```
/*
 * Author : Yannick R. Brodard
 * File name : Primitives2D.cs
 * Version : 0.1.201505191333
 * Description : Primitives 2D for monogame
 */

using HelProject;
using HelProject.Tools;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using System;

namespace HelHelProject.Tools
{
    public class Primitives2D
    {
        /* SINGLETON START */
        private static Primitives2D _instance;

        /// <summary>
        /// Instance of the class
        /// </summary>
        public static Primitives2D Instance
        {
            get
            {
                if (_instance == null)
                    _instance = new Primitives2D();
                return _instance;
            }
        }

        /// <summary>
        /// Creates a primitive 2D
        /// </summary>
        private Primitives2D() { /* no code... */ }
        /* SINGLETON END */
        private const int DEFAULT_THICKNESS = 1;
        private Texture2D _pixel;

        /// <summary>
        /// Loads the content of the primitives 2D
        /// </summary>
        public void LoadContent()
        {
            _pixel = new Texture2D(MainGame.Instance.GraphicsDevice, 1, 1);
            _pixel.SetData<Color>(new Color[] { Color.White });
        }

        /// <summary>
        /// Draws a line
        /// </summary>
        /// <param name="sb">Sprite batch</param>
        /// <param name="start">Start of the line</param>
        /// <param name="end">End of the line</param>
        /// <param name="color">Color</param>
        /// <param name="thickness">Thickness of the line</param>
        /// <see
        cref="http://gamedev.stackexchange.com/questions/44015/how-can-i-draw-a-simple-2d-line
        -in-xna-without-using-3d-primitives-and-shders"/>
        public void DrawLine(SpriteBatch sb, Vector2 start, Vector2 end, Color color, int
        thickness = DEFAULT_THICKNESS)
        {
            Vector2 edge = end - start;
            // calculate angle to rotate line
            float angle =
                (float)Math.Atan2(edge.Y, edge.X);


            sb.Draw(this._pixel,
                new Rectangle(// rectangle defines shape of line and position of start of line
                    (int)start.X,
                    (int)start.Y,
                    (int)edge.Length(), //sb will strech the texture to fill this rectangle
                    thickness), //width of line, change this to make thicker line
                null,
                color, //colour of line
                angle,     //angle of line (calulated above)
                new Vector2(0, 0), // point in line about which to rotate
                SpriteEffects.None,
                0);
        }

        /// <summary>
        /// Draws a rectangle
        /// </summary>
        /// <param name="sb">Sprite batch</param>
        /// <param name="start">Start point</param>
        /// <param name="end">End point</param>
        /// <param name="color">Color</param>
        /// <param name="thickness">Thickness of the edge of the rectangle</param>
        public void DrawRectangle(SpriteBatch sb, Vector2 start, Vector2 end, Color color,
        int thickness = DEFAULT_THICKNESS)
        {
            Vector2 pointA = start;
            Vector2 pointB = new Vector2(end.X, start.Y);
            Vector2 pointC = end;
            Vector2 pointD = new Vector2(start.X, end.Y);
            this.DrawLine(sb, pointA, pointB, color, thickness);
            this.DrawLine(sb, pointB, pointC, color, thickness);
            this.DrawLine(sb, pointC, pointD, color, thickness);
            this.DrawLine(sb, pointD, pointA, color, thickness);
        }

        /// <summary>
        /// Draws a rectangle
        /// </summary>
        /// <param name="sb">Sprite batch</param>
        /// <param name="rectangle">Rectangle to draw</param>
        /// <param name="color">Color</param>
        /// <param name="thickness">Thickness of the edge of the rectangle</param>
        public void DrawRectangle(SpriteBatch sb, FRectangle rectangle, Color color, int
        thickness = DEFAULT_THICKNESS)
        {
            Vector2 start = rectangle.Position;
            Vector2 end = new Vector2(rectangle.Position.X + rectangle.Width, rectangle.
            Position.Y + rectangle.Height);
            this.DrawRectangle(sb, start, end, color, thickness);
        }

        /// <summary>
        /// Draws a rectangle
        /// </summary>
        /// <param name="sb">Sprite batch</param>
        /// <param name="position">Position of the rectangle</param>
        /// <param name="width">Width of the rectangle</param>
        /// <param name="height">Height of the rectangle</param>
        /// <param name="color">Color</param>
        /// <param name="thickness">Thickness of the edge of the rectangle</param>
        public void DrawRectangle(SpriteBatch sb, Vector2 position, int width, int height,
        Color color, int thickness = DEFAULT_THICKNESS)
        {
            this.DrawRectangle(sb, position, new Vector2(width + position.X, height +
            position.Y), color, thickness);
        }

        /// <summary>
        /// Draws a rectangle
        /// </summary>
        /// <param name="sb">Sprite batch</param>
        /// <param name="x">X position of the rectangle</param>
        /// <param name="y">Y position of the rectangle</param>
        /// <param name="width">Width of the rectangle</param>
        /// <param name="height">Height of the rectangle</param>
        /// <param name="color">Color</param>
        /// <param name="thickness">Thickness of the edge of the rectangle</param>
        public void DrawRectangle(SpriteBatch sb, int x, int y, int width, int height, Color
        color, int thickness = DEFAULT_THICKNESS)
        {
            this.DrawRectangle(sb, new Vector2(x, y), new Vector2(width + x, height + y),
            color, thickness);
        }

        /// <summary>
        /// Fills a rectangle
        /// </summary>
        /// <param name="sb">Sprite batch</param>
        /// <param name="start">Start point</param>
        /// <param name="end">End point</param>
        /// <param name="color">Filling color</param>
        public void FillRectangle(SpriteBatch sb, Vector2 start, Vector2 end, Color color)
        {
            sb.Draw(this._pixel,
                new Rectangle(// rectangle defines shape of line and position of start of line
                    (int)start.X,
                    (int)start.Y,
                    (int)(end.X - start.X), //sb will strech the texture to fill this
                    rectangle
                    (int)(end.Y - start.Y)), //width of line, change this to make thicker line
                null,
                color, //colour of line
                0f,     //angle of line
                new Vector2(0, 0), // point in line about which to rotate
                SpriteEffects.None,
                0);
        }

        /// <summary>
        /// Fills a rectangle
        /// </summary>
        /// <param name="sb">Sprite batch</param>
        /// <param name="rectangle">Rectangle to fill</param>
        /// <param name="color">Filling color</param>
        public void FillRectangle(SpriteBatch sb, FRectangle rectangle, Color color)
        {
            Vector2 start = rectangle.Position;
            Vector2 end = new Vector2(rectangle.Position.X + rectangle.Width, rectangle.
            Position.Y + rectangle.Height);
            this.FillRectangle(sb, start, end, color);
        }

        /// <summary>
        /// Fills a rectangle
        /// </summary>
        /// <param name="sb">Sprite batch</param>
        /// <param name="position">Position of the rectangle</param>
        /// <param name="width">Width of the rectangle</param>
        /// <param name="height">Height of the rectangle</param>
        /// <param name="color">Filling color</param>
        public void FillRectangle(SpriteBatch sb, Vector2 position, int width, int height,
        Color color)
        {
            this.FillRectangle(sb, position, new Vector2(width + position.X, height +
            position.Y), color);
        }

        /// <summary>
        /// Fills a rectangle
        /// </summary>
        /// <param name="sb">Sprite batch</param>
        /// <param name="x">X position of the rectangle</param>
        /// <param name="y">Y position of the rectangle</param>
        /// <param name="width">Width of the rectangle</param>
        /// <param name="height">Height of the rectangle</param>
        /// <param name="color">Filling color</param>
        public void FillRectangle(SpriteBatch sb, int x, int y, int width, int height, Color
        color)
        {
            this.FillRectangle(sb, new Vector2(x, y), new Vector2(width + x, height + y),
            color);
        }
    }
}
```

```
/*
 * Author : Yannick R. Brodard
 * File name : XmlManager.cs
 * Version : 0.1.201504241035
 * Description : Used to manage the xml fils for the classes
 */

#region USING STATEMENTS
using System;
using System.IO;
using System.Xml.Serialization;
#endregion

namespace HelProject.Tools
{
    public class XmlManager<T>
    {
        private Type _type;

        /// <summary>
        /// Type of the class that the XmlManager represents
        /// </summary>
        public Type TypeClass
        {
            get { return _type; }
            set { _type = value; }
        }

        /// <summary>
        /// Loads a XML
        /// </summary>
        /// <param name="path">path of the xml</param>
        /// <returns>Object of the xml</returns>
        public T Load(string path)
        {
            T instance;
            using (TextReader reader = new StreamReader(path))
            {
                XmlSerializer xml = new XmlSerializer(TypeClass);
                instance = (T)xml.Deserialize(reader);
            }
            return instance;
        }

        /// <summary>
        /// Saves an XML
        /// </summary>
        /// <param name="path">path of the xml file</param>
        /// <param name="obj">object to serialize</param>
        public void Save(string path, object obj)
        {
            using (TextWriter writer = new StreamWriter(path))
            {
                XmlSerializer xml = new XmlSerializer(TypeClass);
                xml.Serialize(writer, obj);
            }
        }
    }
}
```

```
/*
 * Author : Yannick R. Brodard
 * File name : FRectangle.cs
 * Version : 0.1.201505130900
 * Description : Rectangle with float parameters
 */

using HelProject.GameWorld.Map;
using Microsoft.Xna.Framework;

namespace HelProject.Tools
{
    /// <summary>
    /// Rectangle with float parameters
    /// </summary>
    public class FRectangle
    {
        public float X;
        public float Y;
        public float Width;
        public float Height;

        /// <summary>
        /// Creates a rectangle with float parameters
        /// </summary>
        /// <param name="width">Width of the rectangle</param>
        /// <param name="height">Height of the rectangle</param>
        public FRectangle(float width, float height) : this(0f, 0f, width, height) { /* no
        code... */ }

        /// <summary>
        /// Creates a rectangle with float parameters
        /// </summary>
        /// <param name="x">X position of the rectangle</param>
        /// <param name="y">Y position of the rectangle</param>
        /// <param name="width">Width of the rectangle</param>
        /// <param name="height">Height of the rectangle</param>
        public FRectangle(float x, float y, float width, float height)
        {
            X = x;
            Y = y;
            Width = width;
            Height = height;
        }

        /// <summary>
        /// Top position
        /// </summary>
        public float Top
        {
            get { return Y; }
        }

        /// <summary>
        /// Bottom position
        /// </summary>
        public float Bottom
        {
            get { return Y + Height; }
        }

        /// <summary>
        /// Left position
        /// </summary>
        public float Left
        {
            get { return X; }
        }

        /// <summary>
        /// Right position
        /// </summary>
        public float Right
        {
            get { return X + Width; }
        }

        /// <summary>
        /// Position of the rectangle
        /// </summary>
        public Vector2 Position
        {
            get { return new Vector2(X, Y); }
            set
            {
                this.X = value.X;
                this.Y = value.Y;
            }
        }

        /// <summary>
        /// Finds if this rectangle is intersecting with another
        /// </summary>
        /// <param name="rectangle">Other rectangle tested</param>
        /// <returns>Results if it's intersecting</returns>
        /// <see
        cref="http://stackoverflow.com/questions/13390333/two-rectangles-intersection"/>
        public bool Intersects(FRectangle rectangle)
        {
            float X = this.X;
            float Y = this.Y;
            float A = this.Width + X;
            float B = this.Height + Y;
            float X1 = rectangle.X;
            float Y1 = rectangle.Y;
            float A1 = rectangle.Width + X1;
            float B1 = rectangle.Height + Y1;

            if (A < X1 || A1 < X || B < Y1 || B1 < Y)
            {
                return false;
            }
            else
            {
                return true;
            }
        }

        /// <summary>
        /// Finds if a point is intersecting with this rectangle
        /// </summary>
        /// <param name="x">X position of the point</param>
        /// <param name="y">Y position of the point</param>
        /// <returns>Results if it's intersecting</returns>
        public bool Intersects(float x, float y)
        {
            float b = 1f / (float)HCell.TILE_SIZE;
            return this.Intersects(new FRectangle(x, y, b, b));
        }

        /// <summary>
        /// Finds if a vector is intersecting with this rectangle
        /// </summary>
        /// <param name="position">Position</param>
        /// <returns>Results if it's intersecting</returns>
        public bool Intersects(Vector2 position)
        {
            float b = 1f / (float)HCell.TILE_SIZE;
            return this.Intersects(new FRectangle(position.X, position.Y, b, b));
        }

        /// <summary>
        /// Sets the position of the bounds accordingly to the given position
        /// </summary>
        public void SetBoundsWithTexture(Vector2 position, int textureWidth, int
        textureHeight)
        {
            this.X = position.X - (float)textureWidth / 2f / (float)HCell.TILE_SIZE;
            this.Y = position.Y - (float)textureHeight / 2f / (float)HCell.TILE_SIZE;
        }
    }
}
```

```
/*
 * Author : Yannick R. Brodard
 * File name : InputManager.cs
 * Version : 0.1.201504281237
 * Description : Manages all the input in the game
 *               It is a singleton class
 */

#region USING STATEMENTS
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Input;
using System;
using System.Collections.Generic;
#endregion

namespace HelProject.Tools
{
    public class InputManager
    {
        #region ATTRIBUTES
        private static InputManager _instance;
        private KeyboardState _kbState;
        private List<Keys> _downKeys;
        private List<Keys> _pressedKeys;
        private List<Keys> _releasedKeys;
        private MouseState _msState;

        #endregion

        #region PROPRIETIES
        /// <summary>
        /// List of the keys that are released
        /// </summary>
        public List<Keys> ReleasedKeys
        {
            get { return _releasedKeys; }
            private set { _releasedKeys = value; }
        }

        /// <summary>
        /// List of the keys that are at a down state
        /// </summary>
        public List<Keys> DownKeys
        {
            get { return _downKeys; }
            private set { _downKeys = value; }
        }

        /// <summary>
        /// List of the keys that are at a pressed state
        /// </summary>
        public List<Keys> PressedKeys
        {
            get { return _pressedKeys; }
            private set { _pressedKeys = value; }
        }
        /// <summary>
        /// Current keyboard state
        /// </summary>
        public KeyboardState KbState
        {
            get { return _kbState; }
            private set { _kbState = value; }
        }

        /// <summary>
        /// Current mouse state
        /// </summary>
        public MouseState MsState
        {
            get { return _msState; }
            set { _msState = value; }
        }

        /// <summary>
        /// Instance of the class
        /// </summary>
        public static InputManager Instance
        {
            get
            {
                if (_instance == null)
                {
                    _instance = new InputManager();
                }
                return _instance;
            }
        }
        #endregion

        #region CONSTRUCTORS
        /// <summary>
        /// Creates an inputmanager
        /// </summary>
        private InputManager()
        {
            KbState = new KeyboardState();
            this.PressedKeys = new List<Keys>();
            this.ReleasedKeys = new List<Keys>();
            this.DownKeys = new List<Keys>();
        }
        #endregion

        #region METHODS
        /// <summary>
        /// Updates the states of the inputs
        /// </summary>
        /// <param name="gameTime"></param>
        public void Update(GameTime gameTime)
        {
            this.UpdateKeyboardInput();
            this.UpdateMouseInput();
        }
        /// <summary>
        /// Checks if a key is up.
        /// </summary>
        /// <param name="key">Key that is checked</param>
        /// <returns>Boolean</returns>
        public bool IsKeyboardKeyReleased(Keys key)
        {
            foreach (Keys k in this.ReleasedKeys)
            {
                if (key == k)
                    return true;
            }
            return false;
        }

        /// <summary>
        /// Checks if the key is down
        /// </summary>
        /// <param name="key">Key that is checked</param>
        /// <returns>Boolean</returns>
        public bool IsKeyboardKeyDown(Keys key)
        {
            foreach (Keys k in this.DownKeys)
            {
                if (key == k)
                    return true;
            }
            return false;
        }

        /// <summary>
        /// Checks if the key is pressed
        /// </summary>
        /// <param name="key">Key that is checked</param>
        /// <returns>Boolean</returns>
        public bool IsKeyboardKeyPressed(Keys key)
        {
            foreach (Keys k in this.PressedKeys)
            {
                if (key == k)
                    return true;
            }
            return false;
        }
        #endregion

        #region PRIVATE METHODS
        /// <summary>
        /// Updates the states of the keyboard
        /// </summary>
        private void UpdateKeyboardInput()
        {
            if (MainGame.Instance.IsActive)
            {
                this.KbState = Keyboard.GetState();
                // Verifies all the keys of the keyboard
                foreach (Keys key in Enum.GetValues(typeof(Keys)))
                {
                    // If the key is not pressed and it is not in the release key list and
                    is in one of the other two list
                    // we add it to the released key list and remove it in the others
                    if (this.KbState.IsKeyUp(key) && !this.IsKeyboardKeyReleased(key) &&
                        (this.IsKeyboardKeyPressed(key) || this.IsKeyboardKeyDown(key)))
                    {
                        this.DownKeys.Remove(key);
                        this.PressedKeys.Remove(key);
                        this.ReleasedKeys.Add(key);
                    }
                    else if (this.KbState.IsKeyUp(key) && this.IsKeyboardKeyReleased(key))
                    // If it is already in the released key list
                    {                                                                 //
                    remove it
                        this.ReleasedKeys.Remove(key);
                    }
                    else
                    {
                        // If the key is down and is not yet pressed
                        if (this.KbState.IsKeyDown(key) && !this.IsKeyboardKeyDown(key) && !
                        this.IsKeyboardKeyPressed(key))
                        {
                            // remove it from the other lists (to be sure)
                            this.ReleasedKeys.Remove(key);
                            this.PressedKeys.Remove(key);
                            this.DownKeys.Add(key); // and add it to the down key list
                        }
                        else if (this.KbState.IsKeyDown(key) && !this.IsKeyboardKeyPressed(
                        key)) // If it's already in the down key list
                        {
                        // and isn't in the pressed list
                            this.DownKeys.Remove(key); // remove it from the other lists
                            this.ReleasedKeys.Remove(key);
                            this.PressedKeys.Add(key); // add it to the pressed list
                        }
                    }
                }
            }
            else
                this.KbState = new KeyboardState();
        }

        /// <summary>
        /// Updates the states of the mouse
        /// </summar>
        private void UpdateMouseInput()
        {
            if (MainGame.Instance.IsActive)
                this.MsState = Mouse.GetState();
            else
                this.MsState = new MouseState();
        }
        #endregion
    }
```

```
}
```

```
/*
 * Author : Yannick R. Brodard
 * File name : HSpell.cs
 * Version : 0.1.201505070906
 * Description : Abstract class and base of all spells
 */

#region USING STATEMENTS
using HelProject.Features;
using HelProject.GameWorld.Entities;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
#endregion

namespace HelProject.GameWorld.Spells
{
    /// <summary>
    /// Spell of the game
    /// </summary>
    public abstract class HSpell
    {
        #region ATTRIBUTES
        private HHero _hero;
        private FeatureCollection _features;
        private string _name;
        private float _timeOfEffect;
        #endregion

        #region PROPRIETIES
        /// <summary>
        /// Hero that the spell is attached to
        /// </summary>
        public HHero Hero
        {
            get { return _hero; }
            set { _hero = value; }
        }

        /// <summary>
        /// Features of the spell
        /// </summary>
        public FeatureCollection Features
        {
            get { return _features; }
            set { _features = value; }
        }

        /// <summary>
        /// Name of the spell
        /// </summary>
        public string Name
        {
            get { return _name; }
            set { _name = value; }
        }

        /// <summary>
        /// Time of effect once the spell is active
        /// </summary>
        public float TimeOfEffect
        {
            get { return _timeOfEffect; }
            set { _timeOfEffect = value; }
        }
        #endregion

        #region CONSTRUCTORS
        /// <summary>
        /// Creates a spell
        /// </summary>
        /// <param name="hero">Hero that the spell is attached to</param>
        /// <param name="features">Features of the spell</param>
        /// <param name="timeOfEffect">Time of effect once the spell is active</param>
        /// <param name="name">Name of the spell</param>
        public HSpell(HHero hero, FeatureCollection features, float timeOfEffect, string name)
        {
            this.Hero = hero;
            this.Features = features;
            this.TimeOfEffect = timeOfEffect;
            this.Name = name;
        }
        #endregion

        #region METHODS
        /// <summary>
        /// Loads the content of the spell
        /// </summary>
        public virtual void LoadContent() { /* no code... */ }

        /// <summary>
        /// Unloads the content of the spell
        /// </summary>
        public virtual void UnloadContent() { /* no code... */ }

        /// <summary>
        /// Updates the spell in the game loop
        /// </summary>
        /// <param name="gameTime"></param>
        public virtual void Update(GameTime gameTime) { /* no code... */ }

        /// <summary>
        /// Draws the spell
        /// </summary>
        /// <param name="spriteBatch"></param>
        public virtual void Draw(SpriteBatch spriteBatch) { /* no code... */ }
        #endregion
    }
}
```

```
/*
 * Author : Yannick R. Brodard
 * File name : HSpellSingelTarget.cs
 * Version : 0.1.201505070928
 * Description : Abstract class and base of all single targeted spells
 */

using HelProject.Features;
using HelProject.GameWorld.Entities;

namespace HelProject.GameWorld.Spells
{
    /// <summary>
    /// Single targeted spell
    /// </summary>
    public abstract class HSpellSingleTarget : HSpell
    {
        private HEntity _target;

        /// <summary>
        /// Target of the spell
        /// </summary>
        public HEntity Target
        {
            get { return _target; }
            set { _target = value; }
        }

        /// <summary>
        /// Creates a single target spell
        /// </summary>
        /// <param name="hero">Hero that the spell is attached to</param>
        /// <param name="features">Features of the spell</param>
        /// <param name="timeOfEffect">Time of effect once the spell is active</param>
        /// <param name="name">Name of the spell</param>
        /// <param name="target">Target of the spell</param>
        public HSpellSingleTarget(HHero hero, FeatureCollection features, float timeOfEffect,
         string name, HEntity target)
            : base(hero, features, timeOfEffect, name)
        {
            this.Target = target;
        }
    }
}
```

```
/*
 * Author : Yannick R. Brodard
 * File name : HSpellBuff.cs
 * Version : 0.1.201505070911
 * Description : Abstract class and base of all spell-buffs
 */

#region USING STATEMENTS
using HelProject.Features;
using HelProject.GameWorld.Entities;
#endregion

namespace HelProject.GameWorld.Spells
{
    /// <summary>
    /// Spell buff
    /// </summary>
    public abstract class HSpellBuff : HSpell
    {
        /// <summary>
        /// Creates a spell buff
        /// </summary>
        /// <param name="hero">Hero that the spell is attached to</param>
        /// <param name="features">Features of the spell</param>
        /// <param name="timeOfEffect">Time of effect once the spell is active</param>
        /// <param name="name">Name of the spell</param>
        public HSpellBuff(HHero hero, FeatureCollection features, float timeOfEffect, string
        name) : base(hero, features, timeOfEffect, name) { /* no code... */ }
    }
}
```

```
/*
 * Author : Yannick R. Brodard
 * File name : HSpellZone.cs
 * Version : 0.1.201505070916
 * Description : Abstract class and base of all spell-zones
 */

using HelProject.Features;
using HelProject.GameWorld.Entities;

namespace HelProject.GameWorld.Spells
{
    /// <summary>
    /// Spell-zone
    /// </summary>
    public abstract class HSpellZone : HSpell
    {
        private float _range;
        private float _areaOfEffect;

        /// <summary>
        /// Casting range of the spell
        /// </summary>
        public float Range
        {
            get { return _range; }
            set { _range = value; }
        }

        /// <summary>
        /// Area of effect diameter of the spell
        /// </summary>
        public float AreaOfEffect
        {
            get { return _areaOfEffect; }
            set { _areaOfEffect = value; }
        }

        /// <summary>
        /// Creates a spell zone
        /// </summary>
        /// <param name="hero">Hero that the spell is attached to</param>
        /// <param name="features">Features of the spell</param>
        /// <param name="timeOfEffect">Time of effect once the spell is active</param>
        /// <param name="name">Name of the spell</param>
        /// <param name="range">Casting range of the spell</param>
        /// <param name="areaOfEffect">Area of effect diameter of the spell</param>
        public HSpellZone(HHero hero, FeatureCollection features, float timeOfEffect, string
        name, float range, float areaOfEffect)
            : base(hero, features, timeOfEffect, name)
        {
            this.Range = range;
            this.AreaOfEffect = areaOfEffect;
        }
    }
}
```

```
/*
 * Author : Yannick R. Brodard
 * File name : HHero.cs
 * Version : 0.1.201505110841
 * Description : Hero class, controllable entity by the player
 */

using HelProject.Features;
using HelProject.GameWorld.Map;
using HelProject.GameWorld.Spells;
using HelProject.Tools;
using HelProject.UI;
using HelProject.UI.HUD;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using System.Collections.Generic;

namespace HelProject.GameWorld.Entities
{
    /// <summary>
    /// Controllable entity
    /// </summary>
    public class HHero : HEntity
    {
        private HSpell _spellSlot1;
        private HSpell _spellSlot2;
        private HSpell _spellSlot3;
        private HSpell _spellSlot4;
        private FillingBar _playerHealth;

        /// <summary>
        /// Player's health bar
        /// </summary>
        public FillingBar PlayerHealth
        {
            get { return _playerHealth; }
            set { _playerHealth = value; }
        }

        /// <summary>
        /// Spell present in slot 1
        /// </summary>
        public HSpell SpellSlot1
        {
            get { return _spellSlot1; }
            set { _spellSlot1 = value; }
        }

        /// <summary>
        /// Spell present in slot 2
        /// </summary>
        public HSpell SpellSlot2
        {
            get { return _spellSlot2; }
            set { _spellSlot2 = value; }
        }
        /// <summary>
        /// Spell present in slot 3
        /// </summary>
        public HSpell SpellSlot3
        {
            get { return _spellSlot3; }
            set { _spellSlot3 = value; }
        }

        /// <summary>
        /// Spell present in slot 4
        /// </summary>
        public HSpell SpellSlot4
        {
            get { return _spellSlot4; }
            set { _spellSlot4 = value; }
        }

        /// <summary>
        /// Creates a controlable entity
        /// </summary>
        /// <param name="initialFeatures">Initial features of the entity</param>
        /// <param name="position">Position of the enitity</param>
        public HHero(FeatureCollection initialFeatures, Vector2 position, float width, float
        height, string textureName) : base(initialFeatures, position, width, height,
        textureName) { /* no code... */ }

        /// <summary>
        /// Loads the content of the entity
        /// </summary>
        public override void LoadContent()
        {
            base.LoadContent();

            // init player health bar
            FRectangle r = new FRectangle(20, MainGame.Instance.GraphicsDevice.Viewport.
            Height - 170, 30, 150);
            this.PlayerHealth = new FillingBar(FillingBar.FillingDirection.BottomToTop, r,
            Color.DarkRed, Color.Red, new Color(Color.Black, 0.75f),
                                               this.FeatureCalculator.GetTotalLifePoints(),
                                               this.ActualFeatures.LifePoints);
        }

        /// <summary>
        /// Unloads the content of the entity
        /// </summary>
        public override void UnloadContent()
        {
            base.UnloadContent();
        }

        /// <summary>
        /// Updates the entity
        /// </summary>
        /// <param name="gameTime">Current game time</param>
        public override void Update(GameTime gameTime)
        {
            base.Update(gameTime);

            this.UpdateNoMovementKey();
            this.UpdateBasicAttack(gameTime);
            this.UpdateMovement(gameTime);
            this.UpdateTeleportUsage();
            this.PlayerHealth.ActualValue = this.ActualFeatures.LifePoints;
            this.CheckLife();

            PlayScreen.Instance.Camera.Position = this.Position; // Apply the new position
            to the camera
        }

        /// <summary>
        /// Draws the entity on screen
        /// </summary>
        /// <param name="spriteBatch">Sprite batch used for the drawing</param>
        public override void Draw(SpriteBatch spriteBatch)
        {
            base.Draw(spriteBatch);
            this.PlayerHealth.Draw(spriteBatch);
        }

        /// <summary>
        /// Teleports the play to the designed area
        /// </summary>
        /// <param name="map">Map</param>
        /// <param name="position">Position</param>
        public void Teleport(HMap map, Vector2 position)
        {
            PlayScreen.Instance.CurrentMap = map;
            this.Position = position;
            this.Bounds.SetBoundsWithTexture(position, this.Texture.Width, this.Texture.
            Height);
            PlayScreen.Instance.Camera.Position = position;
        }

        /// <summary>
        /// Sets the state to nomovement if the left shift is pressed
        /// </summary>
        private void UpdateNoMovementKey()
        {
            if (InputManager.Instance.IsKeyboardKeyDown(Keys.LeftShift) ||
                InputManager.Instance.IsKeyboardKeyPressed(Keys.LeftShift) ||
                InputManager.Instance.IsKeyboardKeyReleased(Keys.LeftShift))
                this.State = EntityState.NoMovement;
        }

        /// <summary>
        /// Checks if the player is attacking something
        /// </summary>
        private void UpdateBasicAttack(GameTime gameTime)
        {
            if (InputManager.Instance.MsState.LeftButton == ButtonState.Pressed)
            {
                if (PlayScreen.Instance.SelectionAssistant.SelectedObjects.Count > 0)
                {
                    if (PlayScreen.Instance.SelectionAssistant.SelectedObjects[0] is HHostile)
                    {
                        HHostile target = PlayScreen.Instance.SelectionAssistant.
                        SelectedObjects[0] as HHostile;
                        if (target.Bounds.Intersects(this.AttackBounds))
                        {
                            this.State = EntityState.MeleeAttacking;
                            this.BasicMeleeAttack(target, gameTime);
                        }
                    }
                }
            }
        }

        /// <summary>
        /// Checks if the player is using a teleporter
        /// </summary>
        private void UpdateTeleportUsage()
        {
            List<HCell> adjacentCells = PlayScreen.Instance.CurrentMap.GetAdjacentCells((int)
            this.Position.X, (int)this.Position.Y, 1, 1, true);
            int count = adjacentCells.Count;
            for (int i = 0; i < count; i++)
            {
                if (adjacentCells[i].Type == "teleporteasy")
                {
                    if (this.Bounds.Intersects(adjacentCells[i].Bounds))
                        this.Teleport(PlayScreen.Instance.MapDifficultyEasy, PlayScreen.
                        Instance.MapDifficultyEasy.GetRandomFloorPoint());
                }

                if (adjacentCells[i].Type == "teleportmedium")
                {
                    if (this.Bounds.Intersects(adjacentCells[i].Bounds))
                        this.Teleport(PlayScreen.Instance.MapDifficultyMedium, PlayScreen.
                        Instance.MapDifficultyMedium.GetRandomFloorPoint());
                }

                if (adjacentCells[i].Type == "teleporthard")
                {
                    if (this.Bounds.Intersects(adjacentCells[i].Bounds))
                        this.Teleport(PlayScreen.Instance.MapDifficultyHard, PlayScreen.
                        Instance.MapDifficultyHard.GetRandomFloorPoint());
                }
            }
        }

        /// <summary>
        /// Updates the movement of the character
        /// </summary>
        /// <param name="gameTime">Game time</param>
        /// <returns>Did a movement happen ?</returns>
        private void UpdateMovement(GameTime gameTime)
        {
            Vector2 newPosition = this.Position; // gets the current position
            MouseState ms = InputManager.Instance.MsState; // gets the current state of the
            mouse

            // is the right button of the mouse clicked ?
            if ((ms.LeftButton == ButtonState.Pressed) &&
                (this.State != EntityState.MeleeAttacking) &&
                (this.State != EntityState.RangeAttacking) &&
                (this.State != EntityState.NoMovement))
            {
                // Update hero state to running
                this.State = EntityState.Running;

                Vector2 mouseVector = ms.Position.ToVector2(); // Gets mouse position
                Vector2 direction = mouseVector - ScreenManager.Instance.
                GetCorrectScreenPosition(this.Position, PlayScreen.Instance.Camera.Position,
                32); // Gets the direction of the mouse from the player
                direction.Normalize(); // Normalize the direction vector

                float elapsedTime = (float)gameTime.ElapsedGameTime.TotalSeconds; // gets
                the elapsed time in seconds from the last update
                newPosition += direction * elapsedTime * (this.FeatureCalculator.
                GetTotalMovementSpeed()); // Calculates the new position
                FRectangle newBounds = new FRectangle(this.Bounds.Width, this.Bounds.Height);
                 // ready the new bounds of the character
                newBounds.SetBoundsWithTexture(newPosition, this.Texture.Width, this.Texture.
                Height);

                this.ApplyFluidMovement(direction, newPosition, newBounds, elapsedTime);

                this.Direction = direction; // Update the direction the hero is facing
            }
        }
    }
}
```

```
/*
 * Author : Yannick R. Brodard
 * File name : HHostile.cs
 * Version : 0.1.201505182014
 * Description : Base class for hostile entities
 */

using HelHelProject.Tools;
using HelProject.Features;
using HelProject.GameWorld.Map;
using HelProject.Tools;
using HelProject.UI;
using HelProject.UI.HUD;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace HelProject.GameWorld.Entities
{
    public class HHostile : HEntity
    {
        private const float ALERT_FOV_MUTLIPLIER = 1.85f;

        private FRectangle _fieldOfView;
        private FRectangle _alertedFieldOfView;
        private bool _isAlerted;
        private FillingBar _healthBar;

        /// <summary>
        /// Health bar of the hostile
        /// </summary>
        public FillingBar HealthBar
        {
            get { return _healthBar; }
            set { _healthBar = value; }
        }

        /// <summary>
        /// The unit is alerted
        /// </summary>
        public bool IsAlerted
        {
            get { return _isAlerted; }
            set { _isAlerted = value; }
        }

        /// <summary>
        /// Field of view of the hostile
        /// </summary>
        public FRectangle FieldOfView
        {
            get { return _fieldOfView; }
            set { _fieldOfView = value; }
        }
        /// <summary>
        /// Field of view of the hositle when this one is alerted
        /// </summary>
        public FRectangle AlertedFieldOfView
        {
            get { return _alertedFieldOfView; }
            set { _alertedFieldOfView = value; }
        }

        /// <summary>
        /// Creates a hostile creature
        /// </summary>
        /// <param name="initialFeatures">The initial features</param>
        /// <param name="position">Position</param>
        /// <param name="width">Width (in-game unit)</param>
        /// <param name="height">Height (in-game unit)</param>
        /// <param name="textureName">Name of the texture</param>
        public HHostile(FeatureCollection initialFeatures, Vector2 position, float width,
        float height, string textureName, float fieldOfView = 8.125f)
            : base(initialFeatures, position, width, height, textureName)
        {
            this.IsAlerted = false;
            this.FieldOfView = new FRectangle(fieldOfView, fieldOfView);
            this.AlertedFieldOfView = new FRectangle(fieldOfView * ALERT_FOV_MUTLIPLIER,
            fieldOfView * ALERT_FOV_MUTLIPLIER);
            this.HealthBar = new FillingBar(FillingBar.FillingDirection.LeftToRight, new
            FRectangle(30, 5), Color.Black, Color.Red, Color.Black,
                this.FeatureCalculator.GetTotalLifePoints(), this.ActualFeatures.LifePoints);
        }

        /// <summary>
        /// Loads the content of the hostile
        /// </summary>
        public override void LoadContent()
        {
            base.LoadContent();
            this.IsAlerted = false;
        }

        /// <summary>
        /// Unloads the content of the hostile
        /// </summary>
        public override void UnloadContent()
        {
            base.UnloadContent();
            this.IsAlerted = false;
        }

        /// <summary>
        /// Updates the mechanismes of the hostile
        /// </summary>
        /// <param name="gameTime">Game time</param>
        public override void Update(GameTime gameTime)
        {
            base.Update(gameTime);
            this.CenterFieldOfView();
            this.UpdatePursuit(gameTime);
            this.UpdateHealthBar();
            this.CheckLife();
        }

        /// <summary>
        /// Draws the hostile
        /// </summary>
        /// <param name="spriteBatch">Sprite batch</param>
        public override void Draw(SpriteBatch spriteBatch)
        {
            base.Draw(spriteBatch);
            this.HealthBar.Draw(spriteBatch);

            if (MainGame.DEBUG_MODE)
            {
                if (this.IsAlerted)
                {
                    Vector2 start = ScreenManager.Instance.GetCorrectScreenPosition(this.
                    AlertedFieldOfView.Position, PlayScreen.Instance.Camera.Position);
                    Vector2 end = ScreenManager.Instance.GetCorrectScreenPosition(new Vector2
                    (this.AlertedFieldOfView.Position.X + this.AlertedFieldOfView.Width, this
                    .AlertedFieldOfView.Position.Y + this.AlertedFieldOfView.Height),
                    PlayScreen.Instance.Camera.Position);
                    end.X += 1f;
                    end.Y += 1f;
                    Primitives2D.Instance.DrawRectangle(spriteBatch, start, end, Color.Blue);
                }
                else
                {
                    Vector2 start = ScreenManager.Instance.GetCorrectScreenPosition(this.
                    FieldOfView.Position, PlayScreen.Instance.Camera.Position);
                    Vector2 end = ScreenManager.Instance.GetCorrectScreenPosition(new Vector2
                    (this.FieldOfView.Position.X + this.FieldOfView.Width, this.FieldOfView.
                    Position.Y + this.FieldOfView.Height), PlayScreen.Instance.Camera.
                    Position);
                    end.X += 1f;
                    end.Y += 1f;
                    Primitives2D.Instance.DrawRectangle(spriteBatch, start, end, Color.Blue);
                }
            }
        }

        /// <summary>
        /// Updates the pursuit mechanism of the hostile
        /// </summary>
        /// <param name="gameTime">Game time</param>
        private void UpdatePursuit(GameTime gameTime)
        {
            if (this.FieldOfView.Intersects(PlayScreen.Instance.PlayableCharacter.Bounds) ||
               (this.IsAlerted && this.AlertedFieldOfView.Intersects(PlayScreen.Instance.
               PlayableCharacter.Bounds)))
            {
                this.IsAlerted = true;
                this.UpdateAttackOnPlayer(gameTime);
                this.UpdateMovementTowardsPlayer(gameTime);
            }
            else
            {
                this.IsAlerted = false;
            }
        }

        /// <summary>
        /// Attacks the player
        /// </summary>
        private void UpdateAttackOnPlayer(GameTime gameTime)
        {
            HHero target = PlayScreen.Instance.PlayableCharacter;
            if (target.Bounds.Intersects(this.AttackBounds))
            {
                this.State = EntityState.MeleeAttacking;
                this.BasicMeleeAttack(target, gameTime);
            }
        }

        /// <summary>
        /// Updates the movement of the hostile so it can reach the player
        /// </summary>
        /// <param name="gameTime">Game time</param>
        private void UpdateMovementTowardsPlayer(GameTime gameTime)
        {
            if ((this.State != EntityState.MeleeAttacking) &&
                (this.State != EntityState.RangeAttacking) &&
                (this.State != EntityState.NoMovement))
            {
                this.State = EntityState.Running;
                Vector2 newPosition = this.Position;
                Vector2 heroPosition = new Vector2(PlayScreen.Instance.PlayableCharacter.
                Position.X, PlayScreen.Instance.PlayableCharacter.Position.Y);
                Vector2 direction = heroPosition - newPosition; // new position is still
                actual position
                direction.Normalize();

                float elapsedTime = (float)gameTime.ElapsedGameTime.TotalSeconds;
                newPosition += direction * elapsedTime * this.FeatureCalculator.
                GetTotalMovementSpeed();
                FRectangle newBounds = new FRectangle(this.Bounds.Width, this.Bounds.Height);
                 // ready the new bounds of the character
                newBounds.SetBoundsWithTexture(newPosition, this.Texture.Width, this.Texture.
                Height);

                this.ApplyFluidMovement(direction, newPosition, newBounds, elapsedTime);
                this.Direction = direction;
            }
        }

        /// <summary>
        /// Updates the health bar of the hostile
        /// </summary>
        private void UpdateHealthBar()
        {
            this.HealthBar.ActualValue = this.ActualFeatures.LifePoints;
            Vector2 hbPos = new Vector2((this.Position.X - this.HealthBar.Container.Width / 2
             / HCell.TILE_SIZE) + 1f / HCell.TILE_SIZE, this.Position.Y - this.Texture.Height
             / HCell.TILE_SIZE);
            this.HealthBar.Container.Position = ScreenManager.Instance.
            GetCorrectScreenPosition(hbPos, PlayScreen.Instance.Camera.Position);
        }

        /// <summary>
        /// Centers the field of view to the position
        /// </summary>
        private void CenterFieldOfView()
        {
            float x = 0, y = 0;
            x = this.Position.X - this.FieldOfView.Width / 2f;
            y = this.Position.Y - this.FieldOfView.Height / 2f;
            this.FieldOfView.Position = new Vector2(x, y);
            x = this.Position.X - this.AlertedFieldOfView.Width / 2f;
            y = this.Position.Y - this.AlertedFieldOfView.Height / 2f;
            this.AlertedFieldOfView.Position = new Vector2(x, y);
        }
    }
}
```

```
/*
 * Author : Yannick R. Brodard
 * File name : HEntity.cs
 * Version : 0.5.201505110823
 * Description : Base abstract class for the entities of the game
 */

using HelHelProject.Tools;
using HelProject.Features;
using HelProject.GameWorld.Map;
using HelProject.Tools;
using HelProject.UI;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using System;
using System.Collections.Generic;

namespace HelProject.GameWorld.Entities
{
    /// <summary>
    /// Base abstract class for the entities of the game
    /// </summary>
    public abstract class HEntity : HObject
    {
        public const float DEFAULT_STRENGHT = 5.0f;
        public const float DEFAULT_AGILITY = 5.0f;
        public const float DEFAULT_VITALITY = 5.0f;
        public const float DEFAULT_MAGIC = 5.0f;
        public const float DEFAULT_ATTACKSPEED = 0.6f;
        public const float DEFAULT_MINUMUMDAMAGE = 1.0f;
        public const float DEFAULT_MAXIMUMDAMAGE = 3.0f;
        public const float DEFAULT_MANAREGENERATION = 1.0f;
        public const float DEFAULT_MOVEMENTSPEED = 5.0f;
        public const float DEFAULT_LIFEPOINTS = 100.0f;
        public const float DEFAULT_ATTACKBOUND_WIDTH = 1.8f;
        public const float DEFAULT_ATTACKBOUND_HEIGHT = 2.7f;

        private FeatureCollection _initialFeatures;
        private FeatureCollection _actualFeatures;
        private FeatureCollection _maximizedFeatures;
        private FeatureManager _featureCalculator;
        private EntityState _state;
        private Vector2 _direction;
        private FRectangle _bounds;
        private FRectangle _attackBounds;
        private Texture2D _texture;
        private Random _rand;
        private double _lastAttackTime;
        private bool _isDead;

        /// <summary>
        /// Is the entity dead
        /// </summary>
        public bool IsDead
        {
            get { return _isDead; }
            set { _isDead = value; }
        }

        /// <summary>
        /// Attack bounds of the entity
        /// </summary>
        public FRectangle AttackBounds
        {
            get { return _attackBounds; }
            set { _attackBounds = value; }
        }

        /// <summary>
        /// Texture of the entity
        /// </summary>
        public Texture2D Texture
        {
            get { return _texture; }
            set { _texture = value; }
        }

        /// <summary>
        /// Bounds of the entity
        /// </summary>
        public FRectangle Bounds
        {
            get { return _bounds; }
            set { _bounds = value; }
        }

        /// <summary>
        /// Direction the character is facing
        /// </summary>
        public Vector2 Direction
        {
            get { return _direction; }
            set { _direction = value; }
        }

        /// <summary>
        /// State of the entity
        /// </summary>
        public EntityState State
        {
            get { return _state; }
            set { _state = value; }
        }

        /// <summary>
        /// Maximized features
        /// </summary>
        public FeatureCollection MaximizedFeatures
        {
            get { return _maximizedFeatures; }
            set { _maximizedFeatures = value; }
        }

        /// <summary>
        /// Feature manager to calculate the actual features
        /// </summary>
        public FeatureManager FeatureCalculator
        {
            get { return _featureCalculator; }
            set { _featureCalculator = value; }
        }

        /// <summary>
        /// Actual features of the entity
        /// </summary>
        public FeatureCollection ActualFeatures
        {
            get { return _actualFeatures; }
            set { _actualFeatures = value; }
        }

        /// <summary>
        /// Initial feature of the entity
        /// </summary>
        public FeatureCollection InitialFeatures
        {
            get { return _initialFeatures; }
            set { _initialFeatures = value; }
        }

        /// <summary>
        /// Creates an entity
        /// </summary>
        public HEntity() : this(Vector2.Zero) { /* no code... */ }

        /// <summary>
        /// Creates an entity
        /// </summary>
        /// <param name="position">Position of the entity</param>
        public HEntity(Vector2 position)
            : this(new FeatureCollection()
            {
                Strenght = DEFAULT_STRENGHT,
                Vitality = DEFAULT_VITALITY,
                Agility = DEFAULT_AGILITY,
                Magic = DEFAULT_MAGIC,
                InitialAttackSpeed = DEFAULT_ATTACKSPEED,
                MinimumDamage = DEFAULT_MINUMUMDAMAGE,
                MaximumDamage = DEFAULT_MAXIMUMDAMAGE,
                InitialManaRegeneration = DEFAULT_MANAREGENERATION,
                InitialMovementSpeed = DEFAULT_MOVEMENTSPEED,
                InitialLifePoints = DEFAULT_LIFEPOINTS
            }, position, 0f, 0f, null) { /* no code... */ }

        /// <summary>
        /// Creates an entity
        /// </summary>
        /// <param name="initialFeatures">Initial Features of the enitity</param>
        /// <param name="position">Position of the entity</param>
        public HEntity(FeatureCollection initialFeatures, Vector2 position, float width,
        float height, string textureName)
```

```
            : base(true, position)
        {
            this.InitialFeatures = initialFeatures;
            this.Position = position;
            this.FeatureCalculator = new FeatureManager(this.InitialFeatures);
            this.ActualFeatures = this.FeatureCalculator.GetCalculatedFeatures();
            this.MaximizedFeatures = (FeatureCollection)this.ActualFeatures.Clone();
            this.State = EntityState.Idle;
            this.Texture = TextureManager.Instance.GetTexture(textureName);
            this.Bounds = new FRectangle(width, height);
            this.Bounds.SetBoundsWithTexture(position, this.Texture.Width, this.Texture.
            Height);
            this.AttackBounds = new FRectangle(DEFAULT_ATTACKBOUND_WIDTH,
            DEFAULT_ATTACKBOUND_HEIGHT);
            this.AttackBounds.X = this.Position.X - this.AttackBounds.Width / 2f;
            this.AttackBounds.Y = this.Position.Y - this.AttackBounds.Height / 2f;
            this._rand = new Random();
            this._lastAttackTime = 0d;
            this.IsDead = false;
        }

        /// <summary>
        /// Updates the entity
        /// </summary>
        /// <param name="gameTime"></param>
        public override void Update(GameTime gameTime)
        {
            base.Update(gameTime);
            this.State = EntityState.Idle;

            // Centers the attack bounds of the entity
            this.AttackBounds.X = this.Position.X - this.AttackBounds.Width / 2f;
            this.AttackBounds.Y = this.Position.Y - this.AttackBounds.Height / 2f;
        }

        /// <summary>
        /// Draws the entity
        /// </summary>
        /// <param name="spriteBatch"></param>
        public override void Draw(SpriteBatch spriteBatch)
        {
            base.Draw(spriteBatch);

            if (this.Texture != null && this.Bounds != null)
            {
                Vector2 boundsPosA = new Vector2(this.Bounds.X, this.Bounds.Y);

                Vector2 position = ScreenManager.Instance.GetCorrectScreenPosition(boundsPosA
                , PlayScreen.Instance.Camera.Position);
                spriteBatch.Draw(this.Texture, position, Color.White);
            }

            if (MainGame.DEBUG_MODE)
            {
                Vector2 start = ScreenManager.Instance.GetCorrectScreenPosition(this.
                AttackBounds.Position, PlayScreen.Instance.Camera.Position);
                Vector2 end = ScreenManager.Instance.GetCorrectScreenPosition(new Vector2(
                this.AttackBounds.Position.X + this.AttackBounds.Width, this.AttackBounds.
                Position.Y + this.AttackBounds.Height), PlayScreen.Instance.Camera.Position);
                end.X += 1f;
                end.Y += 1f;
                Primitives2D.Instance.DrawRectangle(spriteBatch, start, end, Color.Red);
            }
        }

        /// <summary>
        /// Corrects the movement. When the entity touches an unwalkable object, it slides
        instead of stopping completly
        /// </summary>
        /// <param name="direction">Direction of the movement</param>
        /// <param name="newPosition">The new calculated position</param>
        /// <param name="newBounds">The new bounds according to the position</param>
        /// <param name="elapsedTime">Elapsed game time in seconds</param>
        public void ApplyFluidMovement(Vector2 direction, Vector2 newPosition, FRectangle
        newBounds, float elapsedTime)
        {
            // Is the position of the hero on a walkable area ?
            if (this.IsCharacterSurfaceWalkable(newPosition, newBounds))
            {
                this.Position = newPosition; // Apply the new position to the hero
                this.Bounds = newBounds; // Apply the new bounds to the hero
            }
            else // try with the biggest axis
            {
                newPosition = this.Position;
                float nX = (direction.X >= 0) ? direction.X : direction.X * -1;
                float nY = (direction.Y >= 0) ? direction.Y : direction.Y * -1;

                if (nX > nY)
                {
                    newPosition += new Vector2(direction.X, 0.0f) * elapsedTime * (this.
                    FeatureCalculator.GetTotalMovementSpeed());
                }
                else if (nX < nY)
                {
                    newPosition += new Vector2(0.0f, direction.Y) * elapsedTime * (this.
                    FeatureCalculator.GetTotalMovementSpeed());
                }

                newBounds.SetBoundsWithTexture(newPosition, this.Texture.Width, this.Texture.
                Height);

                if (this.IsCharacterSurfaceWalkable(newPosition, newBounds))
                {
                    this.Position = newPosition; // Apply the new position to the hero
                    this.Bounds = newBounds; // Apply the new bounds to the hero
                }
                else // try with the smallest axis
                {
                    newPosition = this.Position;

                    if (nX < nY)
                    {
                        newPosition += new Vector2(direction.X, 0.0f) * elapsedTime * (this.
                        FeatureCalculator.GetTotalMovementSpeed());
                    }
                    else if (nX > nY)
                    {
                        newPosition += new Vector2(0.0f, direction.Y) * elapsedTime * (this.
                        FeatureCalculator.GetTotalMovementSpeed());
                    }

                    newBounds.SetBoundsWithTexture(newPosition, this.Texture.Width, this.
                    Texture.Height);

                    if (this.IsCharacterSurfaceWalkable(newPosition, newBounds))
                    {
                        this.Position = newPosition; // Apply the new position to the hero
                        this.Bounds = newBounds; // Apply the new bounds to the hero
                    }
                }
            }
        }

        /// <summary>
        /// Checks if the surface where the hero is present if walkable
        /// </summary>
        /// <param name="position">Position of the hero</param>
        /// <param name="bounds">Bounds of the hero</param>
        /// <returns></returns>
        public bool IsCharacterSurfaceWalkable(Vector2 position, FRectangle bounds)
        {
            bool validArea = true;
            List<HCell> unwalkableAdjacentCells = PlayScreen.Instance.CurrentMap.
            GetAdjacentUnwalkableCells((int)this.Position.X, (int)this.Position.Y, 1, 1);
            List<HHostile> hostiles = PlayScreen.Instance.CurrentMap.Hostiles;
            HHero hero = PlayScreen.Instance.PlayableCharacter;
            int nbrCells = unwalkableAdjacentCells.Count;
            int nbrHostiles = hostiles.Count;

            for (int i = 0; i < nbrCells; i++)
            {
                if (bounds.Intersects(unwalkableAdjacentCells[i].Bounds))
                    validArea = false;
            }

            for (int i = 0; i < nbrHostiles; i++)
            {
                if (this != hostiles[i] && bounds.Intersects(hostiles[i].Bounds))
                    validArea = false;
            }

            if (this != hero && bounds.Intersects(hero.Bounds))
                validArea = false;

            return validArea;
        }

        /// <summary>
        /// Basic melee attack
        /// </summary>
        /// <param name="target">Targeted enemy</param>
        public void BasicMeleeAttack(HEntity target, GameTime gameTime)
        {
            double currentTime = gameTime.TotalGameTime.TotalSeconds;
            float secPerAttack = 1f / this.ActualFeatures.AttackSpeed;

            if (currentTime - this._lastAttackTime >= secPerAttack)
            {
                float minDmg = this.ActualFeatures.MinimumDamage;
                float maxDmg = this.ActualFeatures.MaximumDamage;
                float receivedDamage = (float)this._rand.NextDouble();
                receivedDamage = (maxDmg - minDmg) * receivedDamage + minDmg;
                float realReceivedDamage = this.FeatureCalculator.GetReceivedPhysicalDamage(
                receivedDamage);
                realReceivedDamage = realReceivedDamage * (this.ActualFeatures.Strenght / 100
                 + 1);
                target.ActualFeatures.LifePoints -= realReceivedDamage;
                this._lastAttackTime = currentTime;
            }
        }

        /// <summary>
        /// Checks if the hostile is dead
        /// </summary>
        public void CheckLife()
        {
            if (this.ActualFeatures.LifePoints <= 0f)
                this.IsDead = true;
        }

        /// <summary>
        /// State of the entity
        /// </summary>
        public enum EntityState
        {
            Idle,
            Running,
            MeleeAttacking,
            RangeAttacking,
            SpellCasting,
            NoMovement,
        }
    }
}
```

```
/*
 * Author : Yannick R. Brodard
 * File name : HMap.cs
 * Version : 0.5.201505151012
 * Description : The map class, creates a map
 */
/* Helped by : http://www.csharpprogramming.tips/2013/07/Rouge-like-dungeon-generation.html */

#region USING STATEMENTS
using HelProject.GameWorld.Entities;
using HelProject.Tools;
using HelProject.UI;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.Graphics;
using System;
using System.Collections.Generic;
using System.Text;
using System.Xml;
using System.Xml.Serialization;
#endregion

namespace HelProject.GameWorld.Map
{
    /// <summary>
    /// Used the create a map for the game
    /// </summary>
    public class HMap
    {
        #region CONSTANTS
        protected const int WALKABLE_AJDACENT_WALL_QUANTITY_LIMIT = 5;
        protected const int NONWALKABLE_AJDACENT_WALL_QUANTITY_LIMIT = 4;
        protected const int DEFAULT_NONWALKABLE_CELLS_PERCENTAGE = 45;
        protected const int DEFAULT_SMOOTHNESS = 5;
        protected const int MINIMUM_SMOOTHNESS = 1;
        protected const int MINIMUM_HEIGHT = 10;
        protected const int MINIMUM_WIDTH = 10;
        protected const int MAXIMUM_HEIGHT = 800;
        protected const int MAXIMUM_WIDTH = 800;
        #endregion

        #region ATTRIBUTES
        private Random rand = new Random(); // Randomizer
        private HCell[,] _cells; // Cells of the map
        private int _height; // Height of the map
        private int _width; // Width of the map
        private int _nonWalkableSpacePercentage; // Percentage of walkable area in the map
        private ContentManager _content; // content manager
        private float _scale;
        private List<HHostile> _hostiles; // enemies of the map
        private List<HItem> _onFloorItems;
        #endregion

        #region PROPRIETIES
        /// <summary>
        /// Items currently on the floor
        /// </summary>
        public List<HItem> OnFloorItems
        {
            get { return _onFloorItems; }
            set { _onFloorItems = value; }
        }

        /// <summary>
        /// Enemies present in the map
        /// </summary>
        public List<HHostile> Hostiles
        {
            get { return _hostiles; }
            set { _hostiles = value; }
        }

        /// <summary>
        /// Scale of the map
        /// </summary>
        public float Scale
        {
            get { return _scale; }
            set { _scale = value; }
        }

        /// <summary>
        /// Percentage of walkable area in the map
        /// </summary>
        public int NonWalkableSpacePercentage
        {
            get { return _nonWalkableSpacePercentage; }
            private set { _nonWalkableSpacePercentage = value; }
        }

        /// <summary>
        /// Height of the map
        /// </summary>
        public int Height
        {
            get { return _height; }
            private set { _height = value; }
        }

        /// <summary>
        /// Width of the map
        /// </summary>
        public int Width
        {
            get { return _width; }
            private set { _width = value; }
        }

        /// <summary>
        /// Cells of the map
        /// </summary>
        public HCell[,] Cells
        {
            get { return _cells; }
            private set { _cells = value; }
        }
        #endregion

        #region CONSTRUCTORS
        /// <summary>
        /// Creates a map from given cells
        /// </summary>
        /// <param name="cells">Cells of the map</param>
        /// <param name="scale">Scale of the map</param>
        public HMap(HCell[,] cells, float scale = 1.0f)
        {
            this.Width = cells.GetLength(0);
            this.Height = cells.GetLength(1);
            this.NonWalkableSpacePercentage = 0;
            this.Scale = scale;
            this.Cells = cells;
        }

        /// <summary>
        /// Creates a map full of non-walkable cells
        /// </summary>
        /// <param name="height">Height of the map</param>
        /// <param name="width">Width of the map</param>
        /// <param name="nonWalkableSpacePercentage">Amount (percentage) of non-walkable
        area in the map for random filling</param>
        /// <remarks>
        /// Use the 'Make' methods to transform the map
        /// </remarks>
        public HMap(int width, int height, float scale = 1.0f, int nonWalkableSpacePercentage
         = HMap.DEFAULT_NONWALKABLE_CELLS_PERCENTAGE)
        {
            this.Height = Math.Min(HMap.MAXIMUM_HEIGHT, Math.Max(height, HMap.MINIMUM_HEIGHT
            ));
            this.Width = Math.Min(HMap.MAXIMUM_WIDTH, Math.Max(width, HMap.MINIMUM_WIDTH));
            this.NonWalkableSpacePercentage = nonWalkableSpacePercentage;
            this.Scale = scale;
            this.Hostiles = new List<HHostile>();
            this.OnFloorItems = new List<HItem>();

            this.ClearMap();
            this.MakeRandomlyFilledMap();
        }
        #endregion

        #region PUBLIC METHODS
        /// <summary>
        /// Fills the map randomly with borders
        /// </summary>
        public void MakeRandomlyFilledMap()
        {
            this.ClearMap();

            int mapMiddle = 0; // tmp variable

            // X is only created once
            for (int x = 0, y = 0; y < this.Height; y++)
            {
                for (x = 0; x < this.Width; x++)
                {
                    // Fills the edges with walls
                    if (x == 0)
                    {
                        this.Cells[x, y] = new HCell(false, new Vector2(x, y));
                    }
                    else if (y == 0)
                    {
                        this.Cells[x, y] = new HCell(false, new Vector2(x, y));
                    }
                    else if (x == this.Width - 1)
                    {
                        this.Cells[x, y] = new HCell(false, new Vector2(x, y));
                    }
                    else if (y == this.Height - 1)
                    {
                        this.Cells[x, y] = new HCell(false, new Vector2(x, y));
                    }
                    else
                    {
                        mapMiddle = (this.Height / 2);
                        // the middle always has a walkable cell for space logic
                        if (y == mapMiddle)
                        {
                            this.Cells[x, y] = new HCell(true, new Vector2(x, y));
                        }
                        else
                        {
                            // Fills the rest with a random ratio
                            this.Cells[x, y] = new HCell(!this.RandomPercent(this.
                            NonWalkableSpacePercentage), new Vector2(x, y));
                        }
                    }
                }
            }
        }

        /// <summary>
        /// Clears the map, all the cells are null
        /// </summary>
        public void ClearMap()
        {
            this.Cells = new HCell[this.Width, this.Height];
        }

        /// <summary>
        /// Creates a map full of non-walkable cells
        /// </summary>
        public void MakeFullMap()
        {
            for (int y = 0; y < this.Height; y++)
            {
                for (int x = 0; x < this.Width; x++)
                {
                    this.Cells[x, y] = new HCell(false, new Vector2(x, y));
                }
            }
        }

        /// <summary>
        /// Transforms the cells in the map to correspond to a cavern
        /// </summary>
        /// <remarks>
        /// It is best to call the RandomFillMap method before this one to
        /// get the best results
        /// </remarks>
        public void MakeCaverns(int smoothness = DEFAULT_SMOOTHNESS)
        {
            //smoothness = Math.Max(MINIMUM_SMOOTHNESS, smoothness);

            for (int i = 0; i < smoothness; i++) // repeating the carverns algo makes the
            caverns smoother on the edges
            {                                    // and gives a more natural look
                HCell[,] grid = new HCell[this.Width, this.Height];
                for (int x = 0, y = 0; y < this.Height; y++)
                {
                    for (x = 0; x < this.Width; x++)
                    {
                        grid[x, y] = new HCell(PlaceCellLogic(x, y), this.Cells[x, y].
                        Position);
                        //this.SetCell(x, y, PlaceCellLogic(x, y));
                    }
                }
                for (int x = 0, y = 0; y < this.Height; y++)
                {
                    for (x = 0; x < this.Width; x++)
                    {
                        this.Cells[x, y] = grid[x, y];
                    }
                }
            }
        }

        /// <summary>
        /// Places a walkable cell depending on it's neighbors
        /// </summary>
        /// <param name="x">X position of the cell</param>
        /// <param name="y">Y position of the cell</param>
        /// <param name="cell"></param>
        public bool PlaceCellLogic(int x, int y)
        {
            int nbUnwalkableCells = this.GetNumberOfAdjacentUnwalkableCells(x, y, 1, 1);

            HCell cell = this.GetCell(x, y);
            // Checks if the cell is non-walkable
            if (cell.IsWalkable == false)
            {
                // if their is too much non-walkable cells around it
                if (nbUnwalkableCells >= HMap.NONWALKABLE_AJDACENT_WALL_QUANTITY_LIMIT)
                {
                    return false;
                }
                if (nbUnwalkableCells < 2)
                {
                    return true;
                }

            }
            else // if it's walkable
            {
                // if their is too much walls around it, smooth it
                if (nbUnwalkableCells >= HMap.WALKABLE_AJDACENT_WALL_QUANTITY_LIMIT)
                {
                    return false;
                }
            }
            return true;
        }

        /// <summary>
        /// Gets the number of adjacent non-walkable cells around the specified cell
        /// </summary>
        /// <param name="x">X position of the specified cell</param>
        /// <param name="y">Y position of the specified cell</param>
        /// <param name="scopeX">X scope to scan around the specified cell</param>
        /// <param name="scopeY">Y scope to scan around the specified cell</param>
        /// <returns>numbers of non-walkable cells around the specified cell</returns>
        public int GetNumberOfAdjacentUnwalkableCells(int x, int y, int scopeX, int scopeY)
        {
            // INITIALISATION
            int startX = x - scopeX;
            int startY = y - scopeY;
            int endX = x + scopeX;
            int endY = y + scopeY;

            int iX = startX;
            int iY = startY;

            int wallCounter = 0;

            for (iY = startY; iY <= endY; iY++)
            {
                for (iX = startX; iX <= endX; iX++)
                {
                    if (!(iX == x && iY == y))
                    {
                        if (this.IsCellNonwalkable(iX, iY))
                        {
                            wallCounter += 1;
                        }
                    }
                }
            }

            return wallCounter;
        }

        /// <summary>
        /// Gets the adjacent non-walkable cells around the given point and scope
        /// </summary>
        /// <param name="x">X position</param>
        /// <param name="y">Y position</param>
        /// <param name="scopeX">Scope on the X axis</param>
        /// <param name="scopeY">Scope on the Y axis</param>
        /// <returns>A list of the adjacent non-walkable cells</returns>
        public List<HCell> GetAdjacentUnwalkableCells(int x, int y, int scopeX, int scopeY)
        {
            List<HCell> unwalkableCells = new List<HCell>();

            int startX = x - scopeX;
            int startY = y - scopeY;
            int endX = x + scopeX;
            int endY = y + scopeY;

            int iX = startX;
            int iY = startY;

            for (iY = startY; iY <= endY; iY++)
            {
                for (iX = startX; iX <= endX; iX++)
                {
                    if (!(iX == x && iY == y))
                    {
                        if (this.IsCellNonwalkable(iX, iY))
                        {
                            unwalkableCells.Add(this.GetCell(iX, iY));
                        }
                    }
                }
            }

            return unwalkableCells;
        }

        /// <summary>
        /// Gets the adjacent cells of the position
        /// </summary>
        /// <param name="x">X position</param>
        /// <param name="y">Y position</param>
        /// <param name="scopeX">X scope</param>
        /// <param name="scopeY">Y scope</param>
        /// <param name="includeDesignatedCell">Include the specified cell ?</param>
        /// <returns>Adjacent cells</returns>
        public List<HCell> GetAdjacentCells(int x, int y, int scopeX, int scopeY, bool 
        includeDesignatedCell = false)
        {
            List<HCell> adjacentcells = new List<HCell>();

            int startX = x - scopeX;
            int startY = y - scopeY;
            int endX = x + scopeX;
            int endY = y + scopeY;

            int iX = startX;
            int iY = startY;
            for (iY = startY; iY <= endY; iY++)
            {
                for (iX = startX; iX <= endX; iX++)
                {
                    if (!(iX == x && iY == y))
                    {
                        adjacentcells.Add(this.GetCell(iX, iY));
                    }
                    else
                    {
                        if (includeDesignatedCell)
                        {
                            adjacentcells.Add(this.GetCell(iX, iY));
                        }
                    }
                }
            }

            return adjacentcells;
        }

        /// <summary>
        /// Verifies if the specified cell is walkable
        /// </summary>
        /// <param name="x">X position of the cell</param>
        /// <param name="y">Y position of the cell</param>
        /// <returns>Result in boolean</returns>
        public bool IsCellNonwalkable(int x, int y)
        {
            // Verifies if the cell is out of bounds
            if (this.IsCellOutOfBounds(x, y))
            {
                return true; // Consider it non-walkable if it is
            }

            HCell cell = this.GetCell(x, y);

            if (cell.IsWalkable == false)
            {
                return true;
            }
            else
            {
                return false;
            }
        }

        /// <summary>
        /// Verifies if the specified cell is out of bound
        /// </summary>
        /// <param name="x">X position of the cell</param>
        /// <param name="y">Y position of the cell</param>
        /// <returns>Result in boolean</returns>
        public bool IsCellOutOfBounds(int x, int y)
        {
            if (x < 0 || y < 0)
            {
                return true;
            }
            else if (x > this.Width - 1 || y > this.Height - 1)
            {
                return true;
            }
            return false;
        }

        /// <summary>
        /// Gets a copy of the specified cell
        /// </summary>
        /// <param name="x">X position of the cell</param>
        /// <param name="y">Y position of the cell</param>
        /// <returns>Copy of the cell</returns>
        public HCell GetCellCopy(int x, int y)
        {
            return new HCell(this.Cells[x, y].IsWalkable, this.Cells[x, y].Position);
        }

        /// <summary>
        /// Gets the specified cell
        /// </summary>
        /// <param name="x">X position of the cell</param>
        /// <param name="y">Y position of the cell</param>
        /// <returns>Specified cell</returns>
        public HCell GetCell(int x, int y)
        {
            return this.Cells[x, y];
        }

        /// <summary>
        /// Sets the cell
        /// </summary>
        /// <param name="x">X position of the cell</param>
        /// <param name="y">Y position of the cell</param>
        /// <param name="isWalkable">Specify the walkability of the cell</param>
        public void SetCell(int x, int y, bool isWalkable)
        {
            this.Cells[x, y].IsWalkable = isWalkable;
        }

        /// <summary>
        /// Loads the content
        /// </summary>
        public void LoadContent()
        {
            this._content = new ContentManager(ScreenManager.Instance.Content.ServiceProvider
            , "Content");
            if (this.Hostiles == null)
            {
                this.Hostiles = new List<HHostile>();
            }

            if (this.OnFloorItems == null)
                this.OnFloorItems = new List<HItem>();
        }

        /// <summary>
        /// Unloads the content
        /// </summary>
        public void UnloadContent()
        {
            this._content.Unload();
        }

        /// <summary>
        /// Draws the map
        /// </summary>
        /// <param name="spriteBatch">Spritebatch for drawing</param>
        /// <param name="camera">Camera to determine where to draw</param>
        public void Draw(SpriteBatch spriteBatch, Camera camera)
        {
            int sizeOfSprites = HCell.TILE_SIZE;

            // determins the start point for the drawing, so it doesn't draw useless cells
            Point startPoint = new Point((int)camera.Position.X - (int)(camera.Width / 2 /
            sizeOfSprites + 1),
                                         (int)camera.Position.Y - (int)(camera.Height / 2 /
                                         sizeOfSprites) - 1);

            // determins the end point for the drawing, so it doesn't draw useless cells
            Point endPoint = new Point((int)camera.Position.X + (int)(camera.Width / 2 /
            sizeOfSprites + 1),
                                         (int)camera.Position.Y + (int)(camera.Height / 2 /
                                         sizeOfSprites + 2));

            // For each cell from the start to end point, it draws it
            for (int y = startPoint.Y; y < endPoint.Y; y++)
            {
                for (int x = startPoint.X; x < endPoint.X; x++)
                {
                    if (!this.IsCellOutOfBounds(x, y))
                    {
                        HCell cell = this.GetCell(x, y);
                        Vector2 position = ScreenManager.Instance.GetCorrectScreenPosition(
                        cell.Position, camera.Position);

                        spriteBatch.Draw(TextureManager.Instance.GetTexture(cell.Type),
                        position, null, null, null, 0.0f, new Vector2(this.Scale, this.Scale
                        ), Color.White);
                    }
                }
            }

            FRectangle limits = new FRectangle(startPoint.X, startPoint.Y, endPoint.X -
            startPoint.X, endPoint.Y - endPoint.Y);
            this.DrawItems(spriteBatch, camera, limits);
        }

        /// <summary>
        /// Draws all the items that are on the floor of the map
        /// </summary>
        /// <param name="spriteBatch">Sprite batch</param>
        /// <param name="camera">Camera of the game</param>
        /// <param name="limits">Limits where the item will be drawn</param>
        public void DrawItems(SpriteBatch spriteBatch, Camera camera, FRectangle limits)
        {
            int nbrItem = this.OnFloorItems.Count;
            for (int i = 0; i < nbrItem; i++)
            {
                this.OnFloorItems[i].Draw(spriteBatch);
            }
        }

        /// <summary>
        /// Gets a random walkable area
        /// </summary>
        /// <returns>Position of the walkable position</returns>
        public Vector2 GetRandomFloorPoint()
        {
            bool foundPosition = false;
            Vector2 position = Vector2.One;
            while (!foundPosition)
            {
                int rX = rand.Next(0, this.Width);
                int rY = rand.Next(0, this.Height);

                HCell foundCell = this.GetCell(rX, rY);

                if (foundCell.IsWalkable && foundCell.Type.Contains("floor"))
                {
                    int unWalkableCells = this.GetNumberOfAdjacentUnwalkableCells(rX, rY, 1,
                    1);
                    if (unWalkableCells == 0)
                    {
                        int nbHostiles = this.Hostiles.Count;
                        bool noIntersection = true;
                        for (int i = 0; i < nbHostiles; i++)
                        {
                            if (foundCell.Bounds.Intersects(this.Hostiles[i].Bounds))
                            {
                                noIntersection = false;
                            }
                        }

                        if (PlayScreen.Instance.PlayableCharacter != null)
                        {
                            if (PlayScreen.Instance.PlayableCharacter.Bounds.Intersects(
                            foundCell.Bounds))
                                noIntersection = false;
                        }

                        if (noIntersection)
                        {
                            foundPosition = true;
                            position = foundCell.Position;
                        }
                    }
                }
            }

            return position;
        }

        /// <summary>
        /// Decorates the map
        /// </summary>
        public void DecorateMap()
        {
            for (int y = 0; y < this.Height; y++)
            {
                for (int x = 0; x < this.Width; x++)
                {
                    if (this.GetNumberOfAdjacentUnwalkableCells(x, y, 1, 1) >= 8)
                    {
                        this.Cells[x, y].Type = "wallblack";
                    }
                    else
                    {
                        if (this.Cells[x, y].Type == "wall")
                        {
                            if (this.GetLeftCell(x, y) != null && this.GetLeftCell(x, y).
                            IsWalkable == false &&
                                this.GetRightCell(x, y) != null && this.GetRightCell(x, y).
                                IsWalkable == false)
                            {
                                this.Cells[x, y].Type = "wallnoborders";

                                if (this.GetBottomCell(x, y) != null && this.GetBottomCell(x,
                                 y).IsWalkable == false)
                                    this.Cells[x, y].Type = "wallnobordersndb";
                            }
                            else
                            {
                                if (this.GetLeftCell(x, y) != null && this.GetLeftCell(x, y).
                                IsWalkable == false)
                                {
                                    this.Cells[x, y].Type = "wallnoleftborder";

                                    if (this.GetBottomCell(x, y) != null && this.
                                    GetBottomCell(x, y).IsWalkable == false)
                                        this.Cells[x, y].Type = "wallnoleftborderndb";
                                }
                                else
                                {
                                    if (this.GetRightCell(x, y) != null && this.GetRightCell(
                                    x, y).IsWalkable == false)
                                    {
                                        this.Cells[x, y].Type = "wallnorightborder";

                                        if (this.GetBottomCell(x, y) != null && this.
                                        GetBottomCell(x, y).IsWalkable == false)
                                            this.Cells[x, y].Type = "wallnorightborderndb";
                                    }
                                }
                            }
                        }
                        if (this.Cells[x, y].Type == "wall" && this.GetBottomCell(x, y) !=
                        null && this.GetBottomCell(x, y).IsWalkable == false)
                        {
                            this.Cells[x, y].Type = "wallndb";
                        }
                    }
                }
            }
        }

        /// <summary>
        /// Gets the top cell
        /// </summary>
        /// <param name="x">X position of the cell</param>
        /// <param name="y">Y postiion of the cell</param>
        /// <returns>Top cell</returns>
        public HCell GetTopCell(int x, int y)
        {
            if (this.IsCellOutOfBounds(x, y - 1))
                return null;
            return this.GetCell(x, y - 1);
        }

        /// <summary>
        /// Gets the bottom cell
        /// </summary>
        /// <param name="x">X position of the cell</param>
        /// <param name="y">Y postiion of the cell</param>
        /// <returns>Bottom cell</returns>
        public HCell GetBottomCell(int x, int y)
        {
            if (this.IsCellOutOfBounds(x, y + 1))
                return null;
            return this.GetCell(x, y + 1);
        }

        /// <summary>
        /// Gets the Left cell
        /// </summary>
        /// <param name="x">X position of the cell</param>
        /// <param name="y">Y postiion of the cell</param>
        /// <returns>Left cell</returns>
        public HCell GetLeftCell(int x, int y)
        {
            if (this.IsCellOutOfBounds(x - 1, y))
                return null;
            return this.GetCell(x - 1, y);
        }

        /// <summary>
        /// Gets the Right cell
        /// </summary>
        /// <param name="x">X position of the cell</param>
        /// <param name="y">Y postiion of the cell</param>
        /// <returns>Right cell</returns>
        public HCell GetRightCell(int x, int y)
        {
            if (this.IsCellOutOfBounds(x + 1, y))
                return null;
            return this.GetCell(x + 1, y);
        }
        #endregion

        #region PRIVATE METHODS

        /// <summary>
        /// Returns a bool depending on a given percentage
        /// </summary>
        /// <param name="percent">Percentage for it to be true</param>
        /// <returns>True or false depending on the given percentage</returns>
        private bool RandomPercent(int percent)
        {
            if (percent >= rand.Next(1, 101))
            {
                return true;
            }
            return false;
        }
        #endregion

        #region STATIC METHODS
        /// <summary>
        /// Save the cells in an XML file
        /// </summary>
        /// <param name="path">Path of the file</param>
        public static void SaveToXml(HMap map, string path)
        {
            XmlTextWriter writer = null;
            writer = new XmlTextWriter(path, UTF8Encoding.Default);
            writer.Formatting = Formatting.Indented;

            writer.WriteStartElement("Map");
            writer.WriteStartElement("Dimensions");
            writer.WriteElementString("Width", map.Width.ToString());
            writer.WriteElementString("Height", map.Height.ToString());
            writer.WriteEndElement();
            writer.WriteStartElement("Cells");

            for (int y = 0; y < map.Height; y++)
            {
                for (int x = 0; x < map.Width; x++)
                {
                    writer.WriteStartElement("Cell");
                    writer.WriteElementString("X", map.GetCell(x, y).Position.X.ToString());
                    writer.WriteElementString("Y", map.GetCell(x, y).Position.Y.ToString());
                    writer.WriteElementString("IsWalkable", map.GetCell(x, y).IsWalkable.
                    ToString());
                    writer.WriteElementString("Type", map.GetCell(x, y).Type);
                    writer.WriteEndElement();
                }
            }

            writer.WriteEndElement();
            writer.WriteEndElement();
            writer.Close();
        }

        /// <summary>
        /// Loads a map from an xml file
        /// </summary>
        /// <param name="path">Path of the file</param>
        /// <returns>Cells of the map</returns>
        public static HCell[,] LoadFromXml(string path)
        {
            XmlTextReader reader = new XmlTextReader(path);

            string currentElement = String.Empty;
            int w = 0;
            int h = 0;
            List<int> posXs = new List<int>();
            List<int> posYs = new List<int>();
            List<bool> isWalkables = new List<bool>();
            List<string> types = new List<string>();
            while (reader.Read())
            {
                switch (reader.NodeType)
                {
                    case XmlNodeType.Element:
                        currentElement = reader.Name;
                        break;
                    case XmlNodeType.Text:
                        if (currentElement == "Width")
                            w = Convert.ToInt32(reader.Value);
                        if (currentElement == "Height")
                            h = Convert.ToInt32(reader.Value);
                        if (currentElement == "X")
                            posXs.Add(Convert.ToInt32(reader.Value));
                        if (currentElement == "Y")
                            posYs.Add(Convert.ToInt32(reader.Value));
                        if (currentElement == "IsWalkable")
                            isWalkables.Add(Convert.ToBoolean(reader.Value));
                        if (currentElement == "Type")
                            types.Add(reader.Value);
                        break;
                    default:
                        break;
                }
            }

            HCell[,] cells = new HCell[w, h];

            for (int i = 0; i < w * h; i++)
            {
                int x = posXs[i];
                int y = posYs[i];
                bool isWalkable = isWalkables[i];
                string type = types[i];
                cells[x, y] = new HCell(isWalkable, new Vector2(x, y), type);
            }
            return cells;
        }
        #endregion
    }
}
```

```
/*
 * Author : Yannick R. Brodard
 * File name : HCell.cs
 * Version : 0.1.201504240835
 * Description : Cell class, for the map
 */

#region USING STATEMENTS
using HelProject.Tools;
using Microsoft.Xna.Framework;
using System;
using System.Xml.Serialization;
#endregion

namespace HelProject.GameWorld.Map
{
    /// <summary>
    /// Cell of a map
    /// </summary>
    public class HCell : HObject
    {
        public const int TILE_SIZE = 32;

        private FRectangle _bounds;
        private string _type;

        /// <summary>
        /// Bounds of the cells
        /// </summary>
        [XmlIgnore]
        public FRectangle Bounds
        {
            get { return _bounds; }
            set { _bounds = value; }
        }

        /// <summary>
        /// Type of the cell
        /// </summary>
        /// <remarks>
        /// Often corresponds with a texture
        /// </remarks>
        public string Type
        {
            get { return _type; }
            set { _type = value.ToLower(); }
        }

        #region CONSTRUCTORS
        /// <summary>
        /// Cell that represents a part of the map
        /// </summary>
        public HCell() : this(new Vector2(DEFAULT_POSITION_X_VALUE, DEFAULT_POSITION_Y_VALUE
        )) { /* no code... */ }

        /// <summary>
        /// Cell that represents a part of the map
        /// </summary>
        /// <param name="position">The position of the cell</param>
        /// <remarks>
        /// The cell position is rounded to the base digit.
        /// </remarks>
        public HCell(Vector2 position) : this(DEFAULT_IS_WALKABLE_VALUE, position) { /* no
        code... */ }

        /// <summary>
        /// Cell that represents a part of the map
        /// </summary>
        /// <param name="isWalkable">The cell can be 'walked' on by entities</param>
        /// <param name="position">The position of the cell</param>
        /// <remarks>
        /// The cell position is rounded to the base digit.
        /// </remarks>
        public HCell(bool isWalkable, Vector2 position) : this(isWalkable, position, String.
        Empty) { /* no code... */ }

        /// <summary>
        /// Cell that represents a part of the map
        /// </summary>
        /// <param name="isWalkable">The cell can be 'walked' on by entities</param>
        /// <param name="position">The position of the cell</param>
        /// <param name="type">Type of the cell</param>
        /// <remarks>
        /// The cell position is rounded to the base digit.
        /// The type of the often corresponds with a texture
        /// </remarks>
        public HCell(bool isWalkable, Vector2 position, string type)
        {
            this.IsWalkable = isWalkable;
            this.Position = new Vector2((int)position.X, (int)position.Y); // casted to
            integer, to only have round numbers for cells
            this.Bounds = new FRectangle(position.X, position.Y, 1f, 1f);

            if (type == String.Empty)
                this.Type = (isWalkable) ? "floorlava" : "wall";
            else
                this.Type = type;
        }
        #endregion
    }
}
```

```
/*
 * Author : Yannick R. Brodard
 * File name : HEntity.cs
 * Version : 0.2.201504240836
 * Description : Base abstact class for 'things' in the game world
 */

#region USING STATEMENTS
using HelProject.Tools;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
#endregion

namespace HelProject.GameWorld
{
    /// <summary>
    /// Abstract class for all entities of the game
    /// </summary>
    public abstract class HObject
    {
        protected const bool DEFAULT_IS_WALKABLE_VALUE = false;
        protected const float DEFAULT_POSITION_X_VALUE = 0.0f;
        protected const float DEFAULT_POSITION_Y_VALUE = 0.0f;

        #region ATTRIBUTES
        private Vector2 _position;
        private bool _isWalkable;             // true if the object can be walked on by
        enitites
        #endregion

        #region PROPRIETIES
        /// <summary>
        /// Position of the entity
        /// </summary>
        public Vector2 Position
        {
            get { return _position; }
            set { _position = value; }
        }

        /// <summary>
        /// Can be walked by entities
        /// </summary>
        public bool IsWalkable
        {
            get { return _isWalkable; }
            set { _isWalkable = value; }
        }
        #endregion

        #region CONSTRUCTORS
        /// <summary>
        /// Creates an object
        /// </summary>
        public HObject() : this(DEFAULT_IS_WALKABLE_VALUE, new Vector2(
        DEFAULT_POSITION_X_VALUE, DEFAULT_POSITION_Y_VALUE)) { /* no code... */ }
        /// <summary>
        /// Creates an object
        /// </summary>
        /// <param name="position">Position of the object</param>
        public HObject(Vector2 position) : this(DEFAULT_IS_WALKABLE_VALUE, position) { /* no
        code... */ }

        /// <summary>
        /// Creates an object
        /// </summary>
        /// <param name="isWalkable">Can the object be 'walked' on by entities</param>
        /// <param name="position">Position of the object</param>
        public HObject(bool isWalkable, Vector2 position)
        {
            this.IsWalkable = isWalkable;
            this.Position = position;
        }

        /// <summary>
        /// Override this to load content
        /// </summary>
        public virtual void LoadContent() { /* no code... */ }

        /// <summary>
        /// Override this to unload content
        /// </summary>
        public virtual void UnloadContent() { /* no code... */ }

        /// <summary>
        /// Override this to update object
        /// </summary>
        /// <param name="gameTime"></param>
        public virtual void Update(GameTime gameTime) { /* no code... */ }

        /// <summary>
        /// Override this to draw object
        /// </summary>
        /// <param name="spriteBatch"></param>
        public virtual void Draw(SpriteBatch spriteBatch) { /* no code... */ }
        #endregion
    }
}
```

```
/*
 * Author : Yannick R. Brodard
 * File name : HItem.cs
 * Version : 0.1.201505071037
 * Description : Class for the items
 */

using HelProject.Features;
using HelProject.Tools;
using HelProject.UI;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;

namespace HelProject.GameWorld
{
    /// <summary>
    /// Item class
    /// </summary>
    public class HItem : HObject
    {
        private const bool DEFAULT_ISONFLOOR_VALUE = true;

        private string _name;
        private ItemTypes _itemType;
        private bool _isOnFloor;
        private FeatureCollection _features;
        private string _imageName;
        private string _description;

        /// <summary>
        /// description or summary, or story, just additional content for the eyes
        /// </summary>
        public string Description
        {
            get { return _description; }
            set { _description = value; }
        }

        /// <summary>
        /// Name of the related texture2D
        /// </summary>
        public string ImageName
        {
            get { return _imageName; }
            set { _imageName = value; }
        }

        /// <summary>
        /// Name of the item
        /// </summary>
        public string Name
        {
            get { return _name; }
            set { _name = value; }
        }

        /// <summary>
        /// Type of the item
        /// </summary>
        public ItemTypes ItemType
        {
            get { return _itemType; }
            set { _itemType = value; }
        }

        /// <summary>
        /// Is the item on the floor
        /// </summary>
        public bool IsOnFloor
        {
            get { return _isOnFloor; }
            set { _isOnFloor = value; }
        }

        /// <summary>
        /// Given features of the item
        /// </summary>
        public FeatureCollection Features
        {
            get { return _features; }
            set { _features = value; }
        }

        /// <summary>
        /// Creates an empty item
        /// </summary>
        public HItem() : this("DEFAULT_ITEM", ItemTypes.Sword, new FeatureCollection(),
        "cursor_normal", false, Vector2.Zero, string.Empty) { /* no code... */ }

        /// <summary>
        /// Creates an item on the floor
        /// </summary>
        /// <param name="name">Name of the item</param>
        /// <param name="type">Type of the item</param>
        /// <param name="features">Given features of the item</param>
        /// <param name="imageName">Image name</param>
        /// <remarks>
        /// IsOnFloor == true
        /// </remarks>
        public HItem(string name, ItemTypes type, FeatureCollection features, string
        imageName) : this(name, type, features, imageName, DEFAULT_ISONFLOOR_VALUE, Vector2.
        Zero, string.Empty) { /* no code... */ }

        /// <summary>
        /// Creates an item
        /// </summary>
        /// <param name="name">Name of the item</param>
        /// <param name="type">Type of the item</param>
        /// <param name="features">Given features of the item</param>
        /// <param name="isOnFloor">Is the item on the floor</param>
        /// <param name="imageName">Image name</param>
        /// <param name="position">Position of the item (IG unit)</param>
        public HItem(string name, ItemTypes type, FeatureCollection features, string
        imageName, bool isOnFloor, Vector2 position) : this(name, type, features, imageName,
        isOnFloor, position, string.Empty) { /* no code... */ }

        /// <summary>
        /// Creates an item
        /// </summary>
        /// <param name="name">Name of the item</param>
        /// <param name="type">Type of the item</param>
        /// <param name="features">Given features of the item</param>
        /// <param name="isOnFloor">Is the item on the floor</param>
        /// <param name="imageName">Image name</param>
        /// <param name="position">Position of the item (IG unit)</param>
        /// <param name="description">Summary of the weapon</param>
        public HItem(string name, ItemTypes type, FeatureCollection features, string
        imageName, bool isOnFloor, Vector2 position, string description)
        {
            this.Name = name;
            this.ItemType = type;
            this.Features = features;
            this.IsOnFloor = isOnFloor;
            this.ImageName = imageName;
            this.Position = position;
            this.IsWalkable = true;
            this.Description = description;
        }
        /// <summary>
        /// Draws the item
        /// </summary>
        /// <param name="spriteBatch">Sprite batch</param>
        public override void Draw(SpriteBatch spriteBatch)
        {
            base.Draw(spriteBatch);
            Vector2 position = ScreenManager.Instance.GetCorrectScreenPosition(this.Position,
             PlayScreen.Instance.Camera.Position);
            spriteBatch.Draw(TextureManager.Instance.GetTexture(this.ImageName), position,
            null, null, null, 0.0f, new Vector2(1f, 1f), Color.White);
        }

        /// <summary>
        /// Item types
        /// </summary>
        /// <remarks>
        /// Weapons : 0 to 6,
        /// Accessories : 7 to 11,
        /// Armors : 12 to 17
        /// </remarks>
        public enum ItemTypes
        {
            // WEAPONS : 0 -> 6
            Sword = 0,
            TwoHandedSword = 1,
            Axe = 2,
            TwoHandedAxe = 3,
            Wand = 4,
            Staff = 5,
            Bow = 6,

            // ACCESSORIES : 7 -> 11
            Amulet = 7,
            Ring = 8,
            Shield = 9,
            Quiver = 10,
            PowerSource = 11,

            // ARMORS : 12 -> 17
            Head = 12,
            Shoulders = 13,
            Body = 14,
            Hands = 15,
            Legs = 16,
            Feet = 17
        }
    }
}
```

```
/*
 * Author : Yannick R. Brodard
 * File name : HUDManager.cs
 * Version : 0.1.201505191335
 * Description : Manager for the in-game HUD (Heads-up display)
 */

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace HelProject.UI.HUD
{
    /// <summary>
    /// Manager for the in-game HUD (Heads-up display)
    /// </summary>
    public class HUDManager
    {
        /* SINGLETON START */
        private static HUDManager _instance;

        /// <summary>
        /// Instance of the HUD
        /// </summary>
        public static HUDManager Instance
        {
            get
            {
                if (_instance == null)
                    _instance = new HUDManager();
                return _instance;
            }
        }

        /// <summary>
        /// Constructor
        /// </summary>
        private HUDManager() { /* no code... */ }
        /* SINGLETON END */

        private FillingBar _playerHealth;
        private FillingBar _playerMana;

        /// <summary>
        /// Filling bar for the player's health
        /// </summary>
        public FillingBar PlayerHealth
        {
            get { return _playerHealth; }
            set { _playerHealth = value; }
        }

        /// <summary>
        /// Filling bar for the player's mana
        /// </summary>
        public FillingBar PlayerMana
        {
            get { return _playerMana; }
            set { _playerMana = value; }
        }
    }
}
```

```
/*
 * Author : Yannick R. Brodard
 * File name : FillingBar.cs
 * Version : 0.1.201505191335
 * Description : Represents a filling bar
 */


using HelHelProject.Tools;
using HelProject.Tools;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using System;

namespace HelProject.UI.HUD
{
    /// <summary>
    /// Class of the filling bar
    /// </summary>
    public class FillingBar
    {
        private const float FILLER_MINIMUM = 0f;
        private const float FILLER_MAXIMUM = 100f;

        private FRectangle _container;
        private Color _borderColor;
        private Color _fillerColor;
        private Color _backgroundColor;
        private FillingDirection _movementDirection;
        private float _maxValue;
        private float _actualValue;

        /// <summary>
        /// Actual value of the filling
        /// </summary>
        public float ActualValue
        {
            get { return _actualValue; }
            set { _actualValue = value; }
        }

        /// <summary>
        /// Maximum value of the filling
        /// </summary>
        public float MaxValue
        {
            get { return _maxValue; }
            set { _maxValue = value; }
        }

        /// <summary>
        /// Filling percentage
        /// </summary>
        public float FillerPercentage
        {
            get { return this.ActualValue * 100f / this.MaxValue; }
        }
        /// <summary>
        /// Direction of the filling
        /// </summary>
        public FillingDirection MovementDirection
        {
            get { return _movementDirection; }
            set { _movementDirection = value; }
        }

        /// <summary>
        /// Container of the bar
        /// </summary>
        public FRectangle Container
        {
            get { return _container; }
            set
            {
                _container = value;
            }
        }

        /// <summary>
        /// Color of the border
        /// </summary>
        public Color BorderColor
        {
            get { return _borderColor; }
            set { _borderColor = value; }
        }

        /// <summary>
        /// Color of the filler
        /// </summary>
        public Color FillerColor
        {
            get { return _fillerColor; }
            set { _fillerColor = value; }
        }

        /// <summary>
        /// Color of the background
        /// </summary>
        public Color BackgroundColor
        {
            get { return _backgroundColor; }
            set { _backgroundColor = value; }
        }

        /// <summary>
        /// Creates a filling bar
        /// </summary>
        public FillingBar(FillingDirection fillingDirection, FRectangle rectangle, Color
        borderColor, Color fillerColor, Color backgroundColor,
            float maxValue = FILLER_MAXIMUM, float actualValue = FILLER_MAXIMUM, int
            borderThickness = 1)
        {
            this.MovementDirection = fillingDirection;
            this.Container = rectangle;
            this.BorderColor = borderColor;
            this.FillerColor = fillerColor;
            this.BackgroundColor = backgroundColor;
            this.MaxValue = maxValue;
            this.ActualValue = actualValue;
        }

        /// <summary>
        /// Draws the filling bar
        /// </summary>
        /// <param name="spriteBatch">Sprite batch</param>
        public void Draw(SpriteBatch spriteBatch)
        {
            Primitives2D.Instance.FillRectangle(spriteBatch, this.Container, this.
            BackgroundColor);
            Vector2 start = Vector2.Zero;
            Vector2 end = Vector2.Zero;
            int fillingPos = 0;
            switch (this.MovementDirection)
            {
                case FillingDirection.LeftToRight:
                    fillingPos = (int)(((this.FillerPercentage / 100f) * this.Container.Width
                    ) + this.Container.X);
                    start = new Vector2(this.Container.Position.X + 1, this.Container.
                    Position.Y + 1);
                    end = new Vector2(fillingPos, this.Container.Position.Y + this.Container.
                    Height - 1);
                    Primitives2D.Instance.FillRectangle(spriteBatch, start, end, this.
                    FillerColor);
                    break;
                case FillingDirection.RightToLeft:
                    fillingPos = (int)((this.Container.X + this.Container.Width) - ((this.
                    FillerPercentage / 100f) * this.Container.Width));
                    start = new Vector2(fillingPos, this.Container.Position.Y + 1);
                    end = new Vector2(this.Container.Position.Y + this.Container.Width - 1,
                    this.Container.Position.Y + this.Container.Height - 1);
                    Primitives2D.Instance.FillRectangle(spriteBatch, start, end, this.
                    FillerColor);
                    break;
                case FillingDirection.TopToBottom:
                    fillingPos = (int)(((this.FillerPercentage / 100f) * this.Container.
                    Height) + this.Container.Y);
                    start = new Vector2(this.Container.Position.X + 1, this.Container.
                    Position.Y + 1);
                    end = new Vector2(this.Container.Position.X + this.Container.Width - 1,
                    fillingPos);
                    Primitives2D.Instance.FillRectangle(spriteBatch, start, end, this.
                    FillerColor);
                    break;
                case FillingDirection.BottomToTop:
                    fillingPos = (int)((this.Container.Y + this.Container.Height) - ((this.
                    FillerPercentage / 100f) * this.Container.Height));
                    start = new Vector2(this.Container.Position.X + 1, fillingPos + 1);
                    end = new Vector2(this.Container.Position.X + this.Container.Width - 1,
                    this.Container.Position.Y + this.Container.Height - 1);
                    Primitives2D.Instance.FillRectangle(spriteBatch, start, end, this.
                    FillerColor);
                    break;
            }
            Primitives2D.Instance.DrawRectangle(spriteBatch, this.Container, this.BorderColor
            );
        }

        /// <summary>
        /// Direction of the filling
        /// </summary>
        public enum FillingDirection
        {
            LeftToRight,
            RightToLeft,
            TopToBottom,
            BottomToTop
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Linq;
using System.Text;
using System.Xml.Serialization;

namespace HelProject.UI.Menu
{
    public class MenuItem
    {
        private const LinkTypes DEFAULT_LINK_TYPE = LinkTypes.MENU;
        private const Image DEFAULT_IMAGE = null;

        private LinkTypes _linkType;
        private Image _itemImage;

        private int _id;

        public int Id
        {
            get { return _id; }
            set { _id = value; }
        }

        /// <summary>
        /// Image of the item
        /// </summary>
        [XmlElement("Image")]
        public Image ItemImage
        {
            get { return _itemImage; }
            set { _itemImage = value; }
        }

        /// <summary>
        /// Type of the link of the item
        /// </summary>
        //[XmlIgnore]
        public LinkTypes LinkType
        {
            get { return _linkType; }
            set { _linkType = value; }
        }

        /// <summary>
        /// Creates a menu item
        /// </summary>
        public MenuItem() : this(DEFAULT_LINK_TYPE, DEFAULT_IMAGE) { /* no code... */ }

        /// <summary>
        /// Creates a menu item
        /// </summary>
        /// <param name="linkType">Type of the link of the item</param>
        /// <param name="img">Image of the item</param>
        public MenuItem(LinkTypes linkType, Image img)
        {
            this.LinkType = linkType;
            this.ItemImage = img;
        }

        /// <summary>
        /// Types of links in the game
        /// </summary>
        public enum LinkTypes
        {
            [XmlEnum("0")]
            MENU = 0,
            [XmlEnum("1")]
            GAME = 1,
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace HelProject.UI.Menu
{
    public class MenuManager
    {
    }
}
```

```
/*
 * Author : Yannick R. Brodard
 * File name : MenuScreen.cs
 * Version : 0.1.201504241035
 * Description : Main menu screen of the game
 */

#region USING STATEMENTS
using HelProject.Tools;
using HelProject.UI.Menu;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Xml.Serialization;
#endregion

namespace HelProject.UI.Menu
{
    /// <summary>
    /// Title screen of the game
    /// </summary>
    public class MenuScreen : GameScreen
    {
        private const int DEFAULT_MENU_POS_X = 0;
        private const int DEFAULT_MENU_POS_Y = 0;
        private const int DEFAULT_HIGHLIGHT_INDEX = 0;

        private Point _menuPosition;
        private List<MenuItem> _items;
        private int _highlightIndex;
        private Image _backgroundImage;
        private Image _selectionIndicator;

        /// <summary>
        /// Background image of the screen
        /// </summary>
        [XmlElement("Image")]
        public Image BackgroundImage
        {
            get { return _backgroundImage; }
            set { _backgroundImage = value; }
        }

        /// <summary>
        /// Index of the highlighted item of the menu
        /// </summary>
        public int HighlightIndex
        {
            get { return _highlightIndex; }
            set { _highlightIndex = value; }
        }

        /// <summary>
        /// Position of the menu
        /// </summary>
        public Point MenuPosition
        {
            get { return _menuPosition; }
            set { _menuPosition = value; }
        }

        /// <summary>
        /// Items of the menu
        /// </summary>
        public List<MenuItem> Items
        {
            get { return _items; }
            set { _items = value; }
        }

        /// <summary>
        /// Creates an empty menu
        /// </summary>
        public MenuScreen() :
            this(new Point(DEFAULT_MENU_POS_X, DEFAULT_MENU_POS_Y), new List<MenuItem>()) {
            /* no code... */ }

        /// <summary>
        /// Creates a menu
        /// </summary>
        /// <param name="menuPosition">Position of the menu</param>
        /// <param name="menuItems">Items of the menu</param>
        public MenuScreen(Point menuPosition, List<MenuItem> menuItems)
            : this(menuPosition, menuItems, DEFAULT_HIGHLIGHT_INDEX) { /* no code... */ }

        public MenuScreen(Point menuPosition, List<MenuItem> menuItems, int highlightIndex)
        {
            this.MenuPosition = menuPosition;
            this.Items = menuItems;
            this.HighlightIndex = highlightIndex;
            this._selectionIndicator = new Image();
            this._selectionIndicator.ImagePath = "MenuScreen/selectorw";
        }

        /// <summary>
        /// Loads the content of the screen
        /// </summary>
        public override void LoadContent()
        {
            base.LoadContent();
            this.BackgroundImage.LoadContent();
            this._selectionIndicator.LoadContent();
            foreach (MenuItem itm in this.Items)
            {
                itm.ItemImage.LoadContent();
            }
        }

        /// <summary>
        /// Unloads the content of the screen
        /// </summary>
        public override void UnloadContent()
        {
            base.UnloadContent();
            this.BackgroundImage.UnloadContent();
            this._selectionIndicator.UnloadContent();
            foreach (MenuItem itm in this.Items)
            {
                itm.ItemImage.UnloadContent();
            }
        }

        /// <summary>
        /// Updates the content of the screen
        /// </summary>
        /// <param name="gameTime"></param>
        public override void Update(GameTime gameTime)
        {
            base.Update(gameTime);
            this.updateInputSelection(gameTime);
        }

        /// <summary>
        /// Draws the content of the screen
        /// </summary>
        /// <param name="spriteBatch"></param>
        public override void Draw(SpriteBatch spriteBatch)
        {
            this.BackgroundImage.Draw(spriteBatch);
            this.drawIndicator(spriteBatch);
        }

        /// <summary>
        /// Draws the indicator
        /// </summary>
        /// <param name="spriteBatch"></param>
        private void drawIndicator(SpriteBatch spriteBatch)
        {
            int i = 0;
            foreach (MenuItem itm in this.Items)
            {
                if (i == this.HighlightIndex)
                {
                    int totalOffSet = ((itm.ItemImage.SourceRect.Width / 2) + (this.
                    _selectionIndicator.SourceRect.Width / 2) + 10);

                    this._selectionIndicator.Scale = new Vector2(1, 1);
                    this._selectionIndicator.Position = new Vector2(
                        itm.ItemImage.Position.X - totalOffSet, itm.ItemImage.Position.Y);
                    this._selectionIndicator.Draw(spriteBatch);

                    this._selectionIndicator.Scale = new Vector2(-1, -1);
                    this._selectionIndicator.Position = new Vector2(
                        itm.ItemImage.Position.X + totalOffSet, itm.ItemImage.Position.Y);
                    this._selectionIndicator.Draw(spriteBatch);
                }
                itm.ItemImage.Draw(spriteBatch);
                i++;
            }
        }

        /// <summary>
        ///  Input management
        /// </summary>
        /// <param name="gameTime"></param>
        private void updateInputSelection(GameTime gameTime)
        {
            int itemNbr = this.Items.Count;

            if (InputManager.Instance.IsKeyboardKeyDown(Keys.Down))
            {
                this.HighlightIndex++;
                if (this.HighlightIndex >= itemNbr)
                    this.HighlightIndex = 0;
            }

            if (InputManager.Instance.IsKeyboardKeyReleased(Keys.Up))
            {
                this.HighlightIndex--;
                if (this.HighlightIndex < 0)
                    this.HighlightIndex = itemNbr - 1;
            }

            Point mousePosition = InputManager.Instance.MsState.Position;
            bool enterKeyDown = InputManager.Instance.IsKeyboardKeyDown(Keys.Enter);
            // initialisation to only create variable ONCE
            int i = 0; // Index
            int posX = 0; // position of the item
            int posY = 0; // position of the item
            int offSetX = 0; // Offset from the center
            int offSetY = 0; // Offset from the center
            foreach (MenuItem itm in this.Items)
            {
                /* MOUSE SELECTION */
                posX = (int)itm.ItemImage.Position.X;
                posY = (int)itm.ItemImage.Position.Y;
                offSetX = itm.ItemImage.SourceRect.Width / 2;
                offSetY = itm.ItemImage.SourceRect.Height / 2;

                if ((mousePosition.Y > (posY - offSetY)) &&
                    (mousePosition.Y < (posY + offSetY)) &&
                    (mousePosition.X > (posX - offSetX)) &&
                    (mousePosition.X < (posX + offSetX)))
                {
                    this.HighlightIndex = i;
                }
                updateInputEntries(gameTime, itm, enterKeyDown);
                /* END MOUSE SELECTION */
                i++;
            }
        }

        /// <summary>
        ///  Input management
        /// </summary>
        /// <param name="gameTime"></param>
        private void updateInputEntries(GameTime gameTime, MenuItem itm, bool enter)
        {
            if (enter)
            {
                if (itm.Id == 1 && this.HighlightIndex == 0)
                {
                    this.UnloadContent();
                    GameScreen ps = PlayScreen.Instance;
                    ScreenManager.Instance.Transition(ps);
                }
                else if (itm.Id == 3 && this.HighlightIndex == 2)
                {
                    MainGame.Instance.Exit();
                }
            }
        }
    }
}
```

```
/*
 * Author : Yannick R. Brodard
 * File name : GameScreen.cs
 * Version : 0.1.201504241035
 * Description : Is the base class for all the screens of the game
 */

#region USING STATEMENTS
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.Graphics;
using System;
using System.Xml.Serialization;
#endregion

namespace HelProject.UI
{
    /// <summary>
    /// Base class for all the screens of the game
    /// </summary>
    public abstract class GameScreen
    {
        [XmlIgnore]
        private Type _type;
        private ContentManager _content;

        /// <summary>
        /// Content of the gamescreen
        /// </summary>
        protected ContentManager Content
        {
            get { return _content; }
            set { _content = value; }
        }

        /// <summary>
        /// Type of the class
        /// </summary>
        [XmlIgnore]
        public Type TypeClass
        {
            get { return _type; }
            set { _type = value; }
        }

        /// <summary>
        /// Creates a game screen
        /// </summary>
        public GameScreen()
        {
            TypeClass = this.GetType(); // sets the type of the gamescreen
        }

        /// <summary>
        /// Loads the content of the screen
        /// </summary>
        public virtual void LoadContent()
        {
            this.Content = new ContentManager(ScreenManager.Instance.Content.ServiceProvider,
             "Content");
        }

        /// <summary>
        /// Unloads the content of the screen
        /// </summary>
        public virtual void UnloadContent()
        {
            Content.Unload();
        }

        /// <summary>
        /// Updates the content of the screen
        /// </summary>
        /// <param name="gameTime">Game time</param>
        public virtual void Update(GameTime gameTime) { /* no code... */ }

        /// <summary>
        /// Draws the content of the screen
        /// </summary>
        /// <param name="spriteBatch">Sprite batch</param>
        public virtual void Draw(SpriteBatch spriteBatch) { /* no code... */ }
    }
}
```

```
/*
 * Author : Yannick R. Brodard
 * File name : Image.cs
 * Version : 0.1.201504241405
 * Description : Image class, this manages all the images and text needed for the game
 */

#region USING STATEMENTS
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.Graphics;
using System;
using System.Collections.Generic;
using System.Xml.Serialization;
#endregion

namespace HelProject.UI
{
    public class Image
    {
        #region ATTRIBUTES FOR PROP
        private float _alphaChannel;
        private string _text, _fontName, _imagePath;
        private Vector2 _position, _scale;
        private Texture2D _texture;
        private Rectangle _sourceRect;
        private string _effects;
        private bool _isActive;
        #endregion

        #region ATTRIBUTES
        private Vector2 _origin;
        private ContentManager _content;
        private RenderTarget2D _renderTarget;
        private SpriteFont _font;
        #endregion

        #region PROPRIETIES
        /// <summary>
        /// Aplha channel for transparacy
        /// </summary>
        [XmlElement("Alpha")]
        public float AlphaChannel
        {
            get { return _alphaChannel; }
            set { _alphaChannel = value; }
        }

        /// <summary>
        /// Getter of the font
        /// </summary>
        public SpriteFont Font
        {
            get { return _font; }
        }

        /// <summary>
        /// Path of the image file
        /// </summary>
        [XmlElement("Path")]
        public string ImagePath
        {
            get { return _imagePath; }
            set { _imagePath = value; }
        }

        /// <summary>
        /// Name of the font
        /// </summary>
        public string FontName
        {
            get { return _fontName; }
            set { _fontName = value; }
        }

        /// <summary>
        /// Text for the image
        /// </summary>
        public string Text
        {
            get { return _text; }
            set { _text = value; }
        }

        /// <summary>
        /// Scale of the image
        /// </summary>
        public Vector2 Scale
        {
            get { return _scale; }
            set { _scale = value; }
        }

        /// <summary>
        /// Position of the image
        /// </summary>
        public Vector2 Position
        {
            get { return _position; }
            set { _position = value; }
        }

        /// <summary>
        /// Rectangle around the image
        /// </summary>
        public Rectangle SourceRect
        {
            get { return _sourceRect; }
            set { _sourceRect = value; }
        }

        /// <summary>
        /// Texture 2D, media for the image
        /// </summary>
        [XmlIgnore]
        public Texture2D Texture
        {
            get { return _texture; }
            set { _texture = value; }
        }

        /// <summary>
        /// Is the image active
        /// </summary>
        public bool IsActive
        {
            get { return _isActive; }
            set { _isActive = value; }
        }
        #endregion

        #region CONSTRUCTORS
        /// <summary>
        /// Makes an empty image
        /// </summary>
        public Image()
        {
            ImagePath = Text = String.Empty;
            FontName = "Lane";
            Position = Vector2.Zero;
            Scale = Vector2.One;
            AlphaChannel = 1.0f;
            SourceRect = Rectangle.Empty;
        }

        /// <summary>
        /// Effects of the image
        /// </summary>
        public string Effects
        {
            get { return _effects; }
            set { _effects = value; }
        }
        #endregion

        #region METHODS
        /// <summary>
        /// Loads the content of the image
        /// </summary>
        public void LoadContent()
        {
            _content = new ContentManager(ScreenManager.Instance.Content.ServiceProvider,
            "Content");

            // Gets the image if there is  one
            if (ImagePath != String.Empty)
            {
                this.Texture = this._content.Load<Texture2D>(ImagePath);
            }

            // loads the font
            this._font = _content.Load<SpriteFont>(FontName);

            Vector2 dimensions = Vector2.Zero;

            // Sets the width
            if (Texture != null)
                dimensions.X += Texture.Width;
            dimensions.X += _font.MeasureString(Text).X;

            // Sets the height
            if (Texture != null)
                dimensions.Y = Math.Max(Texture.Height, _font.MeasureString(Text).Y);
            else
                dimensions.Y = _font.MeasureString(Text).Y;

            // Creates the rectangle with the dimensions
            if (SourceRect == Rectangle.Empty)
            {
                SourceRect = new Rectangle(0, 0, (int)dimensions.X, (int)dimensions.Y);
            }

            /* Create the image */
            this._renderTarget = new RenderTarget2D(ScreenManager.Instance.SMGraphicsDevice,
            (int)dimensions.X, (int)dimensions.Y);
            ScreenManager.Instance.SMGraphicsDevice.SetRenderTarget(_renderTarget);
            ScreenManager.Instance.SMGraphicsDevice.Clear(Color.Transparent);
            ScreenManager.Instance.SMSpriteBatch.Begin();
            if (Texture != null)
                ScreenManager.Instance.SMSpriteBatch.Draw(Texture, Vector2.Zero, Color.White);
            ScreenManager.Instance.SMSpriteBatch.DrawString(_font, Text, Vector2.Zero, Color.
            White);
            ScreenManager.Instance.SMSpriteBatch.End();

            // Places the new image in the texture
            this.Texture = _renderTarget;

            // Gives back the render target to default
            ScreenManager.Instance.SMGraphicsDevice.SetRenderTarget(null);
        }

        /// <summary>
        /// Unloads the content of the image
        /// </summary>
        public void UnloadContent()
        {
            _content.Unload();
        }

        /// <summary>
        /// Updates the image
        /// </summary>
        /// <param name="gameTime"></param>
        public void Update(GameTime gameTime) { /* no code... */ }

        /// <summary>
        /// Draws the image
        /// </summary>
        /// <param name="spriteBatch"></param>
        public void Draw(SpriteBatch spriteBatch)
        {
            _origin = new Vector2(SourceRect.Width / 2, SourceRect.Height / 2);
            spriteBatch.Draw(Texture, Position, SourceRect, Color.White * AlphaChannel, 0.0f,
             _origin, Scale, SpriteEffects.None, 0.0f);
        }
        #endregion
    }
}
```

```
/*
 * Author : Yannick R. Brodard
 * File name : ScreenManager.cs
 * Version : 0.2.201504271045
 * Description : All the screens of the game are manage here
 */

#region USING STATEMENTS
using HelProject.Tools;
using HelProject.UI.Menu;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.Graphics;
#endregion

namespace HelProject.UI
{
    /// <summary>
    /// Singleton class, all the screens of the game are managed here
    /// </summary>
    public class ScreenManager
    {
        #region PROTECTED CONSTANTS
        protected const int DEFAULT_SCREEN_WIDTH = 1280;
        protected const int DEFAULT_SCREEN_HEIGHT = 720;
        protected const int DEFAULT_SPLASH_SCREEN_TIME = 3;
        #endregion

        #region TRANSITION PRIVATE VARIABLES
        private bool _transitionDelayActive;
        private int _transitionTime;
        private double _transitionFirstCount;
        private GameScreen _transitionScreen;
        #endregion

        #region PRIVATE VARIABLES
        private static ScreenManager _instance; // instance of this class
        private XmlManager<GameScreen> _xmlGameScreenManager; //xml manager for the screens
        private GameScreen _currentScreen; // current screen shown in the game
        #endregion

        #region PUBLIC VARIABLES
        public GraphicsDevice SMGraphicsDevice;
        public SpriteBatch SMSpriteBatch;
        #endregion

        #region PROPRIETIES
        /// <summary>
        /// Dimensions of the screen
        /// </summary>
        public Vector2 Dimensions { get; private set; }

        /// <summary>
        /// Content of the screen
        /// </summary>
        public ContentManager Content { get; private set; }
        /// <summary>
        /// Singleton instance of this class
        /// </summary>
        public static ScreenManager Instance
        {
            get
            {
                if (_instance == null)
                {
                    _instance = new ScreenManager();
                }
                return _instance;
            }
        }
        #endregion

        #region CONSTRUCTORS
        /// <summary>
        /// Creates a screen manager
        /// </summary>
        private ScreenManager()
        {
            // Initialisation
            this.Dimensions = new Vector2(DEFAULT_SCREEN_WIDTH, DEFAULT_SCREEN_HEIGHT); //
            fix the dim of the window
            this._transitionDelayActive = false; // init transition variables
            this._transitionTime = 0;
            this._transitionScreen = null;

            // shows the first splash screen
            this._currentScreen = this.PrepareScreen("Load/SplashScreen.xml", ScreenTypes.
            SPLASH);
            (this._currentScreen as SplashScreen).NextScreen = this.PrepareScreen(
            "Load/MenuScreen1.xml", ScreenTypes.MENU);
        }
        #endregion

        #region METHODS
        /// <summary>
        /// Loads the content
        /// </summary>
        /// <param name="content"></param>
        public void LoadContent(ContentManager content)
        {
            this.Content = new ContentManager(content.ServiceProvider, "Content");
            this._currentScreen.LoadContent();
        }

        /// <summary>
        /// Unloads the content
        /// </summary>
        public void UnloadContent()
        {
            this._currentScreen.UnloadContent();
        }

        /// <summary>
        /// Updates the content
        /// </summary>
        /// <param name="gameTime"></param>
        public void Update(GameTime gameTime)
        {
            this._currentScreen.Update(gameTime);

            // Transition mechanism
            if (this._transitionDelayActive)
            {
                double currentTime = gameTime.TotalGameTime.TotalSeconds; // gets the
                current time

                // initialise the first count if it's the first time it passes here
                if (this._transitionFirstCount < 0.0d)
                {
                    this._transitionFirstCount = currentTime;
                }
                else
                {
                    // calculates the time difference between the first count and the
                    current count
                    double diff = currentTime - (double)this._transitionFirstCount;

                    // if this time is superior or equal to the specified transition time
                    // the Transition method is called with the specified screen
                    if (diff >= (double)this._transitionTime)
                    {
                        this.Transition(this._transitionScreen);
                    }
                }
            }
        }

        /// <summary>
        /// Draws the content
        /// </summary>
        /// <param name="spriteBatch"></param>
        public void Draw(SpriteBatch spriteBatch)
        {
            _currentScreen.Draw(spriteBatch);
        }

        /// <summary>
        /// Transitions the screen to another one
        /// </summary>
        /// <param name="nextScreen"></param>
        public void Transition(GameScreen nextScreen)
        {
            // resets the transition variables
            this._transitionTime = 0;
            this._transitionDelayActive = false;
            this._transitionFirstCount = -1.0d;

            this.UnloadContent(); // unloads the content of the current screen
            this._currentScreen = nextScreen; // place the new screen
            this._currentScreen.LoadContent(); // loads the new screen
        }

        /// <summary>
        /// Activates a screen transition for the specified time
        /// </summary>
        /// <param name="nextScreen">Next screen that will appear</param>
        /// <param name="time">Time before transition</param>
        public void Transition(GameScreen nextScreen, int time)
        {
            this._transitionScreen = nextScreen;
            this._transitionTime = time;
            this._transitionDelayActive = true;
            this._transitionFirstCount = -1.0d;
        }

        /// <summary>
        /// Prepares an initialized screen
        /// </summary>
        /// <param name="loadContent">Path to the XML file for the initialization
        information</param>
        /// <param name="screenType">Type of the screen</param>
        /// <returns>The prepared screen</returns>
        public GameScreen PrepareScreen(string loadContent, ScreenTypes screenType)
        {
            GameScreen preparedScreen;

            switch (screenType)
            {
                case ScreenTypes.SPLASH:
                    preparedScreen = new SplashScreen();
                    break;
                case ScreenTypes.MENU:
                    preparedScreen = new MenuScreen();
                    break;
                case ScreenTypes.INGAME:
                    preparedScreen = new SplashScreen();
                    break;
                case ScreenTypes.LOADING:
                    preparedScreen = new SplashScreen();
                    break;
                default:
                    preparedScreen = new SplashScreen();
                    break;
            }

            this._xmlGameScreenManager = new XmlManager<GameScreen>();
            this._xmlGameScreenManager.TypeClass = preparedScreen.TypeClass;
            preparedScreen = _xmlGameScreenManager.Load(loadContent);

            return preparedScreen;
        }

        /// <summary>
        /// Gives the current screen type of the game
        /// </summary>
        /// <returns></returns>
        public ScreenTypes GetCurrentScreenType()
        {
            if (this._currentScreen is SplashScreen)
            {
                return ScreenTypes.SPLASH;
            }
            else if (this._currentScreen is MenuScreen)
            {
                return ScreenTypes.MENU;
            }
            else
            {
                return ScreenTypes.LOADING;
            }
        }

        /// <summary>
        /// Gets the correct position on the screen
        /// </summary>
        /// <param name="pos">Position of the object</param>
        /// <param name="tileSize">Size of a tile</param>
        /// <param name="scale">Scale</param>
        /// <returns>On screen position</returns>
        public Vector2 GetCorrectScreenPosition(Vector2 pos, Vector2 cameraPostion, int
        tileSize = HelProject.GameWorld.Map.HCell.TILE_SIZE, float scale = 1.0f)
        {
            float offSetX = 0f, offSetY = 0f;
            offSetX = -cameraPostion.X;
            offSetY = -cameraPostion.Y;

            return new Vector2(pos.X * (float)tileSize * scale + // X Pos
                               offSetX * (float)tileSize * scale +
                               this.Dimensions.X / 2f,
                               pos.Y * (float)tileSize * scale + // Y Pos
                               offSetY * (float)tileSize * scale +
                               this.Dimensions.Y / 2f);
        }
        #endregion

        #region PUBLIC ENUMERATORS
        /// <summary>
        /// Available screen types
        /// </summary>
        public enum ScreenTypes
        {
            SPLASH, // screen with an image
            MENU,   // screen with an interactive menu
            INGAME, // screen with integrated gameplay
            LOADING // screen used for loading moments
        };
        #endregion
    }
}
```

```
using Microsoft.Xna.Framework.Graphics;
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;
using System.Xml.Serialization;

namespace HelProject.UI
{
    /// <summary>
    /// Gets all the textures
    /// </summary>
    public class TextureManager
    {
        private static TextureManager _instance;
        private IDictionary<string, string> _texturesPaths;
        private IDictionary<string, Texture2D> _loadedTextures;

        private XmlSerializer _serializer;

        /// <summary>
        /// Path of all the textures
        /// </summary>
        /// <remarks>
        /// key = name of the texture
        /// value = path to texture
        /// </remarks>
        public IDictionary<string, string> TexturesPaths
        {
            get { return _texturesPaths; }
            set { _texturesPaths = value; }
        }

        /// <summary>
        /// Loaded textures
        /// </summary>
        public IDictionary<string, Texture2D> LoadedTextures
        {
            get { return _loadedTextures; }
            set { _loadedTextures = value; }
        }

        /// <summary>
        /// Instance of the class
        /// </summary>
        public static TextureManager Instance
        {
            get
            {
                if (_instance == null)
                    _instance = new TextureManager();
                return _instance;
            }
        }

        /// <summary>
        /// Private constructor
        /// </summary>
        private TextureManager()
        {
            this._texturesPaths = new Dictionary<string, string>();
            this._loadedTextures = new Dictionary<string, Texture2D>();
            this._serializer = new XmlSerializer(typeof(TemporaryDictionnaryItem[]), new
            XmlRootAttribute() { ElementName = "items" });
        }

        /// <summary>
        /// Loads all the textures paths from the given file
        /// </summary>
        /// <param name="path">Path of the initialization file (.xml)</param>
        public void Load(string path)
        {
            using (TextReader reader = new StreamReader(path))
            {
                this._texturesPaths = ((TemporaryDictionnaryItem[])this._serializer.
                Deserialize(reader)).ToDictionary(i => i.id, i => i.path);
            }

            foreach (KeyValuePair<string, string> entry in this._texturesPaths)
            {
                this._loadedTextures.Add(entry.Key, MainGame.Instance.Content.Load<Texture2D
                >(entry.Value));
            }

        }

        /// <summary>
        /// Save all the textures paths to the given location
        /// </summary>
        /// <param name="path">Path of the location/file</param>
        public void Save(string path)
        {
            using (TextWriter writer = new StreamWriter(path))
            {
                this._serializer.Serialize(writer, this._texturesPaths.Select(kv => new
                TemporaryDictionnaryItem() { id = kv.Key, path = kv.Value }).ToArray());
            }
        }

        /// <summary>
        /// Gets the texture by the key
        /// </summary>
        /// <param name="key"></param>
        public Texture2D GetTexture(string key)
        {
            Texture2D texture;

            if (this.LoadedTextures.ContainsKey(key))
            {
                texture = this.LoadedTextures[key];
            }
            else
            {
                texture = this.LoadedTextures["notexture"];
            }

            return texture;
        }
    }

    /// <summary>
    /// Class used for the serialization/deserialization of the TextureManager class
    /// </summary>
    public class TemporaryDictionnaryItem
    {
        [XmlAttribute]
        public string id;
        [XmlAttribute]
        public string path;
    }
}
```

```
/*
 * Author : Yannick R. Brodard
 * File name : Camera.cs
 * Version : 0.1.201505120855
 * Description : Camera class for the screen
 */

using HelProject.GameWorld.Map;
using HelProject.Tools;
using Microsoft.Xna.Framework;

namespace HelProject.UI
{
    /// <summary>
    /// Camera class for the screen
    /// </summary>
    public class Camera
    {
        private float _zoom;
        private Vector2 _position;
        private int _width;
        private int _height;

        /// <summary>
        /// Height of the camera
        /// </summary>
        /// <remarks>
        /// Usually the height of the window of the game
        /// </remarks>
        public int Height
        {
            get { return _height; }
            set { _height = value; }
        }

        /// <summary>
        /// Width of the camera
        /// </summary>
        /// <remarks>
        /// Usually the width of the window of the game
        /// </remarks>
        public int Width
        {
            get { return _width; }
            set { _width = value; }
        }

        /// <summary>
        /// Gets the view port of the camera
        /// </summary>
        public Rectangle ViewPort
        {
            get
            {
                return new Rectangle((int)this.Position.X, (int)this.Position.Y,
                                     (int)this.Position.X + this.Width, (int)this.Position.Y
                                     + this.Height);
            }
        }

        /// <summary>
        /// Position of the camera, relative to the map
        /// </summary>
        public Vector2 Position
        {
            get { return _position; }
            set { _position = value; }
        }

        /// <summary>
        /// Zoom effect
        /// </summary>
        public float Zoom
        {
            get { return _zoom; }
            set { _zoom = value; }
        }

        /// <summary>
        /// Create a camera
        /// </summary>
        /// <param name="position">Position of the camera</param>
        /// <param name="width">Width of the camera</param>
        /// <param name="height">Height of the camera</param>
        /// <param name="zoom">Zoom of the camera</param>
        public Camera(Vector2 position, int width, int height, float zoom = 1.0f)
        {
            this.Position = position;
            this.Width = width;
            this.Height = height;
            this.Zoom = zoom;
        }

        /// <summary>
        /// Gets the current mouse position relative to the map
        /// </summary>
        /// <returns>Mouse position relative to the map</returns>
        public Vector2 GetMousePositionRelativeToMap()
        {
            Vector2 pos = InputManager.Instance.MsState.Position.ToVector2();
            Vector2 firstCellPos = ScreenManager.Instance.GetCorrectScreenPosition(PlayScreen
            .Instance.CurrentMap.Cells[0, 0].Position, this.Position) / (float)HCell.
            TILE_SIZE;
            float offSetX = -firstCellPos.X;
            float offSetY = -firstCellPos.Y;

            pos /= (float)HCell.TILE_SIZE;

            pos.X += offSetX;
            pos.Y += offSetY;

            return pos;
        }
    }
```

```
}
```

```
using HelProject.GameWorld;
using Microsoft.Xna.Framework;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace HelProject.UI
{
    public class SelectionAid
    {
        private List<HObject> _selectedObjects;
        private PlayScreen playScreen;

        /// <summary>
        /// Selected objects
        /// </summary>
        public List<HObject> SelectedObjects
        {
            get { return _selectedObjects; }
            set { _selectedObjects = value; }
        }

        /// <summary>
        /// Creates a selection aider
        /// </summary>
        public SelectionAid()
        {
            this.SelectedObjects = new List<HObject>();
            playScreen = PlayScreen.Instance;
        }

        public void Update(GameTime gameTime)
        {
            this.SelectedObjects.Clear();

            int nbHostiles = playScreen.CurrentMap.Hostiles.Count;
            for (int i = 0; i < nbHostiles; i++)
            {
                if (playScreen.CurrentMap.Hostiles[i].Bounds.Intersects(playScreen.Camera.
                GetMousePositionRelativeToMap()))
                {
                    this.SelectedObjects.Add(playScreen.CurrentMap.Hostiles[i]);
                }
            }
        }
    }
}
```

```
/*
 * Author : Yannick R. Brodard
 * File name : PlayScreen.cs
 * Version : 0.3.201505120902
 * Description : Screen for the gameplay
 */

#region USING STATEMENTS
using HelHelProject.Tools;
using HelProject.Features;
using HelProject.GameWorld;
using HelProject.GameWorld.Entities;
using HelProject.GameWorld.Map;
using HelProject.Tools;
using HelProject.UI.HUD;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using System;
using System.Collections.Generic;
#endregion

namespace HelProject.UI
{
    /// <summary>
    /// Screen for the gameplay
    /// </summary>
    public class PlayScreen : GameScreen
    {
        private HMap _mapDifficultyEasy;
        private HMap _mapDifficultyMedium;
        private HMap _mapDifficultyHard;
        private HMap _mapTown;
        private HMap _currentMap;
        private HHero _hero;
        private static PlayScreen _instance;
        private Camera _camera;

        private SelectionAid _selectionAssistant;

        private SpriteFont font;

        /// <summary>
        /// Selection assistant for the game
        /// </summary>
        public SelectionAid SelectionAssistant
        {
            get { return _selectionAssistant; }
            set { _selectionAssistant = value; }
        }

        /// <summary>
        /// Starting point of the game : The town
        /// </summary>
        public HMap MapTown
        {
            get { return _mapTown; }
            set { _mapTown = value; }
        }

        /// <summary>
        /// Map of the game with easy difficulty
        /// </summary>
        public HMap MapDifficultyEasy
        {
            get { return _mapDifficultyEasy; }
            set { _mapDifficultyEasy = value; }
        }

        /// <summary>
        /// Map of the game with medium difficulty
        /// </summary>
        public HMap MapDifficultyMedium
        {
            get { return _mapDifficultyMedium; }
            set { _mapDifficultyMedium = value; }
        }

        /// <summary>
        /// Map of the game with hard difficulty
        /// </summary>
        public HMap MapDifficultyHard
        {
            get { return _mapDifficultyHard; }
            set { _mapDifficultyHard = value; }
        }

        /// <summary>
        /// Current map where the hero is
        /// </summary>
        public HMap CurrentMap
        {
            get { return _currentMap; }
            set { _currentMap = value; }
        }

        /// <summary>
        /// Playable character
        /// </summary>
        public HHero PlayableCharacter
        {
            get { return _hero; }
            set { _hero = value; }
        }

        /// <summary>
        /// Instance of the play screen
        /// </summary>
        /// <remarks>
        /// This is a singleton class
        /// </remarks>
        public static PlayScreen Instance
        {
            get
            {
                if (_instance == null)
                    _instance = new PlayScreen();
                return _instance;
            }
        }

        /// <summary>
        /// Camera of the play screen
        /// </summary>
        public Camera Camera
        {
            get { return _camera; }
            set { _camera = value; }
        }

        /// <summary>
        /// Private constructor
        /// </summary>
        private PlayScreen() { /* no code... */ }

        /// <summary>
        /// Loads the content of the window
        /// </summary>
        public override void LoadContent()
        {
            base.LoadContent();
            font = Content.Load<SpriteFont>("Lane");

            this.LoadMaps();
            this.LoadPlayableCharacter();
            this.LoadHostiles();
            this.SelectionAssistant = new SelectionAid();

            // Camera initialisation, gets the width and height of the window
            // and the position of the hero
            this.Camera = new Camera(this.PlayableCharacter.Position, MainGame.Instance.
            GraphicsDevice.Viewport.Width, MainGame.Instance.GraphicsDevice.Viewport.Height);

            XmlManager<HItem> item = new XmlManager<HItem>();
            item.TypeClass = typeof(HItem);
            FeatureCollection itFeatures = new FeatureCollection();
            itFeatures.SetAllToZero();
            itFeatures.InitialAttackSpeed = 1.4f;
            itFeatures.MinimumDamage = 15.0f;
            itFeatures.MaximumDamage = 24.0f;
            itFeatures.Strenght = 6.0f;
            itFeatures.Vitality = 3.0f;
            itFeatures.AttackSpeed = 15.0f;
            HItem it = new HItem("Death blade", HItem.ItemTypes.Sword, itFeatures,
            "cursor_normal", true, new Vector2(50f, 50f), "Blade of the death maiden");

            item.Save(Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments) +
            "\\item.xml", it);

            this.MapDifficultyEasy.OnFloorItems.Add(it);
        }
        /// <summary>
        /// Unloads the content of the window
        /// </summary>
        public override void UnloadContent()
        {
            base.UnloadContent();

            this.UnloadMaps();
            this.PlayableCharacter.UnloadContent();
        }

        /// <summary>
        /// Updates the mechanisms
        /// </summary>
        /// <param name="gameTime"></param>
        public override void Update(GameTime gameTime)
        {
            this.UpdateMapControl();
            this.PlayableCharacter.Update(gameTime);
            this.UpdateHostiles(gameTime);
            this.ManageDeadEntities();
            this.SelectionAssistant.Update(gameTime);
        }

        /// <summary>
        /// Draws on the window
        /// </summary>
        /// <param name="spriteBatch"></param>
        public override void Draw(SpriteBatch spriteBatch)
        {
            this.CurrentMap.Draw(spriteBatch, this.Camera);
            this.PlayableCharacter.Draw(spriteBatch);
            this.DrawHostiles(spriteBatch);
            this.DrawSelection(spriteBatch);

            if (MainGame.DEBUG_MODE)
            {
                Primitives2D.Instance.FillRectangle(spriteBatch, 0, 0, 200, 150, new Color(
                Color.LightBlue, 0.5f));
                Primitives2D.Instance.DrawRectangle(spriteBatch, 0, 0, 200, 150, Color.Black,
                 5);

                spriteBatch.DrawString(font, "Camera position (IG unit)", new Vector2(10, 10
                ), Color.Black);
                spriteBatch.DrawString(font, "X => " + Camera.Position.X.ToString(), new
                Vector2(15, 30), Color.Black);
                spriteBatch.DrawString(font, "Y => " + Camera.Position.Y.ToString(), new
                Vector2(15, 50), Color.Black);
                spriteBatch.DrawString(font, "Mouse position (IG unit)", new Vector2(10, 80),
                 Color.Black);
                spriteBatch.DrawString(font, "X => " + Camera.GetMousePositionRelativeToMap
                ().X, new Vector2(15, 100), Color.Black);
                spriteBatch.DrawString(font, "Y => " + Camera.GetMousePositionRelativeToMap
                ().Y.ToString(), new Vector2(15, 120), Color.Black);
            }
        }
        /// <summary>
        /// Map initialization
        /// </summary>
        private void LoadMaps()
        {
            this.MapTown = new HMap(HMap.LoadFromXml("Load/MapTown.xml"));
            this.MapTown.LoadContent();
            this.MapTown.DecorateMap();

            this.MapDifficultyEasy = new HMap(125, 125, 1f);
            this.MapDifficultyEasy.MakeCaverns();
            this.MapDifficultyEasy.LoadContent();
            this.MapDifficultyEasy.DecorateMap();

            this.MapDifficultyMedium = new HMap(125, 125, 1f);
            this.MapDifficultyMedium.MakeCaverns();
            this.MapDifficultyMedium.LoadContent();
            this.MapDifficultyMedium.DecorateMap();

            this.MapDifficultyHard = new HMap(125, 125, 1f);
            this.MapDifficultyHard.MakeCaverns();
            this.MapDifficultyHard.LoadContent();
            this.MapDifficultyHard.DecorateMap();

            this.CurrentMap = this.MapTown;
        }

        /// <summary>
        /// Loads the hostiles on the different maps
        /// </summary>
        private void LoadHostiles()
        {
            FeatureCollection f = new FeatureCollection();
            f.SetToDraugrLvlOne();

            for (int i = 0; i < 75; i++)
            {
                this.MapDifficultyEasy.Hostiles.Add(new HHostile(f, this.MapDifficultyEasy.
                GetRandomFloorPoint(), 1f, 1.5f, "draugr"));
            }

            for (int i = 0; i < 175; i++)
            {
                this.MapDifficultyMedium.Hostiles.Add(new HHostile(f, this.
                MapDifficultyMedium.GetRandomFloorPoint(), 1f, 1.5f, "draugr"));
            }

            for (int i = 0; i < 250; i++)
            {
                this.MapDifficultyHard.Hostiles.Add(new HHostile(f, this.MapDifficultyHard.
                GetRandomFloorPoint(), 1f, 1.5f, "draugr"));
            }
        }

        /// <summary>
        /// Draws the hostiles on the current map
        /// </summary>
        /// <param name="sb">Sprite batch</param>
        private void DrawHostiles(SpriteBatch sb)
        {
            List<HHostile> hostiles = this.CurrentMap.Hostiles;
            int nbrHostiles = hostiles.Count;

            for (int i = 0; i < nbrHostiles; i++)
            {
                hostiles[i].Draw(sb);
            }
        }

        /// <summary>
        /// Updates the mechanismes of the hostiles
        /// </summary>
        /// <param name="gameTime">Game time</param>
        private void UpdateHostiles(GameTime gameTime)
        {
            List<HHostile> hostiles = this.CurrentMap.Hostiles;
            int nbrHostiles = hostiles.Count;

            for (int i = 0; i < nbrHostiles; i++)
            {
                hostiles[i].Update(gameTime);
            }
        }

        /// <summary>
        /// Removes dead hostiles
        /// </summary>
        private void ManageDeadEntities()
        {
            List<HHostile> hostiles = this.CurrentMap.Hostiles;
            List<HHostile> hostilesToDestroy = new List<HHostile>();
            int nbrHostiles = hostiles.Count;

            for (int i = 0; i < nbrHostiles; i++)
            {
                if (hostiles[i].IsDead)
                    hostilesToDestroy.Add(hostiles[i]);
            }

            int nbrHostilesToDestroy = hostilesToDestroy.Count;
            for (int i = 0; i < nbrHostilesToDestroy; i++)
            {
                hostilesToDestroy[i].UnloadContent();
                hostiles.Remove(hostilesToDestroy[i]);
            }

            if (this.PlayableCharacter.IsDead)
            {
                PlayScreen.Instance.TransitionToMap(PlayScreen.Instance.MapTown);
                this.PlayableCharacter.ActualFeatures.LifePoints = this.PlayableCharacter.
                MaximizedFeatures.LifePoints;
                this.PlayableCharacter.IsDead = false;
            }
        }

        /// <summary>
        /// Unloads the maps
        /// </summary>
        private void UnloadMaps()
        {
            this.CurrentMap = null;
            this.MapDifficultyEasy.UnloadContent();
            this.MapDifficultyMedium.UnloadContent();
            this.MapDifficultyHard.UnloadContent();
            this.MapTown.UnloadContent();
        }

        /// <summary>
        /// Draws the selection of the mouse
        /// </summary>
        /// <param name="sb">Sprite batch</param>
        private void DrawSelection(SpriteBatch sb)
        {
            int nbObjects = this.SelectionAssistant.SelectedObjects.Count;
            for (int i = 0; i < nbObjects; i++)
            {
                if (this.SelectionAssistant.SelectedObjects[i] is HHostile)
                {
                    HHostile hostile = this.SelectionAssistant.SelectedObjects[i] as HHostile;
                    Vector2 start = ScreenManager.Instance.GetCorrectScreenPosition(hostile.
                    Bounds.Position, this.Camera.Position);
                    Vector2 end = ScreenManager.Instance.GetCorrectScreenPosition(new Vector2
                    (hostile.Bounds.Position.X + hostile.Bounds.Width, hostile.Bounds.
                    Position.Y + hostile.Bounds.Height), this.Camera.Position);
                    end.X += 1f;
                    end.Y += 1f;
                    Primitives2D.Instance.DrawRectangle(sb, start, end, Color.Yellow, 2);
                }
            }
        }

        /// <summary>
        /// Loads the playable character
        /// </summary>
        private void LoadPlayableCharacter()
        {
            FeatureCollection f = new FeatureCollection()
            {
                Strenght = HEntity.DEFAULT_STRENGHT,
                Vitality = HEntity.DEFAULT_VITALITY,
                Agility = HEntity.DEFAULT_AGILITY,
                Magic = HEntity.DEFAULT_MAGIC,
                InitialAttackSpeed = HEntity.DEFAULT_ATTACKSPEED,
                MinimumDamage = HEntity.DEFAULT_MINUMUMDAMAGE,
                MaximumDamage = HEntity.DEFAULT_MAXIMUMDAMAGE,
                InitialManaRegeneration = HEntity.DEFAULT_MANAREGENERATION,
                InitialMovementSpeed = HEntity.DEFAULT_MOVEMENTSPEED,
                InitialLifePoints = HEntity.DEFAULT_LIFEPOINTS
            };
            Vector2 position = this.CurrentMap.GetRandomFloorPoint();
            this.PlayableCharacter = new HHero(f, position, 1f, 1.5f, "hero");
            this.PlayableCharacter.LoadContent();
        }

        /// <summary>
        /// Update method to update the map control mechanisms
        /// </summary>
        private void UpdateMapControl()
        {
            if (InputManager.Instance.IsKeyboardKeyDown(Keys.F8))
            {
                this.CurrentMap.MakeRandomlyFilledMap();
                this.CurrentMap.MakeCaverns();
                this.CurrentMap.DecorateMap();
            }

            if (InputManager.Instance.IsKeyboardKeyDown(Keys.F9))
            {
                if (this.CurrentMap == this.MapDifficultyEasy)
                    TransitionToMap(this.MapDifficultyMedium);
                else if (this.CurrentMap == this.MapDifficultyMedium)
                    TransitionToMap(this.MapDifficultyHard);
                else if (this.CurrentMap == this.MapDifficultyHard)
                    TransitionToMap(this.MapTown);
                else if (this.CurrentMap == this.MapTown)
                    TransitionToMap(this.MapDifficultyEasy);
            }
        }

        /// <summary>
        /// Transitions to another map
        /// </summary>
        /// <param name="map"></param>
        public void TransitionToMap(HMap map)
        {
            this.CurrentMap = map;
            this.PlayableCharacter.Position = this.CurrentMap.GetRandomFloorPoint();
            this.PlayableCharacter.Bounds.SetBoundsWithTexture(this.PlayableCharacter.
            Position, this.PlayableCharacter.Texture.Width, this.PlayableCharacter.Texture.
            Height);
            this.Camera.Position = this.PlayableCharacter.Position;
        }
    }
}
```

```
/*
 * Author : Yannick R. Brodard
 * File name : SplashScreen.cs
 * Version : 0.1.201504241035
 * Description : Fills the screen with an image
 */

#region USING STATEMENTS
using HelProject.Tools;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using System.Xml.Serialization;
#endregion

namespace HelProject.UI
{
    /// <summary>
    /// Splash screen
    /// </summary>
    public class SplashScreen : GameScreen
    {
        private Image _backgroundImage;
        private GameScreen _nextScreen;

        /// <summary>
        /// Next screen after the splash screen
        /// </summary>
        public GameScreen NextScreen
        {
            get { return _nextScreen; }
            set { _nextScreen = value; }
        }

        /// <summary>
        /// Background Image of the splashscreen
        /// </summary>
        [XmlElement("Image")]
        public Image BackgroundImage
        {
            get { return _backgroundImage; }
            set { _backgroundImage = value; }
        }

        /// <summary>
        /// Loads the content of the screen
        /// </summary>
        public override void LoadContent()
        {
            base.LoadContent();
            BackgroundImage.LoadContent();
        }

        /// <summary>
        /// Unloads the content of the screen
        /// </summary>
        public override void UnloadContent()
        {
            base.UnloadContent();
            BackgroundImage.UnloadContent();
        }

        /// <summary>
        /// Updates the content screen
        /// </summary>
        /// <param name="gameTime"></param>
        public override void Update(GameTime gameTime)
        {
            base.Update(gameTime);
            BackgroundImage.Update(gameTime);
            if (InputManager.Instance.ReleasedKeys.Count > 0 ||
                InputManager.Instance.MsState.LeftButton == ButtonState.Pressed ||
                InputManager.Instance.MsState.RightButton == ButtonState.Pressed ||
                InputManager.Instance.MsState.MiddleButton == ButtonState.Pressed)
            {
                if (this.NextScreen == null)
                {
                    this.NextScreen = new SplashScreen();
                    this.NextScreen = ScreenManager.Instance.PrepareScreen(
                    "Load/MenuScreen1.xml",
                                                                           ScreenManager.
                                                                           ScreenTypes.SPLASH
                                                                           );
                }

                ScreenManager.Instance.Transition(this.NextScreen);
            }
        }

        /// <summary>
        /// Draws the content of the screen
        /// </summary>
        /// <param name="spriteBatch"></param>
        public override void Draw(SpriteBatch spriteBatch)
        {
            BackgroundImage.Draw(spriteBatch);
        }
    }
}
```

```
/*
 * Author : Yannick R. Brodard
 * File name : FeatureCollection.cs
 * Version : 0.1.201505041040
 * Description : Represents a collection of all possible features
 */

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace HelProject.Features
{
    public class FeatureCollection : Object, ICloneable
    {
        private float _initialAttackSpeed;
        private float _initialMovementSpeed;
        private float _initialManaRegeneration;
        private float _initialLifePoints;
        private float _strenght;
        private float _agility;
        private float _vitality;
        private float _magic;
        private float _attackSpeed;
        private float _minimumDamage;
        private float _maximumDamage;
        private float _minimumMagicDamage;
        private float _maximumMagicDamage;
        private float _armor;
        private float _magicResistance;
        private float _lifeRegeneration;
        private float _manaRegeneration;
        private float _movementSpeed;
        private float _lifePoints;

        /// <summary>
        /// Imposed life points
        /// </summary>
        public float InitialLifePoints
        {
            get { return _initialLifePoints; }
            set { _initialLifePoints = value; }
        }

        /// <summary>
        /// Imposed attack speed (attacks per second)
        /// </summary>
        public float InitialAttackSpeed
        {
            get { return _initialAttackSpeed; }
            set { _initialAttackSpeed = value; }
        }

        /// <summary>
        /// Imposed movement speed (meters / second)
        /// </summary>
        public float InitialMovementSpeed
        {
            get { return _initialMovementSpeed; }
            set { _initialMovementSpeed = value; }
        }

        /// <summary>
        /// Imposed mana regeneration (mana per second)
        /// </summary>
        public float InitialManaRegeneration
        {
            get { return _initialManaRegeneration; }
            set { _initialManaRegeneration = value; }
        }

        /// <summary>
        /// Strenght points
        /// </summary>
        public float Strenght
        {
            get { return _strenght; }
            set { _strenght = value; }
        }

        /// <summary>
        /// Agility points
        /// </summary>
        public float Agility
        {
            get { return _agility; }
            set { _agility = value; }
        }

        /// <summary>
        /// Vitality points
        /// </summary>
        public float Vitality
        {
            get { return _vitality; }
            set { _vitality = value; }
        }

        /// <summary>
        /// Magic points
        /// </summary>
        public float Magic
        {
            get { return _magic; }
            set { _magic = value; }
        }

        /// <summary>
        /// Attack speed buff percentage
        /// </summary>
        /// <example>25%</example>
        public float AttackSpeed
        {
            get { return _attackSpeed; }
            set { _attackSpeed = value; }
        }

        /// <summary>
        /// Minimum damage
        /// </summary>
        public float MinimumDamage
        {
            get { return _minimumDamage; }
            set { _minimumDamage = value; }
        }

        /// <summary>
        /// Maximum damage
        /// </summary>
        public float MaximumDamage
        {
            get { return _maximumDamage; }
            set { _maximumDamage = value; }
        }

        /// <summary>
        /// Minimum magic damage
        /// </summary>
        public float MinimumMagicDamage
        {
            get { return _minimumMagicDamage; }
            set { _minimumMagicDamage = value; }
        }

        /// <summary>
        /// Maximum magic damage
        /// </summary>
        public float MaximumMagicDamage
        {
            get { return _maximumMagicDamage; }
            set { _maximumMagicDamage = value; }
        }

        /// <summary>
        /// Armor points
        /// </summary>
        public float Armor
        {
            get { return _armor; }
            set { _armor = value; }
        }

        /// <summary>
        /// Magic resistance points
        /// </summary>
        public float MagicResistance
        {
            get { return _magicResistance; }
            set { _magicResistance = value; }
        }
        /// <summary>
        /// Life regeneration (life per second)
        /// </summary>
        public float LifeRegeneration
        {
            get { return _lifeRegeneration; }
            set { _lifeRegeneration = value; }
        }

        /// <summary>
        /// Mana regeneration (mana per second)
        /// </summary>
        public float ManaRegeneration
        {
            get { return _manaRegeneration; }
            set { _manaRegeneration = value; }
        }

        /// <summary>
        /// Movement speed buff (percentage)
        /// </summary>
        /// <example>25%</example>
        public float MovementSpeed
        {
            get { return _movementSpeed; }
            set { _movementSpeed = value; }
        }

        /// <summary>
        /// Life points
        /// </summary>
        public float LifePoints
        {
            get { return _lifePoints; }
            set { _lifePoints = value; }
        }

        /// <summary>
        /// Creates a feature collection
        /// </summary>
        public FeatureCollection()
        {
            this._agility = 0f;
            this._armor = 0f;
            this._attackSpeed = 0f;
            this._initialAttackSpeed = 0f;
            this._initialMovementSpeed = 0f;
            this._initialManaRegeneration = 0f;
            this._lifeRegeneration = 0f;
            this._magic = 0f;
            this._magicResistance = 0f;
            this._manaRegeneration = 0f;
            this._maximumDamage = 0f;
            this._maximumMagicDamage = 0f;
            this._minimumDamage = 0f;
            this._minimumMagicDamage = 0f;
            this._movementSpeed = 0f;
            this._strenght = 0f;
            this._vitality = 0f;
            this._initialLifePoints = 0f;
            this._lifePoints = 0f;
        }

        /// <summary>
        /// Clones the object
        /// </summary>
        /// <returns>object</returns>
        public object Clone()
        {
            FeatureCollection obj = new FeatureCollection();

            obj.InitialAttackSpeed = this.InitialAttackSpeed;
            obj.InitialLifePoints = this.InitialLifePoints;
            obj.InitialManaRegeneration = this.InitialManaRegeneration;
            obj.InitialMovementSpeed = this.InitialMovementSpeed;
            obj.LifePoints = this.LifePoints;
            obj.LifeRegeneration = this.LifeRegeneration;
            obj.Magic = this.Magic;
            obj.MagicResistance = this.MagicResistance;
            obj.ManaRegeneration = this.ManaRegeneration;
            obj.MaximumDamage = this.MaximumDamage;
            obj.MaximumMagicDamage = this.MaximumMagicDamage;
            obj.MinimumDamage = this.MinimumDamage;
            obj.MinimumMagicDamage = this.MinimumMagicDamage;
            obj.MovementSpeed = this.MovementSpeed;
            obj.Strenght = this.Strenght;
            obj.Vitality = this.Vitality;

            return obj;
        }

        /// <summary>
        /// Sets all features to zero
        /// </summary>
        public void SetAllToZero()
        {
            this._agility = 0f;
            this._armor = 0f;
            this._attackSpeed = 0f;
            this._initialAttackSpeed = 0f;
            this._initialMovementSpeed = 0f;
            this._initialManaRegeneration = 0f;
            this._lifeRegeneration = 0f;
            this._magic = 0f;
            this._magicResistance = 0f;
            this._manaRegeneration = 0f;
            this._maximumDamage = 0f;
            this._maximumMagicDamage = 0f;
            this._minimumDamage = 0f;
            this._minimumMagicDamage = 0f;
            this._movementSpeed = 0f;
            this._strenght = 0f;
            this._vitality = 0f;
            this._initialLifePoints = 0f;
            this._lifePoints = 0f;
        }

        /// <summary>
        /// Sets the features for a draugr level 1
        /// </summary>
        public void SetToDraugrLvlOne()
        {
            this._agility = 0f;
            this._armor = .0f;
            this._attackSpeed = 0f;
            this._initialAttackSpeed = 0.6f;
            this._initialMovementSpeed = 3.0f;
            this._initialManaRegeneration = .0f;
            this._lifeRegeneration = .0f;
            this._magic = 0f;
            this._magicResistance = .0f;
            this._manaRegeneration = 0f;
            this._maximumDamage = 3f;
            this._maximumMagicDamage = .0f;
            this._minimumDamage = 1f;
            this._minimumMagicDamage = .0f;
            this._movementSpeed = .0f;
            this._strenght = 0f;
            this._vitality = 0f;
            this._initialLifePoints = 30.0f;
            this._lifePoints = .0f;
        }
    }
}
```

```
/*
 * Author : Yannick R. Brodard
 * File name : FeatureManager.cs
 * Version : 0.1.201505071332
 * Description : Manages the calculation of the feature for the entities
 */

using HelProject.GameWorld;
using HelProject.GameWorld.Spells;
using System.Collections.Generic;

namespace HelProject.Features
{
    /// <summary>
    /// Manages the calculation of the feature for the entities
    /// </summary>
    public class FeatureManager
    {
        public const float LIFE_PER_VITALITY = 100.0f;

        private List<HItem> _activeItems;
        private List<HSpell> _activeSpells;
        private FeatureCollection _initialFeatures;

        /// <summary>
        /// Items worn by the hero
        /// </summary>
        public List<HItem> ActiveItems
        {
            get { return _activeItems; }
            set { _activeItems = value; }
        }

        /// <summary>
        /// Currently casted spells of the hero
        /// </summary>
        public List<HSpell> ActiveSpells
        {
            get { return _activeSpells; }
            set { _activeSpells = value; }
        }

        /// <summary>
        /// Initial features of the hero
        /// </summary>
        public FeatureCollection InitialFeatures
        {
            get { return _initialFeatures; }
            set { _initialFeatures = value; }
        }

        /// <summary>
        /// Creates a feature manager
        /// </summary>
        /// <param name="features">Initial features of the hero</param>
        /// <param name="spells">Active spells of the hero</param>
        /// <param name="items">Worn items of the hero</param>
        /// <remarks>Null values will be initialized (except the "features"
        parameter).</remarks>
        public FeatureManager(FeatureCollection features, List<HSpell> spells = null, List<
        HItem> items = null)
        {
            this.InitialFeatures = features;

            if (spells != null)
                this.ActiveSpells = spells;
            else
                this.ActiveSpells = new List<HSpell>();

            if (items != null)
                this.ActiveItems = items;
            else
                this.ActiveItems = new List<HItem>();
        }

        /// <summary>
        /// Gets the calculated features
        /// </summary>
        /// <returns>Feature collection</returns>
        /// <remarks>For movement speed, use the initial movement speed</remarks>
        public FeatureCollection GetCalculatedFeatures()
        {
            return new FeatureCollection()
            {
                Strenght = this.GetTotalStrenght(),
                Agility = this.GetTotalAgility(),
                Vitality = this.GetTotalVitality(),
                Magic = this.GetTotalMagic(),
                Armor = this.GetTotalArmor(),
                AttackSpeed = this.GetTotalAttackSpeed(),
                LifeRegeneration = this.GetTotalLifeRegeneration(),
                MagicResistance = this.GetTotalMagicResistance(),
                ManaRegeneration = this.GetTotalManaRegeneration(),
                MaximumDamage = this.GetTotalMaximumDamage(),
                MaximumMagicDamage = this.GetTotalMaximumMagicDamage(),
                MinimumDamage = this.GetTotalMinimumDamage(),
                MinimumMagicDamage = this.GetTotalMinimumMagicDamage(),
                InitialMovementSpeed = this.GetTotalMovementSpeed(),
                LifePoints = this.GetTotalLifePoints(),
            };
        }

        /// <summary>
        /// Calaculates the received damage
        /// </summary>
        /// <param name="damage">Damage given</param>
        /// <returns>Real received damage</returns>
        public float GetReceivedPhysicalDamage(float damage)
        {
            return damage - (this.GetTotalArmor() * 0.075f * damage);
        }

        /// <summary>
        /// Calculates the total strenght within the items, spells and initial values
        /// </summary>
        /// <returns>Total strenght</returns>
        public float GetTotalStrenght()
        {
            float str = this.InitialFeatures.Strenght;

            for (int i = 0; i < this.ActiveSpells.Count; i++)
            {
                str += this.ActiveSpells[i].Features.Strenght;
            }

            for (int i = 0; i < this.ActiveItems.Count; i++)
            {
                str += this.ActiveItems[i].Features.Strenght;
            }

            return str;
        }

        /// <summary>
        /// Calculates the total agility within the items, spells and initial values
        /// </summary>
        /// <returns>Total agility</returns>
        public float GetTotalAgility()
        {
            float agi = this.InitialFeatures.Agility;

            for (int i = 0; i < this.ActiveSpells.Count; i++)
            {
                agi += this.ActiveSpells[i].Features.Agility;
            }

            for (int i = 0; i < this.ActiveItems.Count; i++)
            {
                agi += this.ActiveItems[i].Features.Agility;
            }

            return agi;
        }

        /// <summary>
        /// Calculates the total Vitality within the items, spells and initial values
        /// </summary>
        /// <returns>Total Vitality</returns>
        public float GetTotalVitality()
        {
            float vit = this.InitialFeatures.Vitality;

            for (int i = 0; i < this.ActiveSpells.Count; i++)
            {
                vit += this.ActiveSpells[i].Features.Vitality;
            }

            for (int i = 0; i < this.ActiveItems.Count; i++)
            {
                vit += this.ActiveItems[i].Features.Vitality;
            }
            return vit;
        }

        /// <summary>
        /// Calculates the total Magic within the items, spells and initial values
        /// </summary>
        /// <returns>Total Magic</returns>
        public float GetTotalMagic()
        {
            float mag = this.InitialFeatures.Magic;

            for (int i = 0; i < this.ActiveSpells.Count; i++)
            {
                mag += this.ActiveSpells[i].Features.Magic;
            }

            for (int i = 0; i < this.ActiveItems.Count; i++)
            {
                mag += this.ActiveItems[i].Features.Magic;
            }

            return mag;
        }

        /// <summary>
        /// Calculates the total AttackSpeed within the items, spells and initial values
        /// </summary>
        /// <returns>Total AttackSpeed</returns>
        /// <remarks>Gets the highest initial attack speed and adds all the attack speed
        percentages</remarks>
        public float GetTotalAttackSpeed()
        {
            float attsp = this.InitialFeatures.InitialAttackSpeed;
            float totAttspBuff = this.InitialFeatures.AttackSpeed;

            for (int i = 0; i < this.ActiveSpells.Count; i++)
            {
                if (this.ActiveSpells[i].Features.InitialAttackSpeed > attsp)
                    attsp = this.ActiveSpells[i].Features.InitialAttackSpeed;

                totAttspBuff += this.ActiveSpells[i].Features.AttackSpeed;
            }

            for (int i = 0; i < this.ActiveItems.Count; i++)
            {
                if (this.ActiveItems[i].Features.InitialAttackSpeed > attsp)
                    attsp = this.ActiveItems[i].Features.InitialAttackSpeed;

                totAttspBuff += this.ActiveItems[i].Features.AttackSpeed;
            }

            return attsp * (totAttspBuff / 100.0f + 1.0f);
        }

        /// <summary>
        /// Calculates the total MinimumDamage within the items, spells and initial values
        /// </summary>
        /// <returns>Total MinimumDamage</returns>
        /// <remarks>Gets the total of minimum damage and adds 1% more per Strenght</remarks>
        public float GetTotalMinimumDamage()
        {
            float minDmg = this.InitialFeatures.MinimumDamage;

            for (int i = 0; i < this.ActiveSpells.Count; i++)
            {
                minDmg += this.ActiveSpells[i].Features.MinimumDamage;
            }

            for (int i = 0; i < this.ActiveItems.Count; i++)
            {
                minDmg += this.ActiveItems[i].Features.MinimumDamage;
            }

            return minDmg * (this.GetTotalStrenght() / 100.0f + 1.0f);
        }

        /// <summary>
        /// Calculates the total MaximumDamage within the items, spells and initial values
        /// </summary>
        /// <returns>Total MaximumDamage</returns>
        /// <remarks>Gets the total of maximum damage and adds 1% more per Strenght</remarks>
        public float GetTotalMaximumDamage()
        {
            float maxDmg = this.InitialFeatures.MaximumDamage;

            for (int i = 0; i < this.ActiveSpells.Count; i++)
            {
                maxDmg += this.ActiveSpells[i].Features.MaximumDamage;
            }

            for (int i = 0; i < this.ActiveItems.Count; i++)
            {
                maxDmg += this.ActiveItems[i].Features.MaximumDamage;
            }

            return maxDmg * (this.GetTotalStrenght() / 100.0f + 1.0f);
        }

        /// <summary>
        /// Calculates the total MinimumMagicDamage within the items, spells and initial
        values
        /// </summary>
        /// <returns>Total MinimumMagicDamage</returns>
        /// <remarks>Gets the total of minimum magic damage and adds 1% more per Magic
        point</remarks>
        public float GetTotalMinimumMagicDamage()
        {
            float minMagDmg = this.InitialFeatures.MinimumMagicDamage;

            for (int i = 0; i < this.ActiveSpells.Count; i++)
            {
                minMagDmg += this.ActiveSpells[i].Features.MinimumMagicDamage;
            }
            for (int i = 0; i < this.ActiveItems.Count; i++)
            {
                minMagDmg += this.ActiveItems[i].Features.MinimumMagicDamage;
            }

            return minMagDmg * (this.GetTotalMagic() / 100.0f + 1.0f);
        }

        /// <summary>
        /// Calculates the total MaximumMagicDamage within the items, spells and initial
        values
        /// </summary>
        /// <returns>Total MaximumMagicDamage</returns>
        /// <remarks>Gets the total of maximum magic damage and adds 1% more per Magic
        point</remarks>
        public float GetTotalMaximumMagicDamage()
        {
            float maxMagDmg = this.InitialFeatures.MaximumMagicDamage;

            for (int i = 0; i < this.ActiveSpells.Count; i++)
            {
                maxMagDmg += this.ActiveSpells[i].Features.MaximumMagicDamage;
            }

            for (int i = 0; i < this.ActiveItems.Count; i++)
            {
                maxMagDmg += this.ActiveItems[i].Features.MaximumMagicDamage;
            }

            return maxMagDmg * (this.GetTotalMagic() / 100.0f + 1.0f);
        }

        /// <summary>
        /// Calculates the total Armor within the items, spells and initial values
        /// </summary>
        /// <returns>Total Armor</returns>
        public float GetTotalArmor()
        {
            float arm = this.InitialFeatures.Armor;

            for (int i = 0; i < this.ActiveSpells.Count; i++)
            {
                arm += this.ActiveSpells[i].Features.Armor;
            }

            for (int i = 0; i < this.ActiveItems.Count; i++)
            {
                arm += this.ActiveItems[i].Features.Armor;
            }

            return arm;
        }

        /// <summary>
        /// Calculates the total MagicResistance within the items, spells and initial values
        /// </summary>
        /// <returns>Total MagicResistance</returns>
        public float GetTotalMagicResistance()
        {
            float magRes = this.InitialFeatures.MagicResistance;

            for (int i = 0; i < this.ActiveSpells.Count; i++)
            {
                magRes += this.ActiveSpells[i].Features.MagicResistance;
            }

            for (int i = 0; i < this.ActiveItems.Count; i++)
            {
                magRes += this.ActiveItems[i].Features.MagicResistance;
            }

            return magRes;
        }

        /// <summary>
        /// Calculates the total LifeRegeneration within the items, spells and initial values
        /// </summary>
        /// <returns>Total LifeRegeneration</returns>
        public float GetTotalLifeRegeneration()
        {
            float lifReg = this.InitialFeatures.LifeRegeneration;

            for (int i = 0; i < this.ActiveSpells.Count; i++)
            {
                lifReg += this.ActiveSpells[i].Features.LifeRegeneration;
            }

            for (int i = 0; i < this.ActiveItems.Count; i++)
            {
                lifReg += this.ActiveItems[i].Features.LifeRegeneration;
            }

            return lifReg;
        }

        /// <summary>
        /// Calculates the total ManaRegeneration within the items, spells and initial values
        /// </summary>
        /// <returns>Total ManaRegeneration</returns>
        public float GetTotalManaRegeneration()
        {
            float manReg = this.InitialFeatures.ManaRegeneration;

            for (int i = 0; i < this.ActiveSpells.Count; i++)
            {
                manReg += this.ActiveSpells[i].Features.ManaRegeneration;
            }

            for (int i = 0; i < this.ActiveItems.Count; i++)
            {
                manReg += this.ActiveItems[i].Features.ManaRegeneration;
            }
            return manReg;
        }

        /// <summary>
        /// Calculates the total MovementSpeed within the items, spells and initial values
        /// </summary>
        /// <returns>Total MovementSpeed</returns>
        /// <remarks>Gets the highest initial movement speed and adds all the movement speed percentages</remarks>
        public float GetTotalMovementSpeed()
        {
            float iniMovSp = this.InitialFeatures.InitialMovementSpeed;
            float totMovSpBuff = this.InitialFeatures.MovementSpeed;

            for (int i = 0; i < this.ActiveSpells.Count; i++)
            {
                if (this.ActiveSpells[i].Features.InitialMovementSpeed > iniMovSp)
                    iniMovSp = this.ActiveSpells[i].Features.InitialMovementSpeed;

                totMovSpBuff += this.ActiveSpells[i].Features.MovementSpeed;
            }

            for (int i = 0; i < this.ActiveItems.Count; i++)
            {
                if (this.ActiveItems[i].Features.InitialMovementSpeed > iniMovSp)
                    iniMovSp = this.ActiveItems[i].Features.InitialMovementSpeed;

                totMovSpBuff += this.ActiveItems[i].Features.MovementSpeed;
            }

            return iniMovSp * (totMovSpBuff / 100.0f + 1.0f);
        }

        /// <summary>
        /// Calculates the total LifePoints within the items, spells and initial values
        /// </summary>
        /// <returns>Total LifePoints</returns>
        public float GetTotalLifePoints()
        {
            float iniLifePoints = this.InitialFeatures.InitialLifePoints;

            for (int i = 0; i < this.ActiveSpells.Count; i++)
            {
                if (iniLifePoints < this.ActiveSpells[i].Features.InitialLifePoints)
                    iniLifePoints = this.ActiveSpells[i].Features.InitialLifePoints;
            }

            for (int i = 0; i < this.ActiveItems.Count; i++)
            {
                if (iniLifePoints < this.ActiveItems[i].Features.InitialLifePoints)
                    iniLifePoints = this.ActiveItems[i].Features.InitialLifePoints;
            }

            return this.GetTotalVitality() * LIFE_PER_VITALITY + iniLifePoints;
        }
    }
}
```

```
#region Using Statements
using System;
using System.Collections.Generic;
using System.Linq;
#endregion

namespace HelProject
{
#if WINDOWS || LINUX
    /// <summary>
    /// The main class.
    /// </summary>
    public static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            using (var game = MainGame.Instance)
                game.Run();
        }
    }
#endif
}
```

```
using HelHelProject.Tools;
using HelProject.Tools;
using HelProject.UI;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using System;

namespace HelProject
{
    /// <summary>
    /// This is the main type for your game
    /// </summary>
    public class MainGame : Game
    {
        public const bool DEBUG_MODE = true;

        private static MainGame _instance;
        private GraphicsDeviceManager _graphics;
        private SpriteBatch _spriteBatch;
        private Vector2 _cursorPosition;

        /// <summary>
        /// Instance of the Main Game
        /// </summary>
        public static MainGame Instance
        {
            get
            {
                if (_instance == null)
                    _instance = new MainGame();
                return _instance;
            }
        }

        /// <summary>
        /// private Constructor
        /// </summary>
        private MainGame()
            : base()
        {
            _graphics = new GraphicsDeviceManager(this);
            Content.RootDirectory = "Content";
        }

        /// <summary>
        /// Allows the game to perform any initialization it needs to before starting to run.
        /// This is where it can query for any required services and load any non-graphic
        /// related content. Calling base. Initialize will enumerate through any components
        /// and initialize them as well.
        /// </summary>
        protected override void Initialize()
        {
            // TODO: Add your initialization logic here
            //this.IsMouseVisible = true;

            _graphics.PreferredBackBufferWidth = (int)ScreenManager.Instance.Dimensions.X;
            _graphics.PreferredBackBufferHeight = (int)ScreenManager.Instance.Dimensions.Y;
            _graphics.ApplyChanges();

            TextureManager.Instance.Load("Load/Textures.xml");

            this._cursorPosition = new Vector2();
            this.Window.Title = "Hel: The pixelated horror";
            this.Window.Position = new Point(GraphicsDevice.DisplayMode.Width / 2 - (int)
            ScreenManager.Instance.Dimensions.X / 2,
                                             GraphicsDevice.DisplayMode.Height / 2 - (int)
                                             ScreenManager.Instance.Dimensions.Y / 2);

            base.Initialize();
        }

        /// <summary>
        /// LoadContent will be called once per game and is the place to load
        /// all of your content.
        /// </summary>
        protected override void LoadContent()
        {
            // Create a new SpriteBatch, which can be used to draw textures.
            _spriteBatch = new SpriteBatch(GraphicsDevice);

            // TODO: use this.Content to load your game content here
            Primitives2D.Instance.LoadContent();
            ScreenManager.Instance.SMGraphicsDevice = GraphicsDevice;
            ScreenManager.Instance.SMSpriteBatch = _spriteBatch;
            ScreenManager.Instance.LoadContent(Content);
        }

        /// <summary>
        /// UnloadContent will be called once per game and is the place to unload
        /// all content.
        /// </summary>
        protected override void UnloadContent()
        {
            // TODO: Unload any non ContentManager content here
            ScreenManager.Instance.UnloadContent();
        }

        /// <summary>
        /// Allows the game to run logic such as updating the world,
        /// checking for collisions, gathering input, and playing audio.
        /// </summary>
        /// <param name="gameTime">Provides a snapshot of timing values.</param>
        protected override void Update(GameTime gameTime)
        {
            //if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed ||
            Keyboard.GetState().IsKeyDown(Keys.Escape))
            //Exit();

            // TODO: Add your update logic here

            ScreenManager.Instance.Update(gameTime);
            InputManager.Instance.Update(gameTime);
            this._cursorPosition.X = InputManager.Instance.MsState.X;
            this._cursorPosition.Y = InputManager.Instance.MsState.Y;
            base.Update(gameTime);
        }

        /// <summary>
        /// This is called when the game should draw itself.
        /// </summary>
        /// <param name="gameTime">Provides a snapshot of timing values.</param>
        protected override void Draw(GameTime gameTime)
        {
            GraphicsDevice.Clear(Color.Black);

            // TODO: Add your drawing code here

            _spriteBatch.Begin();
            ScreenManager.Instance.Draw(_spriteBatch);
            if (this.IsActive)
                _spriteBatch.Draw(TextureManager.Instance.LoadedTextures["cursor_normal"],
                this._cursorPosition, Color.White);
            _spriteBatch.End();

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