Střední škola informatiky, poštovnictví a finančnictví Brno, příspěvková organizace

Ročníková práce

Brno 2017 Marian Dolinský

Střední škola informatiky, poštovnictví a finančnictví Brno, příspěvková organizace

Hra Snake

Ročníková práce

Vedoucí práce:
Mgr. František Skalka

Vypracoval: Marian Dolinský
Studijní obor: IT
Číslo oboru: 18-20-M/01
Třída: IT3

Brno 2017

Prohlašuji, že jsem ročníkovou práci vypracoval samostatně s přispěním vedoucího práce a použil jsem jen literaturu a informační zdroje uvedené v kapitole literatura.

Děkuji Mgr. Františku Skalkovi za odborné vedení a cenné rady, které mi poskytl při zpracování této ročníkové práce.

Souhlasím s půjčováním a zpřístupněním ročníkové práce.

V Brně 15. května 2017 .....

Obsah

1 Úvod

Pro letošní ročníkovou práci jsem si, stejně jako minulý rok, zvolil téma „Hra“ a stejně tak jsem znovu vytvořil téměř celosvětově známou hru Snake, tentokrát však pomocí objektově orientovaného programovacího jazyka C#, který mi umožnil hru napsat mnohem efektivněji. Původně jsem se měl zúčastnit projektu firmy Sodat a jako ročníkovou práci vytvořit „aplikaci pro bezpečnou skartaci dat“, ale protože jsem nikde nenašel knihovny poskytující tuto funkčnost, které bych mohl ve své aplikaci použít, vrátil jsem se k tématu hra.

Jako úložiště pro tuto práci jsem použil GitHub, kde je tato ročníková práce veřejně dostupná pod licencí MIT. GitHub je online úložiště kódu pracující na principu Gitu, což je distribuovaný systém správy verzí – umožňuje mi vrátit se ke každé jednotlivé verzi nahrané na server, a můžu tak jednoduše zjistit, kdy a kde vznikla nějaká chyba, a jednoduše ji opravit.

Pro vývoj hry jsem také použil jednu svoji knihovnu, taktéž open source, dostupnou na GitHubu a distribuovanou přes NuGet – NotifyPropertyChangedBase, která pomáhá implementovat INotifyPropertyChanged interface často používaný pro data binding. Aplikace ji sice zatím naplno nevyužívá, jelikož z ní dědí třída AppData, která ještě nemá vypracované uživatelské rozhraní, ale tato knihovna mi pomůže v budoucnu, právě až toto rozhraní bude hotové.

Abych mohl obsah třídy uložit jako text, používám taktéž open source knihovnu Newtonsoft.Json, která slouží k serializaci a deserializaci dat do/z JSONu a která je taktéž volně ke stažení na NuGetu.

2 Zdrojový kód

2.1 SnakeTheResurrection.Data.AppData

```
using Newtonsoft.Json;
using NotifyPropertyChangedBase;
using SnakeTheResurrection.Utilities;
using System;

namespace SnakeTheResurrection.Data
{
    public sealed class AppData : NotifyPropertyChanged
    {
        private static readonly string filePath = Environment.GetFolderPath(Environ-
ment.SpecialFolder.ApplicationData) + $@"\{Constants.APP_NAME}\AppData.json";

        public static AppData Current { get; private set; }

        [JsonIgnore]
        public bool ShowLoadingError { get; set; }
        public bool EnableDiagonalMovement
        {
            get { return (bool)GetValue(); }
            set { SetValue(value); }
        }
        public bool ForceGameBoardBorders
        {
            get { return (bool)GetValue(); }
            set { SetValue(value); }
        }

        public AppData()
        {
            RegisterProperty(nameof(EnableDiagonalMovement), typeof(bool), true);
            RegisterProperty(nameof(ForceGameBoardBorders), typeof(bool), false);
        }

        public void Save()
        {
            FileHelper.SaveObject(this, filePath);
        }

        public static void Load()
        {
#if DEBUG
            if (Current != null)
            {
                throw new Exception("You're not doing it right ;)");
            }
#endif

            var loadObjectAsyncResult = FileHelper.LoadObject<AppData>(filePath);
            Current                   = loadObjectAsyncResult.Object;
            Current.ShowLoadingError  = !loadObjectAsyncResult.Success;

            Current.PropertyChanged += (sender, e) =>
            {
                Current.Save();
            };
        }
```

```
    }
}
```

2.2 SnakeTheResurrection.Data.Profile

```
using System;

namespace SnakeTheResurrection.Data
{
    public sealed class Profile
    {
        public string Name { get; set; }
        public ConsoleColor Color { get; set; }
        public SnakeControls SnakeControls { get; set; }

        public Profile()
        {
            SnakeControls = new SnakeControls();
        }
    }

    public sealed class SnakeControls
    {
        public ConsoleKey Up { get; set; }
        public ConsoleKey Down { get; set; }
        public ConsoleKey Left { get; set; }
        public ConsoleKey Right { get; set; }

        public SnakeControls()
        {
            Up      = ConsoleKey.UpArrow;
            Down    = ConsoleKey.DownArrow;
            Left    = ConsoleKey.LeftArrow;
            Right   = ConsoleKey.RightArrow;
        }
    }
}
```

2.3 SnakeTheResurrection.Utilities.Cheats

```
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;

namespace SnakeTheResurrection.Utilities
{
    public static class Cheats
    {
        public enum CheatCode
        {
            Nothing
        }

        private static readonly Dictionary<CheatCode, bool> cheatCodeInfo = new Dic-
tionary<CheatCode, bool>
        {
            { CheatCode.Nothing, false }
```

```
        };

        private static CancellationTokenSource previousCts;

        public static ReadOnlyDictionary<CheatCode, bool> CheatCodeInfo
        {
            get
            {
                return new ReadOnlyDictionary<CheatCode, bool>(cheatCodeInfo);
            }
        }

        public static ConsoleKeyInfo ValidateCheat(ConsoleKeyInfo pressedKeyInfo)
        {
            if (char.IsLetter(pressedKeyInfo.KeyChar))
            {
                // We don't currently support more cheats starting with the same
letter
                string currentCheatCode = Enum.Get-
Names(typeof(CheatCode)).FirstOrDefault(c => char.ToLower(c[0]) ==
char.ToLower(pressedKeyInfo.KeyChar));

                if (currentCheatCode != null)
                {
                    for (int i = 1; i < currentCheatCode.Length; i++)
                    {
                        pressedKeyInfo = Console.ReadKey(true);

                        if (char.ToLower(currentCheatCode[i]) !=
char.ToLower(pressedKeyInfo.KeyChar))
                        {
                            return pressedKeyInfo;
                        }
                    }

                    CheatCode currentCode = (CheatCode)Enum.Parse(typeof(CheatCode),
currentCheatCode);
                    cheatCodeInfo[currentCode] = !cheatCodeInfo[currentCode];

                    CancellationTokenSource currentCts = new CancellationToken-
Source();

                    Task.Run(async () =>
                    {
                        const string CHEAT_ACTIVATED_MESSAGE    = " Cheat activated
";
                        const string CHEAT_DEACTIVATED_MESSAGE  = " Cheat deac-
tivated ";


                        previousCts?.Cancel();

                        lock (Symtext.SyncRoot)
                        {
                            Symtext.SetCursorPosition(0, 0);
                            Symtext.FontSize            = 1;
                            Symtext.BackgroundColor     = ConsoleColor.Gray;
                            Symtext.ForegroundColor     = ConsoleColor.Black;
                            Symtext.HorizontalAlignment = HorizontalAlignment.None;
                            Symtext.VerticalAlignment   = VerticalAlignment.None;
```

```
                            Renderer.RemoveFromBuffer(0, 0, Symtext.CharHeight,
Symtext.GetSymtextWidth(CHEAT_DEACTIVATED_MESSAGE));
                            Symtext.Write(cheatCodeInfo[currentCode] ? CHEAT_ACTI-
VATED_MESSAGE : CHEAT_DEACTIVATED_MESSAGE);
                            Renderer.RenderFrame();
                        }

                        await Task.Delay(TimeSpan.FromSeconds(5));

                        if (!currentCts.IsCancellationRequested)
                        {
                            lock (Symtext.SyncRoot)
                            {
                                Renderer.RemoveFromBuffer(0, 0, Symtext.CharHeight,
Symtext.GetSymtextWidth(CHEAT_DEACTIVATED_MESSAGE));
                                Renderer.RenderFrame();
                            }
                        }
                    }, currentCts.Token);

                    previousCts = currentCts;
                }
            }

            return pressedKeyInfo;
        }
    }
}
```

2.4 SnakeTheResurrection.Utilities.DebugHelper

```
using System;
using System.Diagnostics;

namespace SnakeTheResurrection.Utilities
{
    public static class DebugHelper
    {
        [Conditional("DEBUG")]
        public static void OperationInfo(string objectName, string operationName,
bool success)
        {
            Debug.WriteLine($"{DateTime.Now:HH:mm:ss}: {objectName} {operationName}
{(success ? "succeeded" : "failed")}");
        }
    }
}
```

2.5 SnakeTheResurrection.Utilities.DllImports

```
using System;
using System.Diagnostics;
using System.Runtime.InteropServices;

namespace SnakeTheResurrection.Utilities
{
    public unsafe static class DllImports
    {
        private const int STD_OUTPUT_HANDLE         = -11;
        private const int KEY_PRESSED               = 0x8000;
```

```
        private const int SW_MAXIMIZE               = 3;
        private const int CONSOLE_FULLSCREEN_MODE   = 1;
        private const int CONSOLE_WINDOWED_MODE     = 2;
        private const int GWL_STYLE                 = -16;
        private const int WS_OVERLAPPED             = 0;
        private const int WS_CAPTION                = 0xC00000;
        private const int WS_SYSMENU                = 0x80000;
        private const int WS_MINIMIZEBOX            = 0x20000;
        private const int WS_MAXIMIZEBOX            = 0x10000;
        private const int INVALID_HANDLE_VALUE      = -1;
        private const int NULL                      = 0;

        private static readonly IntPtr mainWindowHandle;

        public static IntPtr StdOutputHandle { get; }

        public static bool ConsoleFullscreen
        {
            get
            {
                uint lpModeFlags;
                ExceptionHelper.ValidateMagic(GetConsoleDisplayMode(out lpModeF-
lags));

                return lpModeFlags == CONSOLE_FULLSCREEN_MODE;
            }
            set
            {
                COORD lpNewScreenBufferDimensions;

                if (!SetConsoleDisplayMode(StdOutputHandle, (uint)(value ? CON-
SOLE_FULLSCREEN_MODE : CONSOLE_WINDOWED_MODE), out lpNewScreenBufferDimensions))
                {
                    // Compatibility with Windows Vista, 7, 8.x
                    ShowWindow(mainWindowHandle, SW_MAXIMIZE);
                }
            }
        }

        static DllImports()
        {
            StdOutputHandle = GetStdHandle(STD_OUTPUT_HANDLE);
            ExceptionHelper.ValidateMagic(StdOutputHandle != new IntPtr(INVALID_HAN-
DLE_VALUE) && StdOutputHandle != new IntPtr(NULL));

            mainWindowHandle = Process.GetCurrentProcess().MainWindowHandle;
        }

        public static void DisableWindowButtons()
        {
            // Backup
            // SetWindowLong(mainWindowHandle, GWL_STYLE, GetWindowLong(mainWindow-
Handle, GWL_STYLE) & ~(WS_OVERLAPPED | WS_CAPTION | WS_SYSMENU | WS_MINIMIZEBOX |
WS_MAXIMIZEBOX));
            SetWindowLong(mainWindowHandle, GWL_STYLE, GetWindowLong(mainWindowHan-
dle, GWL_STYLE) & ~WS_CAPTION);
        }

        public static bool IsKeyDown(ConsoleKey key)
        {
            return (GetKeyState((int)key) & KEY_PRESSED) != 0;
```

```
        }

        public static unsafe void SetFont(string fontName, short x, short y)
        {
            CONSOLE_FONT_INFOEX info = new CONSOLE_FONT_INFOEX()
            {
                dwFontSize = new COORD(x, y)
            };

            info.cbSize = (uint)Marshal.SizeOf(info);

            Marshal.Copy(fontName.ToCharArray(), 0, new IntPtr(info.FaceName), font-
Name.Length);
            ExceptionHelper.ValidateMagic(SetCurrentConsoleFontEx(StdOutputHandle,
false, ref info));
        }

        public static int MessageBox(string message, string title, uint type = 0 |
0x10, bool exitProgram = true)
        {
            int output = MessageBox((IntPtr)0, message, title, type);

            if (exitProgram)
            {
                Program.ExitWithError();
            }

            return output;
        }

        [DllImport("kernel32.dll", SetLastError = true)]
        private static extern IntPtr GetStdHandle(int nStdHandle);

        [DllImport("kernel32.dll", SetLastError = true)]
        private static extern bool GetConsoleDisplayMode(out uint lpModeFlags);

        [DllImport("kernel32.dll", SetLastError = true)]
        private static extern bool SetConsoleDisplayMode(IntPtr hConsoleOutput, uint
dwFlags, out COORD lpNewScreenBufferDimensions);

        [DllImport("kernel32.dll", SetLastError = true)]
        private static extern bool SetCurrentConsoleFontEx(IntPtr hConsoleOutput,
bool bMaximumWindow, ref CONSOLE_FONT_INFOEX lpConsoleCurrentFontEx);

        [DllImport("user32.dll")]
        private static extern short GetKeyState(int key);

        [DllImport("user32.dll", SetLastError = true)]
        private static extern int GetWindowLong(IntPtr hWnd, int nIndex);

        [DllImport("user32.dll")]
        private static extern int SetWindowLong(IntPtr hWnd, int nIndex, int dwNew-
Long);

        [DllImport("user32.dll")]
        public static extern bool ShowWindow(IntPtr hWnd, int cmdShow);

        [DllImport("user32.dll", CharSet = CharSet.Unicode)]
        private static extern int MessageBox(IntPtr hWnd, string lpText, string
lpCaption, uint uType);
```

```
        [StructLayout(LayoutKind.Sequential, CharSet = CharSet.Unicode)]
        private unsafe struct CONSOLE_FONT_INFOEX
        {
            public uint cbSize;
            public uint nFont;
            public COORD dwFontSize;
            public int FontFamily;
            public int FontWeight;
            public fixed char FaceName[32];
        }

        [DebuggerDisplay("{X},{Y}")]
        [StructLayout(LayoutKind.Sequential)]
        public struct COORD
        {
            public short X;
            public short Y;

            public COORD(short x, short y)
            {
                X = x;
                Y = y;
            }
        }

        [StructLayout(LayoutKind.Explicit)]
        public struct CHAR_UNION
        {
            [FieldOffset(0)]
            public char UnicodeChar;
            [FieldOffset(0)]
            public byte AsciiChar;
        }

        [StructLayout(LayoutKind.Explicit)]
        public struct CHAR_INFO
        {
            [FieldOffset(0)]
            public CHAR_UNION Char;
            [FieldOffset(2)]
            public short Attributes;
        }

        [StructLayout(LayoutKind.Sequential)]
        public struct SMALL_RECT
        {
            public short Left;
            public short Top;
            public short Right;
            public short Bottom;

            public SMALL_RECT(short left, short top, short right, short bottom)
            {
                Left    = left;
                Top     = top;
                Right   = right;
                Bottom  = bottom;
            }
        }
    }
}
```

2.6 SnakeTheResurrection.Utilities.ExceptionHelper

```
using System;

namespace SnakeTheResurrection.Utilities
{
    public static class ExceptionHelper
    {
        public static void ValidateObjectNotNull(object obj, string parameterName)
        {
            if (obj == null)
            {
                throw new ArgumentNullException(parameterName);
            }
        }

        public static void ValidateStringNotNullOrWhiteSpace(string str, string pa-
rameterName)
        {
            if (string.IsNullOrWhiteSpace(str))
            {
                throw new ArgumentException("Value cannot be white space or null.",
parameterName);
            }
        }

        public static void ValidateEnumValueDefined(Enum enumValue, string parame-
terName)
        {
            if (!Enum.IsDefined(enumValue.GetType(), enumValue))
            {
                throw new ArgumentOutOfRangeException(parameterName);
            }
        }

        public static void ValidateNumberGreaterOrEqual(int value, int min, string
parameterName)
        {
            if (value < min)
            {
                throw new ArgumentOutOfRangeException(parameterName, $"Value
({value}) is out of range (smaller than {min}).");
            }
        }

        public static void ValidateNumberSmallerOrEqual(int value, int max, string
parameterName)
        {
            if (value > max)
            {
                throw new ArgumentOutOfRangeException(parameterName, $"Value
({value}) is out of range (greater than {max}).");
            }
        }

        public static void ValidateNumberInRange(int value, int min, int max, string
parameterName)
        {
            if (value < min || value > max)
            {
```

```
                throw new ArgumentOutOfRangeException(parameterName, $"Value
({value}) is out of range ({min} - {max}).");
            }
        }

        public static void ValidateMagic(bool magic)
        {
            if (!magic)
            {
                ThrowMagicException();
            }
        }

        public static void ThrowMagicException()
        {
            // To be able to debug the call stack etc
#if DEBUG
            throw new Exception();
#else
            DllImports.MessageBox(@"We are so sorry but some unknown dark power pre-
vented us from doing the required magic ¯\_(ツ)_/¯", "No magic");
#endif
        }
    }
}
```

2.7 SnakeTheResurrection.Utilities.FileHelper

```
using Newtonsoft.Json;
using System.IO;

namespace SnakeTheResurrection.Utilities
{
    // Ported from my StorageFileHelper from UWPHelper - https://github.com/brambor-
man/UWPHelper/blob/master/UWPHelper/Utilities/StorageFileHelper.cs
    public static class FileHelper
    {
        public static bool SaveObject(object obj, string filePath)
        {
            ExceptionHelper.ValidateStringNotNullOrWhiteSpace(filePath,
nameof(filePath));

            bool success    = true;
            string fileName = Path.GetFileName(filePath);

            try
            {
                string folderPath = filePath.Substring(0, filePath.Length - file-
Name.Length);

                if (!Directory.Exists(folderPath))
                {
                    Directory.CreateDirectory(folderPath);
                }

                File.WriteAllText(filePath, JsonConvert.SerializeObject(obj), Con-
stants.encoding);
            }
            catch
            {
                success = false;
```

```
            }

            DebugHelper.OperationInfo(fileName, "saving", success);
            return success;
        }

        public static LoadObjectAsyncResult<T> LoadObject<T>(string filePath) where
T : class, new()
        {
            ExceptionHelper.ValidateStringNotNullOrWhiteSpace(filePath,
nameof(filePath));

            if (!File.Exists(filePath))
            {
                return new LoadObjectAsyncResult<T>(new T(), true);
            }

            bool success    = true;
            T obj           = null;

            // Reading from the file could fail while the file is used by another
proccess
            try
            {
                string json = File.ReadAllText(filePath, Constants.encoding);

                if (!string.IsNullOrWhiteSpace(json))
                {
                    obj = JsonConvert.DeserializeObject<T>(json);
                }
            }
            catch
            {
                success = false;
            }

            DebugHelper.OperationInfo(Path.GetFileName(filePath), "loading", suc-
cess);
            return new LoadObjectAsyncResult<T>(obj ?? new T(), success);
        }

        public sealed class LoadObjectAsyncResult<T> where T : class, new()
        {
            public T Object { get; }
            public bool Success { get; }

            public LoadObjectAsyncResult(T @object, bool success)
            {
                Object  = @object;
                Success = success;
            }
        }
    }
}
```

2.8 SnakeTheResurrection.Utilities.InputHelper

```
using System;
using System.Collections.Generic;
using System.Threading;
```

```
using System.Threading.Tasks;

namespace SnakeTheResurrection.Utilities
{
    public static class InputHelper
    {
        private static readonly List<ConsoleKey> cache = new List<ConsoleKey>();

        private static CancellationTokenSource cts;
        private static Task inputCachingTask;

        public static void StartCaching()
        {
            if (cts != null)
            {
                throw new InvalidOperationException();
            }

            ClearCache();
            cts = new CancellationTokenSource();
            inputCachingTask = Task.Factory.StartNew(() =>
            {
                while (!cts.IsCancellationRequested)
                {
                    if (Console.KeyAvailable)
                    {
                        cache.Add(Console.ReadKey(true).Key);
                    }

                    Thread.Sleep(10);
                }
            }, cts.Token);
        }

        public static void StopCaching()
        {
            if (cts == null)
            {
                throw new InvalidOperationException();
            }

            try
            {
                cts.Cancel();
                inputCachingTask.Wait();
            }
            finally
            {
                inputCachingTask.Dispose();
                inputCachingTask = null;

                cts.Dispose();
                cts = null;
            }
        }

        public static void ClearCache()
        {
            cache.Clear();
        }
```

```
        public static bool WasKeyPressed(ConsoleKey key)
        {
            if (cts != null)
            {
                throw new InvalidOperationException();
            }

            return DllImports.IsKeyDown(key) || cache.Contains(key);
        }
        public static ConsoleKeyInfo ReadKey()
        {
            return Cheats.ValidateCheat(Console.ReadKey(true));
        }

        public static void ClearInputBuffer()
        {
            while (Console.KeyAvailable)
            {
                Console.ReadKey(true);
            }
        }
    }
}
```

2.9 SnakeTheResurrection.Utilities.ListMenu

```
using System;
using System.Collections.Generic;

namespace SnakeTheResurrection.Utilities
{
    public sealed class ListMenu
    {
        private List<MenuItem> _items;
        private int _selectedIndex;

        public List<MenuItem> Items
        {
            get { return _items; }
            set
            {
                if (!ReferenceEquals(_items, value))
                {
                    ExceptionHelper.ValidateObjectNotNull(value, nameof(Items));
                    _items = value;
                }
            }
        }
        public int SelectedIndex
        {
            get { return _selectedIndex; }
            set
            {
                if (_selectedIndex != value)
                {
                    ExceptionHelper.ValidateNumberInRange(value, 0, Items.Count - 1,
nameof(SelectedIndex));
                    _selectedIndex = value;
                }
            }
```

```
        }
        public MenuItem SelectedItem
        {
            get
            {
                return Items[SelectedIndex];
            }
        }

        public ListMenu()
        {
            Items = new List<MenuItem>();
        }

        public void InvokeResult()
        {
            GetResult();
            SelectedItem.Action();
        }

        public int GetResult()
        {
            if (Items.Count < 1)
            {
                throw new InvalidOperationException("Cannot draw menu with no
items.");
            }

            int? symtextCursorTop = null;

            while (true)
            {
                lock (Symtext.SyncRoot)
                {
                    Symtext.SetCenteredTextProperties();

                    if (symtextCursorTop == null)
                    {
                        symtextCursorTop    = Symtext.CursorTop;
                    }
                    else
                    {
                        Symtext.CursorTop   = symtextCursorTop.Value;
                    }

                    for (int i = 0; i < Items.Count; i++)
                    {
                        Symtext.ForegroundColor = Constants.FOREGROUND_COLOR;
                        Symtext.BackgroundColor = i == SelectedIndex ? Constants.AC-
CENT_COLOR_DARK : Constants.BACKGROUND_COLOR;

                        Symtext.WriteLine($" {Items[i].Text} ");
                    }
                }

                Renderer.RenderFrame();

                bool handled = false;

                while (!handled)
                {
```

```
                    switch (InputHelper.ReadKey().Key)
                    {
                        case ConsoleKey.UpArrow:
                            if (SelectedIndex != 0)
                            {
                                handled = true;
                                SelectedIndex--;

                                if (string.IsNullOrWhiteSpace(SelectedItem.Text))
                                {
                                    SelectedIndex--;
                                }
                            }

                            break;

                        case ConsoleKey.DownArrow:
                            if (SelectedIndex != Items.Count - 1)
                            {
                                handled = true;
                                SelectedIndex++;

                                if (string.IsNullOrWhiteSpace(SelectedItem.Text))
                                {
                                    SelectedIndex++;
                                }
                            }

                            break;

                        case ConsoleKey.LeftArrow:
                            {
                                MenuSwitchItem selectedMenuSwitchItem = SelectedItem
as MenuSwitchItem;

                                if (selectedMenuSwitchItem != null)
                                {
                                    handled = true;
                                    selectedMenuSwitchItem.IsOn = true;
                                }

                                break;
                            }

                        case ConsoleKey.RightArrow:
                            {
                                MenuSwitchItem selectedMenuSwitchItem = SelectedItem
as MenuSwitchItem;

                                if (selectedMenuSwitchItem != null)
                                {
                                    handled = true;
                                    selectedMenuSwitchItem.IsOn = false;
                                }

                                break;
                            }

                        case ConsoleKey.Enter:
                            {
                                handled = true;
```

```
                                MenuSwitchItem selectedMenuSwitchItem = SelectedItem
as MenuSwitchItem;

                                if (selectedMenuSwitchItem != null)
                                {
                                    selectedMenuSwitchItem.IsOn = !selected-
MenuSwitchItem.IsOn;
                                    break;
                                }
                                else
                                {
                                    Renderer.ClearBuffer();
                                    return SelectedIndex;
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
```

2.10 SnakeTheResurrection.Utilities.MenuItems

```
using System;

namespace SnakeTheResurrection.Utilities
{
    public class MenuItem
    {
        public string Text { get; set; }
        public Action Action { get; set; }

        public MenuItem(string text, Action action)
        {
            Text    = text;
            Action  = action;
        }
    }

    public sealed class MenuSwitchItem : MenuItem
    {
        private readonly Func<bool> isOnGetter;
        private readonly Action<bool> isOnSetter;

        public bool IsOn
        {
            get { return isOnGetter(); }
            set { isOnSetter(value); }
        }

        public MenuSwitchItem(string text) : this(text, null, null)
        {

        }

        public MenuSwitchItem(string text, Func<bool> isOnGetter, Action<bool> isOn-
Setter) : base(text, null)
        {
            if (isOnSetter != null)
```

```
            {
                ExceptionHelper.ValidateObjectNotNull(isOnGetter, nameof(isOnGet-
ter));
            }

            bool _isOn = false;

            this.isOnGetter = isOnGetter ?? (() => _isOn);
            this.isOnSetter = isOnSetter ?? (value => _isOn = value);
        }
    }
}
```

2.11 SnakeTheResurrection.Utilities.Renderer

```
using System;
using System.Collections.Generic;
using System.Runtime.InteropServices;

namespace SnakeTheResurrection.Utilities
{
    public static class Renderer
    {
        private static readonly object syncRoot = new object();

        private static readonly int bufferHeight;
        private static readonly int bufferWidth;
        private static readonly short[] lpAttribute;

        private static Dictionary<object, ConsoleColor[,]> bufferBackups;

        public static ConsoleColor[,] Buffer { get; private set; }

        static Renderer()
        {
            lock (syncRoot)
            {
                // This is brighter, but the other one looks more retro xD
                // DllImports.SetFont("Consolas", 2, 2);
                DllImports.SetFont("Lucida Console", 1, 1);
                DllImports.DisableWindowButtons();
                DllImports.ConsoleFullscreen = true;

                // Make it a real fullscreen :D
                Console.SetWindowSize(Console.LargestWindowWidth, Console.Larg-
estWindowHeight);

                short windowHeight  = (short)Console.WindowHeight;
                short windowWidth   = (short)Console.WindowWidth;
                Console.SetBufferSize(windowWidth, windowHeight);

                DllImports.CHAR_INFO[] lpBuffer = new DllImports.CHAR_INFO[window-
Width * windowHeight];

                for (int i = 0; i < lpBuffer.Length; i++)
                {
                    // Fill the buffer with black full chars
                    lpBuffer[i].Char.AsciiChar = 219;
                }
```

```
                DllImports.SMALL_RECT lpWriteRegion = new DllImports.SMALL_RECT(0,
0, windowWidth, windowHeight);
                ExceptionHelper.ValidateMagic(WriteConsoleOutput(DllImports.StdOut-
putHandle, lpBuffer, new DllImports.COORD(windowWidth, windowHeight), new DllIm-
ports.COORD(), ref lpWriteRegion));

                Console.CursorVisible = false;

                Buffer          = new ConsoleColor[Console.WindowHeight, Con-
sole.WindowWidth];
                lpAttribute     = new short[Buffer.Length];

                bufferHeight    = Buffer.GetLength(0);
                bufferWidth     = Buffer.GetLength(1);
            }
        }

        public static void RenderFrame()
        {
            lock (syncRoot)
            {
                for (int row = 0; row < bufferHeight; row++)
                {
                    for (int column = 0; column < bufferWidth; column++)
                    {
                        lpAttribute[(row * bufferWidth) + column] =
(short)Buffer[row, column];
                    }
                }

                int lpNumberOfAttrsWritten;
                ExceptionHelper.ValidateMagic(WriteConsoleOutputAttribute(DllIm-
ports.StdOutputHandle, lpAttribute, lpAttribute.Length, new DllImports.COORD(), out
lpNumberOfAttrsWritten));
            }
        }

        public static void AddToBuffer(ConsoleColor[,] element, int x, int y)
        {
            lock (syncRoot)
            {
                int elementWidth = element.GetLength(1);

                for (int row = 0; row < element.GetLength(0); row++)
                {
                    Array.Copy(element, row * elementWidth, Buffer, ((y + row) *
bufferWidth) + x, elementWidth);
                }
            }
        }

        public static void AddToBuffer(ConsoleColor color, int x, int y, int width,
int height)
        {
            lock (syncRoot)
            {
                for (int row = y; row < y + height; row++)
                {
                    for (int column = x; column < x + width; column++)
                    {
                        Buffer[row, column] = color;
```

```
                }
            }
        }
    }

    public static void RemoveFromBuffer(int x, int y, int height, int width)
    {
        AddToBuffer(Constants.BACKGROUND_COLOR, x, y, width, height);
    }

    public static void ClearBuffer()
    {
        Array.Clear(Buffer, 0, Buffer.Length);
        // AddToBuffer(Constants.BACKGROUND_COLOR, 0, 0, bufferHeight, buffer-
Width);
    }

    public static object BackupBuffer()
    {
        lock (syncRoot)
        {
            if (bufferBackups == null)
            {
                bufferBackups = new Dictionary<object, ConsoleColor[,]>();
            }

            object key = new object();
            bufferBackups.Add(key, Buffer);
            Buffer = new ConsoleColor[bufferHeight, bufferWidth];

            return key;
        }
    }

    public static void RestoreBuffer(object key)
    {
        lock (syncRoot)
        {
            ExceptionHelper.ValidateObjectNotNull(key, nameof(key));
            ExceptionHelper.ValidateObjectNotNull(bufferBackups, null);

            Buffer = bufferBackups[key];

            if (bufferBackups.Count == 0)
            {
                bufferBackups = null;
            }
        }
    }

    [DllImport("kernel32.dll")]
    private static extern bool WriteConsoleOutputAttribute(IntPtr hConsoleOut-
put, short[] lpAttribute, int nLength, DllImports.COORD dwWriteCoord, out int lpNum-
berOfAttrsWritten);

    [DllImport("kernel32.dll", SetLastError = true)]
    private static extern bool WriteConsoleOutput(IntPtr hConsoleOutput, DllIm-
ports.CHAR_INFO[] lpBuffer, DllImports.COORD dwBufferSize, DllImports.COORD dwBuff-
erCoord, ref DllImports.SMALL_RECT lpWriteRegion);
    }
}
```

2.12 SnakeTheResurrection.Utilities.Symtext

```
using System;

namespace SnakeTheResurrection.Utilities
{
    // Font inspired by Symtext (4/26/2017): http://www.dafont.com/symtext.font
    public static class Symtext
    {
        private const bool X = true;
        private const bool _ = false;

        // All chars should be 7 rows tall
        #region Alphabet
        private static readonly bool[,] a = new bool[,]
        {
            { _, _, _, _ },
            { X, X, X, X },
            { X, _, _, X },
            { X, X, X, X },
            { X, _, _, X },
            { X, _, _, X },
            { _, _, _, _ }
        };
        private static readonly bool[,] b = new bool[,]
        {
            { _, _, _, _ },
            { X, X, X, _ },
            { X, _, X, _ },
            { X, X, X, _ },
            { X, _, _, X },
            { X, X, X, X },
            { _, _, _, _ }
        };
        private static readonly bool[,] c = new bool[,]
        {
            { _, _, _, _ },
            { X, X, X, X },
            { X, _, _, _ },
            { X, _, _, _ },
            { X, _, _, _ },
            { X, X, X, X },
            { _, _, _, _ }
        };
        private static readonly bool[,] d = new bool[,]
        {
            { _, _, _, _ },
            { X, X, X, _ },
            { X, _, _, X },
            { X, _, _, X },
            { X, _, _, X },
            { X, X, X, X },
            { _, _, _, _ }
        };
        private static readonly bool[,] e = new bool[,]
        {
            { _, _, _, _ },
            { X, X, X, X },
            { X, _, _, _ },
            { X, X, X, _ },
            { X, _, _, _ },
```

```
    { X, X, X, X },
    { _, _, _, _ }
};
private static readonly bool[,] f = new bool[,]
{
    { _, _, _, _ },
    { X, X, X, X },
    { X, _, _, _ },
    { X, X, X, _ },
    { X, _, _, _ },
    { X, _, _, _ },
    { _, _, _, _ }
};
private static readonly bool[,] g = new bool[,]
{
    { _, _, _, _ },
    { X, X, X, X },
    { X, _, _, _ },
    { X, _, _, X },
    { X, _, _, X },
    { X, X, X, X },
    { _, _, _, _ }
};
private static readonly bool[,] h = new bool[,]
{
    { _, _, _, _ },
    { X, _, _, X },
    { X, _, _, X },
    { X, X, X, X },
    { X, _, _, X },
    { X, _, _, X },
    { _, _, _, _ }
};
private static readonly bool[,] i = new bool[,]
{
    { _ },
    { X },
    { X },
    { X },
    { X },
    { X },
    { _ }
};
private static readonly bool[,] j = new bool[,]
{
    { _, _, _, _ },
    { _, _, _, X },
    { _, _, _, X },
    { _, _, _, X },
    { _, _, _, X },
    { X, X, X, X },
    { _, _, _, _ }
};
private static readonly bool[,] k = new bool[,]
{
    { _, _, _, _ },
    { X, _, X, _ },
    { X, _, X, _ },
    { X, X, X, _ },
    { X, _, _, X },
    { X, _, _, X },
```

```
    { _, _, _, _ }
};
private static readonly bool[,] l = new bool[,]
{
    { _, _, _, _ },
    { X, _, _, _ },
    { X, _, _, _ },
    { X, _, _, _ },
    { X, _, _, _ },
    { X, X, X, X },
    { _, _, _, _ }
};
private static readonly bool[,] m = new bool[,]
{
    { _, _, _, _, _ },
    { X, X, X, X, X },
    { X, _, X, _, X },
    { X, _, X, _, X },
    { X, _, _, _, X },
    { X, _, _, _, X },
    { _, _, _, _, _ }
};
private static readonly bool[,] n = new bool[,]
{
    { _, _, _, _ },
    { X, X, X, _ },
    { X, _, _, X },
    { X, _, _, X },
    { X, _, _, X },
    { X, _, _, X },
    { _, _, _, _ }
};
private static readonly bool[,] o = new bool[,]
{
    { _, _, _, _ },
    { X, X, X, X },
    { X, _, _, X },
    { X, _, _, X },
    { X, _, _, X },
    { X, X, X, X },
    { _, _, _, _ }
};
private static readonly bool[,] p = new bool[,]
{
    { _, _, _, _ },
    { X, X, X, X },
    { X, _, _, X },
    { X, X, X, X },
    { X, _, _, _ },
    { X, _, _, _ },
    { _, _, _, _ }
};
private static readonly bool[,] q = new bool[,]
{
    { _, _, _, _ },
    { X, X, X, X },
    { X, _, _, X },
    { X, _, _, X },
    { X, _, X, X },
    { X, X, X, X },
    { _, _, _, _ }
```

```
};
private static readonly bool[,] r = new bool[,]
{
    { _, _, _, _ },
    { X, X, X, _ },
    { X, _, X, _ },
    { X, X, X, X },
    { X, _, _, X },
    { X, _, _, X },
    { _, _, _, _ }
};
private static readonly bool[,] s = new bool[,]
{
    { _, _, _, _ },
    { X, X, X, X },
    { X, _, _, _ },
    { X, X, X, X },
    { _, _, _, X },
    { X, X, X, X },
    { _, _, _, _ }
};
private static readonly bool[,] t = new bool[,]
{
    { _, _, _, _, _ },
    { X, X, X, X, X },
    { _, _, X, _, _ },
    { _, _, X, _, _ },
    { _, _, X, _, _ },
    { _, _, X, _, _ },
    { _, _, _, _, _ }
};
private static readonly bool[,] u = new bool[,]
{
    { _, _, _, _ },
    { X, _, _, X },
    { X, _, _, X },
    { X, _, _, X },
    { X, _, _, X },
    { X, X, X, X },
    { _, _, _, _ }
};
private static readonly bool[,] v = new bool[,]
{
    { _, _, _, _ },
    { X, _, _, X },
    { X, _, _, X },
    { X, _, _, X },
    { X, _, X, _ },
    { _, X, _, _ },
    { _, _, _, _ }
};
private static readonly bool[,] w = new bool[,]
{
    { _, _, _, _, _ },
    { X, _, _, _, X },
    { X, _, _, _, X },
    { X, _, X, _, X },
    { X, _, X, _, X },
    { X, X, X, X, X },
    { _, _, _, _, _ }
};
```

```
private static readonly bool[,] x = new bool[,]
{
    { _, _, _, _ },
    { X, _, _, X },
    { X, _, _, X },
    { _, X, X, _ },
    { X, _, _, X },
    { X, _, _, X },
    { _, _, _, _ }
};
private static readonly bool[,] y = new bool[,]
{
    { _, _, _, _ },
    { X, _, _, X },
    { X, _, _, X },
    { X, X, X, X },
    { _, _, _, X },
    { X, X, X, X },
    { _, _, _, _ }
};
private static readonly bool[,] z = new bool[,]
{
    { _, _, _, _ },
    { X, X, X, X },
    { _, _, _, X },
    { _, X, X, _ },
    { X, _, _, _ },
    { X, X, X, X },
    { _, _, _, _ }
};

private static readonly bool[,] ý = new bool[,]
{
    { _, _, X, _ },
    { X, _, X, X },
    { X, _, _, X },
    { X, X, X, X },
    { _, _, _, X },
    { X, X, X, X },
    { _, _, _, _ }
};
#endregion
#region Numbers
private static readonly bool[,] _0 = new bool[,]
{
    { _, _, _, _, _ },
    { X, X, X, X, X },
    { X, _, _, _, X },
    { X, _, X, _, X },
    { X, _, _, _, X },
    { X, X, X, X, X },
    { _, _, _, _, _ }
};
private static readonly bool[,] _1 = new bool[,]
{
    { _, _ },
    { X, X },
    { _, X },
    { _, X },
    { _, X },
    { _, X },
```

```
    { _, _ }
};
private static readonly bool[,] _2 = new bool[,]
{
    { _, _, _, _ },
    { X, X, X, X },
    { _, _, _, X },
    { X, X, X, X },
    { X, _, _, _ },
    { X, X, X, X },
    { _, _, _, _ }
};
private static readonly bool[,] _3 = new bool[,]
{
    { _, _, _, _ },
    { X, X, X, X },
    { _, _, _, X },
    { _, _, X, X },
    { _, _, _, X },
    { X, X, X, X },
    { _, _, _, _ }
};
private static readonly bool[,] _4 = new bool[,]
{
    { _, _, _, _ },
    { X, _, _, _ },
    { X, _, _, X },
    { X, X, X, X },
    { _, _, _, X },
    { _, _, _, X },
    { _, _, _, _ }
};
private static readonly bool[,] _5 = new bool[,]
{
    { _, _, _, _ },
    { X, X, X, X },
    { X, _, _, _ },
    { X, X, X, X },
    { _, _, _, X },
    { X, X, X, X },
    { _, _, _, _ }
};
private static readonly bool[,] _6 = new bool[,]
{
    { _, _, _, _ },
    { X, X, X, X },
    { X, _, _, _ },
    { X, X, X, X },
    { X, _, _, X },
    { X, X, X, X },
    { _, _, _, _ }
};
private static readonly bool[,] _7 = new bool[,]
{
    { _, _, _, _ },
    { X, X, X, X },
    { _, _, _, X },
    { _, _, _, X },
    { _, _, _, X },
    { _, _, _, X },
    { _, _, _, _ }
```

```
};
private static readonly bool[,] _8 = new bool[,]
{
    { _, _, _, _ },
    { X, X, X, X },
    { X, _, _, X },
    { X, X, X, X },
    { X, _, _, X },
    { X, X, X, X },
    { _, _, _, _ }
};
private static readonly bool[,] _9 = new bool[,]
{
    { _, _, _, _ },
    { X, X, X, X },
    { X, _, _, X },
    { X, X, X, X },
    { _, _, _, X },
    { _, _, _, X },
    { _, _, _, _ }
};
#endregion
#region Special characters
private static readonly bool[,] space = new bool[,]
{
    { _, _, _ },
    { _, _, _ },
    { _, _, _ },
    { _, _, _ },
    { _, _, _ },
    { _, _, _ },
    { _, _, _ }
};
private static readonly bool[,] plus = new bool[,]
{
    { _, _, _ },
    { _, _, _ },
    { _, X, _ },
    { X, X, X },
    { _, X, _ },
    { _, _, _ },
    { _, _, _ }
};
private static readonly bool[,] minus = new bool[,]
{
    { _, _, _ },
    { _, _, _ },
    { _, _, _ },
    { X, X, X },
    { _, _, _ },
    { _, _, _ },
    { _, _, _ }
};
private static readonly bool[,] cross = new bool[,]
{
    { _, _, _ },
    { _, _, _ },
    { X, _, X },
    { _, X, _ },
    { X, _, X },
    { _, _, _ },
```

```
    { _, _, _ }
};
private static readonly bool[,] slash = new bool[,]
{
    { _, _, _ },
    { _, _, X },
    { _, _, X },
    { _, X, _ },
    { X, _, _ },
    { X, _, _ },
    { _, _, _ }
};
private static readonly bool[,] equals = new bool[,]
{
    { _, _, _ },
    { _, _, _ },
    { X, X, X },
    { _, _, _ },
    { X, X, X },
    { _, _, _ },
    { _, _, _ }
};
private static readonly bool[,] percents = new bool[,]
{
    { _, _, _ },
    { X, _, X },
    { _, _, X },
    { _, X, _ },
    { X, _, _ },
    { X, _, X },
    { _, _, _ }
};
private static readonly bool[,] quotationMark = new bool[,]
{
    { X, X },
    { X, X },
    { _, _ },
    { _, _ },
    { _, _ },
    { _, _ },
    { _, _ }
};
private static readonly bool[,] apostrophe = new bool[,]
{
    { X },
    { X },
    { _ },
    { _ },
    { _ },
    { _ },
    { _ }
};
private static readonly bool[,] hash = new bool[,]
{
    { _, _, _, _, _ },
    { _, X, _, X, _ },
    { X, X, X, X, X },
    { _, X, _, X, _ },
    { X, X, X, X, X },
    { _, X, _, X, _ },
    { _, _, _, _, _ }
```

```
};
private static readonly bool[,] comma = new bool[,]
{
    { _ },
    { _ },
    { _ },
    { _ },
    { _ },
    { X },
    { X }
};
private static readonly bool[,] dot = new bool[,]
{
    { _ },
    { _ },
    { _ },
    { _ },
    { _ },
    { X },
    { _ }
};
private static readonly bool[,] colon = new bool[,]
{
    { _ },
    { _ },
    { _ },
    { X },
    { _ },
    { X },
    { _ }
};
private static readonly bool[,] questionMark = new bool[,]
{
    { _, _, _, _ },
    { X, X, X, X },
    { _, _, _, X },
    { _, X, X, X },
    { _, _, _, _ },
    { _, X, _, _ },
    { _, _, _, _ }
};
private static readonly bool[,] exclamationMark = new bool[,]
{
    { _ },
    { X },
    { X },
    { X },
    { _ },
    { X },
    { _ }
};
private static readonly bool[,] arrowLeft = new bool[,]
{
    { _, _, _ },
    { _, _, X },
    { _, X, _ },
    { X, _, _ },
    { _, X, _ },
    { _, _, X },
    { _, _, _ }
};
```

```
private static readonly bool[,] arrowRight = new bool[,]
{
    { _, _, _ },
    { X, _, _ },
    { _, X, _ },
    { _, _, X },
    { _, X, _ },
    { X, _, _ },
    { _, _, _ }
};
private static readonly bool[,] squareBracketLeft = new bool[,]
{
    { _, _ },
    { X, X },
    { X, _ },
    { X, _ },
    { X, _ },
    { X, X },
    { _, _ }
};
private static readonly bool[,] squareBracketRight = new bool[,]
{
    { _, _ },
    { X, X },
    { _, X },
    { _, X },
    { _, X },
    { X, X },
    { _, _ }
};
private static readonly bool[,] copyrightMark = new bool[,]
{
    { _, X, X, X, X, _ },
    { X, _, _, _, _, X },
    { X, _, X, X, _, X },
    { X, _, X, _, _, X },
    { X, _, X, X, _, X },
    { X, _, _, _, _, X },
    { _, X, X, X, X, _ }
};
#endregion

private static ConsoleColor[,] characterSpacingBackgroundFiller;

private static int _cursorLeft;
private static int _cursorTop;
private static int _fontSize;
private static SymtextScalingStyle _scalingStyle;
private static ConsoleColor _foregroundColor;
private static ConsoleColor _backgroundColor;
private static HorizontalAlignment _horizontalAlignment;
private static VerticalAlignment _verticalAlignment;

private static int CharacterSpacing
{
    get
    {
        return FontSize;
    }
}
```

```
        public static object SyncRoot { get; }
        public static int CursorLeft
        {
            get { return _cursorLeft; }
            set
            {
                lock (SyncRoot)
                {
                    if (_cursorLeft != value)
                    {
                        ExceptionHelper.ValidateNumberInRange(value, 0, Console.Win-
dowWidth, nameof(CursorLeft));
                        _cursorLeft = value;
                    }
                }
            }
        }
        public static int CursorTop
        {
            get { return _cursorTop; }
            set
            {
                lock (SyncRoot)
                {
                    if (_cursorTop != value)
                    {
                        ExceptionHelper.ValidateNumberInRange(value, 0, Console.Win-
dowHeight, nameof(CursorTop));
                        _cursorTop = value;
                    }
                }
            }
        }
        public static int FontSize
        {
            get { return _fontSize; }
            set
            {
                lock (SyncRoot)
                {
                    if (_fontSize != value)
                    {
                        ExceptionHelper.ValidateNumberGreaterOrEqual(value, 0,
nameof(FontSize));
                        _fontSize = value;

                        characterSpacingBackgroundFiller = new ConsoleColor[Char-
Height, value];
                        FillCharacterSpacingBackgroundFiller();
                    }
                }
            }
        }
        public static SymtextScalingStyle ScalingStyle
        {
            get { return _scalingStyle; }
            set
            {
                lock (SyncRoot)
                {
                    if (_scalingStyle != value)
```

```
                    {
                        ExceptionHelper.ValidateEnumValueDefined(value, nameof(Scal-
ingStyle));
                        _scalingStyle = value;
                    }
                }
            }
        }
        public static ConsoleColor ForegroundColor
        {
            get { return _foregroundColor; }
            set
            {
                lock (SyncRoot)
                {
                    if (_foregroundColor != value)
                    {
                        ExceptionHelper.ValidateEnumValueDefined(value, nameof(Fore-
groundColor));
                        _foregroundColor = value;
                    }
                }
            }
        }
        public static ConsoleColor BackgroundColor
        {
            get { return _backgroundColor; }
            set
            {
                lock (SyncRoot)
                {
                    if (_backgroundColor != value)
                    {
                        ExceptionHelper.ValidateEnumValueDefined(value, nameof(Back-
groundColor));
                        _backgroundColor = value;

                        FillCharacterSpacingBackgroundFiller();
                    }
                }
            }
        }
        public static HorizontalAlignment HorizontalAlignment
        {
            get { return _horizontalAlignment; }
            set
            {
                lock (SyncRoot)
                {
                    if (_horizontalAlignment != value)
                    {
                        ExceptionHelper.ValidateEnumValueDefined(value, nameof(Hori-
zontalAlignment));
                        _horizontalAlignment = value;
                    }
                }
            }
        }
        public static VerticalAlignment VerticalAlignment
        {
            get { return _verticalAlignment; }
```

```
            set
            {
                lock (SyncRoot)
                {
                    if (_verticalAlignment != value)
                    {
                        ExceptionHelper.ValidateEnumValueDefined(value, nameof(Ver-
ticalAlignment));
                        _verticalAlignment = value;
                    }
                }
            }
        }
        public static int CharHeight
        {
            get
            {
                return 7 * FontSize;
            }
        }

        static Symtext()
        {
            SyncRoot            = new object();
            Reset();
        }

        private static void FillCharacterSpacingBackgroundFiller()
        {
            for (int row = 0; row < characterSpacingBackgroundFiller.GetLength(0);
row++)
            {
                for (int column = 0; column < characterSpacingBackground-
Filler.GetLength(1); column++)
                {
                    characterSpacingBackgroundFiller[row, column] = BackgroundColor;
                }
            }
        }

        public static void SetCursorPosition(int left, int top)
        {
            CursorLeft   = left;
            CursorTop    = top;
        }

        public static void Reset()
        {
            lock (SyncRoot)
            {
                CursorLeft          = 0;
                CursorTop           = 0;
                FontSize            = 1;
                ScalingStyle        = default(SymtextScalingStyle);
                ForegroundColor     = Constants.FOREGROUND_COLOR;
                BackgroundColor     = Constants.BACKGROUND_COLOR;
                HorizontalAlignment = default(HorizontalAlignment);
                VerticalAlignment   = default(VerticalAlignment);
            }
        }
```

```
        public static void Write(object value)
        {
            Write(value, 0);
        }

        public static void Write(object value, int verticalOffset)
        {
            lock (SyncRoot)
            {
                string[] lines = value.ToString().Split('\n');

                switch (VerticalAlignment)
                {
                    case VerticalAlignment.Top:     CursorTop = 0;
break;
                    case VerticalAlignment.Center:  CursorTop = (Console.Win-
dowHeight - (lines.Length * CharHeight)) / 2;     break;
                    case VerticalAlignment.Bottom:  CursorTop = Console.WindowHeight
- (lines.Length * CharHeight);           break;
                }

                CursorTop += verticalOffset;

                for (int i = 0; i < lines.Length; i++)
                {
                    string line = lines[i];

                    switch (HorizontalAlignment)
                    {
                        case HorizontalAlignment.Left:      CursorLeft = 0;
break;
                        case HorizontalAlignment.Center:    CursorLeft = (Con-
sole.WindowWidth - GetSymtextWidth(line)) / 2;    break;
                        case HorizontalAlignment.Right:     CursorLeft = Con-
sole.WindowWidth - GetSymtextWidth(line);          break;
                    }

                    for (int j = 0; j < line.Length; j++)
                    {
                        // Is an escape character probably
                        if (line[j] == '\\')
                        {
                            continue;
                        }

                        CursorLeft += AddRenderedCharToBuffer(line[j], CursorLeft,
CursorTop);

                        if (j != line.Length - 1)
                        {
                            Renderer.AddToBuffer(characterSpacingBackgroundFiller,
CursorLeft, CursorTop);
                            CursorLeft += CharacterSpacing;
                        }
                    }

                    if (i != lines.Length - 1)
                    {
                        CursorTop += CharHeight;
                    }
                }
```

```
    }
}

public static void WriteLine()
{
    Write('\n');
}

public static void WriteLine(object value)
{
    Write(value.ToString() + '\n');
}

public static void WriteLine(object value, int verticalOffset)
{
    Write(value.ToString() + '\n', verticalOffset);
}

public static void WriteTitle(object value, int verticalOffset)
{
    ForegroundColor     = Constants.ACCENT_COLOR;
    BackgroundColor     = Constants.BACKGROUND_COLOR;
    FontSize            = 15;
    HorizontalAlignment = HorizontalAlignment.Center;
    VerticalAlignment   = VerticalAlignment.Center;
    WriteLine(value, verticalOffset);

    HorizontalAlignment = HorizontalAlignment.None;
    VerticalAlignment   = VerticalAlignment.None;

    FontSize = 3;
    WriteLine();
}

public static void SetTextProperties()
{
    ForegroundColor     = Constants.FOREGROUND_COLOR;
    BackgroundColor     = Constants.BACKGROUND_COLOR;
    FontSize            = 2;
    HorizontalAlignment = HorizontalAlignment.None;
    VerticalAlignment   = VerticalAlignment.None;
}

public static void SetCenteredTextProperties()
{
    SetTextProperties();
    HorizontalAlignment = HorizontalAlignment.Center;
}

public static int GetSymtextWidth(string str)
{
    int output = 0;

    foreach (char ch in str)
    {
        output += GetScaledBoolChar(ch).GetLength(1) + CharacterSpacing;
    }

    // We are not adding the character spacing behind the word
    return output - CharacterSpacing;
}
```

```
        private static int AddRenderedCharToBuffer(char ch, int x, int y)
        {
            bool[,] character               = GetScaledBoolChar(ch);

            int characterHeight             = character.GetLength(0);
            int characterWidth              = character.GetLength(1);
            ConsoleColor[,] renderedChar    = new ConsoleColor[characterHeight,
characterWidth];

            for (int row = 0; row < characterHeight; row++)
            {
                for (int column = 0; column < characterWidth; column++)
                {
                    renderedChar[row, column] = character[row, column] ? Foreground-
Color : BackgroundColor;
                }
            }

            Renderer.AddToBuffer(renderedChar, x, y);
            return characterWidth;
        }

        private static bool[,] GetScaledBoolChar(char ch)
        {
            bool[,] original    = GetBoolChar(ch);

            int originalHeight  = original.GetLength(0);
            int originalWidth   = original.GetLength(1);
            bool[,] output      = new bool[originalHeight * FontSize, originalWidth
* FontSize];

            if (ScalingStyle == SymtextScalingStyle.Normal)
            {
                for (int row = 0; row < originalHeight; row++)
                {
                    for (int column = 0; column < originalWidth; column++)
                    {
                        bool currentValue = original[row, column];

                        for (int row2 = 0; row2 < FontSize; row2++)
                        {
                            for (int column2 = 0; column2 < FontSize; column2++)
                            {
                                output[(row * FontSize) + row2, (column * FontSize)
+ column2] = currentValue;
                            }
                        }
                    }
                }
            }
            else
            {
                for (int row = 0; row < originalHeight; row++)
                {
                    for (int column = 0; column < originalWidth; column++)
                    {
                        bool currentValue = original[row, column];

                        for (int difference = 0; difference < FontSize; differ-
ence++)
```

```
                        {
                            output[(row * FontSize) + difference, (column * Font-
Size) + difference] = currentValue;
                        }
                    }
                }
            }

            return output;
        }

        private static bool[,] GetBoolChar(char ch)
        {
            switch (char.ToLower(ch))
            {
                case 'a': return a;
                case 'b': return b;
                case 'c': return c;
                case 'd': return d;
                case 'e': return e;
                case 'f': return f;
                case 'g': return g;
                case 'h': return h;
                case 'i': return i;
                case 'j': return j;
                case 'k': return k;
                case 'l': return l;
                case 'm': return m;
                case 'n': return n;
                case 'o': return o;
                case 'p': return p;
                case 'q': return q;
                case 'r': return r;
                case 's': return s;
                case 't': return t;
                case 'u': return u;
                case 'v': return v;
                case 'w': return w;
                case 'x': return x;
                case 'y': return y;
                case 'z': return z;

                case 'ý': return ý;

                case ' ': return space;

                case '0': return _0;
                case '1': return _1;
                case '2': return _2;
                case '3': return _3;
                case '4': return _4;
                case '5': return _5;
                case '6': return _6;
                case '7': return _7;
                case '8': return _8;
                case '9': return _9;

                case '+': return plus;
                case '-': return minus;
                case '*': return cross;
                case '/': return slash;
```

```
                case '=': return equals;
                case '%': return percents;
                case '"': return quotationMark;
                case '\'': return apostrophe;
                case '#': return hash;
                case ',': return comma;
                case '.': return dot;
                case ':': return colon;
                case '?': return questionMark;
                case '!': return exclamationMark;
                case '<': return arrowLeft;
                case '>': return arrowRight;
                case '[': return squareBracketLeft;
                case ']': return squareBracketRight;
                case '©': return copyrightMark;

                default: ExceptionHelper.ThrowMagicException(); return null;
            }
        }
    }

    public enum SymtextScalingStyle
    {
        Normal,
        Stripped
    }

    public enum HorizontalAlignment
    {
        None,
        Left,
        Center,
        Right
    }

    public enum VerticalAlignment
    {
        None,
        Top,
        Center,
        Bottom
    }
}
```

2.13 SnakeTheResurrection.Constansts

```
using System;
using System.Text;

namespace SnakeTheResurrection
{
    public static class Constants
    {
        public const string APP_SHORT_NAME          = "Snake";
        public const string APP_NAME_ADDITION       = "The Resurrection";
        public const string APP_NAME                = APP_SHORT_NAME + " " +
APP_NAME_ADDITION;
        public const ConsoleColor ACCENT_COLOR      = ConsoleColor.Green;
        public const ConsoleColor ACCENT_COLOR_DARK = ConsoleColor.DarkGreen;
        public const ConsoleColor FOREGROUND_COLOR  = ConsoleColor.White;
```

```
        public const ConsoleColor BACKGROUND_COLOR  = ConsoleColor.Black;

        public static readonly Encoding encoding    = Encoding.UTF8;
    }
}
```

2.14 SnakeTheResurrection.Game

```
using SnakeTheResurrection.Data;
using SnakeTheResurrection.Utilities;
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Threading;

namespace SnakeTheResurrection
{
    public static class Game
    {
        private const int BLOCK_SIZE = 5;

        private static int gameBoardLeft;
        private static int gameBoardTop;
        private static int gameBoardRight;
        private static int gameBoardBottom;
        private static int gameBoardWidth;
        private static int gameBoardHeight;

        private static bool BorderlessMode { get; set; }

        public static bool Play(bool multiplayer)
        {
            Renderer.ClearBuffer();
            int delay       = 0;
            int playerCount = 1;

            for (int i = 0; ; i++)
            {
                // Not using switch to be able to use continue and break
                if (i == 0)
                {
                    bool? getGameModeOutput = GetGameMode();

                    if (getGameModeOutput == null)
                    {
                        return false;
                    }
                    else
                    {
                        BorderlessMode = getGameModeOutput.Value;
                    }
                }
                else if (i == 1)
                {
                    int? getDelayOutput = GetDelay();

                    if (getDelayOutput == null)
                    {
                        i -= 2;
                        continue;
```

```
                    }
                    else
                    {
                        delay = getDelayOutput.Value;
                    }
                }
                else if (i == 2)
                {
                    if (multiplayer)
                    {
                        int? getPlayerCountOutput = GetPlayerCount();

                        if (getPlayerCountOutput == null)
                        {
                            i -= 2;
                            continue;
                        }

                        playerCount = getPlayerCountOutput.Value;
                    }
                }
                else
                {
                    break;
                }
            }

            // Using try-finally to execute things even after 'return'
            try
            {
                Renderer.ClearBuffer();
                CreateGameBoard();

                for (int i = 0; i < playerCount; i++)
                {
                    //TODO: Load real profiles here
                    switch (i)
                    {
                        case 0:
                            new Snake(ProfileManager.CurrentProfile, i, player-
Count);
                            break;

                        case 1:
                            {
                                Snake snake = new Snake(new Profile
                                {
                                    Name    = "Frogpanda",
                                    Color   = ConsoleColor.Cyan
                                }, i, playerCount);
                                snake.Profile.SnakeControls.Left    = ConsoleKey.A;
                                snake.Profile.SnakeControls.Up      = ConsoleKey.W;
                                snake.Profile.SnakeControls.Right   = ConsoleKey.D;
                                snake.Profile.SnakeControls.Down    = ConsoleKey.S;

                                break;
                            }

                        case 2:
                            {
                                Snake snake = new Snake(new Profile
```

```
                                {
                                    Name    = "Strawberryraspberry",
                                    Color   = ConsoleColor.Magenta
                                }, i, playerCount);
                                snake.Profile.SnakeControls.Left    = Console-
Key.NumPad4;
                                snake.Profile.SnakeControls.Up      = Console-
Key.NumPad8;
                                snake.Profile.SnakeControls.Right   = Console-
Key.NumPad6;
                                snake.Profile.SnakeControls.Down    = Console-
Key.NumPad5;

                                break;
                            }

                        case 3:
                            {
                                Snake snake = new Snake(new Profile
                                {
                                    Name    = "Lifeescape",
                                    Color   = ConsoleColor.Yellow
                                }, i, playerCount);
                                snake.Profile.SnakeControls.Left    = ConsoleKey.J;
                                snake.Profile.SnakeControls.Up      = ConsoleKey.I;
                                snake.Profile.SnakeControls.Right   = ConsoleKey.L;
                                snake.Profile.SnakeControls.Down    = ConsoleKey.K;

                                break;
                            }
                    }

                    new Berry(10);
                }

                while (Snake.Current.Any(s => s.IsAlive))
                {
                    foreach (Snake snake in Snake.Current)
                    {
                        snake.Update();
                    }

                    foreach (Snake snake in Snake.Current)
                    {
                        snake.LateUpdate();
                    }

                    Renderer.RenderFrame();

                    if (InputHelper.WasKeyPressed(ConsoleKey.Escape))
                    {
                        switch (PauseMenu())
                        {
                            case MenuResult.Restart:
                                return true;

                            case MenuResult.MainMenu:
                                return false;

                            case MenuResult.QuitGame:
                                Program.Exit();
```

```
                                break;
                            }
                        }
#if DEBUG
                        else if (InputHelper.WasKeyPressed(ConsoleKey.B))
                        {
                            while (Console.ReadKey(true).Key != ConsoleKey.B) ;
                        }
#endif

                        InputHelper.StartCaching();
                        Thread.Sleep(delay);
                        InputHelper.StopCaching();
                    }

                    return false;
                }
                finally
                {
                    InputHelper.ClearCache();
                    Berry.Reset();
                    Snake.Reset();
                }
            }

            private static void CreateGameBoard()
            {
                int windowWidthOverlap  = Console.WindowWidth % BLOCK_SIZE;
                int windowHeightOverlap = Console.WindowHeight % BLOCK_SIZE;

                if (windowWidthOverlap >= 1 || windowHeightOverlap >= 1 || AppData.Cur-
rent.ForceGameBoardBorders)
                {
                    windowWidthOverlap  += BLOCK_SIZE * 2;
                    windowHeightOverlap += BLOCK_SIZE * 2;
                }

                gameBoardLeft                   = (int)Math.Round(windowWidthOverlap /
2.0);
                gameBoardTop                    = (int)Math.Round(windowHeightOverlap /
2.0);

                int gameBoardBorderRightSize    = windowWidthOverlap - gameBoardLeft;
                int gameBoardBorderBottomSize   = windowHeightOverlap - gameBoardTop;

                gameBoardRight                  = Console.WindowWidth - gameBoardBorder-
RightSize;
                gameBoardBottom                 = Console.WindowHeight - gameBoardBor-
derBottomSize;

                //TODO: Status bar
                // gameBoardTop                    += gameBoardTop >= 1 ? BLOCK_SIZE :
(BLOCK_SIZE * 2);

                Renderer.AddToBuffer(Constants.ACCENT_COLOR_DARK, 0, 0, gameBoardLeft,
Console.WindowHeight);
                Renderer.AddToBuffer(Constants.ACCENT_COLOR_DARK, gameBoardRight, 0,
gameBoardBorderRightSize, Console.WindowHeight);

                Renderer.AddToBuffer(Constants.ACCENT_COLOR_DARK, 0, 0, Console.Window-
Width, gameBoardTop);
```

```
            Renderer.AddToBuffer(Constants.ACCENT_COLOR_DARK, 0, gameBoardBottom,
Console.WindowWidth, gameBoardBorderBottomSize);

            gameBoardWidth  = gameBoardRight - gameBoardLeft;
            gameBoardHeight = gameBoardBottom - gameBoardTop;
        }

        private static bool? GetGameMode()
        {
            bool? output = null;

            Symtext.WriteTitle("Mode", 7);
            new ListMenu
            {
                Items = new List<MenuItem>
                {
                    new MenuItem("Classic",     () => output = false    ),
                    new MenuItem("Borderless",  () => output = true     ),
                    new MenuItem(null,          null                    ),
                    new MenuItem("Back",        () => output = null     )
                }
            }.InvokeResult();

            return output;
        }

        private static int? GetDelay()
        {
            int? output = null;

            Symtext.WriteTitle("Level", 0);
            new ListMenu
            {
                Items = new List<MenuItem>
                {
                    new MenuItem("Easy",    () => output = 200  ),
                    new MenuItem("Medium",  () => output = 50   ),
                    new MenuItem("Hard",    () => output = 30   ),
                    new MenuItem(null,      null                ),
                    new MenuItem("Back",    () => output = null )
                },
                SelectedIndex = 1
            }.InvokeResult();

            return output;
        }

        private static int? GetPlayerCount()
        {
            //TODO: Selection UI
            return 2;
        }

        private static MenuResult PauseMenu()
        {
            object gameBufferKey    = Renderer.BackupBuffer();
            MenuResult output       = default(MenuResult);

            Symtext.WriteTitle("Pause", 7);
            new ListMenu
            {
```

```
                Items = new List<MenuItem>
                {
                    new MenuItem("Continue",    () => output = MenuResult.Continue
),
                    new MenuItem("Restart",     () => output = MenuResult.Restart
),
                    new MenuItem("Main menu",   () => output = MenuResult.MainMenu
),
                    new MenuItem("Quit game",   () => output = MenuResult.QuitGame
)
                }
            }.InvokeResult();

            Renderer.RestoreBuffer(gameBufferKey);
            return output;
        }

        private enum MenuResult
        {
            Continue,
            Restart,
            MainMenu,
            QuitGame
        }

        private abstract class GameObjectBase
        {
            private int _x;
            private int _y;

            public int X
            {
                get { return _x; }
                protected set
                {
                    if (_x != value)
                    {
                        ExceptionHelper.ValidateNumberInRange(value, gameBoardLeft,
gameBoardRight - Size, nameof(X));
                        _x = value;
                    }
                }
            }
            public int Y
            {
                get { return _y; }
                protected set
                {
                    if (_y != value)
                    {
                        ExceptionHelper.ValidateNumberInRange(value, gameBoardTop,
gameBoardBottom - Size, nameof(Y));
                        _y = value;
                    }
                }
            }
            public abstract int Size { get; }

            public bool HitTest(GameObjectBase g)
            {
```

```
                return X <= g.X + g.Size - 1 && X + Size - 1 >= g.X && Y <= g.Y +
g.Size - 1 && Y + Size - 1 >= g.Y;
            }

            protected bool IsInGameBoard(int newX, int newY)
            {
                return newX >= gameBoardLeft && newY >= gameBoardTop && newX + Size
<= gameBoardRight && newY + Size <= gameBoardBottom;
            }

            public void AlignPosition()
            {
                int alignment = (BLOCK_SIZE % Size) / 2;
                X = X - (X % BLOCK_SIZE) + (gameBoardLeft % BLOCK_SIZE) + alignment;
                Y = Y - (Y % BLOCK_SIZE) + (gameBoardTop % BLOCK_SIZE) + alignment;
            }
        }

        private sealed class Snake : SnakeBody, IEnumerable<SnakeBody>
        {
            private static readonly List<Snake> _current = new List<Snake>();

            public static IEnumerable<Snake> Current
            {
                get
                {
                    foreach (Snake snake in _current.ToList())
                    {
                        if (snake.IsAlive)
                        {
                            yield return snake;
                        }
                        else
                        {
                            _current.Remove(snake);
                        }
                    }
                }
            }

            private SnakeBody tail;
            private int desiredLength = 3;

            private bool _isAlive;

            public bool IsAlive
            {
                get { return _isAlive; }
                set
                {
                    if (!_isAlive)
                    {
                        if (value)
                        {
                            throw new InvalidOperationException("Cannot revive a
dead snake.");
                        }
                        else
                        {
                            throw new InvalidOperationException("Cannot kill a dead
snake.");
```

```
                    }
                }

                _isAlive = value;
            }
        }
        public int Length { get; private set; }

        public Snake(Profile profile, int index, int totalSnakeCount) :
base(GetX(index, totalSnakeCount), gameBoardTop + (gameBoardHeight / 2) -
BLOCK_SIZE, Direction.Up, profile, null)
        {
            _isAlive = true;
            _current.Add(this);
        }

        public void Update()
        {
            if (Length < desiredLength)
            {
                if (tail == null)
                {
                    AlignPosition();
                    tail = this;
                }
                else
                {
                    int newX = tail.X;
                    int newY = tail.Y;

                    Direction inverseDirection = tail.Direction;

                    switch (tail.Direction)
                    {
                        case Direction.Left:
                            inverseDirection = Direction.Right;
                            break;

                        case Direction.UpLeft:
                            inverseDirection = Direction.DownRight;
                            break;

                        case Direction.Up:
                            inverseDirection = Direction.Down;
                            break;

                        case Direction.UpRight:
                            inverseDirection = Direction.DownLeft;
                            break;

                        case Direction.Right:
                            inverseDirection = Direction.Left;
                            break;

                        case Direction.DownRight:
                            inverseDirection = Direction.UpLeft;
                            break;

                        case Direction.Down:
                            inverseDirection = Direction.Up;
                            break;
```

```
                            case Direction.DownLeft:
                                inverseDirection = Direction.UpRight;
                                break;
                        }

                        UpdateCoordinates(inverseDirection, ref newX, ref newY);

                        tail.NextBody    = new SnakeBody(newX, newY, tail.Direction,
Profile, this);
                        tail             = tail.NextBody;
                    }

                    Length++;
                }

                Update(null);

                Berry berry = Berry.Current.FirstOrDefault(b => HitTest(b));

                if (berry != null)
                {
                    desiredLength += berry.Eat();
                }
            }

            public IEnumerator<SnakeBody> GetEnumerator()
            {
                SnakeBody body = this;

                while (body != null)
                {
                    yield return body;
                    body = body.NextBody;
                }
            }

            IEnumerator IEnumerable.GetEnumerator()
            {
                return GetEnumerator();
            }

            public static void Reset()
            {
                _current.Clear();
            }

            private static int GetX(int index, int totalSnakeCount)
            {
                return gameBoardLeft + ((gameBoardWidth / (totalSnakeCount + 1)) *
(index + 1)) - BLOCK_SIZE;
            }
        }

        private class SnakeBody : GameObjectBase
        {
            private const int SIZE = BLOCK_SIZE;

            private readonly List<BendInfo> bendInfo;
            private readonly Snake snake;
```

```
            private bool isNew = true;

            private Direction _direction;

            private bool IsHead
            {
                get { return this is Snake; }
            }

            public override int Size
            {
                get { return SIZE; }
            }
            public Direction Direction
            {
                get { return _direction; }
                private set
                {
                    if (_direction != value)
                    {
                        ExceptionHelper.ValidateEnumValueDefined(value, nameof(Di-
rection));
                        _direction = value;
                    }
                }
            }
            public Profile Profile { get; }
            public SnakeBody NextBody { get; set; }

            public SnakeBody(int x, int y, Direction direction, Profile profile,
Snake snake)
            {
                ExceptionHelper.ValidateObjectNotNull(profile, nameof(profile));

                this.snake  = snake ?? (Snake)this;
                X           = x;
                Y           = y;
                Direction   = direction;
                Profile     = profile;

                if (!IsHead)
                {
                    bendInfo = new List<BendInfo>();
                }
            }

            protected void Update(BendInfo newBendInfo)
            {
                Renderer.RemoveFromBuffer(X, Y, Size, Size);
                bool removeFirstBendInfo = false;

                if (IsHead)
                {
                    Direction originalDirection = Direction;

                    bool up     = InputHelper.WasKeyPressed(Profile.SnakeCon-
trols.Up);
                    bool down   = InputHelper.WasKeyPressed(Profile.SnakeCon-
trols.Down);
                    bool left   = InputHelper.WasKeyPressed(Profile.SnakeCon-
trols.Left);
```

```
                    bool right  = InputHelper.WasKeyPressed(Profile.SnakeCon-
trols.Right);

                    if (up)
                    {
                        bool assigned = false;

                        if (AppData.Current.EnableDiagonalMovement)
                        {
                            if (left)
                            {
                                if (Direction != Direction.DownRight)
                                {
                                    assigned = true;
                                    Direction = Direction.UpLeft;
                                }
                            }
                            else if (right)
                            {
                                if (Direction != Direction.DownLeft)
                                {
                                    assigned = true;
                                    Direction = Direction.UpRight;
                                }
                            }
                        }

                        if (!assigned && Direction != Direction.Down)
                        {
                            Direction = Direction.Up;
                        }
                    }
                    else if (down)
                    {
                        bool assigned = false;

                        if (AppData.Current.EnableDiagonalMovement)
                        {
                            if (left)
                            {
                                if (Direction != Direction.UpRight)
                                {
                                    assigned = true;
                                    Direction = Direction.DownLeft;
                                }
                            }
                            else if (right)
                            {
                                if (Direction != Direction.UpLeft)
                                {
                                    assigned = true;
                                    Direction = Direction.DownRight;
                                }
                            }
                        }

                        if (!assigned && Direction != Direction.Up)
                        {
                            Direction = Direction.Down;
                        }
                    }
```

```
                else if (left)
                {
                    if (Direction != Direction.Right)
                    {
                        Direction = Direction.Left;
                    }
                }
                else if (right)
                {
                    if (Direction != Direction.Left)
                    {
                        Direction = Direction.Right;
                    }
                }

                if (Direction != originalDirection)
                {
                    newBendInfo = new BendInfo(X, Y, Direction);
                }
            }
            else
            {
                if (newBendInfo != null)
                {
                    bendInfo.Add(newBendInfo);
                }

                if (bendInfo.Count >= 1 && bendInfo[0].X == X && bendInfo[0].Y
== Y)
                {
                    Direction = bendInfo[0].Direction;

                    // Need to remove it after passing it using AddRange to the
new SnakeBody
                    removeFirstBendInfo = true;
                }
            }

            // Cannot pass property as ref or out parameter
            int x = X;
            int y = Y;
            UpdateCoordinates(Direction, ref x, ref y);

            if (IsHead && !IsInGameBoard(x, y))
            {
                snake.IsAlive = false;
                return;
            }
            else
            {
                X = x;
                Y = y;
            }

            if (NextBody != null)
            {
                if (NextBody.isNew)
                {
                    NextBody.isNew = false;

                    if (!IsHead)
```

```
                    {
                        NextBody.bendInfo.AddRange(bendInfo);

                        // It's already in the bendInfo list
                        newBendInfo = null;
                    }
                }

                NextBody.Update(newBendInfo);
            }

            if (removeFirstBendInfo)
            {
                bendInfo.RemoveAt(0);
            }
        }

        public void LateUpdate()
        {
            if (IsHead)
            {
                foreach (Snake otherSnake in Snake.Current)
                {
                    foreach (SnakeBody body in otherSnake)
                    {
                        if (!ReferenceEquals(this, body) && HitTest(body))
                        {
                            snake.IsAlive = false;
                            break;
                        }
                    }

                    if (!snake.IsAlive)
                    {
                        break;
                    }
                }
            }

            if (snake.IsAlive)
            {
                Renderer.AddToBuffer(Profile.Color, X, Y, Size, Size);
                NextBody?.LateUpdate();
            }
        }

        protected static void UpdateCoordinates(Direction direction, ref int x,
ref int y)
        {
            if (direction == Direction.UpLeft || direction == Direction.Up ||
direction == Direction.UpRight)
            {
                y -= SIZE;
            }
            else if (direction == Direction.DownLeft || direction == Direc-
tion.Down || direction == Direction.DownRight)
            {
                y += SIZE;
            }
```

```
                if (direction == Direction.UpLeft || direction == Direction.Left ||
direction == Direction.DownLeft)
                {
                    x -= SIZE;
                }
                else if (direction == Direction.UpRight || direction == Direc-
tion.Right || direction == Direction.DownRight)
                {
                    x += SIZE;
                }

                if (BorderlessMode)
                {
                    if (y < gameBoardTop)
                    {
                        y = gameBoardBottom - SIZE;
                    }
                    else if (y > gameBoardBottom - SIZE)
                    {
                        y = gameBoardTop;
                    }

                    if (x < gameBoardLeft)
                    {
                        x = gameBoardRight - SIZE;
                    }
                    else if (x > gameBoardRight - SIZE)
                    {
                        x = gameBoardLeft;
                    }
                }
            }
        }

        private sealed class Berry : GameObjectBase
        {
            private const ConsoleColor x = ConsoleColor.Red;
            private const ConsoleColor _ = Constants.BACKGROUND_COLOR;

            private static readonly List<Berry> _current        = new List<Berry>();
            private static readonly Random random               = new Random();
            private static readonly ConsoleColor[,] texture     = new Con-
soleColor[,]
            {
                { _, x, x, x, _ },
                { x, x, x, x, x },
                { x, x, x, x, x },
                { x, x, x, x, x },
                { _, x, x, x, _ }
            };
            private static readonly int textureSize             = tex-
ture.GetLength(0);

            public static IEnumerable<Berry> Current
            {
                get
                {
                    return _current.AsEnumerable();
                }
            }
```

```
        public override int Size
        {
            get { return textureSize; }
        }
        public ConsoleColor Color { get; }
        public int Power { get; }

        public Berry() : this(1)
        {

        }

        public Berry(int power)
        {
            ExceptionHelper.ValidateNumberGreaterOrEqual(power, 0,
nameof(power));

            Color = ConsoleColor.Red;
            Power = power;

            _current.Add(this);
            GenerateNewPosition(false);
        }

        public int Eat()
        {
            GenerateNewPosition(true);
            return Power;
        }

        private void GenerateNewPosition(bool removePrevious)
        {
            // It will remove the Game board border on 0,0 without this condi-
tion
            if (removePrevious)
            {
                int size = Size;
                Renderer.AddToBuffer(ConsoleColor.White, X, Y, size, size);
            }

            bool regenerate;

            do
            {
                regenerate = false;

                X = random.Next(gameBoardLeft, gameBoardRight - Size);
                Y = random.Next(gameBoardTop, gameBoardBottom - Size);

                // Do not generate berry in a snake xD
                for (int row = Y; row < Y + Size; row++)
                {
                    for (int column = X; column < X + Size; column++)
                    {
                        if (Renderer.Buffer[row, column] != Constants.BACK-
GROUND_COLOR)
                        {
                            regenerate = true;
                            break;
                        }
                    }
```

```
                        if (regenerate)
                        {
                            break;
                        }
                    }
                } while (regenerate);

                AlignPosition();
                Renderer.AddToBuffer(texture, X, Y);
            }

            public static void Reset()
            {
                _current.Clear();
            }
        }

        private sealed class BendInfo
        {
            public int X { get; }
            public int Y { get; }
            public Direction Direction { get; }

            public BendInfo(int x, int y, Direction direction)
            {
                X = x;
                Y = y;
                Direction = direction;
            }
        }

        private enum Direction
        {
            Left,
            UpLeft,
            Up,
            UpRight,
            Right,
            DownRight,
            Down,
            DownLeft
        }
    }
}
```

2.15 SnakeTheResurrection.ProfileManager

```
using SnakeTheResurrection.Data;
using System;
using System.Collections.Generic;

namespace SnakeTheResurrection
{
    public static class ProfileManager
    {
        private static readonly List<Profile> profiles = new List<Profile>();

        public static Profile CurrentProfile { get; private set; }
```

```
        public static void ShowProfileSelection()
        {
            if (profiles.Count < 1)
            {
                CreateNewProfile();
            }

            //TODO: UI for profile selection
            CurrentProfile = profiles[0];
        }

        private static void CreateNewProfile()
        {
            Profile newProfile = new Profile
            {
                Name = "Pandafrog",
                Color = ConsoleColor.Green
            };

            //TODO: UI for customization

            profiles.Add(newProfile);
        }

        public static void SaveProfiles()
        {
            //TODO: Save profiles
        }

        public static void LoadProfiles()
        {
            //TODO: Load profiles
        }
    }
}
```

2.16 SnakeTheResurrection.Program

```
using SnakeTheResurrection.Data;
using SnakeTheResurrection.Utilities;
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Runtime.CompilerServices;

namespace SnakeTheResurrection
{
    public static class Program
    {
        public static void Main(string[] args)
        {
            Console.Title = Constants.APP_NAME;

            // Don't you dare try uncommenting this (ง'̀-'́)ง
            // Console.InputEncoding   = Constants.encoding;
            // Console.OutputEncoding  = Constants.encoding;

            // Run the static constructor of Renderer
            RuntimeHelpers.RunClassConstructor(typeof(Renderer).TypeHandle);

            AppData.Load();
```

```
            ProfileManager.LoadProfiles();

            MainMenu();

#if DEBUG
            throw new Exception("Y u do dis ʘ_ʘ");
#else
            Exit();
#endif
        }

        public static void MainMenu()
        {
            ListMenu mainMenu = new ListMenu
            {
                Items = new List<MenuItem>
                {
                    // I hope we're not filling the call stack using the while in-
stead of calling the method in itself again to restart
                    new MenuItem("Singleplayer",    () => { while (Game.Play(false))
; }    ),
                    new MenuItem("Multiplayer",     () => { while (Game.Play(true))
; }     ),
                    new MenuItem("About",           About
),
                    new MenuItem("Quit game",       () => Exit()
)
                }
            };

            ProfileManager.ShowProfileSelection();

            while (true)
            {
                Renderer.ClearBuffer();
                Symtext.WriteTitle(Constants.APP_SHORT_NAME, 7);

                mainMenu.InvokeResult();
                InputHelper.ClearInputBuffer();
            }
        }

        public static void About()
        {
            bool goBack = false;

            ListMenu aboutMenu = new ListMenu
            {
                Items = new List<MenuItem>
                {
                    new MenuItem("GitHub repo", () => Pro-
cess.Start("https://github.com/bramborman/SnakeTheResurrection")   ),
                    new MenuItem("Back",        () => goBack = true
)
                },
                SelectedIndex = 1
            };

            // Whole screen has to be rendered every time, because opening the link
causes everything on screen to disappear
```

```
            do
            {
                Symtext.WriteTitle("About", 0);
                Symtext.SetCenteredTextProperties();

                Symtext.WriteLine($"{Constants.APP_SHORT_NAME} v2.0.1 '{Con-
stants.APP_NAME_ADDITION}'");
                Symtext.WriteLine("© 2017 Marian Dolinský\n");

                aboutMenu.InvokeResult();
            } while (!goBack);
        }

        public static void Exit([CallerMemberName]string callerMemberName = null)
        {
            AppData.Current.Save();
            ProfileManager.SaveProfiles();
            Environment.Exit(callerMemberName == nameof(Main) ? 1 : 0);
        }

        public static void ExitWithError()
        {
            Environment.Exit(1);
        }
    }
}
```

3 Závěr

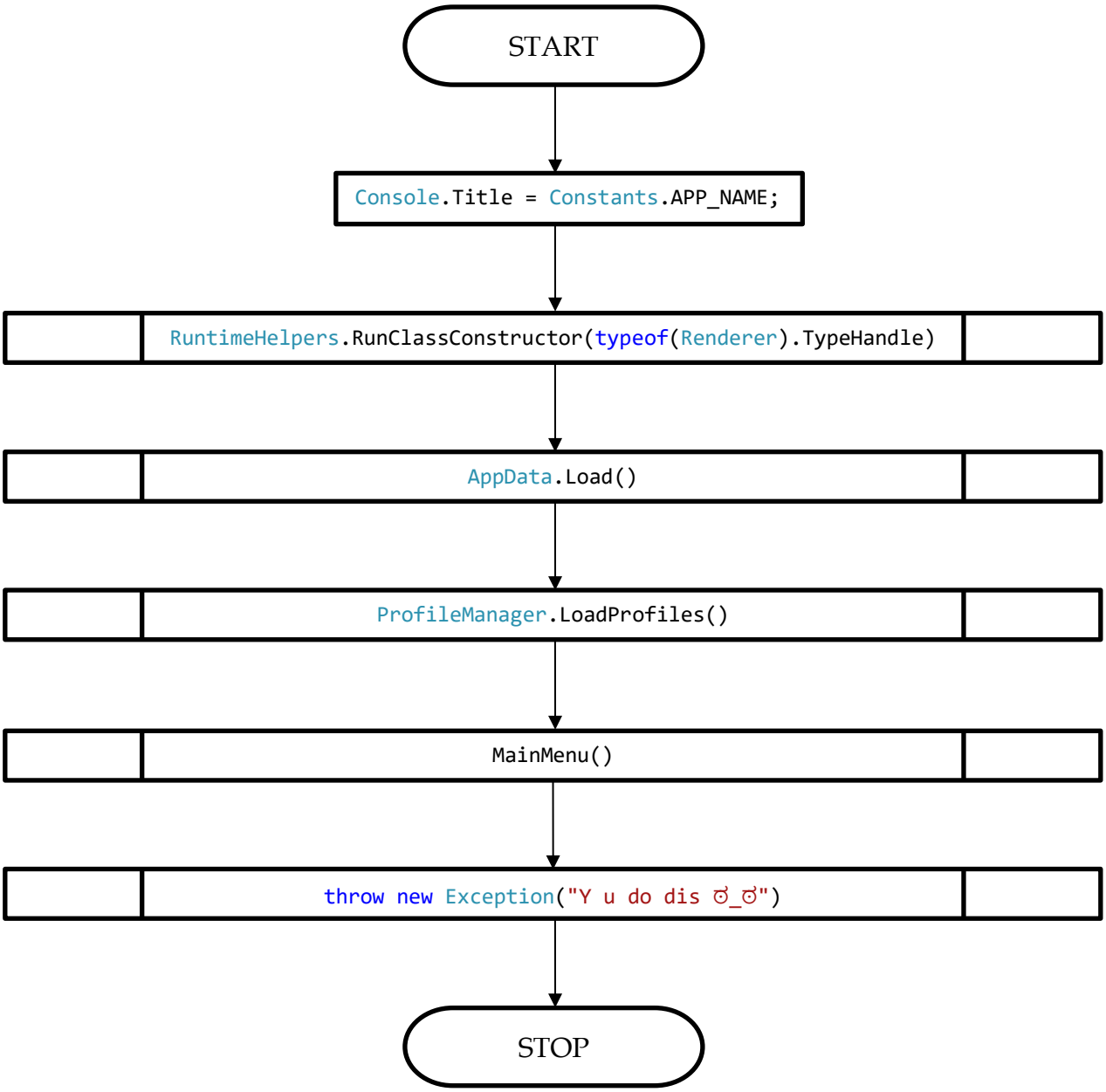
Na rozdíl od loňského roku jsem letos na zpracování této práce strávil pouhých 27 dní. Musím se přiznat, že mě velmi překvapilo, kolik věcí se mi do ní podařilo za tak krátkou dobu implementovat. Hra sice navenek nenabízí tolik funkcí jako loňská verze, ale kód uvnitř je napsaný s přípravou na budoucnost – spousta funkcí, jako např. hra až čtyř hráčů, je sice funkční, ale proto, že jiné funkce, jako např. správce profilů, nejsou ještě dokončené, nejsou ve hře zapnuté. Stejně tak nastavení, které umožní hráči vybrat si vlastní barvu hada, jeho jméno nebo změnit ovládání, či načítání cheat kódů. Kód je pro všechny tyto funkce připravený, jenom pro něj není zatím udělané uživatelské rozhraní.

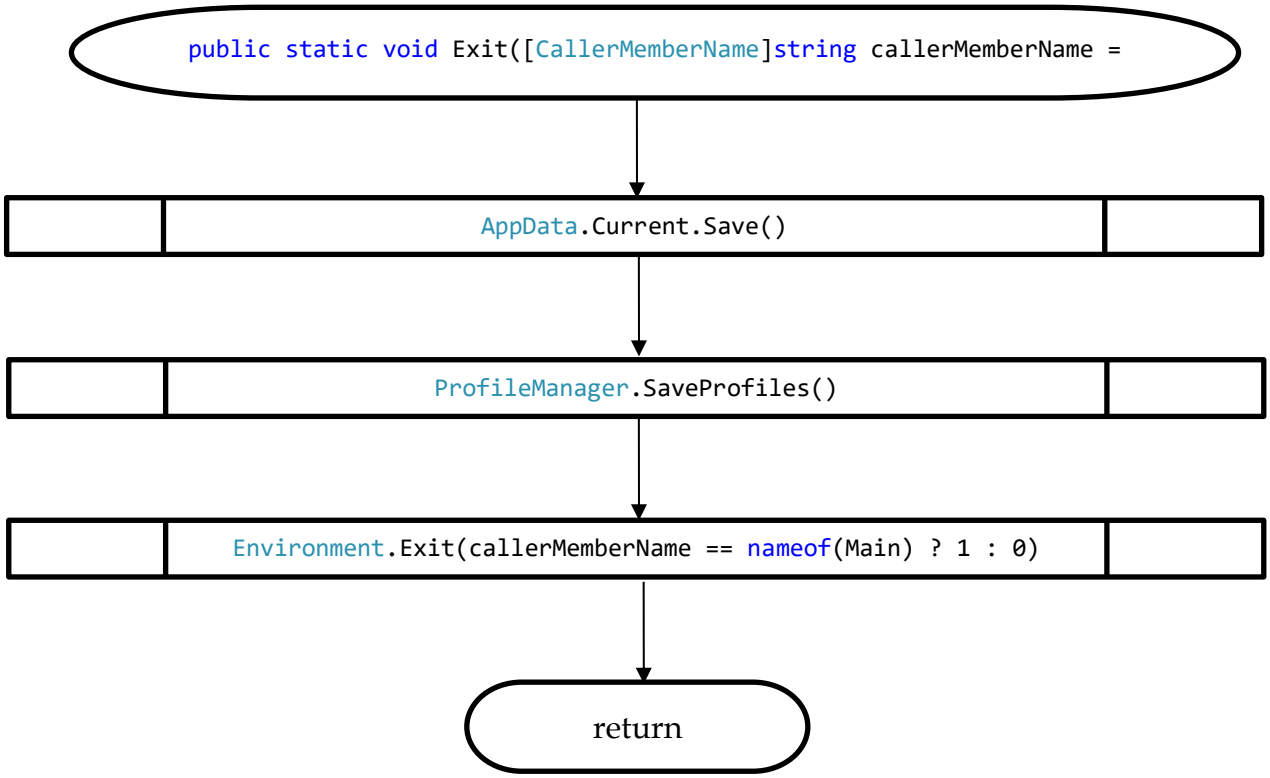
Během práce na této hře jsem se naučil, jak správně využívat tzv. Platform Invoke, který umožňuje z tzv. managed kódu volat nativní funkce systému Windows, definované například v knihovnách kernel32.dll nebo user32.dll a importované pomocí DllImport atributu.

I když mě tato práce zpočátku vůbec nebavila a musel jsem se přemlouvat, abych na ní začal pracovat, v průběhu práce, hlavně pak po objevení způsobu, jak změnit rozlišení konzole na polovinu reálného rozlišení monitoru, mě tato práce začala velice bavit a budu v ní ve volném čase pokračovat. V budoucnu bych rád dokončil zmiňované chybějící funkce a nakonec aplikaci publikoval ve Windows Store, převedenou pomocí Desktop App Converteru (taktéž známém pod názvem Project Centennial) od Microsoftu.

Na závěr bych rád poděkoval Tomáši Lošťákovi, který mi pomohl vyřešit jeden problém týkající se neúplného rozsahu znaků v konzoli v režimu celé obrazovky, dále Davidu Knieradlovi, který mi také pomohl s několika problémy, se kterými jsem se setkal, a pomohl mi program ještě ve svých raných verzích testovat. Speciální díky také patří Alexandře Rakušanové, Jakubu Smejkalovi a Kryštofu Mackovi, kteří taktéž pomohli odhalit pár chyb během testování mého programu.

4 Vývojový diagram




public static void Exit([CallerMemberName]string callerMemberName =
AppData.Current.Save()
ProfileManager.SaveProfiles()
Environment.Exit(callerMemberName == nameof(Main) ? 1 : 0)
return

5 Literatura

Console Functions [online]. [cit. 2017-05-14]. Dostupné z:
`https://msdn.microsoft.com/en-us/library/windows/desktop/ms682073.aspx`

Microsoft MSDN [online]. [cit. 2017-05-14]. Dostupné z:
`https://msdn.microsoft.com/`

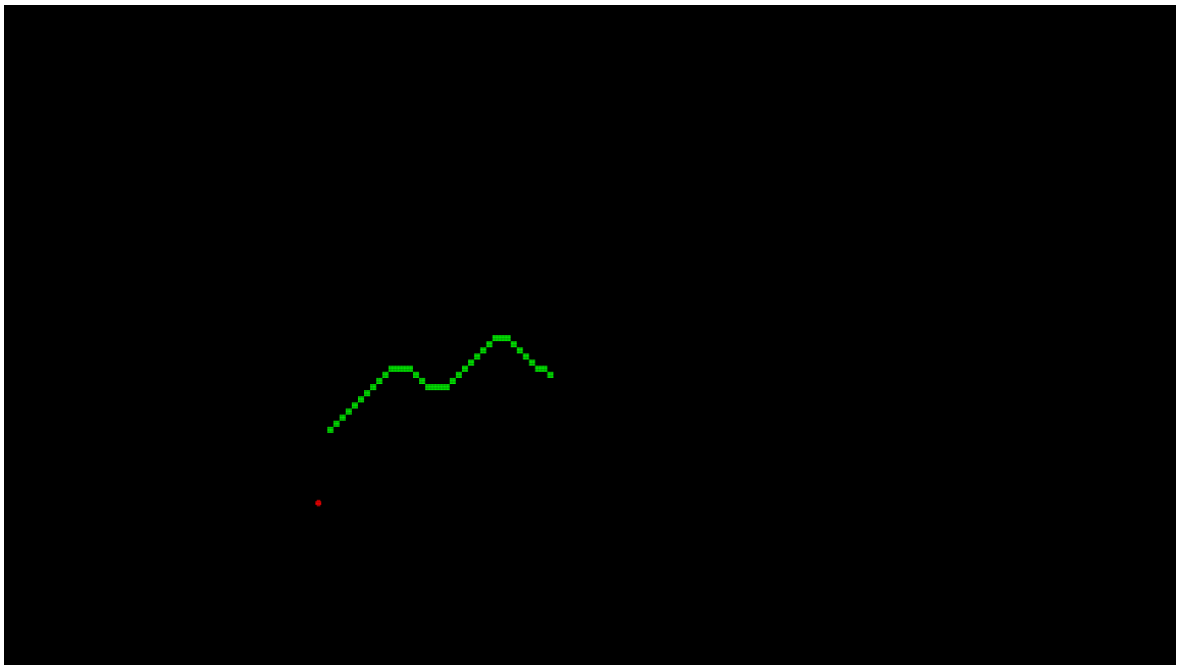
PInvoke.net [online]. [cit. 2017-05-14]. Dostupné z: `http://www.pinvoke.net/`

Stack Overflow [online]. [cit. 2017-05-14]. Dostupné z:
`http://stackoverflow.com/`

Přílohy

A Hlavní menu

B Probíhající hra



C Hra čtyř hráčů