Table des matières

Listings

1 Structure du projet PeripheralControl

Le projet **PeripheralControl** s'organise autour d'une structure simple, pensée pour séparer clairement la logique d'entrée, l'affichage et les classes de configuration. L'arborescence regroupe notamment les fichiers principaux C#, les classes de gestion d'événements et les objets utilitaires utilisés par l'application. Cette organisation facilite la lecture, la maintenance et l'extension du projet.

2 Code source

Cette section rassemble les fichiers constituant le cœur fonctionnel du projet. Ils définissent l'initialisation, la boucle principale de rendu, la gestion des périphériques et les structures de données utilisées pour représenter les entrées d'une manette.

2.1 Program.cs

Ce fichier contient le point d'entrée de l'application.

```
using var game = new PeripheralControl.Game1();
game.Run();
```

Listing 1 – Program.cs

2.2 Game1.cs

Le fichier central de l'application.

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using System;
using System.Collections.Generic;
using System.IO;
using System.Text.Json;
/*
 * Nom : Demiri / Hede
 * Prenom : Leart / Timoléon
 * Date : 02/12/2025
 * Description : Projet test controleur
 * Version : 1
 */
namespace PeripheralControl
{
    public class Game1 : Game
    {
        // Graphique
        private GraphicsDeviceManager _graphics;
        private SpriteBatch _spriteBatch;
        private SpriteFont _font;
        private Texture2D _pixelTex;

        // Etats
        private GamePadState _prevGamePadState;
        private KeyboardState _prevKeyboardState;
        private MouseState _prevMouseState;

        // Historique
        private readonly List<InputEvent> _history = new List<InputEvent>();
        private const int MaxHistoryEntries = 25;

        // Calibration par STICK (gauche / droite)
        private readonly List<StickConfig> _sticks = new List<StickConfig>();
        private int _selectedStickIndex = 0;

        // Drag slider
        private bool _isDraggingSlider = false;
        private int _dragStickIndex = -1;
        private bool _dragDeadzone = false; // true = deadzone, false = sensitivity
```

```
43          // Profils
            private const string ProfileFileName = "profile.json";

            // Temps entre frames (coté jeu)
            private double _lastFrameMs = 0.0;
48
            // Mesure de "latence" d'appui pour le bouton A (duree entre PRESS et ↩
                RELEASE)
            private bool _isButtonALatencyRunning = false;
            private TimeSpan _buttonAPressStart;
            private double _buttonALastDurationMs = 0.0;
53


            /// <summary>
            /// Constructeur du jeu, initialisation de la fenêtre et des configs de base
            /// </summary>
58          public Game1()
            {
                _graphics = new GraphicsDeviceManager(this);
                Content.RootDirectory = "Content";
                IsMouseVisible = true;
63
                _graphics.PreferredBackBufferWidth = 1280;
                _graphics.PreferredBackBufferHeight = 720;
                _graphics.ApplyChanges();

68              // Deux sticks seulement : gauche et droite
                _sticks.Add(new StickConfig("Stick gauche"));
                _sticks.Add(new StickConfig("Stick droit"));
            }

73

            /// <summary>
            /// chargement des ressources graphiques
            /// </summary>

78          protected override void LoadContent()
            {
                _spriteBatch = new SpriteBatch(GraphicsDevice);
                _font = Content.Load<SpriteFont>("Arial");

83              _pixelTex = new Texture2D(GraphicsDevice, 1, 1);
                _pixelTex.SetData(new[] { Color.White });
            }


88          /// <summary>
            /// Boucle de mise à jour, lecture des entrées
            /// </summary>
            /// <param name="gameTime">Infos sur temps de jeu</param>

93          protected override void Update(GameTime gameTime)
            {
                // temps entre deux update coté jeu (en ms)
                _lastFrameMs = gameTime.ElapsedGameTime.TotalMilliseconds;

98              KeyboardState keyboard = Keyboard.GetState();
                MouseState mouse = Mouse.GetState();

                if (keyboard.IsKeyDown(Keys.Escape))
                {
103                 Exit();
                }

                GamePadState state = GamePad.GetState(PlayerIndex.One);

108             if (state.IsConnected)
                {
                    HandleCalibrationKeyboardInput(keyboard);
                    HandleSlidersMouseInput(mouse);
                    UpdateHistory(state, gameTime);
113
                    // Vibration tant que A est maintenu
                    if (state.Buttons.A == ButtonState.Pressed)
                    {
```

```
                    GamePad.SetVibration(PlayerIndex.One, 1.0f, 1.0f);
                }
                else
                {
                    GamePad.SetVibration(PlayerIndex.One, 0f, 0f);
                }
            }
            else
            {
                GamePad.SetVibration(PlayerIndex.One, 0f, 0f);
            }

            _prevGamePadState = state;
            _prevKeyboardState = keyboard;
            _prevMouseState = mouse;

            base.Update(gameTime);
        }

        /// <summary>
        /// Fenetre en quatre panneaux
        /// </summary>
        private void GetPanels(out Rectangle buttonsPanel, out Rectangle ↩
            sticksPanel, out Rectangle calibPanel, out Rectangle historyPanel)
        {
            buttonsPanel = new Rectangle(20, 60, 420, 280);

            sticksPanel = new Rectangle(20, 360, 420, 320);

            calibPanel = new Rectangle(460, 60, 460, 580);

            historyPanel = new Rectangle(940, 60, 320, 580);
        }

        /// <summary>
        /// Petit helper pour savoir si une touche vient juste d'etre pressée
        /// </summary>
        private bool IsKeyJustPressed(KeyboardState current, Keys key)
        {
            return current.IsKeyDown(key) && !_prevKeyboardState.IsKeyDown(key);
        }

        /// <summary>
        /// Gestion des raccourcis clavier pour à la calibration et la latence
        /// </summary>
        private void HandleCalibrationKeyboardInput(KeyboardState keyboard)
        {
            if (_sticks.Count == 0)
            {
                return;
            }

            // Effacer historique + reset de la durée d'appui de A
            if (IsKeyJustPressed(keyboard, Keys.H))
            {
                _history.Clear();
                _buttonALastDurationMs = 0.0;
                _isButtonALatencyRunning = false;
            }

            // Sauvegarder et charger profils
            if (IsKeyJustPressed(keyboard, Keys.F5))
            {
                SaveProfile();
            }

            if (IsKeyJustPressed(keyboard, Keys.F9))
            {
                LoadProfile();
            }
        }


        /// <summary>
        /// Calcul des rectangles pour chaque stick
        /// </summary>
```

```
        /// <returns></returns>
193
        private List<StickSliderRects> GetStickSliderRects()
        {
            Rectangle dummy1;
            Rectangle dummy2;
198         Rectangle calibPanel;
            Rectangle dummy4;

            GetPanels(out dummy1, out dummy2, out calibPanel, out dummy4);

203         List<StickSliderRects> list = new List<StickSliderRects>();

            float lineHeight = 140f;
            int startY = calibPanel.Top + 70;
            int sliderHeight = 12;
208         int totalSliderWidth = calibPanel.Width - 60;

            for (int i = 0; i < _sticks.Count; i++)
            {
                int rowY = (int)(startY + i * lineHeight);
213             int leftX = calibPanel.Left + 30;
                int halfWidth = totalSliderWidth / 2;

                Rectangle deadzoneRect = new Rectangle(
                    leftX,
218                 rowY + 30,
                    halfWidth - 10,
                    sliderHeight
                );

223             Rectangle sensRect = new Rectangle(
                    leftX + halfWidth + 10,
                    rowY + 30,
                    halfWidth - 10,
                    sliderHeight
228             );

                StickSliderRects rects = new StickSliderRects();
                rects.StickIndex = i;
                rects.DeadzoneRect = deadzoneRect;
233             rects.SensRect = sensRect;

                list.Add(rects);
            }

238         return list;
        }


        /// <summary>
243     /// Gestion de la souris sur les sliders
        /// </summary>
        /// <param name="mouse"></param>

        private void HandleSlidersMouseInput(MouseState mouse)
248     {
            List<StickSliderRects> sliderRects = GetStickSliderRects();

            bool leftJustPressed = mouse.LeftButton == ButtonState.Pressed && ↩
                _prevMouseState.LeftButton == ButtonState.Released;
            bool leftReleased = mouse.LeftButton == ButtonState.Released && ↩
                _prevMouseState.LeftButton == ButtonState.Pressed;
253
            Point mousePoint = new Point(mouse.X, mouse.Y);

            if (leftJustPressed)
            {
258             foreach (StickSliderRects s in sliderRects)
                {
                    if (s.DeadzoneRect.Contains(mousePoint))
                    {
                        _isDraggingSlider = true;
263                     _dragStickIndex = s.StickIndex;
                        _dragDeadzone = true;
                        UpdateSliderValueFromMouse(mouse.X, s.DeadzoneRect);
```

```
                    return;
                }
                if (s.SensRect.Contains(mousePoint))
                {
                    _isDraggingSlider = true;
                    _dragStickIndex = s.StickIndex;
                    _dragDeadzone = false;
                    UpdateSliderValueFromMouse(mouse.X, s.SensRect);
                    return;
                }
            }
        }

        if (_isDraggingSlider && mouse.LeftButton == ButtonState.Pressed)
        {
            StickSliderRects s = sliderRects.Find(sl => sl.StickIndex == ↩
                _dragStickIndex);
            if (s != null)
            {
                if (_dragDeadzone)
                {
                    UpdateSliderValueFromMouse(mouse.X, s.DeadzoneRect);
                }
                else
                {
                    UpdateSliderValueFromMouse(mouse.X, s.SensRect);
                }
            }
        }

        if (leftReleased)
        {
            _isDraggingSlider = false;
            _dragStickIndex = -1;
        }
    }

    /// <summary>
    /// Mise à jour de la valeur d'un slider (deadzone ou sensibilité) en ↩
        fonction de la position X de la souris
    /// </summary>
    private void UpdateSliderValueFromMouse(int mouseX, Rectangle rect)
    {
        if (_dragStickIndex < 0 || _dragStickIndex >= _sticks.Count)
        {
            return;
        }

        float t = (mouseX - rect.Left) / (float)rect.Width;
        t = MathHelper.Clamp(t, 0f, 1f);

        StickConfig stick = _sticks[_dragStickIndex];

        if (_dragDeadzone)
        {
            stick.DeadZone = t * 0.9f; // 0 -> 0.9
        }
        else
        {
            float min = 0.1f;
            float max = 3.0f;
            stick.Sensitivity = min + t * (max - min);
        }
    }

    /// <summary>
    /// Mise à jour de l'historique des evenements d'entrée
    /// On mesure la durée entre PRESS et RELEASE du bouton A
    /// </summary>
    private void UpdateHistory(GamePadState state, GameTime gameTime)
    {
        Dictionary<string, ButtonState> buttons = new Dictionary<string, ↩
            ButtonState>()
        {
            { "A",      state.Buttons.A },
            { "B",      state.Buttons.B },
```

```
                { "X",      state.Buttons.X },
                { "Y",      state.Buttons.Y },
                { "Start",  state.Buttons.Start },
                { "Back",   state.Buttons.Back },
                { "L1",     state.Buttons.LeftShoulder },
                { "R1",     state.Buttons.RightShoulder },
                { "L3",     state.Buttons.LeftStick },
                { "R3",     state.Buttons.RightStick },
                { "Up",     state.DPad.Up },
                { "Down",   state.DPad.Down },
                { "Left",   state.DPad.Left },
                { "Right",  state.DPad.Right }
            };

            Dictionary<string, ButtonState> prevButtons = new Dictionary<string, ↩
                ButtonState>()
            {
                { "A",      _prevGamePadState.Buttons.A },
                { "B",      _prevGamePadState.Buttons.B },
                { "X",      _prevGamePadState.Buttons.X },
                { "Y",      _prevGamePadState.Buttons.Y },
                { "Start",  _prevGamePadState.Buttons.Start },
                { "Back",   _prevGamePadState.Buttons.Back },
                { "L1",     _prevGamePadState.Buttons.LeftShoulder },
                { "R1",     _prevGamePadState.Buttons.RightShoulder },
                { "L3",     _prevGamePadState.Buttons.LeftStick },
                { "R3",     _prevGamePadState.Buttons.RightStick },
                { "Up",     _prevGamePadState.DPad.Up },
                { "Down",   _prevGamePadState.DPad.Down },
                { "Left",   _prevGamePadState.DPad.Left },
                { "Right",  _prevGamePadState.DPad.Right }
            };

            foreach (KeyValuePair<string, ButtonState> kvp in buttons)
            {
                ButtonState currentState = kvp.Value;
                ButtonState previousState = prevButtons[kvp.Key];

                if (currentState != previousState)
                {
                    // si c'est le bouton A, on calcule la durée entre PRESS et ↩
                        RELEASE
                    HandleButtonALatency(kvp.Key, currentState, previousState, ↩
                        gameTime.TotalGameTime);

                    string stateText = currentState == ButtonState.Pressed ? ↩
                        "PRESS" : "RELEASE";
                    AddHistoryEvent(string.Format("{0} : {1}", kvp.Key, stateText));
                }
            }

            LogAxisChange("Left Stick X", _prevGamePadState.ThumbSticks.Left.X, ↩
                state.ThumbSticks.Left.X);
            LogAxisChange("Left Stick Y", _prevGamePadState.ThumbSticks.Left.Y, ↩
                state.ThumbSticks.Left.Y);
            LogAxisChange("Right Stick X", _prevGamePadState.ThumbSticks.Right.X, ↩
                state.ThumbSticks.Right.X);
            LogAxisChange("Right Stick Y", _prevGamePadState.ThumbSticks.Right.Y, ↩
                state.ThumbSticks.Right.Y);
            LogAxisChange("LT", _prevGamePadState.Triggers.Left, ↩
                state.Triggers.Left);
            LogAxisChange("RT", _prevGamePadState.Triggers.Right, ↩
                state.Triggers.Right);
        }

        /// <summary>
        /// Gestion de la durée d'appui pour le bouton A
        /// </summary>
        private void HandleButtonALatency(string buttonName, ButtonState current, ↩
            ButtonState previous, TimeSpan time)
        {
            if (buttonName != "A")
            {
                return;
            }

```

```
            // début de l'appui : A passe de RELEASE -> PRESS
            if (previous == ButtonState.Released && current == ButtonState.Pressed)
            {
                _isButtonALatencyRunning = true;
                _buttonAPressStart = time;
            }
            // fin de l'appui : A passe de PRESS -> RELEASE
            else if (previous == ButtonState.Pressed && current == ↩
                ButtonState.Released && _isButtonALatencyRunning)
            {
                TimeSpan delta = time - _buttonAPressStart;
                double ms = delta.TotalMilliseconds;
                if (ms < 0.0)
                {
                    ms = 0.0;
                }
                _buttonALastDurationMs = ms;
                _isButtonALatencyRunning = false;
            }
        }

        /// <summary>
        /// Détection d'un changement assez gros sur un axe pour l'ajouter dans ↩
            l'historique
        /// </summary>
        private void LogAxisChange(string name, float previous, float current)
        {
            if (Math.Abs(current - previous) >= 0.15f)
            {
                AddHistoryEvent(string.Format("{0} : {1:0.00}", name, current));
            }
        }

        /// <summary>
        /// Ajout d'un événement au début de la liste d'historique (en gardant une ↩
            taille max),
        /// en utilisant l'heure du PC
        /// </summary>
        private void AddHistoryEvent(string text)
        {
            _history.Insert(0, new InputEvent(DateTime.Now, text));
            if (_history.Count > MaxHistoryEntries)
            {
                _history.RemoveAt(_history.Count - 1);
            }
        }

        /// <summary>
        /// Dessin d'une ligne avec la texture 1x1 pixel étiré
        /// </summary>
        private void DrawLine(Vector2 start, Vector2 end, Color color, int ↩
            thickness = 2)
        {
            Vector2 edge = end - start;
            float angle = (float)Math.Atan2(edge.Y, edge.X);
            _spriteBatch.Draw(_pixelTex, start, null, color, angle, Vector2.Zero, ↩
                new Vector2(edge.Length(), thickness), SpriteEffects.None, 0);
        }

        /// <summary>
        /// Dessin d'un cercle
        /// </summary>
        private void DrawCircle(Vector2 position, float radius, Color color, int ↩
            thickness = 2)
        {
            int segments = 40;
            Vector2 prev = position + new Vector2(radius, 0);
            for (int i = 1; i <= segments; i++)
            {
                float theta = MathHelper.TwoPi * i / segments;
                Vector2 next = position + new Vector2(
                    radius * (float)Math.Cos(theta),
                    radius * (float)Math.Sin(theta));
                DrawLine(prev, next, color, thickness);
                prev = next;
            }
```

```
        }

        /// <summary>
        /// Dessin d'un rectangle
        /// </summary>
        private void DrawFilledRect(Rectangle rect, Color color)
        {
            _spriteBatch.Draw(_pixelTex, rect, color);
        }

        /// <summary>
        /// Dessin d'un panneau
        /// </summary>
        private void DrawPanel(Rectangle rect, string title)
        {
            DrawFilledRect(rect, new Color(20, 20, 20, 220));

            DrawLine(new Vector2(rect.Left, rect.Top), new Vector2(rect.Right, rect.Top), Color.Gray);
            DrawLine(new Vector2(rect.Right, rect.Top), new Vector2(rect.Right, rect.Bottom), Color.Gray);
            DrawLine(new Vector2(rect.Right, rect.Bottom), new Vector2(rect.Left, rect.Bottom), Color.Gray);
            DrawLine(new Vector2(rect.Left, rect.Bottom), new Vector2( rect.Left, rect.Top), Color.Gray);

            if (!string.IsNullOrEmpty(title))
            {
                _spriteBatch.DrawString(_font, title, new Vector2(rect.Left + 8, rect.Top + 4), Color.LightSkyBlue);
            }
        }

        /// <summary>
        /// Affichage des panneaux et infos de la manette.
        /// </summary>
        protected override void Draw(GameTime gameTime)
        {
            GraphicsDevice.Clear(new Color(10, 10, 20));
            _spriteBatch.Begin();

            GamePadState state = GamePad.GetState(PlayerIndex.One);

            if (!state.IsConnected)
            {
                _spriteBatch.DrawString(_font, "Controller not connected", new Vector2(20, 20), Color.Red);
                _spriteBatch.End();
                base.Draw(gameTime);
                return;
            }

            Rectangle buttonsPanel;
            Rectangle sticksPanel;
            Rectangle calibPanel;
            Rectangle historyPanel;
            GetPanels(out buttonsPanel, out sticksPanel, out calibPanel, out historyPanel);

            DrawPanel(buttonsPanel, "Buttons");
            DrawPanel(sticksPanel, "Sticks / Triggers");
            DrawPanel(calibPanel, "Calibration");
            DrawPanel(historyPanel, "History");

            _spriteBatch.DrawString(_font, "Gamepad connected", new Vector2(20, 20), Color.LimeGreen);

            DrawButtonsState(state, buttonsPanel);
            DrawSticksAndTriggers(state, sticksPanel);
            DrawCalibrationPanel(state, calibPanel);
            DrawHistoryPanel(historyPanel);

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

```
        /// <summary>
        /// Affichage de l'état de tous les boutons
        /// </summary>
        private void DrawButtonsState(GamePadState state, Rectangle panel)
        {
            int columns = 3;
            float marginX = 52f;
            float marginY = 42f;
            float spacingX = (panel.Width - 2 * marginX) / (columns - 1);
            float spacingY = 50f;

            Vector2 startPos = new Vector2(panel.Left + marginX, panel.Top + marginY);

            Dictionary<string, ButtonState> buttons = new Dictionary<string, ButtonState>()
            {
                { "A",     state.Buttons.A },
                { "B",     state.Buttons.B },
                { "X",     state.Buttons.X },
                { "Y",     state.Buttons.Y },
                { "Start", state.Buttons.Start },
                { "Back",  state.Buttons.Back },
                { "L1",    state.Buttons.LeftShoulder },
                { "R1",    state.Buttons.RightShoulder },
                { "L3",    state.Buttons.LeftStick },
                { "R3",    state.Buttons.RightStick },
                { "Up",    state.DPad.Up },
                { "Down",  state.DPad.Down },
                { "Left",  state.DPad.Left },
                { "Right", state.DPad.Right }
            };

            int i = 0;
            foreach (KeyValuePair<string, ButtonState> b in buttons)
            {
                int col = i % columns;
                int row = i / columns;

                Vector2 pos = startPos + new Vector2(col * spacingX, row * spacingY);
                Color color = b.Value == ButtonState.Pressed ? Color.OrangeRed : Color.DimGray;

                float radius = 20f;
                DrawCircle(pos, radius, color, 2);

                Vector2 textSize = _font.MeasureString(b.Key);
                Vector2 textPos = pos - textSize / 2f;
                _spriteBatch.DrawString(_font, b.Key, textPos, Color.White);

                i++;
            }
        }

        /// <summary>
        /// Affichage des sticks et des triggers
        /// </summary>
        private void DrawSticksAndTriggers(GamePadState state, Rectangle panel)
        {
            Vector2 leftStickCenter = new Vector2(panel.Left + 110, panel.Top + 120);
            Vector2 rightStickCenter = new Vector2(panel.Left + 310, panel.Top + 120);

            float rawLX = state.ThumbSticks.Left.X;
            float rawLY = state.ThumbSticks.Left.Y;
            float rawRX = state.ThumbSticks.Right.X;
            float rawRY = state.ThumbSticks.Right.Y;

            StickConfig cfgLeft = _sticks[0];
            StickConfig cfgRight = _sticks[1];

            float adjLX = cfgLeft.Apply(rawLX);
            float adjLY = cfgLeft.Apply(rawLY);
            float adjRX = cfgRight.Apply(rawRX);
            float adjRY = cfgRight.Apply(rawRY);

```

```
            // Stick gauche
            DrawCircle(leftStickCenter, 40, Color.White, 2);
            Vector2 leftStickPos = leftStickCenter + new Vector2(adjLX * 30, ↩
                -adjLY * 30);
            DrawFilledRect(new Rectangle((int)leftStickPos.X - 5, ↩
                (int)leftStickPos.Y - 5, 10, 10), Color.LimeGreen);
618         _spriteBatch.DrawString(
                _font,
                string.Format("L: ({0:0.00}; {1:0.00})", adjLX, adjLY),
                new Vector2(panel.Left + 20, panel.Top + 200),
                Color.LightGray);
623
            // Stick droit
            DrawCircle(rightStickCenter, 40, Color.White, 2);
            Vector2 rightStickPos = rightStickCenter + new Vector2(adjRX * 30, ↩
                -adjRY * 30);
            DrawFilledRect(new Rectangle((int)rightStickPos.X - 5, ↩
                (int)rightStickPos.Y - 5, 10, 10), Color.LimeGreen);
628         _spriteBatch.DrawString(
                _font,
                string.Format("R: ({0:0.00}; {1:0.00})", adjRX, adjRY),
                new Vector2(panel.Left + 220, panel.Top + 200),
                Color.LightGray);
633
            // Triggers
            float lt = state.Triggers.Left;
            float rt = state.Triggers.Right;

638         int barWidth = panel.Width - 60;
            int barHeight = 20;
            int barX = panel.Left + 30;
            int barY1 = panel.Top + 240;
            int barY2 = panel.Top + 280;
643
            DrawFilledRect(new Rectangle(barX, barY1, barWidth, barHeight), ↩
                Color.DimGray);
            DrawFilledRect(new Rectangle(barX, barY2, barWidth, barHeight), ↩
                Color.DimGray);

            DrawFilledRect(new Rectangle(barX, barY1, (int)(barWidth * lt), ↩
                barHeight), Color.Yellow);
648         DrawFilledRect(new Rectangle(barX, barY2, (int)(barWidth * rt), ↩
                barHeight), Color.Yellow);

            _spriteBatch.DrawString(_font, string.Format("LT : {0:0.00}", lt), new ↩
                Vector2(barX, barY1 - 24), Color.White);
            _spriteBatch.DrawString(_font, string.Format("RT : {0:0.00}", rt), new ↩
                Vector2(barX, barY2 - 24), Color.White);
        }
653

        /// <summary>
        /// Affichage du panneau de calibration
        /// </summary>
658     private void DrawCalibrationPanel(GamePadState state, Rectangle panel)
        {
            float lineHeight = 140f;
            Vector2 pos = new Vector2(panel.Left + 20, panel.Top + 40);
663
            _spriteBatch.DrawString(
                _font,
                "H: clear history  F5: save  F9: load",
                new Vector2(panel.Left + 12, panel.Bottom - 28),
668             Color.LightGray);

            List<StickSliderRects> sliderRects = GetStickSliderRects();

            for (int i = 0; i < _sticks.Count; i++)
673         {
                StickConfig stick = _sticks[i];
                StickSliderRects rects = sliderRects[i];

                float rawX;
678             float rawY;
                if (i == 0)
```

```
                {
                    rawX = state.ThumbSticks.Left.X;
                    rawY = state.ThumbSticks.Left.Y;
683             }
                else
                {
                    rawX = state.ThumbSticks.Right.X;
                    rawY = state.ThumbSticks.Right.Y;
688             }

                float adjX = stick.Apply(rawX);
                float adjY = stick.Apply(rawY);

693             string label = string.Format(
                    "{0}  DZ={1:0.00}  S={2:0.00}  Inv={3}",
                    stick.Name,
                    stick.DeadZone,
                    stick.Sensitivity,
698                 stick.Inverted ? "Y" : "N");

                string line2 = string.Format(
                    "Raw=({0:0.00},{1:0.00})  Adj=({2:0.00},{3:0.00})",
                    rawX, rawY, adjX, adjY);
703
                _spriteBatch.DrawString(_font, label, pos, Color.White);
                _spriteBatch.DrawString(_font, line2, pos + new Vector2(0, 20), ↩
                    Color.LightGray);

                // Sliders
708             DrawSlider(rects.DeadzoneRect, stick.DeadZone / 0.9f, "DZ");
                float tSens = (stick.Sensitivity - 0.1f) / (3.0f - 0.1f);
                DrawSlider(rects.SensRect, tSens, "S");

                pos.Y += lineHeight;
713             if (pos.Y > panel.Bottom - 60)
                {
                    break;
                }
            }
718     }

        /// <summary>
        /// Dessin d'un slider horizontal
        /// </summary>
723     private void DrawSlider(Rectangle rect, float t, string label)
        {
            t = MathHelper.Clamp(t, 0f, 1f);

            DrawFilledRect(rect, new Color(50, 50, 50));
728         DrawFilledRect(new Rectangle(rect.Left, rect.Top, (int)(rect.Width * ↩
                t), rect.Height), Color.SteelBlue);

            int knobX = rect.Left + (int)(rect.Width * t);
            Rectangle knob = new Rectangle(knobX - 5, rect.Top - 2, 10, ↩
                rect.Height + 4);
            DrawFilledRect(knob, Color.White);
733
            _spriteBatch.DrawString(_font, label, new Vector2(rect.Left, rect.Top ↩
                - 20), Color.LightGray);
        }

        /// <summary>
738     /// Affichage de l'historique
        /// </summary>
        private void DrawHistoryPanel(Rectangle panel)
        {
            string header = string.Format(
743             "Frame ~ {0:0.0} ms   A press ~ {1:0} ms",
                _lastFrameMs,
                _buttonALastDurationMs);

            _spriteBatch.DrawString(_font, header, new Vector2(panel.Left + 10, ↩
                panel.Top + 16), Color.LightSkyBlue);
748
            Vector2 pos = new Vector2(panel.Left + 10, panel.Top + 40);
            float lineHeight = 24f;
```

```
            foreach (InputEvent e in _history)
            {
                string line = string.Format(
                    "[{0:HH:mm:ss}] {1}",
                    e.Date,
                    e.Text);

                _spriteBatch.DrawString(_font, line, pos, Color.White);
                pos.Y += lineHeight;
                if (pos.Y > panel.Bottom - 20)
                {
                    break;
                }
            }
        }

        /// <summary>
        /// Sauvegarde des paramètres de tous les sticks dans un fichier JSON
        /// </summary>
        private void SaveProfile()
        {
            Profile profile = new Profile();

            foreach (StickConfig stick in _sticks)
            {
                StickConfig copy = new StickConfig(stick.Name);
                copy.DeadZone = stick.DeadZone;
                copy.Sensitivity = stick.Sensitivity;
                copy.Inverted = stick.Inverted;
                profile.Sticks.Add(copy);
            }

            JsonSerializerOptions options = new JsonSerializerOptions();
            options.WriteIndented = true;

            string json = JsonSerializer.Serialize(profile, options);
            File.WriteAllText(ProfileFileName, json);
        }

        /// <summary>
        /// Chargement des paramètres de sticks depuis le fichier JSON
        /// </summary>
        private void LoadProfile()
        {
            if (!File.Exists(ProfileFileName))
            {
                return;
            }

            string json = File.ReadAllText(ProfileFileName);
            Profile profile = JsonSerializer.Deserialize<Profile>(json);
            if (profile == null || profile.Sticks == null)
            {
                return;
            }

            for (int i = 0; i < _sticks.Count && i < profile.Sticks.Count; i++)
            {
                StickConfig saved = profile.Sticks[i];
                StickConfig target = _sticks[i];

                target.DeadZone = saved.DeadZone;
                target.Sensitivity = saved.Sensitivity;
                target.Inverted = saved.Inverted;
            }
        }
    }
}
```

Listing 2 – Game1.cs

2.3 Classes

Les classes suivantes encapsulent les structures logiques nécessaires au fonctionnement du projet. Elles permettent de gérer les événements, de décrire les configurations et de manipuler les zones graphiques associées aux sticks.

2.3.1 InputEvent

Cette classe représente un événement d'entrée détecté par l'application. Elle contient les informations nécessaires pour identifier l'action effectuée et son état.

```
using System;

public class InputEvent
{
    public DateTime Date { get; set; }
    public string Text { get; set; }

    public InputEvent(DateTime date, string text)
    {
        Date = date;
        Text = text;
    }
}
```

Listing 3 – InputEvent.cs

2.3.2 StickConfig

Cette classe stocke la configuration liée aux sticks analogiques. Elle regroupe les valeurs de calibration ainsi que les paramètres utilisés pour déterminer l'affichage ou les limites du mouvement.

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

namespace PeripheralControl
{
    public class StickConfig
    {
        public string Name { get; }
        public float DeadZone { get; set; } = 0.15f;
        public float Sensitivity { get; set; } = 1.0f;
        public bool Inverted { get; set; } = false;

        public StickConfig(string name)
        {
            Name = name;
        }

        // Applique la calibration sur un axe (X ou Y) du stick
        public float Apply(float raw)
        {
            float value = raw;

            // Deadzone
            if (Math.Abs(value) < DeadZone)
            {
                return 0f;
            }

            float sign = Math.Sign(value);
            float magnitude = (Math.Abs(value) - DeadZone) / (1f - DeadZone);
            value = sign * magnitude;

            // Sensibilité
            value *= Sensitivity;

            // Clamp
            value = MathHelper.Clamp(value, -1f, 1f);
```

```
            // Inversion
            if (Inverted)
            {
                value = -value;
            }

            return value;
        }
    }
}
```

Listing 4 – StickConfig.cs

2.3.3 StickSliderRects

Cette classe définit les rectangles et zones graphiques utilisés pour représenter visuellement les positions et déplacements des sticks. Elle sert notamment au rendu des curseurs et des indicateurs analogiques.

```
using Microsoft.Xna.Framework;

namespace PeripheralControl
{
    public class StickSliderRects
    {

        // Index du stick dans la liste

        public int StickIndex;


        // Rectangle slider de deadzone

        public Rectangle DeadzoneRect;


        // Rectangle slider de sensibilité

        public Rectangle SensRect;
    }
}
```

Listing 5 – StickSliderRects.cs

2.3.4 TestCase

Cette classe est la classe qui s'occupe de tout les test en utilisant MSTest.

```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using System;

namespace PeripheralControl
{
    [TestClass]
    public class Tests
    {
        [TestMethod]
        public void StickConfig_InsideDeadZone_ReturnsZero()
        {
            var stick = new StickConfig("Test");
            stick.DeadZone = 0.2f;

            float result = stick.Apply(0.1f);

            Assert.AreEqual(0f, result, 0.0001f);
        }

        [TestMethod]
        public void Profile_Constructor_CreatesEmptyList()
        {
```

```
23              var profile = new Profile();
                Assert.IsNotNull(profile.Sticks);
                Assert.AreEqual(0, profile.Sticks.Count);
            }

28          [TestMethod]
            public void InputEvent_Constructor_StoresDateAndText()
            {
                DateTime now = DateTime.Now;
                string msg = "Test";
33
                var e = new InputEvent(now, msg);

                Assert.AreEqual(now, e.Date);
                Assert.AreEqual(msg, e.Text);
38          }
        }
    }
```

Listing 6 – Tests.cs