How does the client work?

1 General

The client consists of:

`Client.cs`—Startup class
`Control/Controller.cs`—Manages the application
`Control/NetConnector.cs`—Interface between controller and server
`Model/Creature.cs`—location and representation of a creature
`Model/Field.cs`—representation of a tile
`Model/LocalModel.cs`—maintains grid of known tiles and list of creatures
`View/Imagebuffer.cs`—maintains a dictionary of images with string keys
`View/TileDisplayForm.cs`—gets player input and displays model

Its function is based on three events:

1. **Program Start**
   When the program starts the display form and model are created. Everything is made ready for the connection to the server to be made, but this is not done automatically.
2. **Text Input Event**
   When text ended with an enter is typed in the textbox on the display form, it is sent to the controller. If the text is "connect", the connection to the server is made. Otherwise the command is passed on to the NetConnect class to be sent to the server.
3. **Server Input Event**
   When the server sends updates, they are passed on to the controller. The controller updates the model, then makes the form redraw the world.

These three events and the way they are handled are described in detail in the next three sections.

2 Program Start

`Client/Main()`
Runs an instance of the TileDisplayForm class.

`View/TileDisplayForm()`
Creates a form which I'll call 'The Form'.

Calls InitializeComponent().

`View/TileDisplayForm.InitializeComponent()`
Built by the VS2012 Designer.

Creates a textbox and a picturebox and registers the form.load and textbox.textchanged events. Once done the load event will fire.

`View/TileDisplayForm.TileDisplayForm_Load()`
Creates new ImageBuffer with default size 64x64 and a new Controller.

`View/ImageBuffer(64, 64)`
Calls SetDefault(64,64) which creates a default image which is a red square of 64x64 pixels.

Creates a dictionary from strings to images, and adds the pictures “wall”, “floor” and “player” from the images directory.

`Control/Controller(The Form)`
Registers The Form.

Creates a controller which I’ll call ‘The Controller’.

Creates a new LocalModel with size 101x101x101 and a NetConnector with IP 127.0.0.1 and port 4502.

`Model/LocalModel(101,101,101)`
Creates a LocalModel which I’ll call ‘The Model’.

Creates an array of 101x101x101 Fields and an empty List of Creatures.

`Control/Netconnector(The Controller, "127.0.0.1", 4502)`
Registers The Controller.

Creates a NetConnector which I’ll call ‘The Connector’.

Creates a TCPClient and parses the IP to an IPAddress.

3 Text Input Event

`View/TileDisplayForm.TextBox1_TextChanged(_, _)`
If the text in the textbox ends with a newline character, “\r\n” (carriage return + linefeed) is removed from the text, after which The Controller.sendInput(input) is called, after which the textbox is cleared.

`Control/Controller.sendInput(input)`
If the input is “connect”, the Connect method is called (see subsection 2.1), otherwise The Connector.sendData(input) is called.

`Control/NetConnector.sendData(input)`
Replaces all semicolons with colons, since semicolons are used to split lines of input to the server.

Adds a semicolon to the end of the input.

Converts the string to a byte array and sends it as once chunk.

3.1 Connect

`Control/Controller.Connect()`
calls The Connector.Connect().

`Control/NetConnector.Connect()`
Checks if connection is already up using the “running” flag, returns if so.

Calls BeginConnect(IP, port, connectionMade, null) on the TCPClient, which will call back to connectionMade when complete. Sets the “running” flag to true.

If the connection fails, it prints a debug message and calls Disconnect(), which just sets the “running” flag to false.

`Control/NetConnector.connectionMade(_)`
Finishes connection using the client’s EndConnect method. If that fails, a debug message and Disconnect() are called.

Calls startReading().

`Control/NetConnector.startReading()`
Checks if the "running" flag is set. If not, it closes the tcp client.

If so, it creates a byte buffer and calls the BeginRead(buffer, 0, 1024, dataReceived, buffer) method on the client's network stream, which will call back to dataReceived with the buffer as it's argument (encapsulated in an IAsyncResult) when data is received. This is the trigger for the Server Input Event.

4 Server Input Event

`Control/NetConnector.dataReceived(data)`
Checks if the "running" flag is set. If not, it closes the tcp client and returns.

If so, it reads the data and adds it to a message buffer. If there is more data left on the stream, it calls startReading (see above). When all data is read, it calls splitAndHandle.

If the connection fails, a debug message is printed and Disconnect() is called.

`Control/NetConnector.splitAndHandle()`
Splits the received data on semicolons, then adds all of the resulting strings to a list.

If the data ends with a semicolon (as it should), the message buffer should be empty. If the data does not end with a semicolon, we assume more data will be coming to complete the line, so the message buffer should be the incomplete line. We achieve this by putting the last element (everything after the last semicolon) in the message buffer. After that, we remove the last item from the list.

Calls The Controller.doUpdate(list).

Calls startReading() (see above) which will trigger this event again when more data is available.

`Control/Controller.doUpdate(list)`
Calls The Model.update(list).

Calls The Form.drawModel(The Model). (see subsection 3.1).

`LocalModel.update(list)`
Sets the "updating creatures" flag to false

Loops through each string in the list and splits it on commas.

The first part of each command gives the category it's in. The commands are switched on this first part and handed off to specialized methods. (see section 4 for details.)

4.1 Drawing the model

`View/TileDisplayForm.drawModel(The Model)`
Checks if the "can draw" flag is set. If not, it returns.

Sets the "can draw" flag to false.

Creates a bitmap the size of the picturebox, and creates a graphics object for it.

Blacks out the entire bitmap.

Calculates the center of the drawing area.

Calculates the number of tiles that will fit the screen.

Calculates the position of the player on the grid (always in the center).

A loop goes through indexes which are relative positions to the player, for an amount of tiles that will fill the screen. For example, if the screen would fit 11x7 tiles, the loop would go from -5 to 5 for the x-values, and from -3 to 3 for the y-values.

Inside the loop the following two things are done:

Checks if there is a field at the position indicated by player position plus the shift from the loop.

If so, it gets the field at that position from The Model, extracts it's representation, gets the image from the ImageBuffer, and draws it onto the bitmap in the right place, at 64x64 pixels.

Another loop goes over all creatures in the creature list of the model. Inside it, a check is done to see if it's in the view. If so, it's representation is extracted, the image is retrieved from the ImageBuffer, and it is drawn onto the right position (on top of the tile) at 32x32 pixels.

After this the graphics object is disposed of, and the picturebox image is set to the bitmap we've been drawing on.

The "can draw" flag is set to true.

5 Recognized Commands, and actual handling

5.1 LOGIN commands

Server sends "LOGIN,COMPLETE" when login is complete.

Are recognized, but not handled.

5.2 PLAYER commands

Server sends "PLAYER,POSITION,x,y,z" when player position has changed.

Handled by:

`Model/LocalModel.updatePlayer(input)`
Checks if input[1] is "POSITION"

If so: gets x,y,z from the input.

Checks if the player has moved using the current player position and the new values.

If so: calls shiftMap(x,y,z).

`Model/LocalModel.shiftMap(x, y, z)`
Calculates the difference between the current and new position.

Creates a new grid, at the same size as the original one.

Loops over all positions of the old grid. If the field would be in bounds if it were shifted, it's put it it's new position on the new grid. Else, it's dumped.

Updates the grid to be the new shifted one, and updates the current player position.

5.3 TILE commands

Server sends "TILE,DETECTED,x,y,z,representation" when a tile is seen.

Handled by:

`Model/LocalModel.updateTile(input)`
Checks if input[1] is "DETECTED".

If so: gets x,y,z and representation from the input.

Calculates map position from difference with player position.

Adds a new Field to the map, with the correct representation.

5.4 CREATURE commands

Server sends "CREATURE,DETECTED,x,y,z,representation" when a creature is seen. (BUG: nothing is sent when a creature is no longer in sight, which leaves creatures moving out of range on the map.)

Handled by:

`Model/LocalModel.updateCreature(input)` Checks if "updating creatures" flag is set. If not, resets the creature list and sets the flag. This makes sure we have no ghosts of old creatures on our view, but is not graceful and causes problems.

Checks if input[1] is "DETECTED".

If so: gets x,y,z and representation from the input.

Calculates map position from difference with player position.

Adds a new Creature to the creature list.