Eyes of the Dragon Tutorials 4.0

Part 7

What Was That?

I'm writing these tutorials for the MonoGame 3.8.1 framework using Visual Studio 2022. The tutorials will make more sense if they are read in order. You can find the list of tutorials on the Eyes of the Dragon 4.0 page of my web blog. I will be making the project available on GitHub here. It will be included on the page that links to the tutorials.

This is part seven of a series of tutorials I plan to write on creating a role-playing game with MonoGame. I've worked on similar tutorials using XNA and the past version of MonoGame. In the process of writing more tutorials, I discovered better ways of doing some things and had to go back to fix things. I'm hoping in this series to avoid making those same mistakes. Also, I am going to make the game cross-platform. In my previous tutorials, they were focused on Windows only. I want to open things up for macOS, Linux, Android, iOS, and any platform MonoGame supports.

In this tutorial, I am going to cover getting input from the player. Part of it won't work on iOS and Android, and that would be the text box. I will cover text input on those platforms in another tutorial. I will also add a list box and a form. So, let's get started.

Speaking Plain Text

To get started, right-click the Controls folder, select Add and then Class. Name this new class TextBox. Here is the code for the TextBox class.

```
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework;
using System;
using System.Collections.Generic;
using System.Text;

namespace SharedProject.Controls
{
    public class TextBox : Control
    {
        private readonly Texture2D _background;
        private readonly Texture2D _caret;
        private double timer;
        private Color _tint;
        private readonly List<string> validChars = new();

        public TextBox(Texture2D background, Texture2D caret)
            : base()
        {
            Text = "";
            _background = background;
            _caret = caret;
            _tint = Color.Black;
```

```
            foreach (char c in "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRST-
WXYZ0123456789 -_".ToCharArray())
            {
                validChars.Add(c.ToString());
            }
        }

        public override void Draw(SpriteBatch spriteBatch)
        {
            Vector2 dimensions = ControlManager.SpriteFont.MeasureString(Text);
            dimensions.Y = 0;
            spriteBatch.Draw(_background, Position, Color.White);
            spriteBatch.DrawString(
                ControlManager.SpriteFont,
                Text,
                Helper.NearestInt(Position + Vector2.One * 5),
                Color.Black,
                0,
                Vector2.Zero,
                1f,
                SpriteEffects.None,
                1f);
            spriteBatch.Draw(
                _caret,
                Position + dimensions + Vector2.One * 5,
                _tint);
        }

        public override void HandleInput()
        {
            if (!HasFocus)
            {
                return;
            }

            List<Keys> keys = Xin.KeysPressed();

            foreach (Keys key in keys)
            {
                string value = Enum.GetName(typeof(Keys), key);

                if (value == "Back" && Text.Length > 0)
                {
                    Text = Text.Substring(0, Text.Length - 1);
                    return;
                }
                else if (value == "Back")
                {
                    Text = "";
                    return;
                }

                if (value.Length == 2 && value.Substring(0, 1) == "D")
                {
                    value = value.Substring(1);
                }

                if (!Xin.IsKeyDown(Keys.LeftShift) && !Xin.IsKeyDown(Keys.RightShift)
&& !Xin.KeyboardState.CapsLock)
```

```
                {
                    value = value.ToLower();
                }

                if (validChars.Contains(value))
                {
                    if (ControlManager.SpriteFont.MeasureString(Text + value).X < Size.X)
                        Text += value;
                }
            }
        }

        public override void Update(GameTime gameTime)
        {
            timer += 3 * gameTime.ElapsedGameTime.TotalSeconds;
            double sine = Math.Sin(timer);

            _tint = Color.Black * (int)Math.Round(Math.Abs(sine));
        }
    }
}
```

Like all of our controls, this class inherits from the base control class Control. That means it can be added to a control manager and not have to be worried about calling its methods. I added a few new fields. The first is _background, which is the background. The other fields are _caret which, as the name implies, is the caret for the text box. The timer field controls when the caret blinks. The last, validChars, holds the characters that can be entered into the text box. The constructor takes two parameters. The text box's background and the caret. It sets the text to the empty string to prevent null reference exceptions. It sets the _background and _caret fields to the value passed in. Next, it sets the tint to black.

What I do next is loop over all of the characters the text box will accept. They are the lower case letters, upper case letters, digits, space, dash and underscore. I then add each of these characters to the list of valid characters.

In the Draw method, I get the dimensions of the text in the text box, so I know where to draw the caret. I set the Y coordinate to zero, so it will be set at the height of the text box. Next, I draw the background of the text box. When drawing the text, I use the NearestInt method of the Helper class. I pad it five pixels right and down. I just noticed I use the fixed colour black instead of the font colour. I also pad the caret when rendering it.

You could have a whole mess of if statements checking for individual key presses in the HandleInput method. It's a little easier if you check all keys pressed. First, if the control does not have focus, I exit the method. This makes me think. It would be good if the control with the focus was highlighted in some way. I will add that to my to-do list.

The first step is to get the pressed keys in this frame. Then, iterate over the collection of keys. I then used the GetName method of the Enum class. If the key is the backspace key, and there is text in the text box. I shrink the text by one character and exit the method. If it is pressed and there is one character or less, I set the text to the empty string and exit the method.

Next, I handle digits. Digits are the letter D followed by the digit. So, I check to see if the length is done, and the first letter is a D. If it is, the value is set to the last digit.

Next, I check to see if the right or left shift keys are down and that the caps lock is off. If all that is true, I set the value to the ToLower value. Now, I check to see if the key pressed is in the list of valid keys. If it is, and the length of the text box is less than the size of the text box, I add the key to the _text field.

That just leaves the Update method. What I do is update the timer by three times the amount of time that has passed since the last frame. It's a decent blink rate for the caret. I then calculate the sine of it. I do that because sine waves are cyclical and range between one and minus one. I then multiply the colour black by the round of the absolute value of sine. This has the colour blinking between transparent and opaque.

To Do Lists

While we are talking about controls, there are a few others that are important when it comes to the editors. They are a list box and a spin box or right-left selector. I will start with the list box. Right-click the Controls folder in the SharedProject, select Add and then Class. Name this new class ListBox. Here is the code for that class.

```
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework;
using System;
using System.Collections.Generic;
using System.Text;

namespace SharedProject.Controls
{
    public class ListBox : Control
    {
        #region Event Region

        public event EventHandler SelectionChanged;
        public event EventHandler Enter;
        public event EventHandler Leave;

        #endregion

        #region Field Region

        private readonly List<string> _items = new();

        private int _startItem;
        private int _lineCount;
```

```
private readonly Texture2D _image;
private readonly Texture2D _cursor;

private Color _selectedColor = Color.Red;
private int _selectedItem;
private readonly Button _upButton, _downButton;
private double timer;

private bool _mouseOver;

#endregion

#region Property Region

public Color SelectedColor
{
    get { return _selectedColor; }
    set { _selectedColor = value; }
}

public int SelectedIndex
{
    get { return _selectedItem; }
    set { _selectedItem = (int)MathHelper.Clamp(value, 0f, _items.Count); }
}

public string SelectedItem
{
    get { return Items[_selectedItem]; }
}

public List<string> Items
{
    get { return _items; }
}

public bool HasFocus
{
    get { return hasFocus; }
    set
    {
        hasFocus = value;

        if (hasFocus)
            OnEnter(null);
        else
            OnLeave(null);
    }
}

public Rectangle Bounds
{
    get { return new(Helper.V2P(Position), Helper.V2P(Size)); }
}
#endregion

#region Constructor Region
```

```
        public ListBox(Texture2D background, Texture2D downButton, Texture2D upButton,
Texture2D cursor)
            : base()
        {
            HasFocus = false;
            TabStop = true;

            _upButton = new(upButton, ButtonRole.Menu) { Text = "" };
            _downButton = new(downButton, ButtonRole.Menu) { Text = "" };

            _upButton.Click += UpButton_Click;
            _downButton.Click += DownButton_Click;

            this._image = background;
            this.Size = new Vector2(_image.Width, _image.Height);
            this._cursor = cursor;

            _startItem = 0;
            Color = Color.Black;
        }

        private void DownButton_Click(object sender, EventArgs e)
        {
            if (_selectedItem != _items.Count - 1 && timer > 0.1)
            {
                timer = 0;
                _selectedItem++;
                OnSelectionChanged(null);
            }
        }

        private void UpButton_Click(object sender, EventArgs e)
        {
            if (_selectedItem > 0 && timer > 0.1)
            {
                timer = 0;
                _selectedItem--;
                OnSelectionChanged(null);
            }
        }

        #endregion

        #region Abstract Method Region

        public override void Update(GameTime gameTime)
        {
            timer += gameTime.ElapsedGameTime.TotalSeconds;

            _upButton.Update(gameTime);
            _downButton.Update(gameTime);
        }

        public override void Draw(SpriteBatch spriteBatch)
        {
            _lineCount = (int)(Size.Y / SpriteFont.LineSpacing);
            Rectangle d = new((int)Position.X, (int)Position.Y, (int)Size.X,
(int)Size.Y);
            spriteBatch.Draw(_image, d, Color.White);
            Point position = Xin.MouseAsPoint;
```

```
            Rectangle destination = new(0, 0, 100, (int)SpriteFont.LineSpacing);
            _mouseOver = false;

            for (int i = 0; i < _lineCount; i++)
            {
                if (_startItem + i >= _items.Count)
                {
                    break;
                }

                destination.X = (int)Position.X;
                destination.Y = (int)(Position.Y + i * SpriteFont.LineSpacing);

                if ((destination.Contains(position) &&
                    Xin.MouseState.LeftButton == ButtonState.Pressed) ||
                    (destination.Contains(Xin.TouchLocation) &&
                    Xin.TouchPressed()))
                {
                    _mouseOver = true;
                    _selectedItem = _startItem + i;
                    OnSelectionChanged(null);
                }

                float length = 0;
                int j = 0;
                string text = "";

                while (length <= Size.X - _upButton.Width && j < _items[i].Length)
                {
                    j++;
                    length = SpriteFont.MeasureString(_items[i].Substring(0, j)).X;
                    text = _items[i].Substring(0, j);
                }

                if (_startItem + i == _selectedItem)
                {
                    spriteBatch.DrawString(
                        SpriteFont,
                        text,
                        new Vector2(Position.X + 3, Position.Y + i * SpriteFont.LineSpac-
ing + 2),
                        SelectedColor);
                }
                else
                {
                    spriteBatch.DrawString(
                        SpriteFont,
                        text,
                        new Vector2(Position.X + 3, Position.Y + i * SpriteFont.LineSpac-
ing + 2),
                        Color);
                }
            }

            _upButton.Position = new(Position.X + Size.X - _upButton.Width, Position.Y);
            _downButton.Position = new(Position.X + Size.X - _downButton.Width, Posi-
tion.Y + Size.Y - _downButton.Height);

            _upButton.Draw(spriteBatch);
            _downButton.Draw(spriteBatch);
```

```
        }

        public override void HandleInput()
        {
            if (!HasFocus)
            {
                return;
            }

            if (Xin.WasKeyReleased(Keys.Down))
            {
                if (_selectedItem < _items.Count - 1)
                {
                    _selectedItem++;

                    if (_selectedItem >= _startItem + _lineCount)
                    {
                        _startItem = _selectedItem - _lineCount + 1;
                    }

                    OnSelectionChanged(null);
                }
            }
            else if (Xin.WasKeyReleased(Keys.Up))
            {
                if (_selectedItem > 0)
                {
                    _selectedItem--;

                    if (_selectedItem < _startItem)
                    {
                        _startItem = _selectedItem;
                    }

                    OnSelectionChanged(null);
                }
            }
            if (Xin.WasMouseReleased(MouseButtons.Left) && _mouseOver)
            {
                HasFocus = true;
                OnSelectionChanged(null);
            }
            if (Xin.WasKeyReleased(Keys.Enter))
            {
                HasFocus = false;
                OnSelected(null);
            }

            if (Xin.WasKeyReleased(Keys.Escape))
            {
                HasFocus = false;
                OnLeave(null);
            }
        }

        #endregion

        #region Method Region

        protected virtual void OnSelectionChanged(EventArgs e)
```

```
        {
            SelectionChanged?.Invoke(this, e);
        }

        protected virtual void OnEnter(EventArgs e)
        {
            Enter?.Invoke(this, e);
        }

        protected virtual void OnLeave(EventArgs e)
        {
            Leave?.Invoke(this, e);
        }

        #endregion
    }
}
```

This code introduces an issue. That is the addition of a method in the Helper class, V2P, that converts a Vector2 to a Point. I will get to that shortly. First, there is a lot going on in this control. It is a composite control. That is, it has a list that holds the items and two buttons to move the selection up and down.

Let's dig in. There are three events for this control: SelectionChanges, Enter and Leave. As you've probably gathered, the first is when a new item in the list is selected. The second happens when the control is selected. The last is when the control is deselected.

There is a readonly List<string> which is the text of the items. You could use a List<object> and cast the items to a string when rendering them. For our purposes, a List<string> is good enough. There are two fields: _startItem and _lineCount which are used in rendering the list items. More on them later. There is a field, _selectedColor, that controls how the selected item is drawn. There is a field, _selectedItem, that holds which item in the list box is currently selected. The _upButton and _downButton fields are buttons that move the selected item up and down. That last field, timer, counts the amount of time that has passed for the control. It is used because I found that MonoGame can sometimes stutter when it comes to checking if an item was down in one frame and up in the previous frame. So, I use a timer to measure the amount of time that has passed since the event has occurred.

There are properties to expose the _selectedColor, _selectedItem, and value of the selected item. There is also an override of the HasFocus property. The interesting properties are SelectedIndex and HasFocus. The set part of SelectedIndex uses the Clamp method of MathHelper class to ensure that the value is in the range of the bounds of the array. The HasFocus property sets the _hasFocus property and then calls the OnEnter method if the control has received focus and OnLeave if the control has lost focus.

The constructor takes four parameters, the background of the list box, the texture for the down button, the texture for the up button and the texture for the selected item. The _hasFocus field is set to false and the _tabStop property is set to true. I then create the buttons passing in the texture for the button and setting the role to Menu. I also set the Text property to the empty string. Otherwise, you will get a null value exception. I then wire the Click event for the buttons. I set the _image field to the background passed in and the initial size to be the size of the image. The _cursor field is set as the parameter passed in. Finally, I set the _startItem field to zero and the colour to draw the text to black.

In the event handler for the down button, I check to see if the _selectedItem property is not set to the last item in the list of items. Also, I check to see that more than a tenth of a second has passed. This prevents the event

from being fired multiple times. If these conditions are true, I set the timer to zero, increment the selected item and call the OnSelectChanged method, which I will discuss shortly. The event hander for the up butting works in much the same way. The difference is that it checks to see if the _selectedItem is zero. If that is true and the timer is greater than a tenth of a second, the selected item is decremented. It then calls the OnSelectionChanged event handler.

In the Update method, I increment the timer field with the amount of time that has passed since the last frame of the game. Remember that in total, one second corresponds to the value one. So, one-tenth of a second is 0.1, and one full second is represented by one. It then calls the Update method of the buttons.

In the Draw method, I set the _lineCount field to the number of lines that can fit in the size of the list box. That is calculated by taking the height of the control and dividing that by the LineSpacing property of the font and casing that to an integer. I then create a rectangle that will be where the control will be drawn. It is calculated by taking the Position property of the control and the Size property of the control. It then draws the background image. Next, I used a new property, MouseAsPoint of the Xin class, that returns the coordinates of the mouse as a point. I will add this property, and another, once I have finished with the list box. I then create a destination rectangle that is positioned at (0, 0) and is 100 pixels wide and the line spacing property of the font high. Next, it sets the _mouseOver property of the list box to false.

After that, there is a for loop that loops for the number of line items there are. I then check to see if the item at the top of the list box plus i is greater than or equal to the number of items. If that is true, I break out of the loop. The X coordinate of the destination is set to the X coordinate of the position. The Y coordinate of the destination is set to the Y coordinate of the position plus i times the line spacing property of the font. If the destination contains the position of the mouse and the left mouse button is down or there is a touch point in that destination, _mouseOver is set to true, and the _selectedItem is set to the _startItem plus the value of i. The OnSelectionChanged method is called as well.

Next, there are some local variables. The first, length, measures the length of a substring of the current item being drawn. It is used to prevent the line from overflowing the list box. The j variable is the number of characters that are currently being checked. Finally, the text variable is the substring of the current item. There is a while loop that will loop for the number of characters that can safely be drawn. That is calculated by comparing the length of the string to the size of the list box minus the width of the buttons. The first step is to increment the number of characters to be drawn. After that, I use the MeasureString method passing in a substring j characters long. I also set the text variable to that substring.

About the _startItem field. It is set to the first item drawn in the list box. While all the items in the list box can safely be drawn and not overflow the list box's height, it will remain zero. Once there are more items in the list box than items in the list, it will change. When you scroll down, it is incremented. When you scroll up, it is decremented.

Next, there is an if statement that checks to see if the _startItem plus i is the selected item. If it is, that item is drawn using the SelectedColor property. Otherwise, it is drawn in the colour of the control. I added a smidge of padding to render the item. The X coordinate of the up button is set to the X coordinate of the list box plus the width of the list box minus the width of the button. The Y coordinate of the button is the Y coordinate of the list box. The X coordinate of the down button is the same as the X coordinate of the up button. The Y coordinate of the down button is the Y coordinate of the list box plus the height of the list box minus the height of the button. After setting the position of the buttons, I call their Draw methods.

The HandleInput method checks to see if the control has focus. If it does not, the method is exited. In addition

to using the buttons, the list box can be scrolled using the Up and Down keys. So, there is a check to see if the Down key is currently down. If it is, and the selected item is not the last item in the list, I increment the selected item field. Also, there is a check to see I the selected item is greater than or equal to the start item field plus the line count. If it is, I scroll down one item. I then call the OnSelectionChanged method. I do something similar if the up key has been released. If the selected item is greater than zero, I decrement the selected item. If the selected item is less than the start item, I decrement that as well. I then call the OnSelectionChanged method.

After drawing everything, there are three if statements. The first check is to see if the left mouse button was released and if the mouse is over the list box. If they are both true, the list box gains focus, and the OnSelectChanged method is called. If the Enter key is released, the OnSelected method is called. Finally, if the escape key has been released, the list box loses focus, and the OnLeave method is called.

Finally, there are three methods: OnSelectionChanged, OnEnter and OnLeave. The first raises the SelectionChanged event if it is subscribed to. The OnEnter method fires the Enter event if it is subscribed to. Finally, OnLeave fires the Leave event if it is subscribed to.

Circling back to the Helper class. We need a method that will convert a Vector2 to a Point. All I did was add a new method to the Helper class. It casts the X and Y properties of the Vector2 to integers and returns that as a Point.

```
        public static Point V2P(Vector2 vector2)
        {
            return new((int)vector2.X, (int)vector2.Y);
        }
```

Wow, that was a lot of code. I think giving it to you all at once was better than feeding it to you piece by piece. It's hard to say. Reach out to me on Discord or leave a comment on the blog about which approach you find is better.

A Picture Is Worth a Thousand Bytes

Before I get to the form, I need to add a PictureBox. It is used as the background for the form. Right-click the Controls folder in the SharedProject, select Add and then Class. Name this new class PictureBox. Here is the code for the class, sorry there is a lot of it as well.

```
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework;
using System;
using System.Collections.Generic;
using System.Text;

namespace SharedProject.Controls
{
    public enum FillMethod { Clip, Fill, Original, Center }

    public class PictureBox : Control
    {
        #region Field Region

        private Texture2D _image;
        private Rectangle _sourceRect;
        private Rectangle _destRect;
        private FillMethod _fillMethod;
```

```
        private int _width;
        private int _height;

        #endregion

        #region Property Region

        public Texture2D Image
        {
            get { return _image; }
            set { _image = value; }
        }

        public Rectangle SourceRectangle
        {
            get { return _sourceRect; }
            set { _sourceRect = value; }
        }

        public Rectangle DestinationRectangle
        {
            get { return _destRect; }
            set { _destRect = value; }
        }

        public FillMethod FillMethod
        {
            get { return _fillMethod; }
            set { _fillMethod = value; }
        }

        public int Width
        {
            get { return _width; }
            set { _width = value; }
        }

        public int Height
        {
            get { return _height; }
            set { _height = value; }
        }

        #endregion

        #region Constructors

        public PictureBox(Texture2D image, Rectangle destination)
        {
            Image = image;

            DestinationRectangle = destination;

            Width = destination.Width;
            Height = destination.Height;

            if (image != null)
                SourceRectangle = new Rectangle(0, 0, image.Width, image.Height);
            else
                SourceRectangle = new Rectangle(0, 0, 0, 0);
```

```
        Color = Color.White;

        _fillMethod = FillMethod.Original;

        if (SourceRectangle.Width > DestinationRectangle.Width)
        {
            _sourceRect.Width = DestinationRectangle.Width;
        }

        if (SourceRectangle.Height > DestinationRectangle.Height)
        {
            _sourceRect.Height = DestinationRectangle.Height;
        }
    }

    public PictureBox(Texture2D image, Rectangle destination, Rectangle source)
        : this(image, destination)
    {
        SourceRectangle = source;
        Color = Color.White;

        _fillMethod = FillMethod.Original;

        if (SourceRectangle.Width > DestinationRectangle.Width)
        {
            _sourceRect.Width = DestinationRectangle.Width;
        }

        if (SourceRectangle.Height > DestinationRectangle.Height)
        {
            _sourceRect.Height = DestinationRectangle.Height;
        }
    }

    #endregion

    #region Abstract Method Region

    public override void Update(GameTime gameTime)
    {
    }

    public override void Draw(SpriteBatch spriteBatch)
    {
        if (_image != null)
        {
            switch (_fillMethod)
            {
                case FillMethod.Original:
                    _fillMethod = FillMethod.Original;

                    if (SourceRectangle.Width > DestinationRectangle.Width)
                    {
                        _sourceRect.Width = DestinationRectangle.Width;
                    }

                    if (SourceRectangle.Height > DestinationRectangle.Height)
                    {
                        _sourceRect.Height = DestinationRectangle.Height;
```

```
                        }
                        spriteBatch.Draw(Image, DestinationRectangle, SourceRectangle,
Color);
                        break;
                    case FillMethod.Clip:
                        if (DestinationRectangle.Width > SourceRectangle.Width)
                        {
                            _destRect.Width = SourceRectangle.Width;
                        }

                        if (_destRect.Height > DestinationRectangle.Height)
                        {
                            _destRect.Height = DestinationRectangle.Height;
                        }
                        spriteBatch.Draw(Image, DestinationRectangle, SourceRectangle,
Color);
                        break;
                    case FillMethod.Fill:
                        _sourceRect = new(0, 0, Image.Width, Image.Height);
                        spriteBatch.Draw(Image, DestinationRectangle, null, Color);
                        break;
                    case FillMethod.Center:
                        _sourceRect.Width = Image.Width;
                        _sourceRect.Height = Image.Height;
                        _sourceRect.X = 0;
                        _sourceRect.Y = 0;

                        Rectangle dest = new(0, 0, Width, Height);

                        if (Image.Width >= Width)
                        {
                            dest.X = DestinationRectangle.X;
                        }
                        else
                        {
                            dest.X = DestinationRectangle.X + (Width - Image.Width) / 2;
                        }

                        if (Image.Height >= Height)
                        {
                            dest.Y = DestinationRectangle.Y;
                        }
                        else
                        {
                            dest.Y = DestinationRectangle.Y + (Height - Image.Height) /
2;
                        }
                        spriteBatch.Draw(Image, dest, SourceRectangle, Color);
                        break;
                }
            }
        }

        public override void HandleInput()
        {
        }

        #endregion

        #region Picture Box Methods
```

```
        public void SetPosition(Vector2 newPosition)
        {
            _destRect = new Rectangle(
                (int)newPosition.X,
                (int)newPosition.Y,
                _sourceRect.Width,
                _sourceRect.Height);
        }

        #endregion
    }
}
```

There is an enumeration with four values: Clip, Fill, Original, and Center. They define how the picture box will be rendered. If the value is Clip, the image will be clipped if it overflows the size of the picture box. Fill has the image fill the picture box. If it is greater than the bounds of the picture box, the image is scaled to fill the picture box. The Original is similar to the Clip value. I probably could have gotten by with just Clip. Finally, Center with center the image inside the picture box. It clips the edges if they exceed the bounds of the picture box.

The class inherits from Control, so it can be added to a ControlManager instance. There are a total of six fields in the class: _image, _sourceRect, _destRect, _fillMethod, _width, and _height. The fist holds the image to be drawn. The second and third and the source rectangle to be drawn and the destination to be drawn. The _fillMethod field determines the way the image will be drawn. _width and _height are the height and width of the picture box. Also, there are properties to expose the values.

There are two constructors for the class. One takes the image and the desination rectangle to draw the picture box. The second takes a source rectangle in addition to the other parameters. The first constructor sets the Image property to the image passed in. It also sets the destination rectangle. If the Image is not null, the source rectangle is set to the entire image. If it is null, the source rectangle is set to an empty rectangle. The colour is set to white so that it will be drawn without tinting. The fill method is set to original it will be drawn at its original size. If the width of the source rectangle is greater than the width of the destination rectangle, it is set to the width of the destination rectangle, clipping it. Similarly, if the height of the source rectangle is greater than the height of the destination, it is set to the height of the destination, clipping it.

The second constructor takes a third parameter, the source rectangle. It calls the first constructor. I think the only thing it does differently is set the source rectangle to the parameter passed in.

The Update method is implemented but does nothing. All of the magic happens in the Draw method. If the image is not null, there is a switch on the FillMethod passed in. If it is set to Original, it checks if the width of the images is greater than the width of the destination. If it is, the width of the source rectangle is set to the width of the destination. I do the same for the heights. I then draw the image. For Clip, I check to see if the destination is great than the source. If it is, the destination is set to the width. This probably isn't doing what I intended. I will fix it in a future tutorial. It then draws the image to the destination rectangle. Fill draws the entire image into the destination rectangle. So, the width and height of the image are set to the width and height of the image. Fill sets the source rectangle to be the entire image. The destination rectangle is set to the width and height of the image. The image is then rendered. The Center mode is a little tricky. First, the source rectangle is set to the size of the image. Next, a new rectangle is made that has the height and width of the control. If the width of the image is greater than the width of the picture box, the X coordinate of the

destination is set to the X coordinate of the picture box. Otherwise, it is set to the X coordinate of the picture box plus the width of the picture box minus the width of the image divided by 2. I do something similar for the height of the image. I then draw the rectangle. Then I draw the image.

The HandleInput doesn't do anything but needs to be implemented because it inherits from Control. There is a method, SetPosition that positions the picture box. It takes a Vector2, which is the position to place the picture box. Then, it sets the _destRect field to a rectangle that is the position, width, and heigh of the picture box.

Papers Please

Next up is a form. Forms are just fancy control managers. The cool thing is that they inherit from GameState, so they can be used by the state manager. Right-click the Controls folder in the SharedProject, select Add and then Class. Name this new class Form. Here is the code for the Form class. I'm sorry it is a bit long.

```
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework;
using System;
using System.Collections.Generic;
using System.Reflection.Metadata;
using System.Text;

namespace SharedProject.Controls
{
    public abstract class Form : GameState
    {
        private ControlManager _controls;
        protected readonly GraphicsDeviceManager _graphicsDevice;
        private Point _size;
        private Rectangle _bounds;
        private string _title;

        public string Title { get { return _title; } set { _title = value; } }
        public ControlManager Controls { get => _controls; set => _controls = value; }
        public Point Size { get => _size; set => _size = value; }
        public bool FullScreen { get; set; }
        public PictureBox Background { get; private set; }
        public PictureBox TitleBar { get; private set; }
        public Button CloseButton { get; private set; }
        public Rectangle Bounds { get => _bounds; protected set => _bounds = value; }
        public Vector2 Position { get; set; }

        public Form(Game game, Vector2 position, Point size) : base(game)
        {
            Enabled = true;
            Visible = true;
            FullScreen = false;
            _size = size;

            Position = position;
            Bounds = new(Point.Zero, Size);
            _graphicsDevice = (GraphicsDeviceManager)Game.Services.GetService(
                typeof(GraphicsDeviceManager));

            Initialize();
            LoadContent();
            Title = "";
```

```
        }

        public override void Initialize()
        {
            base.Initialize();
        }

        protected override void LoadContent()
        {
            base.LoadContent();
            _controls = new((Game.Content.Load<SpriteFont>("Fonts/MainFont")));

            TitleBar = new(
                Game.Content.Load<Texture2D>("GUI/TitleBar"),
                new(0, 0, Size.X, 20));

            Background = new(
                Game.Content.Load<Texture2D>("GUI/Form"),
                new(
                    0,
                    20,
                    Bounds.Width,
                    Bounds.Height))
            { FillMethod = FillMethod.Fill };

            CloseButton = new(
                Game.Content.Load<Texture2D>("GUI/CloseButton"),
                ButtonRole.Cancel)
            { Position = Vector2.Zero, Color = Color.White, Text = "" };

            CloseButton.Click += CloseButton_Click;

            if (FullScreen)
            {
                TitleBar.Height = 0;
                Background.Position = new();
                Background.Height = _graphicsDevice.PreferredBackBufferHeight;
            }
        }

        private void CloseButton_Click(object sender, EventArgs e)
        {
            StateManager.PopState();
        }

        public override void Update(GameTime gameTime)
        {
            if (Enabled)
            {
                CloseButton.Update(gameTime);
                Controls.Update(gameTime);
            }

            base.Update(gameTime);
        }

        public override void Draw(GameTime gameTime)
        {
            base.Draw(gameTime);
```

```
            if (!Visible) return;

            if (!FullScreen)
            {
                Vector2 size = ControlManager.SpriteFont.MeasureString(Title);
                Vector2 position = new((Bounds.Width - size.X) / 2, 0);
                Label label = new()
                {
                    Text = _title,
                    Color = Color.White,
                    Position = position
                };

                Matrix m = Matrix.CreateTranslation(new Vector3(Position, 0));

                SpriteBatch.Begin(SpriteSortMode.BackToFront, BlendState.AlphaBlend, Sam-
plerState.AnisotropicWrap, null, null, null, m);

                Background.Draw(SpriteBatch);
                TitleBar.Draw(SpriteBatch);

                CloseButton.Draw(SpriteBatch);

                SpriteBatch.End();

                SpriteBatch.Begin();

                label.Position = Helper.NearestInt(position + Position);
                label.Color = Color.White;
                label.Draw(SpriteBatch);

                SpriteBatch.End();

                m = Matrix.CreateTranslation(new Vector3(0, 20, 0) + new Vector3(Posi-
tion, 0));

                SpriteBatch.Begin(SpriteSortMode.FrontToBack, BlendState.AlphaBlend, Sam-
plerState.AnisotropicWrap, null, null, null, m);

                _controls.Draw(SpriteBatch);

                SpriteBatch.End();
            }
            else
            {
                SpriteBatch.Begin();

                Background.DestinationRectangle = new(
                    0,
                    0,
                    _graphicsDevice.PreferredBackBufferWidth,
                    _graphicsDevice.PreferredBackBufferHeight);

                _controls.Draw(SpriteBatch);

                SpriteBatch.End();
            }
        }
    }
}
```

This class inherits from BaseGameState, so it can be used with the state manager. There is a private, might want to make it protected at some point, a field for a control manager. There is a readonly GraphicsDeviceManager field. We need this to get the width and height of the window. There is a Point _size that is the size of the window and a Rectangle _bounds that is the bounds of the window. There is a bit of overlap there. I guess I could have gotten away with just the rectangle. A form also has a title, so there is a field for that. There are properties to expose the title, control manager, Size and if the form should be full-screen or windowed. There are three controls next two picture boxes and a button. The picture boxes hold the title bar and the background. The button is to close the form. There is also a property for the bounds of the form and a Vector2 for the position of the form. There is an overlap, but for reason. I will get to it when I go over rending the form.

The form takes as parameters: a Game object, a Vector2 for its position, and a Point that defines its size. A form is initialized to Enabled and Visible by default. It is windowed, not fullscreen., by default. The constructor then sets the _size field to the size passed in. The Position property is set as the position argument that is passed as well. The Bounds property is set to a rectangle with X and Y coordinates of 0 and the width and height equal to the size argument. I need a GraphicsDeviceManager, so in the game and editor, I will register the GraphicsDeviceManager as a service, and I retrieve it in the base form class. I explicitly call the Initialize and LoadContent methods rather than wait for them to be called.

I really didn't need the Initialize method, but I included it in case it might be required in the future. In the LoadContent method, I call the LoadContent method of the base class. I initialize the control manager passing in the main font that was added in a previous tutorial. I create the title bar next. Its width is set to the width of the form, and its height is set to twenty pixels. The background is created next. It's width is set to the width of the Bounds, and its height is set to the height of the bounds. I set its fill method to fill. Next, I create the close button. I position it in the upper left-hand corner like Apple. Yes, I am an Apple fan girl now. I have an iPhone, iPad, Apple Watch and MacBook. I don't see myself giving up my PC any time soon for a MacStudio. I set the colour to white so there won't be any tinting and the text to the empty string. I also wire the click event handler for the close button. If this is a full-screen window, I set the height of the title bar to zero and the height of the background to the full height of the window.

The event handler for the close button pops the state off of the stack. In the Update method, if the form is Enabled, the Update method of the close button and the control manager is called.

Now, the reason why there is a position and bounds for the form. Forms will seldom have their upper left-hand corner in the upper left-hand corner of the screen. It could be a problem handling that, especially if there are a lot of controls. Enter the solution: translation matrices. What we do is place everything in the upper left-hand corner of the window and then use a translation matrix in rendering to have MonoGame have MonoGame move everything for us. This is also why I added the Offset property to controls. In order to have the click in the right spot, you need to offset the control.

First, the Draw method of the base class is called. Then, if the form is not visible, I exit the method. If the window is not full-screen, I use the sprite font of the control manager to measure the Title property of the form. I then center the title horizontally. I create a label on the fly and set its position to the value just computed, the colour to white, and the text to the _title field. Next, I create the translation matrix passing in the Position property of the window.

I then call Begin on the sprite batch object to begin rendering. I have the sort order back to front, the blend state to alpha blend, the sampler state to PointClamp, null for the next three parameters and finally our matrix. I call the Draw method of the form elements, not the controls on the form. Separately, I call the Draw method

of the title after setting its position. I create a new matrix next, adding twenty pixels for the width of the title bar. I then call the Draw method of the control manager.

If the window is full-screen, I just call Begin on the sprite batch instead of worrying about offsets because everything is relative to the upper left-hand corner. Since there is no need to draw anything but the window background, I don't. I set the destination rectangle of the background to fill the entire screen. I then call the Draw method of the control manager.

I think I will sneak one more item into this tutorial. It is about the length I like to end them, though. I guess what I will do is add a new project for the map editor. Right-click the solution in the Solution Explorer, select Add and then New Project. In the window that comes up, type MonoGame Desktop in the search bar. Select the MonoGame Desktop Application tile and click Next. For the name, enter MapEditor, then click Create. Once Visual Studio has created the project, right-click the MapEditor project, select Add and then Project Reference. In the window that comes up, select RpgLibrary. Before closing the dialog select Shared Project on the left side, then select SharedProject. Right-click the Game1.cs file in the MapEditor project and select Rename. Rename the class to Editor. When prompted, select yes to rename code elements. Replace the MapEditor class with the following code.

Before we get to code, we need a little content. There are some (bad) controls that I use for forms. You can download them from this link. Once you have downloaded them, we need to add them to the SharedProject. In the SharedProject, expand the Content folder and then double-click the Content.mgcb file to open the MGCB Editor. Right-click the GUI folder, select Add and then Existing Item. Select all of the controls to add them. Before closing the MGCB Editor, we need to add a font. Right-click the Fonts folder, select Add and then New Item. From the list, select Sprite Font Description to add a SpriteFont. Name this new font MainFont. Save the project and close the MGCB Editor.

Now, I will add the game state for the editor. Right-click the MapEditor project, select Add and then New Folder. Name this folder GameStates. Now, right-click the GameStates folder, select Add and then Class. Name this new class MainForm. Here is the code for that class.

```
using Microsoft.Xna.Framework;
using SharedProject;
using SharedProject.Controls;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace MapEditor.GameStates
{
    public interface IMainForm
    {
        Form Target { get; }
    }

    public class MainForm : Form, IMainForm
    {
        public Form Target => this;

        public MainForm(Game game, Vector2 position, Point size)
```

```
            : base(game, position, size)
        {
            Game.Services.AddService<IMainForm>(this);
        }

        public override void Initialize()
        {
            base.Initialize();
        }

        protected override void LoadContent()
        {
            base.LoadContent();
        }

        public override void Update(GameTime gameTime)
        {
            base.Update(gameTime);
        }

        public override void Draw(GameTime gameTime)
        {
            base.Draw(gameTime);
        }
    }
}
```

It is just a stub right now. It doesn't actually do anything. The interesting thing, though, is that there is an interface, IMainForm that has a property that returns a type of Form rather than the GameState that is typically returned. I also changed the name to avoid confusion. Otherwise, it just registers itself as a service.

The last thing I'm going to tackle is the Editor class. Replace the Editor class with the following code.

```
using MapEditor.GameStates;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using SharedProject;

namespace MapEditor
{
    public class Editor : Game
    {
        private readonly GraphicsDeviceManager _graphics;
        private SpriteBatch _spriteBatch;
        private readonly GameStateManager _manager;

        public MainForm MainForm { get; private set; }

        public Editor()
        {
            _graphics = new(this)
            {
                PreferredBackBufferWidth = 1920,
                PreferredBackBufferHeight = 1080
            };
```

```
            _graphics.ApplyChanges();

            Services.AddService<GraphicsDeviceManager>(_graphics);

            _manager = new(this);
            Components.Add(_manager);

            Services.AddService<GameStateManager>(_manager);

            Components.Add(new Xin(this));

            Content.RootDirectory = "Content";
            IsMouseVisible = true;
        }

        protected override void Initialize()
        {
            // TODO: Add your initialization logic here

            base.Initialize();
        }

        protected override void LoadContent()
        {
            _spriteBatch = new SpriteBatch(GraphicsDevice);
            Services.AddService<SpriteBatch>(_spriteBatch);

            // TODO: use this.Content to load your game content here
            MainForm = new(this, Vector2.Zero, new(1920, 1080))
            {
                FullScreen= false,
            };

            _manager.PushState(MainForm);
        }

        protected override void Update(GameTime gameTime)
        {
            if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed ||
Keyboard.GetState().IsKeyDown(Keys.Escape))
                Exit();

            // TODO: Add your update logic here

            base.Update(gameTime);
        }

        protected override void Draw(GameTime gameTime)
        {
            GraphicsDevice.Clear(Color.CornflowerBlue);

            // TODO: Add your drawing code here

            base.Draw(gameTime);
        }
    }
}
```

What is new here from boilerplate code is that I made the GraphicsDeviceManager readonly as I don't

ever want a new instance of it. I also added a readonly field for the state manager. Then, there is a public property with a private set accessor for the main form.

In the constructor, I set the width of the window to 1920 and the height of the window to 1080. If your monitor can handle more or less, adjust the size accordingly. I then apply the changes and add the GraphicsDeviceManager as a service. I then instantiate the state manager and add it to the list of components. I also register it as a service. I then add Xin as a component.

In the LoadContent method, I do a few things. First, I register the SpriteBatch as a service. Then I create a new instance of the MainForm setting its FullScreen property to false. I then push it onto the stack.

You can set the MapEditor as the startup project by right-clicking it and selecting Set As Startup Project. If you build and run now, you will be met by a white window with a blue title bar. So, I am going to end this here. There is more that I could pack in, but I will save it for the following tutorials. I don't want you to have too much to digest at once. I encourage you to visit the news page of my site, https://cynthiamcmahon.ca/blog/, for the latest news on my tutorials.

Good luck with your game programming adventures!
Cynthia