Michael Schuld – mike@thehazymind.com – http://www.thehazymind.com

# Moving Cameras and a Simple Input Component

## Adding some Movement to our Camera

The first thing we are going to do to get some more active movement going on in our world is to add some transformation functions to the camera class. Since we are using Quaternions to store the rotations, this is actually fairly easy to do (you'll see in a second.) If you don't understand Quaternions, Google and Wikipedia are your friends, but most people don't really need that many details to figure out this tutorial. Once you have some simple understanding of what they are and how they work, read on.

To get our camera working nicely and in almost any situation we really only need three types of transformations (and one function to provide access to each one). We will need to be able to rotate, revolve, and translate the position of the camera. We will want to add functions for each in relation to the world and to the camera so we can easily create the types of cameras used in most games today. With these six functions we can use combinations of global and local rotate + translate for first person cameras and revolve + translate for third person. Here are the camera additions:

```
public void Rotate(Vector3 axis, float angle) {
    axis = Vector3.Transform(axis, Matrix.CreateFromQuaternion(Rotation));
    Rotation = Quaternion.Normalize(Quaternion.CreateFromAxisAngle(axis, angle) * Rotation);

    UpdateMatrices();
}

public void Translate(Vector3 distance) {
    Position += Vector3.Transform(distance, Matrix.CreateFromQuaternion(Rotation));

    UpdateMatrices();
}

public void Revolve(Vector3 axis, float angle) {
    Vector3 revolveAxis = Vector3.Transform(axis, Matrix.CreateFromQuaternion(Rotation));
    Quaternion rotate = Quaternion.CreateFromAxisAngle(revolveAxis, angle);
    Position = Vector3.Transform(Position - Target, Matrix.CreateFromQuaternion(rotate)) + Target;

    Rotate(axis, angle);
}
```

We could add a few more functions for setting the orientation specifically (instead of relative to its current position and rotation), but I will leave that for later when we get to more advanced uses of the camera. Here are the global counterparts to each of the above functions:

```
public void RotateGlobal(Vector3 axis, float angle) {
    Rotation = Quaternion.Normalize(Quaternion.CreateFromAxisAngle(axis, angle) * Rotation);

    UpdateMatrices();
}

public void TranslateGlobal(Vector3 distance) {
    Position += distance;

    UpdateMatrices();
```

```
}

public void RevolveGlobal(Vector3 axis, float angle) {
    Quaternion rotate = Quaternion.CreateFromAxisAngle(axis, angle);
    Position = Vector3.Transform(Position - Target, Matrix.CreateFromQuaternion(rotate)) + Target;

    RotateGlobal(axis, angle);
}
```

With some usable camera transformations, we need a way to play around with them to prove to ourselves they are all working. Instead of just throwing some movement in manually with a script or two, I decided to just implement an Input component to let us actively move the camera around with the mouse, which will make it much easier to debug if we run into any problems.

## The Input Components

For our input components, we will be setting up a basic input device that all the other devices will inherit from so they can all be added to the input manager the same way. The actual base input device class is fairly simple and only includes a few interface functions for the component manager to call:

```
using System;
using HMEngine.HMComponents;
using Microsoft.Xna.Framework;

namespace HMEngine.HMInputs {
    public abstract class HMInputDevice : IHMUpdatable, IHMInitializable, IDisposable {
        public abstract void Initialize();
        public abstract void Update(GameTime gameTime);
        public abstract void Dispose();
    }
}
```

We will need to add the HMInitializable interface and include it in the component manager with pieces for the disposable items as well exactly how all of our other interfaces have been done:

```
// New Interface
namespace HMEngine.HMComponents {
    public interface IHMInitializable : IHMComponent {
        void Initialize();
    }
}

// In HMComponentManager
private static readonly Dictionary<string, IHMInitializable> initializable =
    new Dictionary<string, IHMInitializable>();

private static readonly Dictionary<string, IDisposable> disposable = new Dictionary<string, IDisposable>();

public static void AddComponent(string name, IHMComponent component) {
    masterList.Add(name, component);

    if (component is IHMInitializable) {
        initializable.Add(name, (IHMInitializable)component);
    }

    // Exisiting Code...

    if (component is IDisposable) {
        disposable.Add(name, (IDisposable)component);
    }
}
```

```
public override void Initialize() {
    foreach (IHMInitializable initialize in initializable.Values) {
        initialize.Initialize();
    }

    base.Initialize();
}

protected override void Dispose(bool disposing) {
    foreach (IDisposable dispose in disposable.Values) {
       dispose.Dispose();
    }

    base.Dispose(disposing);
}
```

The first logical device to add to our set of input devices is the keyboard since we are already using it to watch for keypresses on the F key to go full screen. We will start out by making handlers to deal with the different types of input that can happen on a key: pressing, holding, and releasing. These are all being placed in HMKeyboardDevice.cs in my version of things. We will also be adding files for HMMouseDevice.cs and HMGamePadDevice.cs later on. I'll let you see the code for the whole keyboard device class and go through what it all does below:

```
using System.Collections.Generic;
using System.Linq;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Input;

namespace HMEngine.HMInputs {
    public delegate void HMKeyPressedHandler(IList<Keys> keys);
    public delegate void HMKeyHeldHandler(IList<Keys> keys);
    public delegate void HMKeyReleasedHandler(IList<Keys> keys);

    public sealed class HMKeyboardDevice : HMInputDevice {
        private IEnumerable<Keys> lastPressedKeys;
        private KeyboardState lastState;

        private IList <Keys> pressed;
        private IList <Keys> held;
        private IList <Keys> released;

        public event HMKeyPressedHandler OnKeyPressed;
        public event HMKeyHeldHandler OnKeyHeld;
        public event HMKeyReleasedHandler OnKeyReleased;

        public override void Initialize() { lastState = Keyboard.GetState(); }

        public override void Update(GameTime gameTime) {
            KeyboardState currentState = Keyboard.GetState();
            Keys[] currentPressedKeys = currentState.GetPressedKeys();
            lastPressedKeys = lastState.GetPressedKeys();
            lastState = currentState;

            // Loading Event Lists
            held = currentPressedKeys.Where(key => lastPressedKeys.Contains(key)).ToList();
            pressed = currentPressedKeys.Where(key => !lastPressedKeys.Contains(key)).ToList();
            released = lastPressedKeys.Where(key => !currentPressedKeys.Contains(key)).ToList();

            // Event Calls
            if (null != OnKeyPressed) {
                OnKeyPressed(pressed);
            }
            if (null != OnKeyHeld) {
                OnKeyHeld(held);
            }
            if (null != OnKeyReleased) {
```

```
                OnKeyReleased(released);
            }
        }

        public override void Dispose() { }
    }
}
```

We start out the class by just setting up collections for the keys that were pressed, held, and released so we can call each separately at the appropriate time. Immediately after that, we set the events up using the delegates we previously defined that we call in the update function to act on each key action. The Initialize function is pretty straightforward and just sets up the initial state of the keyboard when the application is first loaded. The update function starts out by just grabbing the key collections of the last state and the current state so we can compare them further down. We then reinitialize each of the key collections to empty sets so none of the data from the last update is still hanging around. For the first comparison we just go through the keys that are currently down and check if they were also down during the last update. This tells us that the key is in a held state and should be added to the held collection, which we do in the loop. If the key wasn't down last frame, then it must have just been pressed and so we add it to the pressed collection in this same loop. The second comparison we do the other direction checking for any keys that were down last update and have since been released and add them to that collection. At the very end of the update we call all of the events with their collections.

## The Input Manager

Before we can actually use the keyboard device, we need to add an input manager to send the device to and to be called by the rest of the engine. Once again this will just be a simple wrapper for the component manager as that is already handling calling initialize and update for us. Here it is:

```
using HMEngine.HMComponents;
using Microsoft.Xna.Framework;

namespace HMEngine.HMInputs {
    public sealed class HMInputManager : GameComponent {
        internal HMInputManager(Game game) : base(game) { }

        public static void AddDevice(string name, HMInputDevice device) {
            HMComponentManager.AddComponent(name, device);
        }
    }
}
```

To finish off the keyboard input, we will move the code for going to fullscreen to a key event in the demo. First remove Update from the HMGame class and add a wrapper for toggling to fullscreen.

```
using HMEngine.HMInputs;

public HMGame(int width, int height) {
    // GraphicsManager Code
    Components.Add(new HMObjectManager(this));
    Components.Add(new HMCameraManager(this));
    Components.Add(new HMEffectManager(this));
    Components.Add(new HMInputManager(this));
    Components.Add(new HMComponentManager(this));

    Window.AllowUserResizing = true;
    Window.ClientSizeChanged += Window_ClientSizeChanged;
```

```
}

public void ToggleFullScreen() { myGraphics.ToggleFullScreen(); }
```

## Updating the Demo

That should actually be about it for new additions and changes to the engine code for the keyboard input device. To test it all further we will simply add a function as an input handler to the demo for checking the F key and calling the fullscreen function. I also added a way to quit the demo with Escape:

```
using System.Collections.Generic;
using System.Linq;
using HMEngine;
using HMEngine.HMEffects;
using HMEngine.HMInputs;
using HMEngine.HMObjects;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Input;

namespace HMDemo {
    internal static class HMDemo {
        private static readonly HMGame game = new HMGame(800, 600);
        private static readonly HMQuad quad = new HMQuad("Content/Textures/hazymind");
        private static readonly HMShader shader = new HMShader("HMEngineContent/Shaders/TransformTexture");
        private static readonly HMKeyboardDevice keyboard = new HMKeyboardDevice();

        public static void Main() {
            HMInputManager.AddDevice("keyboard", keyboard);
            keyboard.OnKeyReleased += keyboard_OnKeyReleased;

            HMEffectManager.AddEffect("TT", shader);
            quad.Shader = "TT";

            HMObjectManager.AddObject("quad", quad);
            game.Run();
        }

        private static void keyboard_OnKeyReleased(IList<Keys> keys) {
            if (keys.Contains(Keys.F)) {
                game.ToggleFullScreen();
            }
            if (keys.Contains(Keys.Escape)) {
                game.Exit();
            }
        }
    }
}
```

## Adding other types of devices

Now that we have a functioning input manager, we need to get to work on writing a few more input devices that it can manage for us. The two we will be adding will be the default mouse device, and a handler for the Xbox 360 gamepad that is compatible with XNA (on the PC too!). The way these two are being implemented is very similar to the keyboard device, but we will need a lot of extra enumerations and other functions to help us out and keep our code clean. These will help us later on to add a bit more functionality than the default states in the framework are able to give us. I will also be adding a new function to the HMInputDevice class that will handle the utility work of checking states and converting from XNA mappings to our own enumerated button sets in later parts of the new input devices. CheckPressedState essentially does exactly what we were doing with our keyboard, only handling it with the HMPressedStates instead of the built in keyboard button states. This will be used by both the mouse

and the gamepad classes, which is why we are including it as a static function in the input device class and not in either of those two explicitly. We would like to use this for our keyboard as well, but the items that it would pass are just a bit too different:

```
using System;
using HMEngine.HMComponents;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Input;

namespace HMEngine.HMInputs {
    public enum HMPressedState {
        Pressed,
        Released,
        Held,
        Idle
    }

    public abstract class HMInputDevice : IHMUpdatable, IHMInitializable, IDisposable {
        public abstract void Initialize();
        public abstract void Update(GameTime gameTime);
        public abstract void Dispose();

        protected static HMPressedState CheckPressedState(ButtonState currentState, HMPressedState lastState) {
            if (currentState == ButtonState.Pressed) {
                if (lastState == HMPressedState.Pressed || lastState == HMPressedState.Held) {
                    return HMPressedState.Held;
                }
                return HMPressedState.Pressed;
            }

            if (lastState != HMPressedState.Released && lastState != HMPressedState.Idle) {
                return HMPressedState.Released;
            }

            return HMPressedState.Idle;
        }
    }
}
```

Now that we have our utility function in place, we can get to implementing the actual mouse and gamepad devices. Both of them are done in a very similar manner to the keyboard, but with uses of our utility mixed in every now and then. As we did with the keyboard class, we start out the mouse device with a list of delegates to create our events out of:

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

namespace HMEngine.HMInputs {
    public enum HMMouseButton {
        LeftButton = 0,
        MiddleButton = 1,
        RightButton = 2,
        XButton1 = 3,
        XButton2 = 4
    }

    public delegate void HMMouseClickedHandler(Point pos, IList<HMMouseButton> buttons);
    public delegate void HMMouseHeldHandler(Point pos, IList<HMMouseButton> buttons);
    public delegate void HMMouseReleasedHandler(Point pos, IList<HMMouseButton> buttons);
    public delegate void HMMouseMovedHandler(Vector2 move);
    public delegate void HMMouseScrolledHandler(int ticks);

    public sealed class HMMouseDevice : HMInputDevice {
        private MouseState mouseState;
```

```
        private Point initialMousePosition;
        private int lastScrollPosition;
        private bool freezeMouse;

        private IList<HMPressedState> buttonStates;
        private IList<ButtonState> pressedStates;

        private IList<HMMouseButton> pressed;
        private IList<HMMouseButton> held;
        private IList<HMMouseButton> released;

        public event HMMouseClickedHandler OnMouseClicked;
        public event HMMouseHeldHandler OnMouseHeld;
        public event HMMouseReleasedHandler OnMouseReleased;
        public event HMMouseMovedHandler OnMouseMoved;
        public event HMMouseScrolledHandler OnMouseScrolled;

        public bool Freeze {
            get { return freezeMouse; }
            set {
                freezeMouse = value;
                if (freezeMouse) {
                    mouseState = Mouse.GetState();
                }
                initialMousePosition = new Point(mouseState.X, mouseState.Y);
            }
        }
```

The members of the mouse device include collections once again for comparing the states and for sending lists of pressed, held, and relseased buttons to each of the event functions (as defined in the delegates.) We also have a freeze value to keep the pointer in place if we need “infinite”movement.

```
        public override void Initialize() {
            mouseState = Mouse.GetState();
            initialMousePosition = new Point(mouseState.X, mouseState.Y);
            lastScrollPosition = mouseState.ScrollWheelValue;

            buttonStates = new List<HMPressedState>();
            for (int i = 0; i < 5; i++) {
                buttonStates.Add(HMPressedState.Idle);
            }
        }
```

The Initialize function starts by tracking where the mouse was on the screen when we started the application and where its wheel was. Using our freeze variable will allow us to reset the position after each movement to give us infinite freedom in both directions. When we are rotating the camera around a target or playing in full screen mode, we can do so endlessly and not be constrained by the screen.

```
        public override void Update(GameTime gameTime) {
            // Mouse Wheel Checks
            int scrollWheelValue = mouseState.ScrollWheelValue;
            int scrollMoved = lastScrollPosition - scrollWheelValue;
            lastScrollPosition = scrollWheelValue;

            // Mouse Move Checks
            mouseState = Mouse.GetState();
            var currentMousePosition = new Point(mouseState.X, mouseState.Y);
            var mouseMoved = new Vector2(currentMousePosition.X - initialMousePosition.X,
                                         currentMousePosition.Y - initialMousePosition.Y);

            if (freezeMouse) {
                Mouse.SetPosition(initialMousePosition.X, initialMousePosition.Y);
            } else {
                initialMousePosition = currentMousePosition;
```

```
            }

            // Mouse Buttons Checks
            pressed = new List<HMMouseButton>();
            held = new List<HMMouseButton>();
            released = new List<HMMouseButton>();
            pressedStates = MousePressedStateArray(mouseState);
            for (int i = 0; i < 5; i++) {
                buttonStates[i] = CheckPressedState(pressedStates[i], buttonStates[i]);
            }
```

Our update function starts similarly to the keyboard update function and just checks the states of all of the buttons and compares them to the states during the last update. We also check against our freeze variable to choose between the two position update actions we can take. Then (below), the different collections are loaded for each event based on the compared states of those buttons and those events are called in exaclty the manner we have done in the keyboard. Finally we see MousePressedStateArray that just takes a MouseState and gives back a collection of the states of each button we can use.

```
            // Loading Event Lists
            for (int i = 0; i < 5; i++) {
                if (buttonStates[i] == HMPressedState.Pressed) {
                    pressed.Add((HMMouseButton)i);
                } else if (buttonStates[i] == HMPressedState.Held) {
                    held.Add((HMMouseButton)i);
                } else if (buttonStates[i] == HMPressedState.Released) {
                    released.Add((HMMouseButton)i);
                }
            }

            // Event Calls
            if (scrollMoved != 0 && null != OnMouseScrolled) {
                OnMouseScrolled(scrollMoved);
            }
            if (mouseMoved.Length() > 0 && null != OnMouseMoved) {
                OnMouseMoved(mouseMoved);
            }
            if (pressed.Count > 0 && null != OnMouseClicked) {
                OnMouseClicked(currentMousePosition, pressed);
            }
            if (held.Count > 0 && null != OnMouseHeld) {
                OnMouseHeld(currentMousePosition, held);
            }
            if (released.Count > 0 && null != OnMouseReleased) {
                OnMouseReleased(currentMousePosition, released);
            }
        }

        private static IList<ButtonState> MousePressedStateArray(MouseState mouse) {
            var states = new List<ButtonState> {
                mouse.LeftButton,
                mouse.MiddleButton,
                mouse.RightButton,
                mouse.XButton1,
                mouse.XButton2
            };

            return states;
        }

        public override void Dispose() { }
    }
}
```

Adding the mouse and some camera movement for both revolving around our quad and moving in and out around it now takes very little additon to the Demo code:

```
using HMEngine.HMCameras;
using Microsoft.Xna.Framework;

// In the Demo class
private static readonly HMMouseDevice mouse = new HMMouseDevice();

// Inside Demo.Main
HMInputManager.AddDevice("mouse", mouse);
mouse.OnMouseMoved += mouse_OnMouseMoved;
mouse.OnMouseScrolled += mouse OnMouseScrolled;

// New functions
private static void mouse_OnMouseMoved(Vector2 move) {
    HMCameraManager.ActiveCamera.Revolve(new Vector3(1, 0, 0), move.Y * 0.01f);
    HMCameraManager.ActiveCamera.RevolveGlobal(new Vector3(0, 1, 0), move.X * 0.01f);
}

private static void mouse_OnMouseScrolled(int ticks) {
    HMCameraManager.ActiveCamera.Translate(new Vector3(0, 0, ticks * 0.01f));
}
```

Now, running the demo will show the camera motion in action. You can move the mouse around to revolve the camera around the quad in the center of our world, and you can use the scroll wheel to move the camera closer or further away from the object in focus as well. I have this set up as a common third person style camera. Using Rotate for the first call and RotateGlobal for the second would be the common first person camera that is used in many games as well.

## Adding the GamePad

The only device left in the framework is the game pad, which will use the exact same concepts as the mouse device, so I will leave it as an exercise for you to look through the source and figure out what everything does. This one is basically a replacement for the mouse device on the Xbox 360, since mice are not supported on that platform. Here is my code for your perusal:

```
using System.Collections.ObjectModel;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Input;

namespace HMEngine.HMInputs {
    public enum HMGamePadButton {
        A = 0,
        B = 1,
        Back = 2,
        LeftShoulder = 3,
        LeftStick = 4,
        RightShoulder = 5,
        RightStick = 6,
        Start = 7,
        X = 8,
        Y = 9
    }

    public enum HMGamePadDPadDirection {
        Down = 0,
        Left = 1,
        Right = 2,
        Up = 3
    }
```

```
public delegate void HMGamePadPressedHandler(IList<HMGamePadButton> buttons);
public delegate void HMGamePadReleasedHandler(IList<HMGamePadButton> buttons);
public delegate void HMGamePadHeldHandler(IList<HMGamePadButton> buttons);
public delegate void HMGamePadDPadPressedHandler(IList<HMGamePadDPadDirection> directions);
public delegate void HMGamePadDPadReleasedHandler(IList<HMGamePadDPadDirection> directions);
public delegate void HMGamePadDPadHeldHandler(IList<HMGamePadDPadDirection> directions);
public delegate void HMGamePadJoystickHandler(Vector2 leftStick, Vector2 rightStick);

public sealed class HMGamePadDevice : HMInputDevice {
    private readonly PlayerIndex playerIndex;
    private GamePadState gamePadState;

    private readonly IList<HMPressedState> buttonStates;
    private readonly IList<HMPressedState> dButtonStates;
    private IList<ButtonState> pressedStates;
    private IList<ButtonState> dPressedStates;

    private IList<HMGamePadButton> pressed;
    private IList<HMGamePadButton> held;
    private IList<HMGamePadButton> released;
    private IList<HMGamePadDPadDirection> dpadPressed;
    private IList<HMGamePadDPadDirection> dpadHeld;
    private IList<HMGamePadDPadDirection> dpadReleased;

    private Vector2 ltStickMove;
    private Vector2 rtStickMove;

    public event HMGamePadPressedHandler OnButtonPressed;
    public event HMGamePadReleasedHandler OnButtonReleased;
    public event HMGamePadHeldHandler OnButtonHeld;
    public event HMGamePadDPadPressedHandler OnDPadPressed;
    public event HMGamePadDPadReleasedHandler OnDPadReleased;
    public event HMGamePadDPadHeldHandler OnDPadHeld;
    public event HMGamePadJoystickHandler OnJoystickMoved;

    public HMGamePadDevice(PlayerIndex player) {
        playerIndex = player;

        gamePadState = GamePad.GetState(playerIndex);
        buttonStates = new List<HMPressedState>();
        dButtonStates = new List<HMPressedState>();
        for (int i = 0; i < 10; i++) {
            buttonStates.Add(HMPressedState.Idle);
        }

        for (int i = 0; i < 4; i++) {
            dButtonStates.Add(HMPressedState.Idle);
        }
    }

    public override void Initialize() { }

    public override void Update(GameTime gameTime) {
        // Button checks including the DPad
        gamePadState = GamePad.GetState(playerIndex);
        pressed = new List<HMGamePadButton>();
        released = new List<HMGamePadButton>();
        held = new List<HMGamePadButton>();
        dpadPressed = new List<HMGamePadDPadDirection>();
        dpadReleased = new List<HMGamePadDPadDirection>();
        dpadHeld = new List<HMGamePadDPadDirection>();
        pressedStates = GamePadPressedStateArray(gamePadState);
        for (int i = 0; i < 10; i++) {
            buttonStates[i] = CheckPressedState(pressedStates[i], buttonStates[i]);
        }
        dPressedStates = DPadPressedStateArray(gamePadState.DPad);
        for (int i = 0; i < 4; i++) {
            dButtonStates[i] = CheckPressedState(dPressedStates[i], dButtonStates[i]);
        }
```

```
            // Checking the Joysticks for movement
            ltStickMove = new Vector2(gamePadState.ThumbSticks.Left.X, gamePadState.ThumbSticks.Left.Y);
            rtStickMove = new Vector2(gamePadState.ThumbSticks.Right.X, gamePadState.ThumbSticks.Right.Y);

            // Loading Event Lists
            for (int i = 0; i < 10; i++) {
                if (buttonStates[i] == HMPressedState.Pressed) {
                    pressed.Add((HMGamePadButton)i);
                } else if (buttonStates[i] == HMPressedState.Held) {
                    held.Add((HMGamePadButton)i);
                } else if (buttonStates[i] == HMPressedState.Released) {
                    released.Add((HMGamePadButton)i);
                }
            }
            for (int i = 0; i < 4; i++) {
                if (dButtonStates[i] == HMPressedState.Pressed) {
                    dpadPressed.Add((HMGamePadDPadDirection)i);
                } else if (dButtonStates[i] == HMPressedState.Held) {
                    dpadHeld.Add((HMGamePadDPadDirection)i);
                } else if (dButtonStates[i] == HMPressedState.Released) {
                    dpadReleased.Add((HMGamePadDPadDirection)i);
                }
            }

            // Event Calls
            if (pressed.Count > 0 && null != OnButtonPressed) {
                OnButtonPressed(pressed);
            }
            if (released.Count > 0 && null != OnButtonReleased) {
                OnButtonReleased(released);
            }
            if (held.Count > 0 && null != OnButtonHeld) {
                OnButtonHeld(held);
            }
            if (dpadPressed.Count > 0 && null != OnDPadPressed) {
                OnDPadPressed(dpadPressed);
            }
            if (dpadReleased.Count > 0 && null != OnDPadReleased) {
                OnDPadReleased(dpadReleased);
            }
            if (dpadHeld.Count > 0 && null != OnDPadHeld) {
                OnDPadHeld(dpadHeld);
            }
            if (ltStickMove.Length() + rtStickMove.Length() > 0 && null != OnJoystickMoved) {
                OnJoystickMoved(ltStickMove, rtStickMove);
            }
        }
        public override void Dispose() { }

        private static IList<ButtonState> GamePadPressedStateArray(GamePadState gamePad) {
            var states = new List<ButtonState> {
                gamePad.Buttons.A,
                gamePad.Buttons.B,
                gamePad.Buttons.Back,
                gamePad.Buttons.LeftShoulder,
                gamePad.Buttons.LeftStick,
                gamePad.Buttons.RightShoulder,
                gamePad.Buttons.RightStick,
                gamePad.Buttons.Start,
                gamePad.Buttons.X,
                gamePad.Buttons.Y
            };

            return states;
        }

        private static IList<ButtonState> DPadPressedStateArray(GamePadDPad dpad) {
            var states = new List<ButtonState> {
                dpad.Down,
                dpad.Left,
                dpad.Right,
```

```
                dpad.Up
            };

            return states;
        }
    }
}

// In HMDemo.cs
private static readonly HMGamePadDevice gamepad = new HMGamePadDevice(PlayerIndex.One);

public static void Main() {
    // Existing Code

    HMInputManager.AddDevice("player1", gamepad);
    gamepad.OnButtonReleased += gamepad_OnButtonReleased;
    gamepad.OnJoystickMoved += gamepad OnJoystickMoved;

    // Existing Code
}

private static void gamepad_OnButtonReleased(IList<HMGamePadButton> buttons) {
    if (buttons.Contains(HMGamePadButton.Back)) {
        game.Exit();
    }
}

private static void gamepad OnJoystickMoved(Vector2 leftStick, Vector2 rightStick) {
    HMCameraManager.ActiveCamera.Revolve(new Vector3(1, 0, 0), leftStick.Y * 0.1f);
    HMCameraManager.ActiveCamera.RevolveGlobal(new Vector3(0, 1, 0), leftStick.X * 0.1f);
    HMCameraManager.ActiveCamera.Translate(new Vector3(0, 0, -rightStick.Y * 0.5f));
}
```