Indoor Positioning System

Challenge Problems and Resources

Developed by:

The teachers, students, and mentors in the Gaming Research Integration for Learning Laboratory® (GRILL®)
Summer 2014
# TABLE OF CONTENTS

1. **CHALLENGE PROBLEM: INDOOR POSITIONING SYSTEM** ........................................... 3
   1.1. **The Tools** ............................................................................................................. 3
   1.2. **The Challenge** ..................................................................................................... 3

2. **OPTION #2: TRACK A CUSTOMER IN A GAME ENVIRONMENT** .......................... 4
   2.1. **The Tools** ............................................................................................................. 4
   2.2. **The Challenge** ..................................................................................................... 4

3. **TUTORIALS OVERVIEW** ............................................................................................ 5

4. **ANDROID STUDIO** .................................................................................................... 5
   4.1. **Downloading Android Studio** ............................................................................... 5
   4.2. **Configuring Android Studio** ................................................................................. 6
       4.2.1. Setting up an Android Virtual Device ............................................................. 7
       4.2.2. Testing the virtual device .................................................................................. 9
   4.3. **Programming an App** ........................................................................................ 9
       4.3.1. Adding Text .................................................................................................... 9
       4.3.2. Adding a button ............................................................................................ 11
       4.3.3. Testing your app .......................................................................................... 13

5. **WINFORM RENDERING** ............................................................................................ 13
   5.1. **Starting a New Project** ....................................................................................... 13
   5.2. **Setting Up the Form** .......................................................................................... 14
       5.2.1. The Properties Window ..................................................................................... 14
       5.2.2. Add a PictureBox ............................................................................................ 16
   5.3. **Writing the Code** .............................................................................................. 18
       5.3.1. Rendering to a Bitmap ...................................................................................... 18
       5.3.2. PanNing and Zooming .................................................................................... 22
   5.4. **All Code** ........................................................................................................... 27
   5.5. **Conclusion** ......................................................................................................... 30

6. **XBEE S2 SETUP** ...................................................................................................... 33
   6.1. **Software Setup** .................................................................................................. 34
   6.2. **Hardware Setup** ............................................................................................... 34
   6.3. **X-CTU** .............................................................................................................. 35

7. **EXPORTING .SKP FILES FROM SKETCHUP TO UNITY WITH TEXTURES** .......... 44
   7.1. **Exporting SketchUp Models** .............................................................................. 44
   7.2. **Importing Models into Unity** ............................................................................. 45
1. CHALLENGE PROBLEM: INDOOR POSITIONING SYSTEM

Modeling and Simulation professionals often leverage off-the-shelf technologies to provide a basis for virtual environments of training or simulation. Utilizing existing technologies reduces the economic investment required to provide a customized environment; therefore, the business or industry moves forward without a large investment in the development of the simulation. This challenge provides students with an opportunity to create an app and virtual environment that allows a user to track an object in an indoor space. This is similar to what industry professionals do, but with consumerism as the topic of interest.

The goal of indoor positioning systems (IPS) is to leverage magnetic, sensor, and/or network technologies to wirelessly locate objects or people inside buildings. This technology is currently being used to enhance location detection during emergency response situations, improve customer satisfaction and experience in retail stores, and serve as a navigation aid to users.

Owners of a multi-level mall want to know where the majority of customers spend their time. This way, they can develop a more efficient store layout and can personalize every customer’s shopping experience. Participating shoppers will download an app on their smartphone that will track their position and send them targeted information dependent upon their location. This material could include coupons, directions, new store openings, and other pertinent information. The smartphone targeted needs to reflect the mobile device the customers use most frequently.

The owners are willing to spend a sizable amount of money to fund this venture, but would prefer to keep the cost low. The anchor stores are willing to house tracking equipment as long as it is no larger than .25L and takes less power than an incandescent light bulb.

1.1. THE TOOLS

This challenge problem is focused on tracking an object inside a building using indoor positioning systems. Math concepts such as triangulation, fingerprinting, trilateration, multilateration (MLAT), dead reckoning, received signal strength indicator (RSSI), location patterning techniques, and operational phases should be considered in order to convert data for mapping. Technology such as Wi-Fi routers, Bluetooth beacons, nodes, software development kits, and a smartphone or tablet could be utilized.

1.2. THE CHALLENGE
Create an app that tracks customer movement in which both accuracy and cost-effectiveness are optimized. This app should have the capacity to record x, y, and z positioning to account for the multi-level floor plans.

2. OPTION #2: TRACK A CUSTOMER IN A GAME ENVIRONMENT

This challenge problem focuses on creating, importing, and implementing a virtual 3D model of a building in order to track a remote person walking through a building in real time.

Note: It would serve the students well for testing purposes to utilize a model of their school in the place of a shopping mall.

2.1. THE TOOLS

Tools such as Unity, SketchUp, computer coding and an Android App could be utilized to work this challenge solution.

2.2. THE CHALLENGE

Track a consumer in a 3D model of the shopping mall using an off-the-shelf game engine.
3. TUTORIALS OVERVIEW

Wright Scholars, in collaboration with educators and the GRILL® team, created the tutorials described below as possible solutions to solve the challenge problems. At the time of creation these were working tutorials; however, with software updates and changes in technology, additional steps may be required. Teachers are encouraged to communicate any issues, problems, or suggested changes to these tutorials to ensure the dissemination of helpful materials to support challenge problem implementation. Below is a description of each tutorial, followed by the full tutorials.

Table 1: Tutorials Provided for Challenge Problem

<table>
<thead>
<tr>
<th>Challenge Problem</th>
<th>Tutorials</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indoor Positioning System</td>
<td>Android Studio (Section 4), WinForms Rendering (Section 5),</td>
</tr>
<tr>
<td></td>
<td>XBee S2 Setup (Section 6), Exporting SKP Files from</td>
</tr>
<tr>
<td></td>
<td>SketchUp to Unity with Textures (Section 7).</td>
</tr>
</tbody>
</table>

4. ANDROID STUDIO

This tutorial documents how to set up Android Studio and create a simple App with buttons and text on Windows 7.

Suggested materials for this tutorial include a computer and an Android device (tablet or phone).

This tutorial includes the following topics:

- Downloading Android Studio
- Configuring Android Studio
- Programming an App

4.1. DOWNLAODING ANDROID STUDIO

1. Download and install the latest version of Android Studio from the following website: https://developer.android.com/sdk/installing/studio.html.
2. Additionally, look under Program Files for a “Java” folder, and verify that jdk 1.6 or above is installed.
3. If jdk 1.6 or above is not installed, download and install the latest version of Java SE from the following website: http://www.oracle.com/technetwork/java/javase/downloads/index.html
4.2. CONFIGURING ANDROID STUDIO

1. Upon opening Android Studio, select New Project..., and give your app a name, such as MyApp. Leave the Company Domain and Project location alone and click Next.
2. Select an appropriate minimum SDK (e.g. API 16: Android 4.1 (Jelly Bean)) and click Next.
3. Select Blank Activity and click Next.
4. Name the Activity (e.g. ButtonActivity) and click Finish.
5. If the Windows Firewall blocks some features of the program, ensure that the Private networks checkbox is checked and click Allow access.

**Note:** Android Studio may take a moment building the Gradle project info.

6. Navigate to the Android SDK Manager.

7. Once it finishes loading, install the recommended new packages by clicking the Install... packages... button. If a warning message displays concerning a folder failing to be moved, close Android Studio (leaving the SDK Manager running) and click Yes to continue. If Android SDK fails to install any packages due to access denial, run Android Studio as an administrator and install the packages.

4.2.1. SETTING UP AN ANDROID VIRTUAL DEVICE

1. Navigate to the AVD Manager.

2. Click the Create... button.

The Edit Android Virtual Device (AVD) screen displays.

3. Name the device after the device to be used (e.g. “Nexus_7”).
4. Select the Device type (e.g. Nexus 7 (1200 x 1920)).
5. Select a Target SDK (e.g. Android 4.4.2 – API Level 19).
6. Select a CPU/ABI (e.g. ARM (armeabi-v7a)).
7. Select the Skin with dynamic hardware controls.
8. Change the RAM to 768.
9. Click OK, and close the AVD Manager.
10. Click on app and select Edit Configurations...
11. Under Target Device, select Emulator.
12. Under Prefer Android Virtual Device, select your device name.
13. Click OK.
4.2.2. TESTING THE VIRTUAL DEVICE

1. Click on Run ‘app’. The virtual device may take several minutes to initiate the first time it is run.
2. Once the device has loaded, unlock the screen.

4.3. PROGRAMMING AN APP

Now that the environment is prepared, we will write the code for the application.

4.3.1. ADDING TEXT

1. If activity_button.xml is not already open, click the Project dropdown window.
   a. Expand My App
   b. Expand App
   c. Expand SRC
   d. Expand Main
   e. Expand Res
   f. Expand layout activity

   ![Project Tree]

2. Click Run App again. The emulator will display your app. At this point, there should already be text displaying, “Hello world!”
3. Select the Text tab at the bottom of the window if it has not already been selected.

   ![Text Tab Selected]

4. Under the heading, TextView, you should see where “Hello world!” has been typed. Replace the text with the following, “Please tap the button”
5. Select the Design tab.
6. Click on the text box and drag it to the middle of the screen to center it.
7. In the Properties window, change the textSize property to 50sp.

```xml
<TextView
    android:text="Hello world!"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />
```
4.3.2. ADDING A BUTTON

1. In the Design tab, drag and drop a Button from the Widgets folder under the Palette toolbox.

2. Under the Properties window, change the text to “Tap Here!”

3. Change the textSize to 50sp.

4. If activity_button.xml is not already open, click the Project dropdown window.
   a. Expand MyApp
   b. Expand App
   c. Expand src
   d. Expand Main
   e. Expand Java
   f. Expand example.myapp
5. Import the following code.

6. Add the following code under **public class ButtonActivity extends Activity** {

7. In the **activity_button.xml**, navigate to the button properties.
8. For the property **onClick**, select **buttonOnIcon** from the dropdown menu.
4.3.3. TESTING YOUR APP

1. Click the Run ‘app’ button and open the window your emulator is running in.
2. After clicking your button, the app should now display “You just made your first button!”

5. WINFORM RENDERING

This tutorial will cover rendering 2D graphics in Windows Forms using Visual Studio Express. We will be covering rendering graphics to a bitmap, panning, and zooming.

Note: While prior programming experience is not required, it would be beneficial to have knowledge of Java and/or basic Object-oriented programming (OOP) concepts.

For this tutorial, you will need Visual Studio Express 2013 for Windows Desktop. This software is available for free download at http://www.visualstudio.com/downloads/download-visual-studio-vs. However, you may need to create a Microsoft Account.

Note: Installing Visual Studio usually takes a while. Feel free to work on something else and then come back to this tutorial.

Now that you have all of the prerequisites downloaded we can begin writing the application.

5.1. STARTING A NEW PROJECT

1. Upon entering Visual Studio, select New Project. If this is not immediately visible, you can find it by opening the File menu.
2. On the left, select Visual C# (pronounced see sharp). You can use other languages to create your application, but we will be using C# throughout this tutorial.

3. Choose Windows Forms Application. Give a name to your solution—for instance: RenderTutorial—and click OK.

**5.2. SETTING UP THE FORM**

You should now find yourself in Design view for the main form of your application. This view allows you to edit your form and add controls to it in a graphical interface.

**5.2.1. THE PROPERTIES WINDOW**
In the bottom-right corner of the window, you should find the Properties window. If you cannot find it, Select View > Properties Window or hold Alt+Enter. This window allows you to set properties for the currently selected control.

1. Your form is given a default (Name) of Form1 (found under Design). You can change this to something more descriptive in the Properties Window, like MainForm. You could also leave this at the default.

   Note: If you do change the form name, you might also want to rename the Form1.cs file in the Solutions Explorer in the top-right corner of your screen.

2. Under the Appearance category, you can set the Text of the window title. In this tutorial it is set to Rendering Tutorial. You can set this to your preference.

3. Scroll down to the Behavior category and set DoubleBuffered to True:
4. Next, set **FormBorderStyle** to **FixedSingle**. This will prevent the control from being resized.

    **Note**: Under the current rendering method, the rendered display will NOT scale with the PictureBox. This is why you need to prevent the display from being scaled.

5. Set **MaximizeBox** under **Window Style** to **False**. This will prevent the form from being maximized.

6. Now that the form is un-scalable, set a size for it. Under **Layout**, give your form a **Size**, in pixels, that is comfortably within your screen resolution. **640, 480** should work for most displays, but you could go larger.

5.2.2. **ADD A PICTUREBOX**

To add a control to the form:

1. Go to the left side of the screen and click on the **Toolbox** to open it. This contains a list of controls.
2. Select the **PictureBox** under **Common Controls** and close the Toolbox by clicking it again.
3. Click and drag on your form to draw a **PictureBox** control.
4. Set the (Name) of the PictureBox (e.g. – MainDisplay, MainPictureBox, etc.) or leave it at the default.
5. Under **Layout**, set the **Dock** type to **Fill**. This will cause the PictureBox to automatically scale to fit the form.
5.3. WRITING THE CODE

If you were to run your application at this point—you can run in Debug mode by clicking the Start button in the Standard Toolbar at the top of the screen—you would find that the application actually launches. However, it does not do much yet. This is because you have not added any code for rendering or given the user a means to interact with the application.

5.3.1. RENDERING TO A BITMAP

In this section, you will draw something for the PictureBox to display on the screen.

5.3.1.1. CREATING THE BITMAP

From this point forward the form will be called MainForm and the PictureBox will be called MainDisplay. This will help maintain consistency, but remember that your form and controls may have different names.
1. Double click the form inside the Design view. (Make sure you are clicking the form and not the PictureBox.) You should be moved to a new document with lines of text.

- This is the .cs file for your main form. It contains all the code that will be executed in this form. Right now, your form’s class declaration should be empty apart from a constructor (the method called to create the class) and a MainForm_Load event handler, which you just created.
- MainForm_Load is called when MainForm is created. You can use this method for one-time instantiations and setting up your program.

2. To set up a bitmap in memory, you will draw graphics to the bitmap and then draw the bitmap to the display. Add the following code inside your class above the constructor (public MainForm):
// Declare Bitmap
private Bitmap buffer;

// Create Graphics object
private Graphics GFX;

- You might see an error in the form of a red underline. In this case, click on Bitmap and select the first option. This lets the compiler know where to look to find the Bitmap object.

```csharp
public partial class MainForm : Form
{
    // Declare Bitmap
    private Bitmap buffer;
    // Create Graphics object
    private Graphics GFX;
    public MainForm()
    {
        // Initialize bitmap image
        // Set MainDisplay's image to the buffer
        MainDisplay.Image = MainBitmap;
        // Create graphics for buffer
        GFX = Graphics.FromImage(MainBitmap);
    }
}
```

- You can set the name of the Bitmap object to whatever you like. In this case, it is buffer.
- This also created a Graphics object which will enable you to draw to the bitmap later.

3. To initialize the bitmap, add the following code to MainForm_Load:

```csharp
// Initialize bitmap image

// Set MainDisplay's image to the buffer
MainDisplay.Image = MainBitmap;

// Create graphics for buffer
GFX = Graphics.FromImage(MainBitmap);
```

This creates the bitmap, sets it as the image for MainDisplay, and makes GFX the Graphics object for the buffer.

5.3.1.2. CREATING A RENDER LOOP

A render loop will allow you to implement motion and animation later. The original code for this game loop, as well as the details of how it works, can be found here:
1. Add the following code to the MainForm class, under MainForm_Load (do not try to put it in MainForm_Load):

```csharp
/**RENDER LOOP CODE**/

void HandleApplicationIdle (object sender, EventArgs e)
{
    while(IsApplicationIdle())
    {
        Render();
    }
}

void Render ()
{
}

bool IsApplicationIdle () { 
    Message result;
    return !PeekMessage(out result, IntPtr.Zero, 0, 0, 0);
}

[StructLayout(LayoutKind.Sequential)]
public struct NativeMessage
{
    public IntPtr Handle;
    public uint Message;
    public IntPtr WParameter;
    public IntPtr LParameter;
    public uint Time;
    public Point Location;
}

[DllImport("user32.dll")]
static extern bool PeekMessage(out Message message, IntPtr window, uint messageFilterMinimum, uint messageFilterMaximum, uint shouldRemoveMessage);
```

2. You will have to fix a few imports manually. Click on StructLayout, which should be red-underlined, move the mouse to the blue box under the beginning of the line, and select the first item in the dropdown:
3. The event handler `HandleApplicationIdle` should fire whenever the application is not busy doing anything else. This handler will call the method `Render`, where you will render your bitmap. However, right now `HandleApplicationIdle` has not been registered as an event handler. To do so, put the following code under `InitializeComponent` in the form constructor:

```csharp
// Handle application idle
Application.Idle += HandleApplicationIdle;
```

5.3.1.3. **DRAWING TO BITMAP**

Now that the render loop is set up, you need to actually draw something to the bitmap. You can do this through the `Graphics` object you created earlier (`GFX`). You can draw whatever you like at this point. In this tutorial, a circle is drawn.

1. Enter the following code in the `Render` method:

```csharp
// Fill background
GFX.Clear(Color.MidnightBlue);
```

This fills the background with the color Midnight Blue.

**Note:** Replace any of the colors in this tutorial with colors that you prefer.

2. Next, draw the circle using the code provided below:

```csharp
// Draw circle
Pen whitepen = new Pen(Color.White, 4F);
int diameter = Math.Min(MainDisplay.Width, MainDisplay.Height);
GFX.DrawEllipse(whitepen, 0, 0, diameter, diameter);
```

- The first line creates a new `Pen` object, which is what will be used to draw the ellipse. It sets the color to white and the width of the tip to 4.
- The second line stores the diameter of the circle. In this case, it set the diameter to be the smallest dimension of the `PictureBox` (that’s what `Math.Min` finds).
- The third line draws the circle. The `Graphics` object actually doesn’t have a method for drawing circles, so it is drawing an ellipse instead. The top-left corner of the ellipse is set to 0, 0, the top-left corner of the `PictureBox`.

3. Call the control’s `Invalidate` method at the end of `Render` to tell `MainDisplay` to redraw itself using the code below:

```csharp
// Redraw MainDisplay
MainDisplay.Invalidate();
```

5.3.2. **PANNING AND ZOOMING**
If you run the program now, you should have a working rendered display that looks something like this:

In this tutorial, you will create a zooming, panning display. You can use this code to create a top-down map.

5.3.2.1. ADDING EVENT HANDLERS

Event handlers are what allow the program to react to certain system events. You can use them to make the program react when the user moves the mouse, clicks, etc.

1. Add the following event handlers inside MainForm that will focus the PictureBox when the mouse moves into it and focus the form when it moves out:

```csharp
private void MainDisplay_MouseEnter(object sender, EventArgs e)
{
    if (!MainDisplay.Focused)
        MainDisplay.Focus();
}

private void MainDisplay_MouseLeave(object sender, EventArgs e)
```
2. Add global variable to the top of the class (where you declared the bitmap buffer):

```csharp
// Declare starting point for panning
private Point startingPoint = Point.Empty;

// Declare moving point for panning
private Point movingPoint = Point.Empty;

// Declare bool to store panning status
private bool panning = false;

// Declare image scale factor
private double scaleFactor = 1;
```

3. You will need to create the event handlers for panning. You will need three separate event handlers:

```csharp
private void MainDisplay_MouseDown(object sender, MouseEventArgs e)
{
    switch (e.Button)
    {
        case MouseButtons.Left:
            panning = true;
            startingPoint = new Point(e.Location.X - movingPoint.X,
                                      e.Location.Y - movingPoint.Y);
            break;
    }
}

private void MainDisplay_MouseUp(object sender, MouseEventArgs e)
{
    panning = false;
}

private void MainDisplay_MouseMove(object sender, MouseEventArgs e)
{
    if (panning)
    {
        movingPoint = new Point(e.Location.X - startingPoint.X,
                                e.Location.Y - startingPoint.Y);
    }
}
```

- The first event handler sets the `startingPoint` and sets `panning` to `true` when the mouse button is pressed.
- The second sets `panning` to `false` when the mouse is released.
- The third event handler moves `movingPoint`, or the origin, in relation to `startingPoint` while `panning` is `true` (the mouse is held down).
**Note:** In the MouseDown event handler, a switch is used to check which mouse button was pressed and a case is added for the left mouse button. You can add cases later to handle other mouse buttons.

4. Use the event handler for the mouse wheel:

```csharp
private void MainDisplay_MouseWheel(object sender, MouseEventArgs e)
{
    if (!panning)
    {
        double zoom = e.Delta > 0 ? .2 : -.2;
        if ((zoom < 0 && scaleFactor > 0.1) || (zoom > 0 && scaleFactor <= 10))
        {
            double mapX = (e.Location.X - movingPoint.X) / scaleFactor;
            double mapY = (e.Location.Y - movingPoint.Y) / scaleFactor;
            scaleFactor += zoom * 0.2;
            int X = Convert.ToInt32(-(mapX * scaleFactor) - e.Location.X);
            int Y = Convert.ToInt32(-(mapY * scaleFactor) - e.Location.Y);
            movingPoint = new Point(X, Y);
        }
    }
}
```

In the second if statement, scaleFactor has been restricted to being between 0.1 and 10, inclusive. You may want to adjust this if you want the user to be able to zoom in or out more.

**Note:** Zooming interferes with panning, which is why the if statement at the top of this code only executes the rest of the code if panning is false.

5. Modify your drawings to be positioned relative to movingPoint and scale according to scaleFactor. Below is the original code modified:

```csharp
// Draw circle
Pen whitepen = new Pen(Color.White, 4F);
int diameter = Convert.ToInt32(Math.Min(MainDisplay.Width,
                                        MainDisplay.Height) * scaleFactor);
GFX.DrawEllipse(whitepen, movingPoint.X, movingPoint.Y, diameter, diameter);
```

Note how movingPoint is now used instead of 0, 0 and all dimensions are multiplied by scaleFactor.

### 5.3.2.2. REGISTER EVENT HANDLERS

If you run the code now, you will notice that the application is still not interactive. This is because you have to register the event handlers for them to take effect.
1. Go back to the Design view – see 5.2 if you’re confused.

**Note:** If you have accidentally closed the Design view, you can reopen it by double-clicking MainForm.cs in Solution Explorer:

![Solution Explorer](Image)

2. Click on MainDisplay to make sure it is selected.
3. Click on the lightning bolt at the top of the Properties window to view the Events for the PictureBox class.
4. Scroll down to the **Mouse** category and select the appropriate event handler for each event.

5. You may have noticed that the MouseWheel event is not listed here. You will have to register this handler manually in the code. In the .cs file, enter the following code in the form constructor under the application idle event:

   ```csharp
   // Handle MouseWheel event for MainDisplay
   MainDisplay.MouseWheel += MainDisplay_MouseWheel;
   ```

5.4. **ALL CODE**

Here is what you should have in your MainForm.cs file once you are finished with this tutorial:

```csharp
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Runtime.InteropServices;
using System.Text;
```
using System.Threading.Tasks;
using System.Windows.Forms;

namespace RenderTutorial
{
    public partial class MainForm : Form
    {
        // Declare Bitmap
        private Bitmap buffer;

        // Declare Graphics object
        private Graphics GFX;

        // Declare starting point for panning
        private Point startingPoint = Point.Empty;

        // Declare moving point for panning
        private Point movingPoint = Point.Empty;

        // Declare bool to store panning status
        private bool panning = false;

        // Declare image scale factor
        private double scaleFactor = 1;

        public MainForm()
        {
            InitializeComponent();

            // Handle application idle
            Application.Idle += HandleApplicationIdle;

            // Handle MouseWheel event for MainDisplay
            MainDisplay.MouseWheel += MainDisplay_MouseWheel;
        }

        private void MainForm_Load(object sender, EventArgs e)
        {
            // Initialize bitmap image
            buffer = new Bitmap(MainDisplay.ClientRectangle.Width,
                                 MainDisplay.ClientRectangle.Height,
                                 System.Drawing.Imaging.PixelFormat.Format32bppRgb);

            // Set MainDisplay’s image to the buffer
            MainDisplay.Image = buffer;

            // Create graphics for buffer
            GFX = Graphics.FromImage(buffer);
        }

        private void MainForm_MouseEnter(object sender, EventArgs e)
        {
            if (!MainDisplay.Focused)
                MainDisplay.Focus();
        }

        private void MainForm_MouseLeave(object sender, EventArgs e)
        {
            if (MainDisplay.Focused)
                MainDisplay.Parent.Focus();
        }
    }
}
private void MainDisplay_MouseDown(object sender, MouseEventArgs e)
{
    switch (e.Button)
    {
        case MouseButtons.Left:
            panning = true;
            startingPoint = new Point(e.Location.X - movingPoint.X,
                                      e.Location.Y - movingPoint.Y);
            break;
    }
}

private void MainDisplay_MouseUp(object sender, MouseEventArgs e)
{
    panning = false;
}

private void MainDisplay_MouseMove(object sender, MouseEventArgs e)
{
    if (panning)
    {
        movingPoint = new Point(e.Location.X - startingPoint.X,
                                 e.Location.Y - startingPoint.Y);
    }
}

private void MainDisplay_MouseWheel(object sender, MouseEventArgs e)
{
    if (!panning)
    {
        double zoom = e.Delta > 0 ? .2 : -.2;
        if (((zoom < 0 && scaleFactor >= 0.1) || (zoom > 0 && scaleFactor <= 10))
            double mapX = (e.Location.X - movingPoint.X) / scaleFactor;
        double mapY = (e.Location.Y - movingPoint.Y) / scaleFactor;
        scaleFactor += zoom * 0.2;
        int X = Convert.ToInt32(((mapX * scaleFactor) - e.Location.X));
        int Y = Convert.ToInt32(((mapY * scaleFactor) - e.Location.Y));
        movingPoint = new Point(X, Y);
    }
}

/**RENDER LOOP CODE**/

void HandleApplicationIdle(object sender, EventArgs e)
{
    while (IsApplicationIdle())
    {
        Render();
    }
}

void Render()
{
    // Fill background
5.5. CONCLUSION

You are done! If you have followed the tutorial up to this point, you now have a fully zooming, panning rendered display. You can draw whatever you like; for more practical applications, you can use the DrawImage method in the Graphics object to create a map.
6. XBEE S2 SETUP

This tutorial documents how to setup two XBee Series 2s to communicate with one another.

Suggested materials for this tutorial include:

- XBee Series 2s (x2)
- XBee dongle
- Arduino Board (Uno)
- XBee Shield
- 9V Battery Adaptor for the Arduino (2.1mm power plug)

This tutorial includes the following topics:

- Software Setup
- Hardware Setup
- X-CTU

![Images of XBee Explorer Dongle, Arduino Uno, 9V Battery Adaptor, XBee Series 2 Module, XBee Shield]
6.1. SOFTWARE SETUP

1. Download and install the X-CTU Software and drivers from the following website:
   http://www.digi.com/support/productdetail?pid=3352&osvid=57&type=utilities

6.2. HARDWARE SETUP

1. Place the XBee Shield on the Arduino Uno.
2. Place one XBee S2 Module on the Explorer Dongle, and set one aside.
3. Plug in the XBee Dongle (with XBee) into the computer. The computer may need to automatically install driver software.

6.3. X-CTU

1. Open the X-CTU Software.
2. Click Add a Radio Module... at the top left:

3. Select the USB Serial Port COM port.

4. Ensure that the settings are as follows:
   a) Baud Rate: 9600
   b) Data Bits: 8
   c) Parity: None
   d) Stop Bits: 1
   e) Flow Control: None
5. If Action Required pops up, manually reset the XBee S2 Module by unpinning and re-pinning the module on the Dongle.

6. Click on the radio module now appearing in the upper left; X-CTU will read its information.
7. Update the firmware on the module by selecting the Update firmware button under Radio Configuration (see below).
   a) Select:
      i) Product Family: XB24-ZB
      ii) Function Set: ZigBee End Device API
      iii) Firmware Version: [Newest] 29A7
   b) Deselect Force Module to maintain its current configuration.
   c) Click Finish.
8. Type in a PAN ID of 1111 and click the write button on the right (the red pencil).
9. Record the PAN ID (under Networking) and the Serial Number High and Serial Number Low (under Addressing) on a notepad.
a) If the Serial Numbers do not show up, try clicking the green refresh button to the right.

10. Disconnect the XBee module by clicking the red x under Radio Modules in the upper left corner.

11. Pin the recently written XBee module onto the XBee shield on the Arduino Uno.
12. Pin the unused XBee module onto the Dongle.

13. Repeat the steps outlined in 1.3 for this XBee module, but set the firmware function to **ZigBee Coordinator API** (instead of ZigBee End Device API).

14. Click the red write buttons to write the previously recorded Serial Numbers to the Destination Address high and Destination Address Low (under Addressing) (see below).
15. Hook up the 9V power supply to the Arduino Uno.
16. At the top left, next to the radio module, click on the blue bubble link button to discover radio nodes and add selected devices. If the discovery fails, try closing the window and resetting the Arduino and XBee shield before trying again.

17. At the top right, under Tools, select Range Test.
18. Click on the radio module on the left and the remote radio device on the right, then click start range test.

19. The packets sent should be received.

20. The XBees are ready to be programmed.
7. EXPORTING .SKP FILES FROM SKETCHUP TO UNITY WITH TEXTURES

This tutorial documents how to maintain the texture of exported SketchUp models when they are imported into Unity.

Suggested materials for this tutorial are SketchUp and Unity.

This tutorial includes the following topics:

- Exporting SketchUp Models
- Importing Models into Unity

7.1. EXPORTING SKETCHUP MODELS

1. Download and install the latest version of SketchUp and Unity from the following websites:
   http://unity3d.com/
   http://www.sketchup.com/
2. Open SketchUp and open a file to export to Unity. Make sure that the model has a texture applied.
3. On the File menu click Export -> 3D Model
4. Select a location to save the file.
5. Type a name in the File Name field.
6. Select 3DS File (*.3ds) from the Export Type dropdown.

Click Export. A 3DS Export Results dialog displays in SketchUp. This means the export was a success, celebrate.

**7.2. IMPORTING MODELS INTO UNITY**

2. Locate the .3ds model saved previously. There should also be a .jpeg file for each texture included on the model.
3. Import the .3ds model and .jpeg(s). Each file will need to be imported separately.
4. Drop the model into the scene.
5. If the model is not textured, drag the textures onto the appropriate surface(s).
6. To add Physics:
   a. Select the model
   b. Right click
   c. Click Add Component
   d. Click Physics
e. Select the physics component you wish to add.