN : 0 ) );

    if ( screen == NULL )
    {
        if ( !alreadyFailed )
        {
            std::cerr << "setFullscreen(" << value << ") failed." << std::
                endl;
            setFullscreen( !value, true );
            return;
        }

        fprintf( stderr, "Unable to create %dx%dx%d screen: %s\n",
```



```
                screenWidth, screenHeight, video_bpp, SDL_GetError(  ) );
        SDL_Quit(  );
        exit( 2 );
    }

    fullscreen = value;
}

void ClientController::setupGL( void )
{
    glClearColor( 0.0f, 0.0f, 0.0f, 0.0f );

    glShadeModel( GL_SMOOTH );

    glEnable( GL_BLEND );
    glBlendFunc( GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA );
    glEnableClientState( GL_VERTEX_ARRAY );

    glViewport( 0, 0, screenWidth, screenHeight );

    glMatrixMode( GL_PROJECTION );
    glLoadIdentity(  );

    float drawWidth = 500.0f;
    float drawHeight =
        ( ( float ) screenHeight / ( float ) screenWidth ) * drawWidth;

    std::
        cout << "drawWidth: " << drawWidth << ", drawHeight: " << drawHeight
        << std::endl;

    glOrtho( -drawWidth / 2, drawWidth / 2, drawHeight / 2, -drawHeight / 2,
             -1.0, 1.0 );

    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity(  );
}

void ClientController::setupSDLNet( void )
{
    // initialize SDL_net
    if ( SDLNet_Init(  ) == -1 )
    {
        printf( "SDLNet_Init: %s\n", SDLNet_GetError(  ) );
        exit( 2 );
    }
}

int main( int argc, char **argv )
{
    cout << "her er en main" << endl;


    // Resolve the argument into an IPaddress type

    ClientController *client;

    client = ClientController::getInstance(  );
    client->run(  );

    return 0;
}
```

## ClientController.h



```
#ifndef CLIENTCONTROLLER_H
#define CLIENTCONTROLLER_H

#include "SDL.h"
#include "SDL_net.h"
#include "View.h"
#include <string>

namespace blaster
{
    class ClientController
    {
      public:
        ~ClientController( void );
        static ClientController *getInstance( void );
        void run( void );
        void quit( void )
        {
            running = false;
        }

        Uint32 getLocalPlayerID( void )
        {
            return localClientID;
        }
        void setLocalPlayerID( Uint32 i )
        {
            localClientID = i;
        }

      private:
        ClientController( void );
        static ClientController *instance;
        void setupGL( void );
        void setupSDL( void );
        void setupSDLNet( void );
        void setFullscreen( bool value, bool alreadyFailed = false );

        View *topView;

        bool running;
        Uint32 screenWidth, screenHeight;
        bool fullscreen;
        SDL_Surface *screen;
        Uint8 video_bpp;
        Uint32 videoflags;

        Uint32 localClientID;
    };

}

#endif
```

## ClientNetworkController.cpp



```
#include "ClientNetworkController.h"
#include <iostream>
#include "SDL_net.h"

using namespace blaster;
using namespace std;

bool ClientNetworkController::connect( const std::string & address,
                                       Uint16 port )
{
    TCPsocket tcpSocket;

    IPaddress ip;

    int addressLength = address.size(  );

    char adr[addressLength + 1];

    strcpy( adr, address.c_str(  ) );

    if ( SDLNet_ResolveHost( &ip, adr, port ) == -1 )
    {
        printf( "SDLNet_ResolveHost: %s\n", SDLNet_GetError(  ) );
        return false;
    }

    // open the tcp socket to the server
    tcpSocket = SDLNet_TCP_Open( &ip );
    if ( !tcpSocket )
    {
        printf( "SDLNet_TCP_Open: %s\n", SDLNet_GetError(  ) );
        return false;
    }

    // open the tcp server socket
    udpsocket = SDLNet_UDP_Open( 0 );

    if ( !udpsocket )
    {
        cerr << "SDLNet_UDP_Open: " << SDLNet_GetError(  ) << endl;
        return false;
    }

    // bind server address to channel 0
    if ( SDLNet_UDP_Bind( udpsocket, 0, &ip ) == -1 )
    {
        printf( "SDLNet_UDP_Bind: %s\n", SDLNet_GetError(  ) );
        return false;
    }

    addConnection( 0, tcpSocket, ip );

    return true;
}

void ClientNetworkController::disconnect( void )
{
    removeConnection( 0 );
}

ClientNetworkController *ClientNetworkController::instance = NULL;

ClientNetworkController *ClientNetworkController::getInstance( void )
```

## ClientNetworkController.cpp



```
{
    if ( instance == NULL )
    {
        instance = new ClientNetworkController(  );
    }

    return instance;
}
```

## ClientNetworkController.h



```
#ifndef CLIENTNETWORKCONTROLLER_H
#define CLIENTNETWORKCONTROLLER_H

#include <string>
#include "SDL.h"

#include "NetworkController.h"

namespace blaster
{

    class ClientNetworkController:public NetworkController
    {
      public:
        static ClientNetworkController *getInstance( void );

        virtual bool connect( const std::string & address, Uint16 port );
        virtual void disconnect( void );

        virtual ~ ClientNetworkController( void )
        {
        }

      private:
          ClientNetworkController( void ):NetworkController(  )
        {
        }
        static ClientNetworkController *instance;

        TCPsocket tcpSocket;
        UDPsocket udpSocket;
    };
}

#endif
```

## GameController.cpp



```
#include "GameController.h"

#include "Level.h"
#include "Player.h"

#include <iostream>

using namespace blaster;
using namespace std;

GameController::GameController( void )
{
    players = new Players(  );
}

GameController::~GameController( void )
{
    delete players;
}

void GameController::think( const float time, const float dt )
{
    this->time = time;

    Level *level = Level::getInstance(  );

    Objects *bullets = level->getBullets(  );
    Objects *ships = level->getShips(  );

    for ( Players::iterator i = players->begin(  );
          i != players->end(  ) && !players->empty(  ); i++ )
    {
        ( *i ).second->think( time, dt );
    }

    for ( Objects::iterator i = ships->begin(  );
          i != ships->end(  ) && !ships->empty(  ); i++ )
    {
        Object *b = ( *i );

        if ( b->needsToBeDeleted(  ) )
        {
            i = ships->erase( i );

            delete b;

            continue;
        }

        b->think( time, dt );
    }

    int polyCount = level->getMap(  )->getPolygonCount(  );
    Polygon **polygons = level->getMap(  )->getPolygonArray(  );

    for ( int i = 0; i < polyCount; i++ )
    {
        Polygon *polygon = polygons[i];

        for ( Objects::iterator it = ships->begin(  ); it != ships->end(  );
              it++ )
        {
            Ship *s = ( Ship * ) * it;
```

## GameController.cpp



```
            if ( s->collides( polygon ) )
            {
                // Let damage be determined by speed
                float speed = s->getVelocity(  ).length(  );

                if ( speed < 10.0f )
                {
                    s->setVelocity( Vector::ZERO );
                }
                else
                {
                    s->takeDamage( speed );

                    // Get the reflection vector from the edge that we hit
                    s->setVelocity( Vector::ZERO - s->getVelocity(  ) );
                }

                s->setColor( 1.0f, 0.0f, 0.0f );
            }
            else
            {
                s->setColor( 0.0f, 1.0f, 1.0f );
            }
        }
    }

    for ( Objects::iterator i = bullets->begin(  );
          i != bullets->end(  ) && !bullets->empty(  ); i++ )
    {
        Object *b = ( *i );

        if ( b->needsToBeDeleted(  ) )
        {
            i = bullets->erase( i );
            delete b;

            continue;
        }

        b->think( time, dt );
    }
    for ( int i = 0; i < polyCount; i++ )
    {
        Polygon *polygon = polygons[i];

        for ( Objects::iterator i = bullets->begin(  );
              i != bullets->end(  ) && !bullets->empty(  ); i++ )
        {
            Bullet *b = ( Bullet * ) * i;

            if ( b->collides( polygon ) )
            {
                b->kill(  );
            }
        }
    }

    for ( Objects::iterator i = ships->begin(  );
          i != ships->end(  ) && !ships->empty(  ); i++ )
    {
        for ( Objects::iterator j = bullets->begin(  );
              j != bullets->end(  ) && !bullets->empty(  ); j++ )
```

## GameController.cpp



```
        {
            Ship *ship = ( Ship * ) * i;
            Object *bullet = *j;

            if ( ship->getOwner(  ) != bullet->getOwner(  ) )
            {
                Polygon **polys = ship->getPolygonArray(  );
                int polycount = ship->getPolygonCount(  );
                bool collided = false;

                for ( int k = 0; k < polycount; k++ )
                {
                    if ( bullet->collides( polys[k] ) )
                    {
                        collided = true;
                        break;
                    }
                }

                if ( collided )
                {
                    ship->takeDamage( 75 );
                    bullet->kill(  );
                }
            }
        }
    }
}

GameController *GameController::instance = NULL;

GameController *GameController::getInstance( void )
{
    if ( instance == NULL )
    {
        instance = new GameController(  );
    }

    return instance;
}

void GameController::addPlayer( Player * p )
{
    players->insert( std::make_pair( p->getID(  ), p ) );
}

Player *GameController::getPlayer( Uint32 id )
{
//  return (*players)[id];
    Players::iterator i = players->find( id );

    return ( *i ).second;
}

void GameController::removePlayer( Uint32 id )
{
    Players::iterator i = players->find( id );

    players->erase( i );

    delete( *i ).second;
}
```

## GameController.h



```
#ifndef GAMECONTROLLER_H
#define GAMECONTROLLER_H

#include "Player.h"
#include "SDL.h"

#include <map>

namespace blaster
{
    typedef std::map < Uint32, Player * >Players;

    class GameController
    {
      public:
        ~GameController( void );

        static GameController *getInstance( void );
        void think( const float time, const float dt );

        Players *players;

        void addPlayer( Player * p );
        Player *getPlayer( Uint32 id );
        void removePlayer( Uint32 id );

        float getTime( void )
        {
            return time;
        }

      private:
         GameController( void );
        float time;

        static GameController *instance;
    };

}

#endif
```

```
#include "InputController.h"

using namespace blaster;

InputController *InputController::instance = NULL;

InputController::InputController( void )
{
    // Key bindings
    forwardKey = SDLK_UP;
    leftKey = SDLK_LEFT;
    rightKey = SDLK_RIGHT;
    bulletKey = SDLK_SPACE;
    quitKey = SDLK_ESCAPE;
    screenKey = SDLK_RETURN;
    killKey = SDLK_k;
    scoreKey = SDLK_TAB;

    // Key state
    moveForward = false;
    rotateLeft = false;
    rotateRight = false;
    fireBullet = false;
    quit = false;
    fullscreen = false;
    kill = false;
    viewScore = false;
}

InputController *InputController::getInstance( void )
{
    if ( instance == NULL )
    {
        instance = new InputController(  );
    }

    return instance;
}

void InputController::think( void )
{
    SDL_Event event;

    while ( SDL_PollEvent( &event ) )
    {
        switch ( event.type )
        {
        case SDL_KEYDOWN:
            {
                keyPressed( event.key.keysym.sym );
                break;
            }

        case SDL_KEYUP:
            {
                keyReleased( event.key.keysym.sym );
                break;
            }

        default:
            break;
        }
```



```
    }
}

void InputController::keyPressed( const SDLKey & key )
{
    // C++ doesn't support switches with non-constant values.
    if ( key == forwardKey )
    {
        moveForward = true;
    }
    else if ( key == leftKey )
    {
        rotateLeft = true;
    }
    else if ( key == rightKey )
    {
        rotateRight = true;
    }
    else if ( key == bulletKey )
    {
        fireBullet = true;
    }
    else if ( key == quitKey )
    {
        quit = true;
    }
    else if ( key == screenKey )
    {
        fullscreen = true;
    }
    else if ( key == killKey )
    {
        kill = true;
    }
    else if ( key == scoreKey )
    {
        viewScore = true;
    }
}

void InputController::keyReleased( const SDLKey & key )
{
    // C++ doesn't support switches with non-constant values.
    if ( key == forwardKey )
    {
        moveForward = false;
    }
    else if ( key == leftKey )
    {
        rotateLeft = false;
    }
    else if ( key == rightKey )
    {
        rotateRight = false;
    }
    else if ( key == bulletKey )
    {
        fireBullet = false;
    }
    else if ( key == quitKey )
    {
        quit = false;
    }
```

## InputController.cpp



```
    else if ( key == screenKey )
    {
        fullscreen = false;
    }
    else if ( key == killKey )
    {
        kill = false;
    }
    else if ( key == scoreKey )
    {
        viewScore = false;
    }

}
```

## InputController.h



```
#ifndef INPUTCONTROLLER_H
#define INPUTCONTROLLER_H

#include "SDL.h"

namespace blaster
{

    class InputController
    {
      public:
        static InputController *getInstance( void );
        void think( void );

        bool moveForward;
        bool rotateLeft;
        bool rotateRight;
        bool fireBullet;
        bool quit;
        bool fullscreen;
        bool kill;
        bool viewScore;

      private:
          InputController( void );

        void keyPressed( const SDLKey & key );
        void keyReleased( const SDLKey & key );

        SDLKey forwardKey;
        SDLKey leftKey;
        SDLKey rightKey;
        SDLKey bulletKey;
        SDLKey quitKey;
        SDLKey screenKey;
        SDLKey killKey;
        SDLKey scoreKey;

        static InputController *instance;
    };

}

#endif
```

```
#include "NetworkController.h"
#include "MessageMakerBase.h"
#include <iostream>
#include <string>

#include "ClientNetworkController.h"
#include "ServerNetworkController.h"

using namespace blaster;
using namespace std;


NetworkController::NetworkController( void )
{
    Connections connections = Connections(  );
    Messages messageInQueue = Messages(  );

    Messages UDPOutQueue = Messages(  );
    Messages TCPOutQueue = Messages(  );

    socketSet = SDLNet_AllocSocketSet( 32 );
    numConnections = 0;
}

// Quick fix for those without Project Builder
#if !defined (BLASTERSERVER) && !defined (BLASTERCLIENT)
#define BLASTERCLIENT
#endif

NetworkController *NetworkController::getInstance( void )
{
#ifdef BLASTERCLIENT
    return ClientNetworkController::getInstance(  );
#endif
#ifdef BLASTERSERVER
    return ServerNetworkController::getInstance(  );
#endif
}

void NetworkController::think( float time, float dt )
{
    if ( numConnections > 0 && SDLNet_CheckSockets( socketSet, 0 ) > 0 )
    {
        readMessages(  );
    }
}


void NetworkController::readMessages( void )
{
    char buffer[1024];
    TCPsocket socket;
    int id = 0;

    for ( Connections::iterator i = connections.begin(  );
          i != connections.end(  ) && !connections.empty(  ); i++ )
    {

        socket = ( ( *i ).second )->tcpsocket;
        id = ( *i ).first;

        if ( SDLNet_SocketReady( socket ) )
        {
```



```
            if ( SDLNet_TCP_Recv( socket, buffer, 1024 ) < 1 )
            {
                removeConnection( id );
            }
        }

    }
    // stuff that reads from tcp-socket and feeds data to message-maker
    // push messages on message queue

    // read data from UDP-socket and push on message-queue via messagemaker
    UDPpacket packet;

    NetMessage *msg = NULL;

    while ( SDLNet_UDP_Recv( udpsocket, &packet ) )
    {
        cerr << "Recived " << packet.len << " bytes of data via UDP" << endl;
        msg = MessageMakerBase::constructMessage( packet.data, packet.len );

        if ( msg )
        {
            messageInQueue.push( msg );
        }
    }

    while ( !messageInQueue.empty(  ) )
    {
        msg = messageInQueue.front(  );

        messageInQueue.pop(  );

        msg->think(  );
    }
}

void NetworkController::flush( void )
{
    UDPpacket *packet;
    NetMessage *message;

    // for each udp
    while ( !UDPOutQueue.empty(  ) )
    {
        message = UDPOutQueue.front(  );
        UDPOutQueue.pop(  );

        Uint32 messageSize = message->getSize(  );

        // create output buffer
        Uint8 *output = new Uint8[messageSize];

        message->serializeTo( output );

        // create package and put output into it

        packet = SDLNet_AllocPacket( messageSize );

        memcpy( packet->data, output, messageSize );

        packet->channel = -1;
        packet->address = getIPFromID( message->getTo(  ) );
```

```
        // we only have one udp-socket
        if ( SDLNet_UDP_Send( udpsocket, -1, packet ) == 0 )
        {
            // if errors
            removeConnection( message->getTo(  ) );
        }

        SDLNet_FreePacket( packet );

        delete message;
    }

    // for each tcp
    while ( !TCPOutQueue.empty(  ) )
    {
        message = UDPOutQueue.front(  );
        UDPOutQueue.pop(  );

        Uint32 messageSize = message->getSize(  );

        Uint8 *output = new Uint8[messageSize];

        message->serializeTo( output );

        if ( !
             ( SDLNet_TCP_Send
               ( getTCPSocketFromID( message->getTo(  ) ), output,
                 message->getSize(  ) ) < messageSize ) )
        {
            // if errors
            removeConnection( message->getTo(  ) );
        }
        delete message;
    }
}

TCPsocket NetworkController::getTCPSocketFromID( int id )
{
    Connection *con = connections[id];

    return con->tcpsocket;
}

IPaddress NetworkController::getIPFromID( int id )
{
    Connection *con = connections[id];

    return con->ip;
}


void NetworkController::pushInMessage( NetMessage * message )
{
    messageInQueue.push( message );
}

NetMessage *NetworkController::popInMessage( void )
{
    NetMessage *message = messageInQueue.front(  );

    messageInQueue.pop(  );

    return message;
```



```
}

Connection *NetworkController::addConnection( int id, TCPsocket & tcpsocket,
                                              IPaddress ip )
{
    cerr << "Adding new connection with id " << id << endl;
    Connection *con = connections[id];

    // return existing connection if exists
    if ( con )
    {
        return con;
    }

    con = new Connection( ip, tcpsocket );
    connections[id] = con;
    SDLNet_TCP_AddSocket( socketSet, tcpsocket );
    numConnections++;

    cerr << "numConnections: " << numConnections << endl;

    return con;
}


void NetworkController::removeConnection( int id )
{
    Connection *con = connections[id];

    if ( con )
    {
        cerr << "Breaking connection with id: " << id << endl;
        SDLNet_TCP_DelSocket( socketSet, con->tcpsocket );
        Connections::iterator i = connections.find( id );
        connections.erase( i );
        numConnections--;
        delete con;

        cerr << "numConnections: " << numConnections << endl;
    }
}


void NetworkController::UDPSendMessage( NetMessage * message )
{
    UDPOutQueue.push( message );
}

void NetworkController::TCPSendMessage( NetMessage * message )
{
    TCPOutQueue.push( message );
}
```

```
#ifndef NETWORKCONTROLLER_H
#define NETWORKCONTROLLER_H

#include "SDL_net.h"
#include "NetworkController.h"
#include "NetMessage.h"
#include "Connection.h"

#include <queue>
#include <map>

using namespace std;

namespace blaster
{
    typedef std::queue < NetMessage * >Messages;
    typedef std::map < int, Connection * >Connections;

    class NetworkController
    {

      public:
        virtual void think( float time, float dt );

        static NetworkController *getInstance( void );
          virtual ~ NetworkController( void )
        {
        }

        void flush( void );
        NetMessage *popMessage( void );

        void TCPSendMessage( NetMessage * message );
        void UDPSendMessage( NetMessage * message );

        // Dummy methods that make sure that both our network-controllers
        // have the same interface
        virtual void startServer( Uint16 port )
        {
        }
        virtual bool connect( const std::string & address, Uint16 port )
        {
            return false;
        }

        virtual void disconnect( void ) {}

        int numConnections;

      private:
        void readMessages( void );

      protected:
        NetworkController( void );
        NetMessage *parseMessage( char *data, int size );

        TCPsocket getTCPSocketFromID( int id );
        IPaddress getIPFromID( int id );

        NetMessage *popInMessage( void );
        void pushMessageOnQueue( NetMessage * message );
        void pushInMessage( NetMessage * message );
```



```
        // *** variables ***

        // used for checking udp/tcp sockets for activity
        SDLNet_SocketSet socketSet;

        // Creates new tcp connection to ip
        Connection *addConnection( int id, TCPsocket & tcpsocket,
                                   IPaddress ip );

        // removes and disconnects an tcp-connection
        virtual void removeConnection( int id );

        // UDPis connectionsless so theres no need to have more than one socket
        UDPsocket udpsocket;

        // map of connections
        Connections connections;
        // queue of incomming messages
        Messages messageInQueue;
        // queue of outgoing messages
        Messages UDPOutQueue;
        Messages TCPOutQueue;
    };
}

#endif
```

```
#include "ServerController.h"
#include "SDL.h"
#include "SDL_net.h"
#include "NetworkController.h"
#include "GameController.h"
#include <iostream>

using namespace blaster;
using namespace std;

// happy now gcc ?
ServerController *ServerController::instance = NULL;

ServerController::ServerController( void )
{
    running = true;

    setupSDL(  );
    setupSDLNet(  );
}

void ServerController::setupSDL(  )
{
    if ( SDL_Init( SDL_INIT_TIMER ) < 0 )
    {
        fprintf( stderr, "Couldn't initialize SDL: %s\n", SDL_GetError(  ) );
        exit( 1 );
    }
}

void ServerController::setupSDLNet( void )
{
    // initialize SDL_net
    if ( SDLNet_Init(  ) == -1 )
    {
        printf( "SDLNet_Init: %s\n", SDLNet_GetError(  ) );
        exit( 2 );
    }
}

void ServerController::run( void )
{
    long launchtime = SDL_GetTicks(  );
    long oldtime = 0;
    long newtime = 0;

    float time = 0.0f;
    float dt = 0.0f;

    NetworkController *network = NetworkController::getInstance(  );

    network->startServer( 1337 );

    while ( running )
    {
        newtime = SDL_GetTicks(  ) - launchtime;
        dt = ( float ) ( newtime - oldtime ) * 0.001f;
        time = ( float ) newtime *0.001f;

        oldtime = newtime;

        /*NetworkController * */ network = NetworkController::getInstance(  );
```



```
        network->think( time, dt );

        // Update the game world
        GameController::getInstance(  )->think( time, dt );

        network->flush(  );
    }

    // shutdown SDL_net
    SDLNet_Quit(  );
    SDL_Quit(  );
}


ServerController *ServerController::getInstance( void )
{
    if ( instance == NULL )
    {
        instance = new ServerController(  );
    }

    return instance;
}



int main( int argc, char **argv )
{
    cout << "Starting ServerController" << endl;
    ServerController *server;

    server = ServerController::getInstance(  );
    server->run(  );

    return ( 0 );
}
```

## ServerController.h



```
#ifndef SERVERCONTROLLER_H
#define SERVERCONTROLLER_H

#include "ServerNetworkController.h"

namespace blaster
{
    class ServerController
    {
      public:
        ~ServerController( void );
        static ServerController *getInstance( void );
        void think( void );
        void run( void );
        void quit( void )
        {
            running = false;
        }

      private:
        //vars
        static ServerController *instance;

          ServerController( void );
        void setupSDL( void );
        void setupSDLNet( void );

        bool running;
    };
}

#endif
```

## ServerNetworkController.cpp



```
#include "ServerNetworkController.h"
#include <iostream>
#include <string>

using namespace blaster;
using namespace std;

ServerNetworkController *ServerNetworkController::instance = NULL;


ServerNetworkController::ServerNetworkController( void ):NetworkController(  )
{
    nextID = 1;
    nextClientCheck = 0;
}

void ServerNetworkController::think( float time, float dt )
{
    // accept new connections
    if ( time > nextClientCheck )
    {
        checkForNewClients(  );

        nextClientCheck += 0.1f;

        if ( nextClientCheck < time )
            nextClientCheck = time;
    }

    // get all incomming stuff
    NetworkController::think( time, dt );
}


void ServerNetworkController::checkForNewClients( void )
{
    IPaddress *remoteip;
    TCPsocket tcpclient;
    Uint32 ipaddr;

    // try to accept a connection
    tcpclient = SDLNet_TCP_Accept( tcpServerSocket );

    // no connection-attempts
    if ( !tcpclient )
        return;

    cerr << "new connection detected..." << endl;

    // get the clients IP and port number
    remoteip = SDLNet_TCP_GetPeerAddress( tcpclient );
    if ( !remoteip )
    {
        cerr << "SDLNet_TCP_GetPeerAddress: " << SDLNet_GetError(  ) << endl;;
        return;
    }


    // print out the clients IP and port number
    ipaddr = SDL_SwapBE32( remoteip->host );
    cerr << "Accepted a connection from "
        << ( ipaddr >> 24 ) << "."
        << ( ( ipaddr >> 16 ) & 0xff ) << "."
```

## ServerNetworkController.cpp



```
        << ( ( ipaddr >> 8 ) & 0xff ) << "."
        << ( ipaddr & 0xff ) << " on port " << ( remoteip->port ) << endl;

    // add connection to io/queue
    addConnection( nextID, tcpclient, *remoteip );
    nextID++;
}


void ServerNetworkController::startServer( Uint16 port )
{
    IPaddress ip;

    // Resolve the argument into an IPaddress type
    if ( SDLNet_ResolveHost( &ip, NULL, port ) == -1 )
    {
        cerr << "Could not resolve localhost" << SDLNet_GetError(  ) << endl;
        exit( 3 );
    }

    cerr << "Resolved host" << endl;

    // open the server socket
    tcpServerSocket = SDLNet_TCP_Open( &ip );
    if ( !tcpServerSocket )
    {
        cerr << "Could not open TCP-server-socket: " << SDLNet_GetError(  ) <<
            endl;
        exit( 4 );
    }

    cerr << "TCP server socket created" << endl;

    // open udp-socket
    udpsocket = SDLNet_UDP_Open( port );
    if ( !udpsocket )
    {
        cerr << "SDLNet_UDP_Open: " << SDLNet_GetError(  ) << endl;
        exit( 2 );
    }

    cerr << "UDP socket opened" << endl;
}



ServerNetworkController *ServerNetworkController::getInstance( void )
{
    if ( instance == NULL )
    {
        instance = new ServerNetworkController(  );
    }

    return instance;
}
```

## ServerNetworkController.h



```
#ifndef SERVERNETWORKCONTROLLER_H
#define SERVERNETWORKCONTROLLER_H

#include "SDL.h"
#include "SDL_net.h"
#include <deque>
#include "ServerNetworkController.h"
#include "NetworkController.h"

namespace blaster
{
    class ServerNetworkController:public NetworkController
    {
      public:
        virtual void think( float time, float dt );
        virtual void startServer( Uint16 port );

        static ServerNetworkController *getInstance( void );
          virtual ~ ServerNetworkController( void )
        {
        }

      private:
        //vars

          TCPsocket tcpServerSocket;
        static ServerNetworkController *instance;
        int port;
        int nextID;

        //functions
          ServerNetworkController( void );

        void checkForNewClients( void );

        float nextClientCheck;
    };
}

#endif
```

```
#include "Font.h"
#include "SDL_endian.h"
#include <fstream>

// Font constructor. Read and upload the texture containing the font
Font::Font( const std::string & filename )
{
    fontname = filename;
}

void Font::reload( void )
{
    loadFont( fontname );
}

void Font::loadFont( const std::string & filename )
{
    Uint32 tmp = 0;
    Sint32 i;

    std::ifstream file;

    file.open( filename.c_str(  ), std::ios::binary );

    if ( !file.is_open(  ) )
        throw "Failed to open font\n";

    // The first 4 bytes should contain the number '6666' if this is a valid fon
t file.
    file.read( ( char * ) &tmp, sizeof( tmp ) );
    tmp = SDL_SwapLE32( tmp );  // Since we're working with values larger than o
ne byte, we need to handle byte-endianness

    if ( tmp != 6666 )
        throw "Wrong font format\n";

    file.read( ( char * ) &tmp, sizeof( tmp ) );
    textureWidth = SDL_SwapLE32( tmp );

    file.read( ( char * ) &tmp, sizeof( tmp ) );
    textureHeight = SDL_SwapLE32( tmp );

    file.read( ( char * ) &tmp, sizeof( tmp ) );
    fontHeight = SDL_SwapLE32( tmp );

    union
    {
        float f;
        Uint32 i;
    }
    converter;

    const GLfloat fTexWidth = ( float ) textureWidth;
    const GLfloat fTexHeight = ( float ) textureHeight;

    // Read each character from the bitmap
    for ( i = 0; i < 256; i++ )
    {
        // Read top v-coord
        file.read( ( char * ) &tmp, sizeof( tmp ) );
        chars[i].texCoords[1] = ( float ) SDL_SwapLE32( tmp ) / fTexHeight;

        // Read left u-coord
```



```
        file.read( ( char * ) &tmp, sizeof( tmp ) );
        chars[i].texCoords[0] = ( float ) SDL_SwapLE32( tmp ) / fTexWidth;

        // Read bottom v-coord
        file.read( ( char * ) &tmp, sizeof( tmp ) );
        chars[i].texCoords[3] = ( float ) SDL_SwapLE32( tmp ) / fTexHeight;

        // Read bottom u-coord
        file.read( ( char * ) &tmp, sizeof( tmp ) );
        chars[i].texCoords[2] = ( float ) SDL_SwapLE32( tmp ) / fTexWidth;

        // Is this character valid? (if not, a space will be displayed instead)
        file.read( ( char * ) &tmp, sizeof( tmp ) );
        chars[i].enabled = ( SDL_SwapLE32( tmp ) != 0 );

        // Read the character width
        file.read( ( char * ) &tmp, sizeof( tmp ) );
        converter.i = SDL_SwapLE32( tmp );
        chars[i].width = converter.f * fontHeight;
    }

    if ( ( file.eof(  ) ) || ( file.fail(  ) ) )
        throw "Font::loadFont - error reading from file";

    // Allocate two buffers to store the actual bitmap information before upload
ing it as a texture

    char *buffer, *buffer2;
    const int bufferSize = textureWidth * textureHeight;

    buffer = ( char * ) malloc( bufferSize );

    if ( buffer == 0 )
        throw "Unable to allocate memory";

    file.read( buffer, bufferSize );

    // We're done reading from the file
    file.close(  );

    // Allocate a buffer large enough to store both pixel data and alpha
    buffer2 = ( char * ) malloc( bufferSize * 2 );

    if ( buffer2 == NULL )
        throw "Unable to allocate memory";

    for ( i = 0; i < bufferSize; i++ )
    {
        buffer2[i * 2] = buffer[i];
        buffer2[i * 2 + 1] = buffer[i];
    }

    // Upload the bitmap as a texture
    glGenTextures( 1, &textureId );
    glBindTexture( GL_TEXTURE_2D, textureId );
    glTexImage2D( GL_TEXTURE_2D, 0, GL_LUMINANCE_ALPHA, textureWidth,
                  textureHeight, 0, GL_LUMINANCE_ALPHA, GL_UNSIGNED_BYTE,
                  buffer2 );
    glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR );
    glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR );
    glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP );
    glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP );
    glBindTexture( GL_TEXTURE_2D, 0 );
```

## Font.cpp



```
    free( buffer );
    free( buffer2 );
}

void Font::draw( const std::string & value )
{
    // Enable texturemapping
    glEnable( GL_TEXTURE_2D );

    // Select the texture associated with this font
    glBindTexture( GL_TEXTURE_2D, textureId );

    char c;                     // holds the current character in the string
    GLfloat x = 0.0f;

    // Draw each character in the string as a textured rectangle (two triangles)
    for ( Uint32 i = 0; i < value.length(  ); i++ )
    {
        c = value[i];

        // Check if the character is represented in the texture
        if ( !chars[c].enabled )
            c = 32;             // replace with the space character

        glBegin( GL_TRIANGLE_STRIP );

        // Top left corner of the current character
        glTexCoord2f( chars[c].texCoords[0], chars[c].texCoords[1] );
        glVertex2f( x, 0.0f );

        // Bottom left
        glTexCoord2f( chars[c].texCoords[0], chars[c].texCoords[3] );
        glVertex2f( x, fontHeight );

        // Move to the right according to the character width
        x += chars[c].width;

        // Top right
        glTexCoord2f( chars[c].texCoords[2], chars[c].texCoords[1] );
        glVertex2f( x, 0.0f );

        // Bottom right
        glTexCoord2f( chars[c].texCoords[2], chars[c].texCoords[3] );
        glVertex2f( x, fontHeight );

        glEnd(  );
    }

    glDisable( GL_TEXTURE_2D );
}

//Font destructor
Font::~Font( void )
{
    glDeleteTextures( 1, &textureId );
}
```

## Font.h



```
#ifndef FONT_H
#define FONT_H

#include "SDL_opengl.h"
#include <string>

class Character
{
  public:
    GLfloat texCoords[4];
    GLfloat width;

    bool enabled;
};

class Font
{
  public:
    Font( const std::string & filename );
     ~Font( void );

    void draw( const std::string & value );

    GLfloat getFontHeight( void )
    {
        return fontHeight;
    }

    void reload( void );

  private:
    void loadFont( const std::string & filename );

    GLfloat fontHeight;
    Character chars[256];
    GLuint textureWidth;
    GLuint textureHeight;
    GLuint textureId;

    std::string fontname;
};

#endif // FONT_H
```

## GameView.cpp



```
#include "ClientController.h"
#include "GameView.h"
#include "GameController.h"
#include "Level.h"
#include "Player.h"

using namespace blaster;

GameView::GameView( View * parent ):View( parent ), viewPosition( 0, 0 )
{
}

void GameView::drawContent( void )
{
    if ( parent )
        parent->drawContent(  );

    Level *level = Level::getInstance(  );
    Map *map = level->getMap(  );

    Objects *ships = level->getShips(  );
    Objects *bullets = level->getBullets(  );

    Uint32 localClient =
        ClientController::getInstance(  )->getLocalPlayerID(  );

    Ship *ship =
        GameController::getInstance(  )->getPlayer( localClient )->
        getShip(  );

    glMatrixMode( GL_PROJECTION );

    glPushMatrix(  );

    if ( ship != NULL )
    {
        viewPosition = ship->getPosition(  );
    }

    glTranslatef( -viewPosition.x, -viewPosition.y, 0.0f );

    glMatrixMode( GL_MODELVIEW );

    map->draw(  );

    for ( Objects::iterator i = bullets->begin(  ); i != bullets->end(  );
          i++ )
    {
        ( *i )->draw(  );
    }

    for ( Objects::iterator i = ships->begin(  ); i != ships->end(  ); i++ )
    {
        ( *i )->draw(  );
    }

    glMatrixMode( GL_PROJECTION );

    glPopMatrix(  );
}
```

## GameView.h



```
#ifndef GAMEVIEW_H
#define GAMEVIEW_H

#include "View.h"
#include "Vector.h"

namespace blaster
{

    class GameView:public View
    {
      public:
        GameView( View * parent );
        virtual void drawContent( void );
      private:
          Vector viewPosition;
    };

}

#endif
```

```
#include "ClientController.h"
#include "ScoreView.h"
#include "InputController.h"
#include "GameController.h"
#include "Player.h"

#include <iostream>

using namespace blaster;
using namespace std;

ScoreView::ScoreView( View * parent ):View( parent )
{
    // get this from somewhere else
    screenWidth = 640;
    screenHeight = 480;

    f = new Font( "verdana.fnt" );
}

void ScoreView::drawContent( void )
{
    if ( parent )
        parent->drawContent(  );

    if ( !InputController::getInstance(  )->viewScore )
    {
        return;
    }

    glMatrixMode( GL_PROJECTION );

    glPushMatrix(  );

    glLoadIdentity(  );

    /* left, right, bottom, top... */
    glOrtho( 0.0f, screenWidth, screenHeight, 0.0f, -1.0f, 1.0f );

    glMatrixMode( GL_MODELVIEW );

    glPushMatrix(  );

    glLoadIdentity(  );

    // calculate box size

    float x1 = screenWidth / 6;
    float y1 = screenHeight / 6;

    float boxWidth = screenWidth - ( screenWidth / 6 ) * 2;
    float x2 = x1 + boxWidth;
    float y2 = screenHeight - ( screenHeight / 6 );

    glColor4f( 0.5, 0.5, 0.8, 0.5 );
    glBegin( GL_QUADS );
    glVertex2f( x1, y1 );
    glVertex2f( x2, y1 );
    glVertex2f( x2, y2 );
    glVertex2f( x1, y2 );
    glEnd(  );

    glTranslatef( x1 + 10, y1 + 10, 0.0 );
```



```
    string playername;
    char score[10];

    Ship *ship;

    Uint32 localClient =
        ClientController::getInstance(  )->getLocalPlayerID(  );

    Player *player = GameController::getInstance(  )->getPlayer( localClient );
// localid

    glColor3f( 1.0, 1.0, 1.0 );

    // insert magical forloop that gets all players and for each does
    // start loop
    ship = player->getShip(  );
    if ( ship )
    {
        playername = player->getName(  );
        glTranslatef( 0, f->getFontHeight(  ) + 10, 0.0 );
        f->draw( playername.c_str(  ) );

        // right collumn
        sprintf( score, "%d", player->getScore(  ) );
        glTranslatef( boxWidth - 100, 0, 0.0 );
        f->draw( score );

        // back again
        glTranslatef( -( boxWidth - 100 ), 0, 0.0 );
    }
    // end loop

    glPopMatrix(  );

    glMatrixMode( GL_PROJECTION );
    glPopMatrix(  );

    glMatrixMode( GL_PROJECTION );
    glPopMatrix(  );
}

void ScoreView::init( void )
{
    if ( parent )
        parent->init(  );

    f->reload(  );
}
```

## ScoreView.h



```
#ifndef SCOREVIEW_H
#define SCOREVIEW_H

#include "View.h"
#include "Font.h"

namespace blaster
{

    class ScoreView:public View
    {
      public:
        ScoreView( View * parent );
        virtual void drawContent( void );
        void init( void );

      private:
          Font * f;
        int screenWidth;
        int screenHeight;
    };

}

#endif
```

## StatusBar.cpp



```
#include "StatusBar.h"
#include "SDL.h"
#include "SDL_opengl.h"
#include <iostream>

using namespace blaster;
using namespace std;

StatusBar::StatusBar( float width, float height )
{
    colorGood = Vector( 0, 1, 0 );
    colorBad = Vector( 1, 0, 0 );

    this->width = width;
    this->height = height;
}

void StatusBar::draw( float max, float current )
{
    float goodness = ( current / max );

    Vector color = colorGood * goodness + colorBad * ( 1 - goodness );

    // draw bar
    glColor3f( color.x, color.y, color.z );
    // padding

    float innerpad = 4;

    float x1 = 0;
    float y1 = 0;

    float x2 = width - innerpad * 2;
    float y2 = height - innerpad * 2;

    x2 = x2 * goodness;

    x1 = x1 + innerpad;
    y1 = y1 + innerpad;

    x2 = x2 + innerpad;
    y2 = y2 + innerpad;


    glBegin( GL_QUADS );
    glVertex2f( x1, y1 );
    glVertex2f( x2, y1 );
    glVertex2f( x2, y2 );
    glVertex2f( x1, y2 );
    glEnd(  );

    // draw box around bar
    glColor3f( 1.0, 1.0, 1.0 );
    glBegin( GL_LINES );
    glVertex2f( 0.0, 0.0 );
    glVertex2f( width, 0.0 );

    glVertex2f( width, 0.0 );
    glVertex2f( width, height );

    glVertex2f( width, height );
    glVertex2f( 0.0, height );
```

## StatusBar.cpp



```
    glVertex2f( 0.0, height );
    glVertex2f( 0.0, 0.0 );
    glEnd(  );

}
```

## StatusBar.h



```
#ifndef STATUSBAR_H
#define STATUSBAR_H

#include "Vector.h"

namespace blaster
{

    class StatusBar
    {
      public:
        StatusBar( float width, float height );

        void draw( float max, float value );


      private:
        float width;
        float height;

        Vector colorGood;
        Vector colorBad;
    };

}

#endif
```

```
#include "ClientController.h"
#include "StatusView.h"
#include "Ship.h"
#include "Player.h"
#include "GameController.h"

#include <iostream>

using namespace blaster;
using namespace std;

StatusView::StatusView( View * parent ):View( parent )
{
    screenWidth = 640;
    screenHeight = 480;

    barWidth = 100.0;
    barHeight = 20.0;

    // padding on left/right for each bar
    barXpad = ( screenWidth / 4 - barWidth ) / 2;

    ammoBar = new StatusBar( barWidth, barHeight );
    fuelBar = new StatusBar( barWidth, barHeight );
    energyBar = new StatusBar( barWidth, barHeight );

    f = new Font( "verdana.fnt" );
    f->reload(  );
}

void StatusView::drawContent( void )
{
    if ( parent )
        parent->drawContent(  );

    Uint32 localClient =
        ClientController::getInstance(  )->getLocalPlayerID(  );

    Player *player = GameController::getInstance(  )->getPlayer( localClient );

    Ship *ship = player->getShip(  );

    // can't draw anything without a ship
    if ( !ship )
        return;

    glMatrixMode( GL_PROJECTION );

    glPushMatrix(  );

    glLoadIdentity(  );

    /* left, right, bottom, top... */
    glOrtho( 0.0f, screenWidth, screenHeight, 0.0f, -1.0f, 1.0f );


    glMatrixMode( GL_MODELVIEW );

    glPushMatrix(  );

    glLoadIdentity(  );
```



```
    // draw bars

    // scorebox
    glTranslatef( barXpad, screenHeight - ( barHeight + 5 ), 0.0f );
    glColor3f( 0.0, 0.7, 0.0 );
    glBegin( GL_LINES );
    glVertex2f( 0.0, 0.0 );
    glVertex2f( barWidth, 0.0 );

    glVertex2f( barWidth, 0.0 );
    glVertex2f( barWidth, barHeight );

    glVertex2f( barWidth, barHeight );
    glVertex2f( 0.0, barHeight );

    glVertex2f( 0.0, barHeight );
    glVertex2f( 0.0, 0.0 );
    glEnd(  );


    glTranslatef( barWidth + 2 * barXpad, 0, 0.0f );
    ammoBar->draw( ship->ammoCapacity(  ), ship->getAmmo(  ) );


    glTranslatef( barWidth + 2 * barXpad, 0, 0.0f );
    fuelBar->draw( ship->fuelCapacity(  ), ship->getFuel(  ) );

    glTranslatef( barWidth + 2 * barXpad, 0, 0.0f );
    energyBar->draw( ship->energyCapacity(  ), ship->getEnergy(  ) );

    // draw strings
    glLoadIdentity(  );
    glColor3f( 1.0, 1.0, 1.0 );

    char buf[256];

    glTranslatef( barXpad,
                  screenHeight - ( barHeight + 5 ) - f->getFontHeight(  ),
                  0.0f );
    glTranslatef( barWidth + 2 * barXpad, 0, 0.0f );
    sprintf( buf, "Ammo: %d", ship->getAmmo(  ) );
    f->draw( buf );

    glTranslatef( barWidth + 2 * barXpad, 0, 0.0f );
    sprintf( buf, "Fuel: %f", ship->getFuel(  ) / 1 );
    f->draw( buf );

    glTranslatef( barWidth + 2 * barXpad, 0, 0.0f );
    sprintf( buf, "Energy: %f", ship->getEnergy(  ) / 1 );
    f->draw( buf );


    glLoadIdentity(  );
    glColor3f( 1.0, 1.0, 1.0 );
    glTranslatef( barXpad + 2, screenHeight - ( barHeight + 3 ), 0.0f );
    sprintf( buf, "Score: %d", player->getScore(  ) );
    f->draw( buf );


    glPopMatrix(  );

    glMatrixMode( GL_PROJECTION );
```

## Statusview.cpp



```
    glPopMatrix(  );


    glMatrixMode( GL_PROJECTION );

    glPopMatrix(  );

}


void StatusView::init( void )
{
    if ( parent )
        parent->init(  );

    f->reload(  );
}
```

## Statusview.h



```
#ifndef STATUS_H
#define STATUS_H

#include "View.h"
#include "Font.h"
#include "Vector.h"
#include "StatusBar.h"

namespace blaster
{

    class StatusView:public View
    {
      public:
        StatusView( View * parent );
        virtual void drawContent( void );
        void init( void );

      private:
          Font * f;
        int screenWidth;
        int screenHeight;
        float barWidth;
        float barHeight;
        float barXpad;

        void drawBar( float width, float height, float cur, float max );

        StatusBar *ammoBar;
        StatusBar *fuelBar;
        StatusBar *energyBar;   // crunchy
    };

}

#endif
```

## View.cpp



```
#include "View.h"

using namespace blaster;

View::View( View * p )
{
    parent = p;
}

View::~View( void )
{
    delete parent;
}

void View::drawFrame( void )
{
    glClear( GL_COLOR_BUFFER_BIT );

    drawContent(  );

    glFlush(  );

    SDL_GL_SwapBuffers(  );
}

void View::init( void )
{
    if ( parent )
        parent->init(  );
}
```

## View.h



```
#ifndef VIEW_H
#define VIEW_H

#include "SDL.h"
#include "SDL_opengl.h"

namespace blaster
{

    class View
    {
      public:
        View( View * p );
        virtual ~ View( void );

        virtual void init( void );
        virtual void drawFrame( void );
        virtual void drawContent( void ) = 0;

      protected:
          View * parent;
    };

}

#endif                          // VIEW_H
```

```
#include "Matrix.h"

#include "Vector.h"

#include "Math.h"
#include "String.h"

#include <iostream>

Matrix::Matrix( void )
{
    m = new float[16];

    loadIdentity(  );
}

Matrix::Matrix( const Matrix & n )
{
    m = new float[16];

    memcpy( m, n.m, sizeof( float ) * 16 );
}

Matrix::~Matrix( void )
{
    delete m;
}

void Matrix::loadIdentity( void )
{
    m[0] = 1.0f;
    m[4] = 0.0f;
    m[8] = 0.0f;
    m[12] = 0.0f;
    m[1] = 0.0f;
    m[5] = 1.0f;
    m[9] = 0.0f;
    m[13] = 0.0f;
    m[2] = 0.0f;
    m[6] = 0.0f;
    m[10] = 1.0f;
    m[14] = 0.0f;
    m[3] = 0.0f;
    m[7] = 0.0f;
    m[11] = 0.0f;
    m[15] = 1.0f;
}

Matrix Matrix::translateMatrix( float a, float b, float c )
{
    Matrix mat;

    mat.m[12] = a;
    mat.m[13] = b;
    mat.m[14] = c;

    return mat;
}

Matrix Matrix::scaleMatrix( float a, float b, float c )
{
    Matrix mat;
```



```
    mat.m[0] = a;
    mat.m[5] = b;
    mat.m[10] = c;

    return mat;
}

Matrix Matrix::rotateZMatrix( float angle )
{
    Matrix mat;

    const float cosa = cos( angle );
    const float sina = sin( angle );

    mat.m[0] = cosa;
    mat.m[4] = -sina;
    mat.m[1] = sina;
    mat.m[5] = cosa;

    return mat;
}

Matrix & operator*=( Matrix & left, const Matrix & right )
{
    float *p = left.m;
    float *q = right.m;

    left.m = new float[16];

    for ( int j = 0; j < 4; j++ )
    {
        for ( int i = 0; i < 4; i++ )
        {
            left.m[i + 4 * j] = p[0 + i] * q[4 * j + 0] +
                p[4 + i] * q[4 * j + 1] +
                p[8 + i] * q[4 * j + 2] + p[12 + i] * q[4 * j + 3];
        }
    }

    delete p;

    return left;
}

Vector operator*( Matrix & matrix, const Vector & v )
{
    float a =
        v.x * matrix.m[0] + v.y * matrix.m[4] + v.z * matrix.m[8] +
        1 * matrix.m[12];
    float b =
        v.x * matrix.m[1] + v.y * matrix.m[5] + v.z * matrix.m[9] +
        1 * matrix.m[13];
    float c =
        v.x * matrix.m[2] + v.y * matrix.m[6] + v.z * matrix.m[10] +
        1 * matrix.m[14];

    return Vector( a, b, c );
}

std::ostream & operator<<( std::ostream & out, const Matrix & m )
{
    for ( int j = 0; j < 4; j++ )
    {
```

## Matrix.cpp



```
        out << "[";

        for ( int i = 0; i < 4; i++ )
        {
            out << "\t" << m.m[i + 4 * j];
        }

        out << "\t]" << std::endl;
    }

    return out;
}
```

## Matrix.h



```
#ifndef MATRIX_H
#define MATRIX_H

#include <iostream>

class Vector;

class Matrix
{
  public:
    Matrix( void );
    Matrix( const Matrix & n );
     ~Matrix( void );

    void loadIdentity( void );

    static Matrix translateMatrix( float a, float b, float c );
    static Matrix scaleMatrix( float a, float b, float c );
    static Matrix rotateZMatrix( float angle );


    float *m;
};

Vector operator*( Matrix & m, const Vector & v );
Matrix & operator*=( Matrix & left, const Matrix & right );

std::ostream & operator<<( std::ostream & out, const Matrix & m );

#endif // MATRIX_H
```

```
#include "Vector.h"
#include "Matrix.h"
#include <iostream>

using namespace std;

const Vector Vector::ZERO( 0.0f, 0.0f, 0.0f );

const Vector Vector::operator+( const Vector & v ) const
{
    return Vector( x + v.x, y + v.y, z + v.z );
}

const Vector Vector::operator-( const Vector & v ) const
{
    return Vector( x - v.x, y - v.y, z - v.z );
}

const float Vector::operator*( const Vector & v ) const
{
    return x * v.x + y * v.y + z * v.z;
}

const Vector Vector::operator*( const float value ) const
{
    return Vector( x * value, y * value, z * value );
}

const Vector Vector::operator/( const float value ) const
{
    return Vector( x / value, y / value, z / value );
}

const Vector Vector::cross( const Vector & v ) const
{
    return Vector( 0, 0, 0 );
}

void Vector::transform( Matrix & matrix )
{
    Vector v = matrix * ( *this );

    x = v.x;
    y = v.y;
    z = v.z;
}

Vector & operator+=( Vector & left, const Vector & right )
{
    left.x += right.x;
    left.y += right.y;
    left.z += right.z;

    return left;
}

Vector & operator*=( Vector & left, const float v )
{
    left.x *= v;
    left.y *= v;
    left.z *= v;

    return left;
```



```
}

std::ostream & operator<<( std::ostream & out, const Vector & v )
{
    out << "(" << v.x << "," << v.y << "," << v.z << ")";

    return out;
}
```

```
#ifndef VECTOR_H
#define VECTOR_H

#include "stdio.h"
#include "math.h"

#include <iostream>

class Matrix;

class Vector
{
  public:
    Vector( void ):x( 0.0f ), y( 0.0f ), z( 0.0f )
    {
    }
    Vector( float x, float y, float z = 0.0f ):x( x ), y( y ), z( z )
    {
    }
    Vector( const Vector & v ):x( v.x ), y( v.y ), z( v.z )
    {
    }

    const Vector operator+( const Vector & v ) const;
    const Vector operator-( const Vector & v ) const;
    const float operator*( const Vector & v ) const;
    const Vector operator*( const float value ) const;
    const Vector operator/( const float value ) const;

//    inline float dot( Vector &rvalue );
    const Vector cross( const Vector & rvalue ) const;

    void transform( Matrix & matrix );

    const float length( void ) const
    {
        return sqrt( x * x + y * y + z * z );
    }
    const Vector normalize( void ) const
    {
        return *this / length(  );
    }

    static const Vector ZERO;

    float x;
    float y;
    float z;
};

std::ostream & operator<<( std::ostream & out, const Vector & v );
Vector & operator+=( Vector & left, const Vector & right );
Vector & operator*=( Vector & left, const float v );

#endif
```