**Study Guide for Final Exam CS345 – Event-driven Programming**

<https://dev.azure.com/riblerClasses/cs345>

**World Coordinates**

Most of the graphics systems that we have dealt with use pixels or some other low-level graphics element as the basic unit that is used by default. These coordinate systems, usually referred to as “device coordinates” provide fine-grained access to the image. However, usually it is more convenient to express the things that we draw on the screen using the same units that are used in our application. For example, if we were drawing football fields it would be convenient to use yards as the basic unit of measure. if we were drawing a ruler, we might want to use millimeters or inches. In some examples we did in class we drew chessboards using the width of one square as the basic unit of measurement. Adopting one of these “world coordinate” systems frees us from having to constantly convert between the units that are natural to our application, and the units used by a particular device.

**Example:**

Suppose that we want to plot 2-dimensional data on a graph where x ranges from -∏ to ∏ and y ranges from -1 to +1. When we plot a point we want to use the x, y coordinate of the point, not the device coordinates.

**Step 1:**

Translate from device units to world units. This is a simple question of determining the number of pixels per world unit.

We can find the number of pixels that we can write to in our window using the ClientRectangle. The width of the ClientRectangle is ClientRectangle.Width and the height is ClientRectangle.Height. So suppose that I am drawing a football field that is 120 years from end to end. The number of pixels per world unit would be ClientRectangle.Width / 120.0f. That is, the width of the window in pixels divided by 120.0f yards. So if our windows was 600 pixels across, we would have 600/120 = 5 pixels per yard. We would make a similar calculation for the height. If the window was 400 pixels in height and a football field is 53 and a third yards in width, there would 400/53.333 pixels per yard, which is 7.5. In windows forms we express this relationship using a call to the Graphics class member function scaleTransform.

graphics.ScaleTransform( ClientRectangle.Width / footballFieldLengthInYards,

ClientREctangle.Height / footballFieldWIdthInYards);

After this call is completed, subsequent calls to the graphics object will be interpreted in terms of yards, rather than pixels. So for example, draw a line from point 1, 1, it will be drawn from pixels(5, 7.5).

Returning to our 2-dimensional plot, the x axis will be 2∏ units wide and the y-axis will be 2 units high. We can express this using the following call to graphics.ScaleTransform().

graphics.ScaleTransform((float) (ClientRectangleWidth / 2\*Math.PI),

(float) (ClientRectanlgeHeight / 2.0f))

Or more generally:

graphicsScaleTransform(ClientReactangle.Width/worldWidth,

ClientRectangle.Height/worldHeight);

If we want to flip the y-axis, so that the y-coordinate decreases rather than increases as we move down the page, we can use ScaleTransform with a negative value. A simple example would be this:

graphics.ScaleTransform(1, -1);

This would cause negative values to be plotted where their positive counterparts would have been plotted previously, and positive values to be plotted above the window, and therefore off-screen.

More often we would combine this with the earlier call to graphics.ScaleTransform like this:

graphicsScaleTransform(ClientReactangle.Width/worldWidth,

-ClientRectangle.Height/worldHeight); // note the minus sign before the ClientRectangle.Height.

**Step 2:**

By default, point (0,0) is always placed in the top-left corner of the display. X values increase as we move from left to right in the display, and y values increase as we move from top to bottom. We can move the point of origin using the TranslateTransform function. This function specifies values that should be added to each x and y coordinate.

A simple use of this function can be used to make room for a menu strip. If we want to move the origin to below a menu strip we can do something like:

graphics.TranslateTransform(0, menuHeight);

This will translate the origin (0,0) to device coordinate (0, menuHeight), effectively moving the origin to the leftmost point immediately below the menu.

In the football field example, we can move the origin to midfield using after we have called ScaleTransfrom with:

graphics.TranslateTransform(fieldLength/2, fieldWidth/2);

In our graphing example, after we flipped the y-axis, the y values in the window ranged from 0 to -2. To achieve the desired range, of 1 to -1, we must move the origin down the page by -1. To place it in the middle of the window, we also add ∏ to the x-coordinate.

The function Graphics.TransformPoints can be used to transform points between world and device coordinate systems.

**Windows Forms – Reference firstCSharp on Class Repository**

General Architecture

Events (mouse clicks, mouse movements, keydown, paint etc.) enter a message queue.

The main program calls Application.Run(mainForm) that processes the message queue.

The programmer associates event handlers (implemented as delegates) with individual events.

Main Form

1. Use View | Toolbox to begin drag and drop population of dialog box.
   1. Add menu strip
   2. Use properties dialog to name menu strip
   3. Add menu items (use & for Alt selection)
   4. Double click menu items to add event handlers

Dialog Boxes

1. Use stock dialog boxes
   1. ColorDialog
   2. OpenFileDialog
   3. PrintDialog
   4. FontDialog
2. Create a new dialog class derived from Form
   1. Add UI items to gather information from user
   2. Add buttons for (OK, Cancel as appropriate)
      1. Set DialogResult = DialogResult.<result>
      2. Close the dialog
3. Using dialog boxes
   1. Create an instance of the dialog
   2. Call the ShowDialog() member function
      1. ShowDialog returns a DialogResult
      2. Check the status of DialogResult and proceed

Paint

When the screen needs to be redrawn, the Paint event handlers are invoked. This can be forced using the Invalidate() call.

Typically, paint will use a world coordinate system to render a window display using the GDI+ graphics system available through the Graphics class.

DrawLine

DrawRectangle

FillRectangle

DrawString

Printing

* 1. Create instance of PrintDialog
  2. Show dialog to obtain printer settings
  3. Create instance of PrintDocument
  4. Copy print settings to PrintDocument
     1. document.PrintSettings = printDialog.PrintSettings;
  5. Add event handler for print page
     1. document.PrintPage += Document\_PrintPage
  6. Start the printing process
     1. document.Print();
  7. In Document\_PrintPage
     1. Extract print rectangle from the PrintPageEventArgs member MarginBounds
     2. Set world coordinates for printing
     3. Paint with PrintPageEventArgs member Graphics

Animation

1. Make sure that the form’s DoubleBuffered property is set to true
2. Create a Timer object to trigger at regular intervals (15-30/second)
3. Create a Tick event that updates the data used to generate the display in Paint
4. Add an Invalidate() call to the Tick event handler to force the Paint even

**WPF – Reference wpfChessExample**

**WPF**

1. Advantages
   1. More natural split between work of programmers and graphic designers
   2. Most things can be accomplished two ways
      1. XAML
      2. Code behind (C# or other .NET language)
   3. Enhanced data binding capability (including support for MVVM)
   4. 3d Capability
   5. A selection of layout managers that make it easier to place UI components
   6. Graphics are “Retained-mode” graphics vs “Immediate-mode” graphics used in Windows Forms
      1. Programmer not responsible for rendering the screen
         1. No Invalidate()
         2. The programmer places something on the screen once.
   7. Newer UI systems are following the trend of separate XML-based UI descriptions
      1. Android
      2. Xamarin
2. XAML
   1. XML-based UI Description Language
   2. XML namespaces
      1. Not really urls (just like namespaces in other languages)
      2. Abbreviations can be created:
         1. xmlns:x=<http://schemas.microsoft.com/winfx/2006/xaml>
         2. x is an abbreviation for the namespace
   3. “Attached Properties”
      1. Allows a child element to set the value of parent property
      2. Example: <Button Grid.Row=0> allows the child Button to set its Row property in the Grid parent.
   4. Layout Managers
      1. Canvas
         1. Simplest of layouts
         2. Specify the position of the element in *device independent pixels.* (px) (1/96 inch), *points* (pt) (1/72 inch), *inches* (in) or *centimeters* (cm).
         3. Positions can be described relative to top, left, bottom, or right using attached properties. For example:

<Rectangle Fill="LightCoral" Canvas.Left="50" Canvas.Top="50" Width="50" Height="50" />

* + 1. DockPanel
       1. Can stack UI elements relative to top, bottom, left, or right.
       2. **Example:** [**https://www.wpftutorial.net/DockPanel.html**](https://www.wpftutorial.net/DockPanel.html)
    2. StackPanel
       1. Orientation can be vertical or horizontal
       2. Simple method to develop UI quickly
       3. Example: <https://www.wpftutorial.net/StackPanel.html>
    3. Grid
       1. Probably the most versatile/powerful layout manager
       2. The Grid can contain <Grid.RowDefinition> and <Grid.ColumnDefinition> tags to describe the number and size of rows and columns in the grid. For example:

<Grid.RowDefinitions>

<!-- "auto means consume just as much as necessary -->

<!-- "a button that is 100 pixels high will consume 100 pixels-->

<!-- "The height of the row is the height of the tallest contained element -->

<RowDefinition Height="auto"/>

<!-- \* means use all the rest of the available space -->

<!-- multipliers can be used with \* specify a proportion of the remaining space

<!-- The second row height will be twice that of the 3rd row, 4 times that of the

<<!--first row. -->

<RowDefinition Height=".5\*"/>

<RowDefinition Height="2\*"/>

<RowDefinition Height="\*"/>

</Grid.RowDefinitions>

<!—Specify 5 columns -->

<Grid.ColumnDefinitions>

<ColumnDefinition/>

<ColumnDefinition/>

<ColumnDefinition/>

<ColumnDefinition/>

<ColumnDefinition/>

</Grid.ColumnDefinitions>

-------------------------------------------

// Grid row and column definitions can also be done in C#

for(int side=0;side<SQUARES\_PER\_SIDE;++side)

{

grid.RowDefinitions.Add(new RowDefinition());

grid.ColumnDefinitions.Add(new ColumnDefinition());

}

// Add a UI element to a grid in C# like this:

Button newButton = new Button();

newButton.BackgroundColor = squareColor;

theGrid.Children.Add(newButton, row, column);

------------------------------------------

<!—Subsequently UI elements can be specified to occupy particular rows and columns -->

<ListBox Grid.Column="0" Grid.Row="1" Grid.ColumnSpan="2" Grid.RowSpan="3">

* + 1. Layout managers can be nested in any combinations.
  1. Specify Margins and Padding
     1. Margins – controls space around outside edges
     2. Padding – controls space around inside edges
     3. Margins and Padding are specified using 1, 2, or 4 values
     4. One value specifies all four sides
        1. Padding=”10”
     5. Two values
        1. First value specifies left and right
        2. Second value specifies top and bottom
        3. Padding=”10,20”
     6. Four values
        1. Specify Left, Top, Right, Bottom (clockwise from left side)
        2. Padding=”10,15,20,25”
  2. XML to C# References

<Button x:Name="firstButton" Margin="20,0,40,0" DockPanel.Dock="Top" Padding="22,0,0,0" Content="Top" Click="FirstButton\_Click" Width="100" Height="25"></Button>

The <Button x:Name=”firstButton” will cause a button to be declared as a window data member named “firstButton.” We can reference this button using this name in the window class. The button is allocated in the InitializeComponent() function.

We also specified the event handler that is called when the button is clicked (“FirstButton\_Click”). Code for this event handler will be added as a private member function of the window.

* 1. XML can specify transforms

<Button Margin="0,10" Grid.Row="2" Grid.Column="3"

MaxHeight="40" MaxWidth="80">

LeftButton

<Button.RenderTransform>

<TransformGroup>

<ScaleTransform ScaleX=".8" ScaleY=".8"></ScaleTransform>

<RotateTransform Angle="12"></RotateTransform>

</TransformGroup>

</Button.RenderTransform>

</Button>

1. Animations with WPF C# - Modify a property value over time

private void BtnSpinner\_Click(object sender, RoutedEventArgs e)

{

DoubleAnimation buttonAnim = new DoubleAnimation();

buttonAnim.From = 0;

buttonAnim.To = 360;

TimeSpan timespan = new TimeSpan(0, 0, 0, 10, 0);

buttonAnim.Duration = new Duration(timespan);

RotateTransform rt = new RotateTransform();

// Specify the point to rotate around relative

// to the top left (point (0,0)) of the element

// that is rotating.

rt.CenterX = btnSpinner.Width / 2;

rt.CenterY = btnSpinner.Height / 2;

//rt.CenterX = 0; // -50;

//rt.CenterY = 0;

// Associate rotate transfrom with the button.

btnSpinner.RenderTransform = rt;

// Start the animation

// The value of RotateTransform.AngleProperty will change over time

rt.BeginAnimation(RotateTransform.AngleProperty, buttonAnim);

}

1. Animation with WPF XML - Storyboards

<Grid>

<Button Width="50" Height="100" Background="#D1313B">

<Image Source="./Resources/0.png"></Image>

<Button.Triggers>

<EventTrigger RoutedEvent="Button.Click" >

<BeginStoryboard>

<Storyboard TargetProperty="Width" >

<DoubleAnimation From="50" To="100"

Duration="0:0:2" AutoReverse="True">

</DoubleAnimation>

</Storyboard>

</BeginStoryboard>

<BeginStoryboard>

<Storyboard TargetProperty="Height">

<DoubleAnimation From="100" To="50"

Duration="0:0:5" AutoReverse="True"/>

</Storyboard>

</BeginStoryboard>

</EventTrigger>

</Button.Triggers>

</Button>

</Grid>

**Xamarin**

1. Purpose: Cross-platform development for mobile/Windows applications
   1. Android
   2. iOS
   3. Windows Phone ( officially defunct as of Dec 10, 2019)
   4. Windows PC
2. Leverages a subset of WPF XAML
3. Uses GestureRecognizers for platform-independent event handling.
4. Example:

private TapGestureRecognizer recognizer = new TapGestureRecognizer();

public MainPage()

{

InitializeComponent()

recognizer.Tapped += Recognizer\_Tapped;

}

private void Recognizer\_Tapped(object sender, EventArgs e)

{

Image image = (Image)sender;

int row = (int)image.GetValue(Grid.RowProperty);

int column = (int)image.GetValue(Grid.ColumnProperty);

// And so on

}

1. Graphics with SkiaSharp – reference SkiaSharpExample
   1. Similar to windows forms
      1. PaintSuface event handler
      2. Canvas similar to Graphics in WinForms
   2. Animation via async calls

async Task AnimationLoop()

{

float x = 0;

float y = 0;

// The delta values say how much the location changes at

// each iteration of the animation loop

float xDelta = 5;

float yDelta = 5;

double refreshRate = 1.0 / 30.0;

Stopwatch stopwatch = new Stopwatch();

stopwatch.Start();

while(stopwatch.Elapsed.TotalSeconds < 2.0)

{

x = x + xDelta;

y = y + yDelta;

currentX = (int)x;

currentY = (int)y;

// Force a redraw

boardCanvas.InvalidateSurface();

await Task.Delay(TimeSpan.FromSeconds(refreshRate));

}

}

**Model view controller (MVC)**

1. **Model** – the underlying data for the application
   1. Examples
      1. Position for chess game
      2. Inventory for a webstore
   2. The basic data structures necessary for representing the state of the system.
   3. The model should not be dependent, or necessarily aware of the views
2. **Views** - The view renders a graphical image based on information that it extracts from the model
   1. There may be any number of views simultaneously using the same model
   2. The view should be concerned exclusively with rendering the user interface.
      1. Generally, it should not directly manipulate the model (that is left to the controller)
      2. It can generate events that are handled by the controller
3. **Controller** 
   1. Responsibilities
      1. Handle requests from the user that affect the model
      2. Inform the views of changes that require them to update

**Model View View-Model (MVVM)**

Reference wpfNotifications and Chapter 28 in Troelsen

1. Model – similar to Model in MVC except
   1. Typically instrumented provide change notifications to views via the observable pattern.
      1. Often implemented with INotifyPropertyChanged and ObservableCollection
2. View – similar to View in MVC except
   1. Bindings typically associated UI components with elements of the model
   2. Subscriptions to model event handlers provide notifications of change
3. View Model – similar to controller except
   1. Provides all data need by the view (probably through binding mechanism)
4. Example of basic Data Binding

<Grid Grid.Row="1" Name="DetailsGrid"

DataContext="{Binding ElementName=cboCars, Path=SelectedItem}">

<Grid.ColumnDefinitions>

<ColumnDefinition Width="Auto" SharedSizeGroup="CarLabels"></ColumnDefinition>

<ColumnDefinition Width="\*"></ColumnDefinition>

</Grid.ColumnDefinitions>

<Grid.RowDefinitions>

<RowDefinition Height="Auto"></RowDefinition>

<RowDefinition Height="Auto"></RowDefinition>

<RowDefinition Height="Auto"></RowDefinition>

<RowDefinition Height="Auto"></RowDefinition>

<RowDefinition Height="Auto"></RowDefinition>

<RowDefinition Height="Auto"></RowDefinition>

</Grid.RowDefinitions>

<Label Grid.Column="0" Grid.Row="0" Content="Make"></Label>

<TextBox Grid.Column="1" Grid.Row="0"

Text="{Binding Path=Make, ValidatesOnDataErrors=True}"></TextBox>

<Label Grid.Column="0" Grid.Row="1" Content="Color"></Label>

<TextBox Grid.Column="1" Grid.Row="1"

1. Example of INotifyPropertyChanged

public partial class InventoryAuto : INotifyPropertyChanged

{

public bool isChanged { get; set; }

public int CarId { get; set; }

public string Make { get; set; }

public string Color { get; set; }

public string PetName { get; set; }

public event PropertyChangedEventHandler PropertyChanged;

}

public string Color

{

set {

If (value != color\_)

{

OnPropertyChanged(nameof(color));

}

protected virtual void OnPropertyChanged(string propertyName)

{

PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName);

}