|  |  |
| --- | --- |
|  |  |
| МИНОБРНАУКИ РОССИИ | | |
| Федеральное государственное бюджетное образовательное учреждение  высшего профессионального образования  **«Московский технологический университет»**  **МИРЭА** | | |

Институт Информационных Технологий

Кафедра Корпоративных Информационных Систем

**ОТЧЕТ**

по Лабораторной Работе №3

на тему

«Графы. Алгоритмы работы с графами»

по дисциплине

«ООП»

Выполнил студент группы ИСБО-11-16 Шайхуллин С.В.

Принял Cтарший преподаватель Мирзоян Д.И.

Выполнено «\_\_\_» \_\_\_\_\_\_\_\_\_2017 г.

Зачтено «\_\_\_» \_\_\_\_\_\_\_\_\_ 2017г.

Москва, 2017

**Теоретическая часть**

В математике, Граф — это абстрактное представление множества объектов и связей между ними. Графом называют пару (V, E) где V это множество вершин, а E множество пар, каждая из которых представляет собой связь (эти пары называют рёбрами).Граф может быть ориентированным или неориентированным. В ориентированном графе, связи являются направленными (то есть пары в E являются упорядоченными, например пары (a, b) и (b, a) это две разные связи). В свою очередь в неориентированном графе, связи ненаправленные, и поэтому если существует связь (a, b) то значит что существует связь (b, a).Степень вершины может быть входящая и исходящая (для неориентированных графов входящая степень равна исходящей).Входящая степень вершины v это количество ребер вида (i, v), то есть количество ребер которые «входят» в v.Исходящая степень вершины v это количество ребер вида (v , i), то есть количество ребер которые «выходят» из v.Это не совсем формальное определение (более формально определение через инцидентность), но оно вполне отражает суть. Путь в графе это конечная последовательность вершин, в которой каждые две вершины идущие подряд соединены ребром. Путь может быть ориентированным или неориентированным в зависимости от графа. На рис 1.а, путем является например последовательность [(1), (4), (5)] на рис 1.б, [(1), (3), (4), (5)].У графов есть ещё много разных свойств (например они могут быть связными, двудольными, полными), но я не буду описывать все эти свойства сейчас, а в следующих частях когда эти понятия понадобятся нам.

Существует два способа представления графа, в виде списков смежности и в виде матрицы смежности. Оба способа подходят для представления ориентированных и неориентированных графов. Матрица смежности. Этот способ является удобным для представления плотных графов, в которых количество рёбер (|E|) примерно равно количеству вершин в квадрате (|V|2). В данном представлении мы заполняем матрицу размером |V| x |V| следущим образом: A[i][j] = 1 (Если существует ребро из i в j)A[i][j] = 0 (Иначе). Данный способ подходит для ориентированных и неориентированных графов. Для неориентированных графов матрица A является симметричной (то есть A[i][j] == A[j][i], т.к. если существует ребро между i и j, то оно является и ребром из i в j, и ребром из j в i). Благодаря этому свойству можно сократить почти в два раза использование памяти, храня элементы только в верхней части матрицы, над главной диагональю). Понятно что с помощью данного способа представления, можно быстро проверить есть ли ребро между вершинами v и u, просто посмотрев в ячейку A[v][u].. С другой стороны этот способ очень громоздкий, так как требует O (|V|2) памяти для хранения матрицы.

Списки смежности. Данный способ представления больше подходит для разреженных графов, то есть графов у которых количество рёбер гораздо меньше чем количество вершин в квадрате (|E| << |V|2).

В данном представлении используется массив Adj содержащий |V| списков. В каждом списке Adj[v] содержатся все вершины u, так что между v и u есть ребро. Память требуемая для представления равна O (|E| + |V|) что является лучшим показателем чем матрица смежности для разреженных графов.

Главный недостаток этого способа представления в том, что нет быстрого способа проверить существует ли ребро (u, v).

**Задание**

Реализовать логическую игру-головоломку согласно техническому заданию.

1. Игровое поле размера, зависящее от размера экрана
2. На котором размещаются гирданты и трубы
3. В правой части экрана находится список доступных труб со случайной минимальный длиной, максимальной длиной и проводимостью границы которых указываются в настройках
4. Положение гидрантов генерируется случайным образом, но не ближе чем минимальной расстояние, указанное в настройках
5. Емкость гидрантов также генерируется в диапазоне, указанном в настройках
6. Переполнение отсутствует, но если емкость гидранта опустеет полностью, то игра оканчивается
7. Подсчет очков ведется как сумма перенесенной воды всех гидрантов на все ходы
8. Каждый ход на время показывается текущий счет
9. Потом от гидранта к гидранту определяется исключительно проводимостью трубы
10. Подсчет очков и условий окончания игры происходит в самом конце хода
11. В игре присутствует три вида труб

Простые

Бонусные

Награда за бонусные

1. По завершению игры пользователь может ввести свое имя и увидеть результат во вкладке score

**Алгоритм решения задачи**

1. Спроектировать основные классы и менеджеры управления игрой
2. Спроектировать удобную инфраструктуру для доступа к управляющему слою из обьектов с логикой
3. Реализовать по сценам весь необходимый функционал и поведение, следуя техническому заданию
4. Добавить немного воображения

**Тестирование**

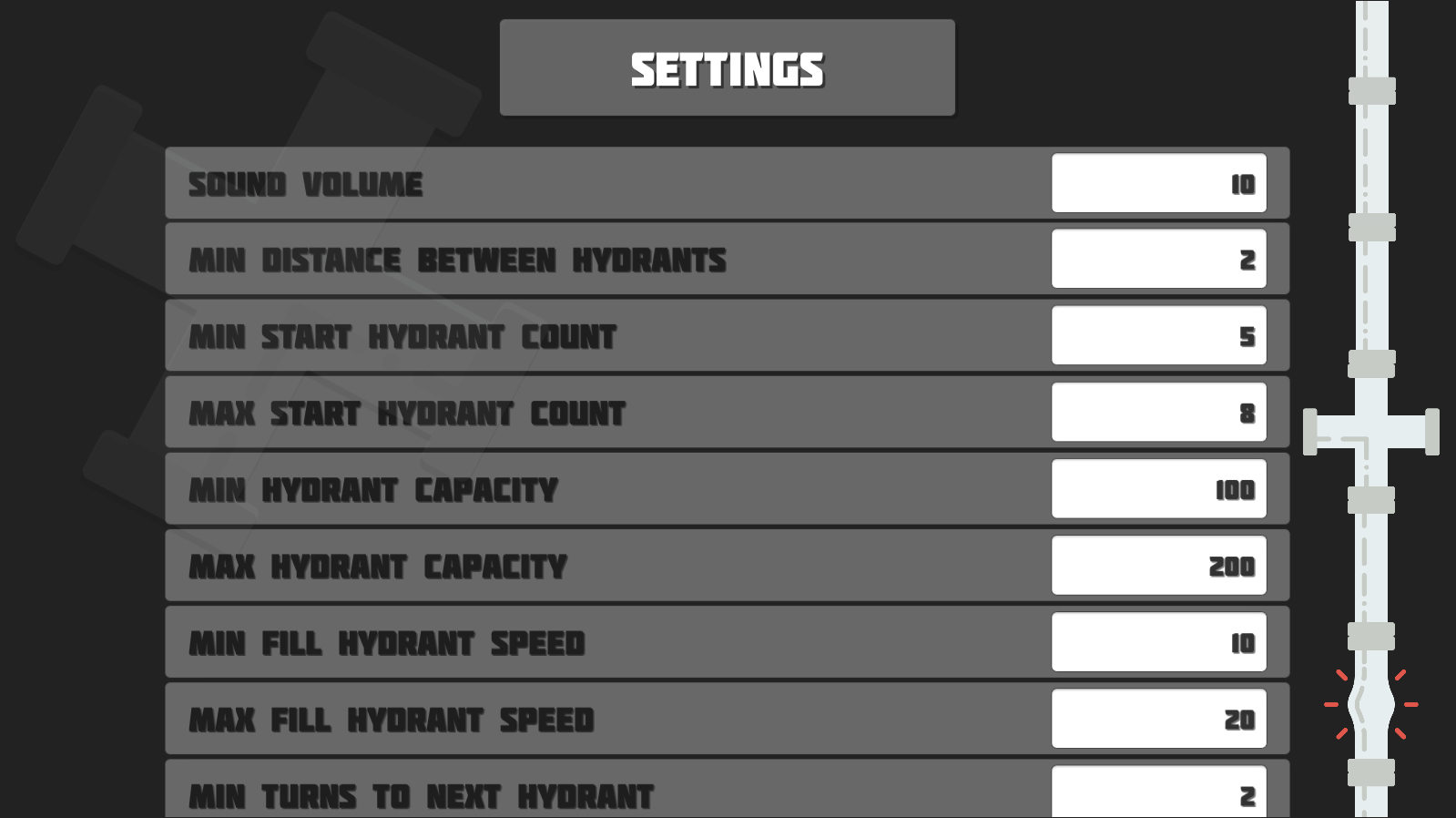
1. Проверить настройки
2. Проверить игру
3. Проверить сохранение результатов

**Заключение**

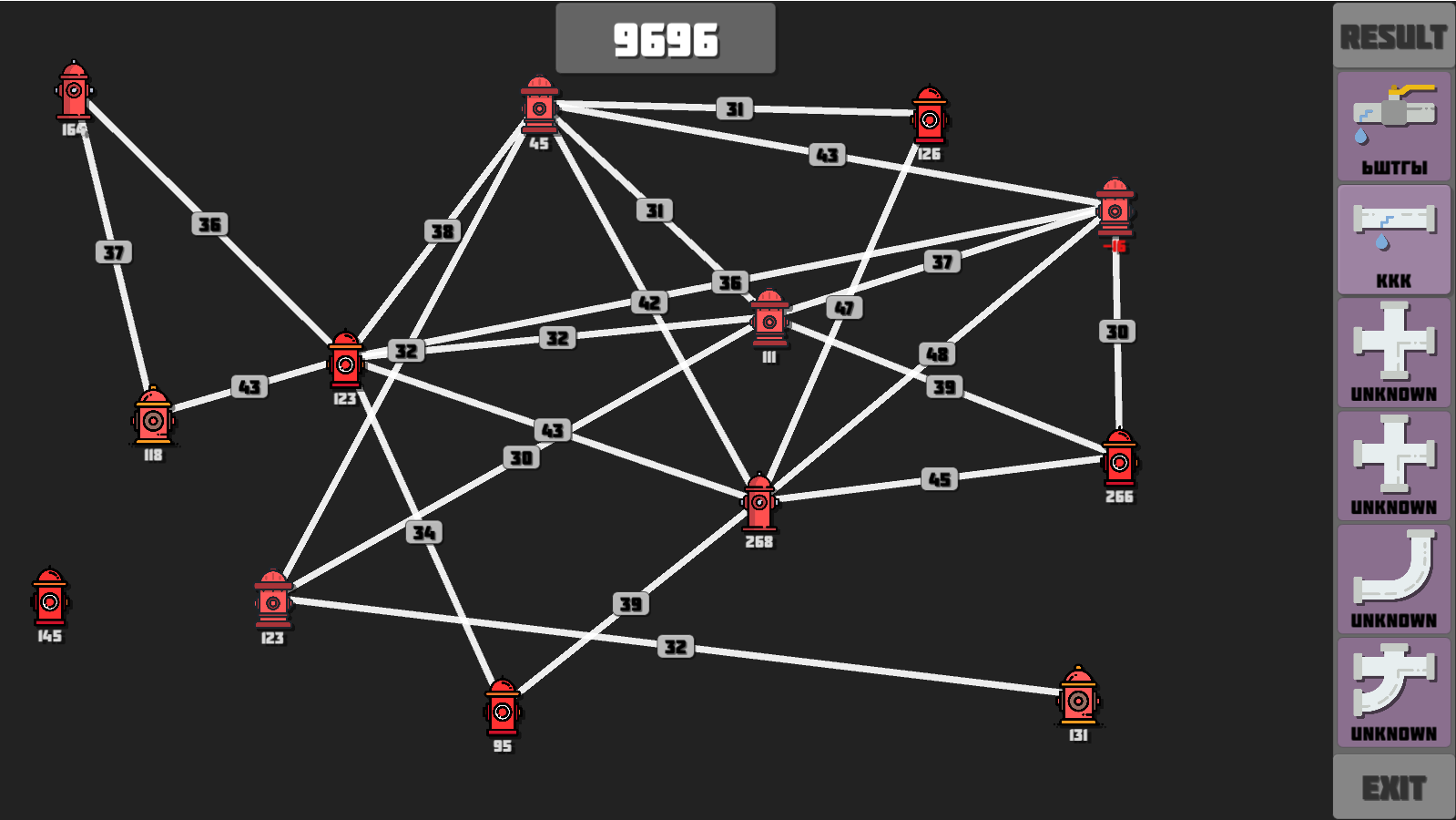
В данной лабораторной работе мы познакомились с понятием графа и его реализацией на платформе .Net с использованием языка C#.

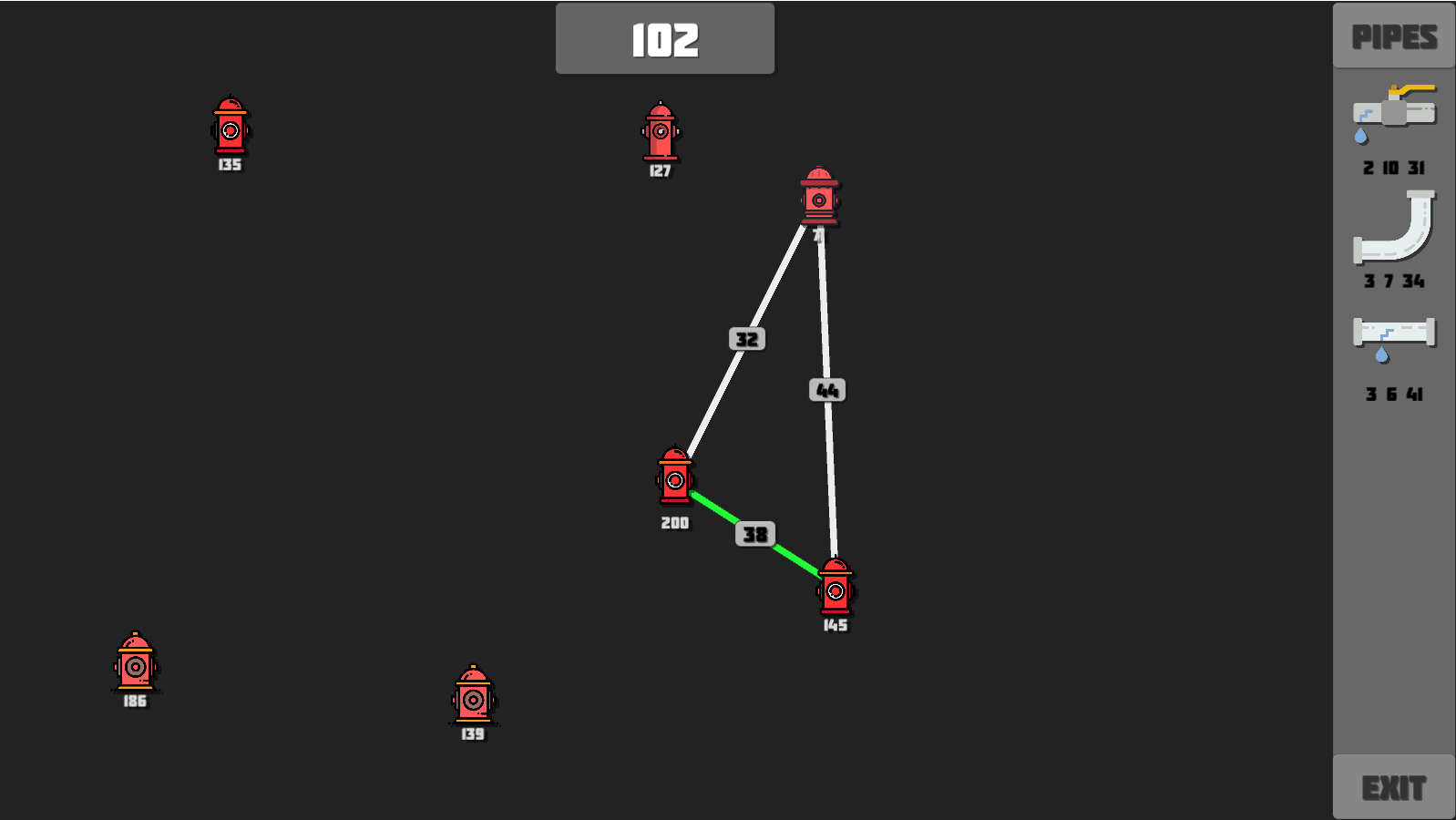
**Пример работы**













**Исходный код**

public abstract class Manager<TManager> : PipeBehaviour where TManager : Manager<TManager>

{

public static TManager Instance { get; private set; }

public abstract bool DestroyOnLoad { get; }

private void Awake()

{

if (Instance == null)

{

Instance = (TManager)this;

if (!DestroyOnLoad)

{

DontDestroyOnLoad(gameObject);

}

}

else

{

Destroy(gameObject);

}

}

private void OnDestroy()

{

if (DestroyOnLoad)

{

Instance = null;

}

}

}

using UnityEngine;

public class PipeBehaviour : MonoBehaviour

{

public string HydrantTag => "Hydrant";

public GameManager Game => GameManager.Instance;

public SettingsManager Settings => SettingsManager.Instance;

public SoundManager Sound => SoundManager.Instance;

public ScoreManager Score => ScoreManager.Instance;

public NotificationController Notification => FindObjectOfType<NotificationController>();

public TInstance Instantiate<TInstance>(GameObject prefab, Transform transform)

{

return Instantiate(prefab, transform, false).GetComponent<TInstance>();

}

public TInstance Instantiate<TInstance>(GameObject prefab, Transform transform, Vector3 position)

{

return Instantiate(prefab, position, Quaternion.identity, transform).GetComponent<TInstance>();

}

public void AddSizeDelta(RectTransform transform)

{

transform.sizeDelta = new Vector2

(

transform.sizeDelta.x,

transform.sizeDelta.y + Settings.PipeItemSizeDelta

);

}

public void RemoveSizeDelta(RectTransform transform)

{

transform.sizeDelta = new Vector2

(

transform.sizeDelta.x,

transform.sizeDelta.y - Settings.PipeItemSizeDelta

);

}

}

using System;

using System.Collections;

using UnityEngine;

using UnityEngine.UI;

public class PipeRenderer : PipeBehaviour

{

public GameObject StreamPowerUi => streamPowerUi;

public int StreamPower => streamPower;

[SerializeField] private LineRenderer lineRenderer;

[SerializeField] private GameObject streamPowerPrefab;

[SerializeField] private GameObject streamPowerUi;

[SerializeField] private int streamPower;

public PipeRenderer WithStartPosition(Vector3 startPosition)

{

lineRenderer.SetPosition(0, startPosition);

lineRenderer.SetPosition(1, startPosition);

Sound.PlayPipeRendererStart();

return this;

}

public PipeRenderer WithEndPosition(Vector3 endPosition)

{

lineRenderer.SetPosition(1, endPosition);

return this;

}

public PipeRenderer WithStreamPower(int streamPower)

{

var origin = lineRenderer.GetPosition(0);

var target = lineRenderer.GetPosition(1);

streamPowerUi = Instantiate

(

streamPowerPrefab,

new Vector3((origin.x + target.x) / 2, (origin.y + target.y) / 2, -1),

Quaternion.identity,

transform

);

streamPowerUi.GetComponentInChildren<Text>().text = streamPower.ToString();

this.streamPower = streamPower;

Sound.PlayPipeRendererEnd();

return this;

}

public PipeRenderer WithColor(Color lineColor)

{

lineRenderer.startColor = lineRenderer.endColor = lineColor;

return this;

}

public void ApplyEndTurn() => StartCoroutine(ApplyEndTurnCoroutine());

private IEnumerator ApplyEndTurnCoroutine()

{

Sound.PlayHydrantEndTurn();

WithColor(Settings.PipeActivatedColor);

streamPowerUi.transform.localScale = new Vector3(

streamPowerUi.transform.localScale.x + 0.1f,

streamPowerUi.transform.localScale.y + 0.1f,

streamPowerUi.transform.localScale.z);

yield return new WaitForSeconds(0.4f);

WithColor(SettingsManager.Instance.PipeConnectedColor);

streamPowerUi.transform.localScale = new Vector3(

streamPowerUi.transform.localScale.x - 0.1f,

streamPowerUi.transform.localScale.y - 0.1f,

streamPowerUi.transform.localScale.z);

}

}

using System;

using System.Collections;

using UnityEngine;

[Serializable]

public class Connection

{

public Hydrant To => to;

public Hydrant From => from;

public PipeRenderer PipeRenderer => pipeRenderer;

[SerializeField] private Hydrant to;

[SerializeField] private Hydrant from;

[SerializeField] private PipeRenderer pipeRenderer;

public Connection(Hydrant from, PipeRenderer pipeRenderer, Hydrant to)

{

this.from = from;

this.pipeRenderer = pipeRenderer;

this.to = to;

}

public void ApplyEndTurn() => pipeRenderer.ApplyEndTurn();

}

using UnityEngine;

using System.Linq;

using System.Collections.Generic;

public class GameFieldController : PipeBehaviour

{

public List<Hydrant> Hydrants => hydrants;

private PipeRenderer LastRenderer => pipeRenderers.Last();

public GameScoreController CurrentScore => currentScoreController;

[SerializeField] private GameObject hydrantPrefab;

[SerializeField] private GameObject pipeRendererPrefab;

[SerializeField] private GameScoreController currentScoreController;

[SerializeField] private Hydrant selectedHydrant;

[SerializeField] private List<Hydrant> hydrants;

[SerializeField] private List<PipeRenderer> pipeRenderers;

public void AddRandom()

{

var position = Settings.GetRandomHydrantPosition();

var repeats = 0;

while (repeats < 100 && hydrants.Any(h => Vector3.Distance(h.transform.position, position) < Settings.MinDistanceBetweenHydrants))

{

position = Settings.GetRandomHydrantPosition();

repeats++;

}

if (repeats == 100)

{

Notification.Show("Can't add more hydrants because settings min distance!");

return;

}

hydrants.Add(Instantiate<Hydrant>(hydrantPrefab, transform, position));

}

public void Select(Hydrant hydrant)

{

if (Game.Sidebar.SelectedPipeItem == null)

{

Notification.Show("Select Pipe Item!");

return;

}

if (selectedHydrant == null)

{

pipeRenderers.Add(Instantiate<PipeRenderer>(pipeRendererPrefab, transform));

selectedHydrant = hydrant;

LastRenderer.WithStartPosition(selectedHydrant.transform.position);

return;

}

if (selectedHydrant == hydrant)

{

Notification.Show("Target can't be selected!");

ResetSelected();

return;

}

var selected = Game.Sidebar.SelectedPipeItem;

if (Vector3.Distance(selectedHydrant.transform.position, hydrant.transform.position) > selected.MaxLength

|| Vector3.Distance(selectedHydrant.transform.position, hydrant.transform.position) < selected.MinLength)

{

ResetSelected();

return;

}

if (Game.LockEndTurn)

{

Notification.Show("Wait for end turn!");

ResetSelected();

return;

}

if (Game.Connections.Any(x => x.From == selectedHydrant && x.To == hydrant || x.From == hydrant && x.To == selectedHydrant))

{

Notification.Show("Connection already exists!");

ResetSelected();

return;

}

AddSelectedConnection(hydrant);

}

private void FixedUpdate()

{

if (selectedHydrant == null)

{

return;

}

var cameraPosition = Camera.main.ScreenToWorldPoint(Input.mousePosition);

var mousePosition = new Vector3(cameraPosition.x, cameraPosition.y, 0);

var distance = Vector3.Distance(selectedHydrant.transform.position, mousePosition);

var color = distance < Game.Sidebar.SelectedPipeItem.MinLength

? Settings.PipeLengthOutOfRangeColor

: distance < Game.Sidebar.SelectedPipeItem.MaxLength

? Settings.PipeLengthInRangeColor

: Settings.PipeLengthOutOfRangeColor;

LastRenderer

.WithEndPosition(mousePosition)

.WithColor(color);

}

private void AddSelectedConnection(Hydrant targetHydrant)

{

var selected = Game.Sidebar.SelectedPipeItem;

LastRenderer

.WithEndPosition(targetHydrant.transform.position)

.WithColor(Settings.PipeConnectedColor)

.WithStreamPower(selected.StreamPower);

Game.Connections.Add(new Connection(selectedHydrant, LastRenderer, targetHydrant));

Game.Sidebar.UseSelected();

selectedHydrant = null;

}

public void ResetSelected()

{

if (selectedHydrant != null)

{

Sound.PlayResetSelected();

selectedHydrant = null;

var last = LastRenderer;

pipeRenderers.Remove(last);

Destroy(last.gameObject);

}

}

}

using System.Linq;

using UnityEngine;

using UnityEngine.SceneManagement;

using UnityEngine.UI;

public class GameOverController : PipeBehaviour

{

[SerializeField] private InputField nicknameInputField;

public void Show()

{

Sound.PlayerGameOverAppear();

gameObject.SetActive(true);

}

public void OnOk()

{

var result = new GameResult

{

PlayerName = string.IsNullOrWhiteSpace(nicknameInputField.text) ? "Unknown" : nicknameInputField.text,

Score = Score.CurrentScore,

Hydrants = Game.GameField.Hydrants.Select(h => new HydrantResult

{

Id = h.Id,

X = h.transform.position.x,

Y = h.transform.position.y,

Z = h.transform.position.z,

Capacity = h.Capacity

}).ToList(),

Connections = Game.Connections.Select(c => new ConnectionResult

{

HydrantId1 = c.From.Id,

HydrantId2 = c.To.Id,

StreamPower = c.PipeRenderer.StreamPower

}).ToList()

};

Sound.PlayButtonClicked();

Score.SaveScore(result);

SceneManager.LoadScene("Menu");

}

}

using UnityEngine;

using UnityEngine.UI;

public class GameScoreController : PipeBehaviour

{

[SerializeField] private Text scoreText;

[SerializeField] private Animator animator;

public void Show()

{

scoreText.text = Score.CurrentScore.ToString();

animator.SetTrigger("ShowScore");

}

}

using UnityEngine;

public class MiniGameController : PipeBehaviour

{

public void StartRandom()

{

Sound.PlayMinigameAppear();

gameObject.SetActive(true);

}

public void OnDone()

{

var reward = Random.Range(1, 4);

for (int count = 0; count < reward; count++)

{

Game.Sidebar.AddReward();

}

Game.EndTurn();

gameObject.SetActive(false);

}

}

using UnityEngine;

using UnityEngine.UI;

public class NotificationController : PipeBehaviour

{

[SerializeField] private Text message;

[SerializeField] private GameObject holder;

public void Show(string message)

{

this.message.text = message;

holder.SetActive(true);

Sound.PlayNotificationAppear();

}

public void OnCancel()

{

holder.SetActive(false);

}

}

using UnityEngine;

using UnityEngine.UI;

using System.Collections.Generic;

using UnityEngine.SceneManagement;

public class PipeItemSidebarController : PipeBehaviour

{

[SerializeField] private RectTransform content;

[SerializeField] private GameObject normalPipeItemPrefab;

[SerializeField] private GameObject bonusPipeItemPrefab;

[SerializeField] private GameObject rewardPipeItemPrefab;

[SerializeField] private PipeItem selectedPipeItem;

[SerializeField] private List<PipeItem> pipeItems;

public PipeItem SelectedPipeItem => selectedPipeItem;

public void AddRandom()

{

var pipeItemPrefab = Random.Range(0, 100) < Settings.BonusPipeChance

? bonusPipeItemPrefab

: normalPipeItemPrefab;

pipeItems.Add(Instantiate<PipeItem>(pipeItemPrefab, content));

AddSizeDelta(content);

}

public void AddReward()

{

pipeItems.Add(Instantiate<PipeItem>(rewardPipeItemPrefab, content));

AddSizeDelta(content);

}

public void Select(PipeItem pipeItem)

{

Sound.PlayPipeItemSelected();

if (selectedPipeItem == pipeItem)

{

selectedPipeItem.Selected = false;

selectedPipeItem = null;

return;

}

if(selectedPipeItem != null)

{

selectedPipeItem.Selected = false;

}

selectedPipeItem = pipeItem;

selectedPipeItem.Selected = true;

Game.GameField.ResetSelected();

}

public void UseSelected()

{

if (selectedPipeItem == null)

{

Notification.Show("Select pipe item!");

return;

}

selectedPipeItem.Use();

pipeItems.Remove(selectedPipeItem);

Destroy(selectedPipeItem.gameObject);

selectedPipeItem = null;

RemoveSizeDelta(content);

}

public void OnExitClicked()

{

Sound.PlayButtonClicked();

SceneManager.LoadScene("Menu");

}

}

using UnityEngine;

using UnityEngine.UI;

public class GameResultHydrant : PipeBehaviour

{

[SerializeField] private int id;

[SerializeField] private Text description;

[SerializeField] private Image hydrantImage;

public int Id => id;

private void Start()

{

hydrantImage.sprite = Settings.GetRandomHydrantSprite();

Sound.PlayHydrantAppear();

}

internal void InitializeWith(HydrantResult hydrant)

{

id = hydrant.Id;

description.text = hydrant.Capacity.ToString();

description.color = hydrant.Capacity > 0

? Settings.PositiveHydrantColor

: Settings.NegativeHydrantColor;

}

}

using UnityEngine;

using UnityEngine.UI;

public class Hydrant : PipeBehaviour

{

private static int uniqueId = 0;

[SerializeField] private int id;

[SerializeField] private int capacity;

[SerializeField] private Animator animator;

[SerializeField] private Text description;

[SerializeField] private Image hydrantSprite;

public int Id => id;

public int Capacity => capacity;

private void Start()

{

id = uniqueId++;

capacity = Settings.GetRandomHydrantCapacity();

description.text = capacity.ToString();

hydrantSprite.sprite = Settings.GetRandomHydrantSprite();

Sound.PlayHydrantAppear();

}

public void ApplyConnection(Connection connection)

{

if(connection.From == this)

{

capacity -= connection.PipeRenderer.StreamPower;

}

else

{

capacity += connection.PipeRenderer.StreamPower;

}

animator.SetTrigger("ApplyConnection");

SetDescription(capacity);

}

private void SetDescription(int capacity)

{

description.text = capacity.ToString();

description.color = capacity > 0

? Settings.PositiveHydrantColor

: Settings.NegativeHydrantColor;

}

public void OnClick()

{

Game.GameField.Select(this);

}

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class BonusPipeItem : PipeItem

{

public override void Use()

{

Game.MiniGame.StartRandom();

Game.Sidebar.AddRandom();

}

}

public class NormalPipeItem : PipeItem

{

public override void Use()

{

Game.Sidebar.AddRandom();

Game.EndTurn();

}

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class RewardPipeItem : PipeItem

{

public override void Use()

{

}

}

using UnityEngine;

using UnityEngine.UI;

public abstract class PipeItem : PipeBehaviour

{

[SerializeField] private Image pipeItemImage;

[SerializeField] private Text pipeItemDescription;

[SerializeField] private int minLength;

[SerializeField] private int maxLength;

[SerializeField] private int streamPower;

public int MinLength => minLength;

public int MaxLength => maxLength;

public int StreamPower => streamPower;

public bool Selected { set { GetComponent<Image>().enabled = value; } }

public abstract void Use();

private void Start()

{

Sound.PlayPipeItemAppear();

pipeItemImage.sprite = Settings.GetRandomPipeSprite();

minLength = Random.Range(

Settings.MinPipeLengthMin, Settings.MinPipeLengthMax);

maxLength = Random.Range(

Settings.MaxPipeLengthMin, Settings.MaxPipeLengthMax);

streamPower = Random.Range(

Settings.MinStreamPower, Settings.MaxStreamPower);

pipeItemDescription.text = $"{minLength} {maxLength} {streamPower}";

}

public void OnClick() => Game.Sidebar.Select(this);

}

using System.Collections;

using System.Collections.Generic;

using System.Linq;

using UnityEngine;

public sealed class GameManager : Manager<GameManager>

{

public override bool DestroyOnLoad => true;

[SerializeField] private bool lockEndTurn;

[SerializeField] private int turnToNextHydrant;

[SerializeField] private PipeItemSidebarController pipeItemSidebarController;

[SerializeField] private GameFieldController gameField;

[SerializeField] private MiniGameController miniGame;

[SerializeField] private GameOverController gameOver;

[SerializeField] private List<Connection> connections;

public PipeItemSidebarController Sidebar => pipeItemSidebarController;

public GameFieldController GameField => gameField;

public MiniGameController MiniGame => miniGame;

public List<Connection> Connections => connections;

public GameOverController GameOver => gameOver;

public bool LockEndTurn => lockEndTurn;

private IEnumerator Start()

{

Sound.PlayGameAmbient();

Score.CurrentScore = 0;

turnToNextHydrant = Settings.GetTurnsForNextHydrant();

var hydrants = Random.Range(Settings.MinStartHydrantCount, Settings.MaxStartHydrantCount);

for (int i = 0; i < hydrants; i++)

{

GameField.AddRandom();

yield return new WaitForSeconds(0.2f);

}

var pipeItems = Random.Range(Settings.MinStartAvaliablePipesCount, Settings.MaxStartAvaliablePipesCount);

for (int i = 0; i < pipeItems; i++)

{

Sidebar.AddRandom();

yield return new WaitForSeconds(0.2f);

}

}

public void EndTurn()

{

lockEndTurn = true;

StartCoroutine(EndTurnCoroutine());

}

private IEnumerator EndTurnCoroutine()

{

foreach (var connection in connections)

{

Score.CurrentScore += connection.PipeRenderer.StreamPower;

yield return new WaitForSeconds(0.2f);

connection.From.ApplyConnection(connection);

connection.ApplyEndTurn();

yield return new WaitForSeconds(0.2f);

connection.To.ApplyConnection(connection);

yield return new WaitForSeconds(0.2f);

}

if (connections.Any(c => c.To.Capacity <= 0 || c.From.Capacity <= 0))

{

Game.GameOver.Show();

yield break;

}

yield return new WaitForSeconds(0.2f);

if (--turnToNextHydrant == 0)

{

Game.GameField.AddRandom();

turnToNextHydrant = Game.Settings.GetTurnsForNextHydrant();

}

GameField.CurrentScore.Show();

lockEndTurn = false;

}

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.SceneManagement;

public class MenuManager : Manager<MenuManager>

{

public override bool DestroyOnLoad => true;

[SerializeField] private Animator holderAnimator;

private void Start()

{

Sound.PlayMenuAmbient();

StartCoroutine(ShowAnimation());

}

private IEnumerator ShowAnimation()

{

yield return new WaitForSeconds(5);

var trigger = Random.Range(0, 9) > 7 ? "Runs" : "Jumps";

holderAnimator.SetTrigger(trigger);

yield return new WaitForSeconds(30);

StartCoroutine(ShowAnimation());

}

public void LoadScene(string sceneName)

{

Sound.PlayButtonClicked();

SceneManager.LoadScene(sceneName);

}

}

using UnityEngine;

using UnityEngine.UI;

using System.Linq;

using System.Collections;

using System.Collections.Generic;

public class GameResultFieldController : PipeBehaviour

{

[SerializeField] private Text selectedGameResultScoreText;

[SerializeField] private GameObject hydrantPrefab;

[SerializeField] private List<GameResultHydrant> hydrants;

[SerializeField] private GameObject pipeRendererPrefab;

[SerializeField] private List<PipeRenderer> pipeRenderers;

[SerializeField] private Coroutine currentGameResultCoroutine;

private IEnumerator Start()

{

foreach (var result in Score.GameResults.AsEnumerable().Reverse())

{

yield return new WaitForSeconds(0.1f);

Score.Sidebar.AddResult(result);

}

}

public void GenerateGameResult(GameResultItem gameResult)

{

selectedGameResultScoreText.text = gameResult.GameResult.Score.ToString();

if (currentGameResultCoroutine != null)

StopCoroutine(currentGameResultCoroutine);

DestroyGameResult();

currentGameResultCoroutine = StartCoroutine(GenerateGameResultCoroutine(gameResult));

}

private IEnumerator GenerateGameResultCoroutine(GameResultItem gameResult)

{

foreach (var data in gameResult.GameResult.Hydrants)

{

var hydrant = Instantiate<GameResultHydrant>(hydrantPrefab, transform, new Vector3(data.X, data.Y, data.Z));

yield return new WaitForFixedUpdate();

hydrant.InitializeWith(data);

hydrants.Add(hydrant);

yield return new WaitForSeconds(0.1f);

}

foreach (var data in gameResult.GameResult.Connections)

{

var hydrant1 = gameResult.GameResult.Hydrants.Single(x => x.Id == data.HydrantId1);

var hydrant2 = gameResult.GameResult.Hydrants.Single(x => x.Id == data.HydrantId2);

var pipeRenderer = Instantiate<PipeRenderer>(pipeRendererPrefab, transform)

.WithStartPosition(new Vector3(hydrant1.X, hydrant1.Y, hydrant1.Z))

.WithEndPosition(new Vector3(hydrant2.X, hydrant2.Y, hydrant2.Z))

.WithStreamPower(data.StreamPower)

.WithColor(Settings.PipeConnectedColor);

pipeRenderers.Add(pipeRenderer);

yield return new WaitForSeconds(0.1f);

}

}

private void DestroyGameResult()

{

foreach (var hydrant in hydrants)

{

Destroy(hydrant.gameObject);

}

hydrants = new List<GameResultHydrant>();

foreach (var pipeRenderer in pipeRenderers)

{

Destroy(pipeRenderer.gameObject);

}

pipeRenderers = new List<PipeRenderer>();

}

}

using UnityEngine;

using UnityEngine.SceneManagement;

public class GameResultSidebarController : PipeBehaviour

{

[SerializeField] private RectTransform content;

[SerializeField] private GameObject gameResultPrefab;

[SerializeField] private GameResultItem selectedGameResultItem;

internal void AddResult(GameResult result)

{

var gameResult = Instantiate<GameResultItem>(gameResultPrefab, content);

gameResult.InitializeWith(result);

AddSizeDelta(content);

}

public void Select(GameResultItem gameResultItem)

{

if(selectedGameResultItem != null)

{

selectedGameResultItem.Selected = false;

}

selectedGameResultItem = gameResultItem;

selectedGameResultItem.Selected = true;

Score.Result.GenerateGameResult(selectedGameResultItem);

}

public void OnExitClicked()

{

Sound.PlayButtonClicked();

SceneManager.LoadScene("Menu");

}

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class ConnectionResult

{

public int HydrantId1 { get; set; }

public int StreamPower { get; set; }

public int HydrantId2 { get; set; }

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class GameResult

{

public string PlayerName { get; set; }

public int Score { get; set; }

public List<HydrantResult> Hydrants { get; set; }

public List<ConnectionResult> Connections { get; set; }

} public class HydrantResult

{

public int Id { get; set; }

public float X { get; set; }

public float Y { get; set; }

public float Z { get; set; }

public int Capacity { get; set; }

}

using UnityEngine;

using UnityEngine.UI;

public class GameResultItem : PipeBehaviour

{

[SerializeField] private Image gameResultImage;

[SerializeField] private Text gameResultNickname;

[SerializeField] private GameResult gameResult;

public GameResult GameResult => gameResult;

public bool Selected { set { GetComponent<Image>().enabled = value; } }

private void Start()

{

Sound.PlayPipeItemAppear();

gameResultImage.sprite = Settings.GetRandomPipeSprite();

}

public void InitializeWith(GameResult gameResult)

{

gameResultNickname.text = gameResult.PlayerName;

this.gameResult = gameResult;

}

public void OnClick()

{

Score.Sidebar.Select(this);

}

}

using System.Collections.Generic;

using System.IO;

using System.Xml.Serialization;

using UnityEngine;

public sealed class ScoreManager : Manager<ScoreManager>

{

public override bool DestroyOnLoad => false;

public GameResultFieldController Result => FindObjectOfType<GameResultFieldController>();

public GameResultSidebarController Sidebar => FindObjectOfType<GameResultSidebarController>();

public int CurrentScore { get; set; }

public List<GameResult> GameResults => gameResults;

[SerializeField] private List<GameResult> gameResults;

private void Start() => LoadScore();

public void LoadScore()

{

if (File.Exists(Settings.GameResultsFilePath))

{

using (var stream = new FileStream(Settings.GameResultsFilePath, FileMode.Open, FileAccess.Read))

{

gameResults = (List<GameResult>)new XmlSerializer(typeof(List<GameResult>)).Deserialize(stream);

}

}

else

{

gameResults = new List<GameResult>();

}

}

public void SaveScore(GameResult result)

{

gameResults.Add(result);

using (var stream = new FileStream(Settings.GameResultsFilePath, FileMode.OpenOrCreate, FileAccess.Write))

{

new XmlSerializer(typeof(List<GameResult>)).Serialize(stream, gameResults);

}

}

}

using UnityEngine;

using UnityEngine.SceneManagement;

using UnityEngine.UI;

public class SettingsController : PipeBehaviour

{

[Header("Game Settings", order = 0)]

[SerializeField] private InputField soundVolume;

[Header("Hydrant settings", order = 1)]

[SerializeField] private InputField minDistanceBetweenHydrants;

[SerializeField] private InputField minStartHydrantCount;

[SerializeField] private InputField maxStartHydrantCount;

[SerializeField] private InputField minHydrantCapacity;

[SerializeField] private InputField maxHydrantCapacity;

[SerializeField] private InputField minTurnsToNextHydrant;

[SerializeField] private InputField maxTurnsToNextHydrant;

[SerializeField] private InputField minFillHydrantSpeed;

[SerializeField] private InputField maxFillHydrantSpeed;

[Header("Pipe settings")]

[SerializeField] private InputField bonusPipeChance;

[SerializeField] private InputField minStartAvaliablePipesCount;

[SerializeField] private InputField maxStartAvaliablePipesCount;

[SerializeField] private InputField minPipeLengthMin;

[SerializeField] private InputField minPipeLengthMax;

[SerializeField] private InputField maxPipeLengthMin;

[SerializeField] private InputField maxPipeLengthMax;

[SerializeField] private InputField minStreamPower;

[SerializeField] private InputField maxStreamPower;

[Header("Minigame settings")]

[SerializeField] private InputField firstMiniGameTimeLimit;

[SerializeField] private InputField secondMiniGameTimeLimit;

[SerializeField] private InputField thirdMiniGameTimeLimit;

private void Start()

{

soundVolume.text = Settings.SoundVolume.ToString();

minDistanceBetweenHydrants.text = Settings.MinDistanceBetweenHydrants.ToString();

minStartHydrantCount.text = Settings.MinStartHydrantCount.ToString();

maxStartHydrantCount.text = Settings.MaxStartHydrantCount.ToString();

minHydrantCapacity.text = Settings.MinHydrantCapacity.ToString();

maxHydrantCapacity.text = Settings.MaxHydrantCapacity.ToString();

minTurnsToNextHydrant.text = Settings.MinTurnsToNextHydrant.ToString();

maxTurnsToNextHydrant.text = Settings.MaxTurnsToNextHydrant.ToString();

minFillHydrantSpeed.text = Settings.MinFillHydrantSpeed.ToString();

maxFillHydrantSpeed.text = Settings.MaxFillHydrantSpeed.ToString();

bonusPipeChance.text = Settings.BonusPipeChance.ToString();

minStartAvaliablePipesCount.text = Settings.MinStartAvaliablePipesCount.ToString();

maxStartAvaliablePipesCount.text = Settings.MaxStartAvaliablePipesCount.ToString();

minPipeLengthMin.text = Settings.MinPipeLengthMin.ToString();

minPipeLengthMax.text = Settings.MinPipeLengthMax.ToString();

maxPipeLengthMin.text = Settings.MaxPipeLengthMin.ToString();

maxPipeLengthMax.text = Settings.MaxPipeLengthMax.ToString();

minStreamPower.text = Settings.MinStreamPower.ToString();

maxStreamPower.text = Settings.MaxStreamPower.ToString();

firstMiniGameTimeLimit.text = Settings.FirstMiniGameTimeLimit.ToString();

secondMiniGameTimeLimit.text = Settings.SecondMiniGameTimeLimit.ToString();

thirdMiniGameTimeLimit.text = Settings.ThirdMiniGameTimeLimit.ToString();

}

public void OnSaveClicked()

{

Sound.PlayButtonClicked();

var settingsDto = Settings.CreateSettingsDto();

settingsDto.SoundVolume = int.Parse(soundVolume.text);

settingsDto.MinDistanceBetweenHydrants = int.Parse(minDistanceBetweenHydrants.text);

settingsDto.MinStartHydrantCount = int.Parse(minStartHydrantCount.text);

settingsDto.MaxStartHydrantCount = int.Parse(maxStartHydrantCount.text);

settingsDto.MinHydrantCapacity = int.Parse(minHydrantCapacity.text);

settingsDto.MaxHydrantCapacity = int.Parse(maxHydrantCapacity.text);

settingsDto.MinFillHydrantSpeed = int.Parse(minFillHydrantSpeed.text);

settingsDto.MaxFillHydrantSpeed = int.Parse(maxFillHydrantSpeed.text);

settingsDto.MinTurnsToNextHydrant = int.Parse(minTurnsToNextHydrant.text);

settingsDto.MaxTurnsToNextHydrant = int.Parse(maxTurnsToNextHydrant.text);

settingsDto.BonusPipeChance = int.Parse(bonusPipeChance.text);

settingsDto.MinStartAvaliablePipesCount = int.Parse(minStartAvaliablePipesCount.text);

settingsDto.MaxStartAvaliablePipesCount = int.Parse(maxStartAvaliablePipesCount.text);

settingsDto.MinPipeLengthMin = int.Parse(minPipeLengthMin.text);

settingsDto.MinPipeLengthMax = int.Parse(minPipeLengthMax.text);

settingsDto.MaxPipeLengthMin = int.Parse(maxPipeLengthMin.text);

settingsDto.MaxPipeLengthMax = int.Parse(maxPipeLengthMax.text);

settingsDto.MinStreamPower = int.Parse(minStreamPower.text);

settingsDto.MaxStreamPower = int.Parse(maxStreamPower.text);

settingsDto.FirstMiniGameTimeLimit = int.Parse(firstMiniGameTimeLimit.text);

settingsDto.SecondMiniGameTimeLimit = int.Parse(secondMiniGameTimeLimit.text);

settingsDto.ThirdMiniGameTimeLimit = int.Parse(thirdMiniGameTimeLimit.text);

if (Validate(settingsDto))

{

Settings.ApplySettings(settingsDto);

SceneManager.LoadScene("Menu");

}

}

public bool Validate(SettingsDto dto)

{

if(dto.SoundVolume < 0 || dto.SoundVolume > 100)

{

Notification.Show($"{nameof(dto.SoundVolume)}: Value must be in [0..100] range!");

return false;

}

if (dto.MinDistanceBetweenHydrants <= 0)

{

Notification.Show($"{nameof(dto.MinDistanceBetweenHydrants)}: Value must be > 0!");

return false;

}

if (dto.MinStartHydrantCount < 2)

{

Notification.Show($"{nameof(dto.MinStartHydrantCount)}: Value must be >= 2!");

return false;

}

if (dto.MinStartHydrantCount > dto.MaxStartHydrantCount)

{

Notification.Show($"{nameof(dto.MinStartHydrantCount)}: Value must be <= {nameof(dto.MaxStartHydrantCount)}!");

return false;

}

if (dto.MinHydrantCapacity <= 0)

{

Notification.Show($"{nameof(dto.MinHydrantCapacity)}: Value must be > 0!");

return false;

}

if (dto.MinHydrantCapacity > dto.MaxHydrantCapacity)

{

Notification.Show($"{nameof(dto.MinHydrantCapacity)}: Value must be <= {nameof(dto.MaxHydrantCapacity)}!");

return false;

}

if (dto.MinFillHydrantSpeed <= 0)

{

Notification.Show($"{nameof(dto.MinFillHydrantSpeed)}: Value must be > 0!");

return false;

}

if (dto.MinFillHydrantSpeed > dto.MaxFillHydrantSpeed)

{

Notification.Show($"{nameof(dto.MinFillHydrantSpeed)}: Value must be <= {nameof(dto.MaxFillHydrantSpeed)}!");

return false;

}

if (dto.MinTurnsToNextHydrant < 0)

{

Notification.Show($"{nameof(dto.MinTurnsToNextHydrant)}: Value must be >= 0!");

return false;

}

if (dto.MinTurnsToNextHydrant > dto.MaxTurnsToNextHydrant)

{

Notification.Show($"{nameof(dto.MinTurnsToNextHydrant)}: Value must be <= {nameof(dto.MaxTurnsToNextHydrant)}!");

return false;

}

if(dto.BonusPipeChance < 0 || dto.BonusPipeChance > 100)

{

Notification.Show($"{nameof(dto.BonusPipeChance)}: Value must be in [0..100] range!");

return false;

}

if (dto.MinStartAvaliablePipesCount <= 0)

{

Notification.Show($"{nameof(dto.MinStartAvaliablePipesCount)}: Value must be > 0!");

return false;

}

if (dto.MinStartAvaliablePipesCount > dto.MaxStartAvaliablePipesCount)

{

Notification.Show($"{nameof(dto.MinStartAvaliablePipesCount)}: Value must be <= {nameof(dto.MaxStartAvaliablePipesCount)}!");

return false;

}

if(dto.MinPipeLengthMin <= 0)

{

Notification.Show($"{nameof(dto.MinPipeLengthMin)}: Value must be > 0!");

return false;

}

if (dto.MinPipeLengthMin > dto.MinPipeLengthMax)

{

Notification.Show($"{nameof(dto.MinPipeLengthMin)}: Value must be <= {nameof(dto.MinPipeLengthMax)}!");

return false;

}

if(dto.MinPipeLengthMax > dto.MaxPipeLengthMin)

{

Notification.Show($"{nameof(dto.MinPipeLengthMax)}: Value must be <= {nameof(dto.MaxPipeLengthMin)}!");

return false;

}

if (dto.MaxPipeLengthMin > dto.MaxPipeLengthMax)

{

Notification.Show($"{nameof(dto.MaxPipeLengthMin)}: Value must be <= {nameof(dto.MaxPipeLengthMax)}!");

return false;

}

if(dto.MinStreamPower <= 0)

{

Notification.Show($"{nameof(dto.MinStreamPower)}: Value must be > 0!");

return false;

}

if (dto.MinStreamPower > dto.MaxStreamPower)

{

Notification.Show($"{nameof(dto.MinStreamPower)}: Value must be <= {nameof(dto.MaxStreamPower)}!");

return false;

}

if(dto.FirstMiniGameTimeLimit < 10 || dto.FirstMiniGameTimeLimit > 100)

{

Notification.Show($"{nameof(dto.FirstMiniGameTimeLimit)}: Value must be in [10..100] range!");

return false;

}

if (dto.SecondMiniGameTimeLimit < 10 || dto.SecondMiniGameTimeLimit > 100)

{

Notification.Show($"{nameof(dto.SecondMiniGameTimeLimit)}: Value must be in [10..100] range!");

return false;

}

if (dto.ThirdMiniGameTimeLimit < 10 || dto.ThirdMiniGameTimeLimit > 100)

{

Notification.Show($"{nameof(dto.ThirdMiniGameTimeLimit)}: Value must be in [10..100] range!");

return false;

}

return true;

}

}

using UnityEngine;

public class SettingsDto

{

public int SoundVolume { get; set; }

// Hydrant

public int MinDistanceBetweenHydrants { get; set; }

public int MinStartHydrantCount { get; set; }

public int MaxStartHydrantCount { get; set; }

public int MinHydrantCapacity { get; set; }

public int MaxHydrantCapacity { get; set; }

public int MinFillHydrantSpeed { get; set; }

public int MaxFillHydrantSpeed { get; set; }

public int MinTurnsToNextHydrant { get; set; }

public int MaxTurnsToNextHydrant { get; set; }

// Pipe

public int BonusPipeChance { get; set; }

public int MinStartAvaliablePipesCount { get; set; }

public int MaxStartAvaliablePipesCount { get; set; }

public int MinPipeLengthMin { get; set; }

public int MinPipeLengthMax { get; set; }

public int MaxPipeLengthMin { get; set; }

public int MaxPipeLengthMax { get; set; }

public int MinStreamPower { get; set; }

public int MaxStreamPower { get; set; }

// Minigame

public int FirstMiniGameTimeLimit { get; set; }

public int SecondMiniGameTimeLimit { get; set; }

public int ThirdMiniGameTimeLimit { get; set; }

// Developer

public int PipeItemSizeDelta { get; set; }

public string SettingsFileName { get; set; }

public string GameResultsFileName { get; set; }

public Color PositiveHydrantColor { get; set; }

public Color NegativeHydrantColor { get; set; }

public Color PipeLengthOutOfRangeColor { get; set; }

public Color PipeLengthInRangeColor { get; set; }

public Color PipeConnectedColor { get; set; }

public Color PipeActivatedColor { get; set; }

}

using UnityEngine;

using System.Collections.Generic;

using System;

using Random = UnityEngine.Random;

using System.IO;

using System.Xml.Serialization;

public sealed class SettingsManager : Manager<SettingsManager>

{

public override bool DestroyOnLoad => false;

[Header("Game Settings")]

[SerializeField]

private int soundVolume;

[Header("Hydrant settings")]

[SerializeField]

private int minDistanceBetweenHydrants;

[SerializeField] private int minStartHydrantCount;

[SerializeField] private int maxStartHydrantCount;

[SerializeField] private int minHydrantCapacity;

[SerializeField] private int maxHydrantCapacity;

[SerializeField] private int minTurnsToNextHydrant;

[SerializeField] private int maxTurnsToNextHydrant;

[SerializeField] private int minFillHydrantSpeed;

[SerializeField] private int maxFillHydrantSpeed;

[Header("Pipe settings")]

[SerializeField]

private int bonusPipeChance;

[SerializeField] private int minStartAvaliablePipesCount;

[SerializeField] private int maxStartAvaliablePipesCount;

[SerializeField] private int minPipeLengthMin;

[SerializeField] private int minPipeLengthMax;

[SerializeField] private int maxPipeLengthMin;

[SerializeField] private int maxPipeLengthMax;

[SerializeField] private int minStreamPower;

[SerializeField] private int maxStreamPower;

[Header("Minigame settings")]

[SerializeField]

private int firstMiniGameTimeLimit;

[SerializeField] private int secondMiniGameTimeLimit;

[SerializeField] private int thirdMiniGameTimeLimit;

[Header("Developer Settings")]

[SerializeField]

private int pipeItemSizeDelta = 78;

[SerializeField] private string settingsFileName;

[SerializeField] private string gameResultsFileName;

[SerializeField] private List<Sprite> pipeSprites;

[SerializeField] private List<Sprite> hydrantSprites;

[SerializeField] private Color positiveHydrantColor;

[SerializeField] private Color negativeHydrantColor;

[SerializeField] private Color pipeLengthOutOfRangeColor;

[SerializeField] private Color pipeLengthInRangeColor;

[SerializeField] private Color pipeConnectedColor;

[SerializeField] private Color pipeActivatedColor;

public int SoundVolume => soundVolume;

// Hydrant

public int MinDistanceBetweenHydrants => minDistanceBetweenHydrants;

public int MinStartHydrantCount => minStartHydrantCount;

public int MaxStartHydrantCount => maxStartHydrantCount;

public int MinHydrantCapacity => minHydrantCapacity;

public int MaxHydrantCapacity => maxHydrantCapacity;

public int MinFillHydrantSpeed => minFillHydrantSpeed;

public int MaxFillHydrantSpeed => maxFillHydrantSpeed;

public int MinTurnsToNextHydrant => minTurnsToNextHydrant;

public int MaxTurnsToNextHydrant => maxTurnsToNextHydrant;

// Pipe

public int BonusPipeChance => bonusPipeChance;

public int MinStartAvaliablePipesCount => minStartAvaliablePipesCount;

public int MaxStartAvaliablePipesCount => maxStartAvaliablePipesCount;

public int MinPipeLengthMin => minPipeLengthMin;

public int MinPipeLengthMax => minPipeLengthMax;

public int MaxPipeLengthMin => maxPipeLengthMin;

public int MaxPipeLengthMax => maxPipeLengthMax;

public int MinStreamPower => minStreamPower;

public int MaxStreamPower => maxStreamPower;

// Minigame

public int FirstMiniGameTimeLimit => firstMiniGameTimeLimit;

public int SecondMiniGameTimeLimit => secondMiniGameTimeLimit;

public int ThirdMiniGameTimeLimit => thirdMiniGameTimeLimit;

// Developer

public int PipeItemSizeDelta => pipeItemSizeDelta;

public string GameResultsFilePath => Path.Combine(Environment.CurrentDirectory, gameResultsFileName);

public string SettingsFilePath => Path.Combine(Environment.CurrentDirectory, settingsFileName);

public Color PositiveHydrantColor => positiveHydrantColor;

public Color NegativeHydrantColor => negativeHydrantColor;

public Color PipeLengthOutOfRangeColor => pipeLengthOutOfRangeColor;

public Color PipeLengthInRangeColor => pipeLengthInRangeColor;

public Color PipeConnectedColor => pipeConnectedColor;

public Color PipeActivatedColor => pipeActivatedColor;

public Sprite GetRandomPipeSprite() => pipeSprites[Random.Range(0, pipeSprites.Count)];

public Sprite GetRandomHydrantSprite() => hydrantSprites[Random.Range(0, hydrantSprites.Count)];

public Vector3 GetRandomHydrantPosition() => new Vector3(Random.Range(-8.4f, 6.77f), Random.Range(-4.15f, 4.15f), 0);

public int GetRandomHydrantCapacity() => Random.Range(minHydrantCapacity, maxHydrantCapacity);

public int GetTurnsForNextHydrant() => Random.Range(minTurnsToNextHydrant, maxTurnsToNextHydrant);

private void Start() => LoadSettings();

public void ApplySettings(SettingsDto settingsDto)

{

// Game

soundVolume = settingsDto.SoundVolume;

minDistanceBetweenHydrants = settingsDto.MinDistanceBetweenHydrants;

minStartHydrantCount = settingsDto.MinStartHydrantCount;

maxStartHydrantCount = settingsDto.MaxStartHydrantCount;

minHydrantCapacity = settingsDto.MinHydrantCapacity;

maxHydrantCapacity = settingsDto.MaxHydrantCapacity;

minTurnsToNextHydrant = settingsDto.MinTurnsToNextHydrant;

maxTurnsToNextHydrant = settingsDto.MaxTurnsToNextHydrant;

minFillHydrantSpeed = settingsDto.MinFillHydrantSpeed;

maxFillHydrantSpeed = settingsDto.MaxFillHydrantSpeed;

bonusPipeChance = settingsDto.BonusPipeChance;

minStartAvaliablePipesCount = settingsDto.MinStartAvaliablePipesCount;

maxStartAvaliablePipesCount = settingsDto.MaxStartAvaliablePipesCount;

minPipeLengthMin = settingsDto.MinPipeLengthMin;

minPipeLengthMax = settingsDto.MinPipeLengthMax;

maxPipeLengthMin = settingsDto.MaxPipeLengthMin;

maxPipeLengthMax = settingsDto.MaxPipeLengthMax;

minStreamPower = settingsDto.MinStreamPower;

maxStreamPower = settingsDto.MaxStreamPower;

firstMiniGameTimeLimit = settingsDto.FirstMiniGameTimeLimit;

secondMiniGameTimeLimit = settingsDto.SecondMiniGameTimeLimit;

thirdMiniGameTimeLimit = settingsDto.ThirdMiniGameTimeLimit;

// Developer

pipeItemSizeDelta = settingsDto.PipeItemSizeDelta;

settingsFileName = settingsDto.SettingsFileName;

gameResultsFileName = settingsDto.GameResultsFileName;

positiveHydrantColor = settingsDto.PositiveHydrantColor;

negativeHydrantColor = settingsDto.NegativeHydrantColor;

pipeLengthOutOfRangeColor = settingsDto.PipeLengthOutOfRangeColor;

pipeLengthInRangeColor = settingsDto.PipeLengthInRangeColor;

pipeConnectedColor = settingsDto.PipeConnectedColor;

pipeActivatedColor = settingsDto.PipeActivatedColor;

SaveSettings(settingsDto);

Sound.UpdateVolumeSettings();

}

public SettingsDto CreateSettingsDto()

{

return new SettingsDto

{

PipeItemSizeDelta = pipeItemSizeDelta,

SettingsFileName = settingsFileName,

GameResultsFileName = gameResultsFileName,

PositiveHydrantColor = positiveHydrantColor,

NegativeHydrantColor = negativeHydrantColor,

PipeLengthOutOfRangeColor = pipeLengthOutOfRangeColor,

PipeLengthInRangeColor = pipeLengthInRangeColor,

PipeConnectedColor = pipeConnectedColor,

PipeActivatedColor = pipeActivatedColor

};

}

private void SaveSettings(SettingsDto settingsDto)

{

using (var stream = new FileStream(Settings.SettingsFilePath, FileMode.OpenOrCreate, FileAccess.Write))

{

new XmlSerializer(typeof(SettingsDto)).Serialize(stream, settingsDto);

}

}

private void LoadSettings()

{

if (!File.Exists(Settings.SettingsFilePath))

{

return;

}

SettingsDto settingsDto;

using (var stream = new FileStream(Settings.SettingsFilePath, FileMode.Open, FileAccess.Read))

{

settingsDto = (SettingsDto)new XmlSerializer(typeof(SettingsDto)).Deserialize(stream);

}

ApplySettings(settingsDto);

}

}

using System.Collections.Generic;

using System.Linq;

using UnityEngine;

public class SoundManager : Manager<SoundManager>

{

public override bool DestroyOnLoad => false;

[SerializeField] private AudioSource ambientChannel;

[SerializeField] private AudioSource soundEffectsChannel1;

[SerializeField] private AudioSource soundEffectsChannel2;

[SerializeField] private List<AudioClip> menuAmbient;

[SerializeField] private List<AudioClip> gameAmbient;

[SerializeField] private List<AudioClip> hydrantAppear;

[SerializeField] private List<AudioClip> hydrantEndTurn;

[SerializeField] private List<AudioClip> notificationAppear;

[SerializeField] private List<AudioClip> pipeItemAppear;

[SerializeField] private List<AudioClip> pipeItemSelected;

[SerializeField] private List<AudioClip> pipeRendererStart;

[SerializeField] private List<AudioClip> pipeRendererEnd;

[SerializeField] private List<AudioClip> buttonClicked;

[SerializeField] private List<AudioClip> minigameAppear;

[SerializeField] private List<AudioClip> gameOverAppear;

[SerializeField] private List<AudioClip> pipeDecorationClick;

[SerializeField] private List<AudioClip> hydrantDecorationClick;

[SerializeField] private List<AudioClip> r2d2DecorationClicked;

[SerializeField] private List<AudioClip> resetSelected;

public void PlayMenuAmbient()

{

if(ambientChannel.clip != menuAmbient.First())

Play(ambientChannel, menuAmbient);

}

public void PlayGameAmbient()

{

if (ambientChannel.clip != gameAmbient.First())

Play(ambientChannel, gameAmbient);

}

public void PlayHydrantAppear() => Play(soundEffectsChannel1, hydrantAppear);

public void PlayHydrantEndTurn() => Play(soundEffectsChannel2, hydrantEndTurn);

public void PlayNotificationAppear() => Play(soundEffectsChannel1, notificationAppear);

public void PlayMinigameAppear() => Play(soundEffectsChannel2, minigameAppear);

public void PlayerGameOverAppear() => Play(soundEffectsChannel1, gameOverAppear);

public void PlayButtonClicked() => Play(soundEffectsChannel1, buttonClicked);

public void PlayPipeRendererStart() => Play(soundEffectsChannel2, pipeRendererStart);

public void PlayPipeRendererEnd() => Play(soundEffectsChannel1, pipeRendererEnd);

public void PlayPipeItemAppear() => Play(soundEffectsChannel1, pipeItemAppear);

public void PlayPipeItemSelected() => Play(soundEffectsChannel1, pipeItemSelected);

public void PlayPipeDecorationClicked() => Play(soundEffectsChannel1, pipeDecorationClick);

public void PlayHydrantDecorationClicked() => Play(soundEffectsChannel1, hydrantDecorationClick);

public void PlayR2D2DecorationClicked() => Play(soundEffectsChannel1, r2d2DecorationClicked);

public void PlayResetSelected() => Play(soundEffectsChannel1, resetSelected);

private void Play(AudioSource channel, List<AudioClip> clip)

{

channel.clip = clip[Random.Range(0, clip.Count)];

channel.pitch = Random.Range(0.9f, 1.1f);

channel.Play();

}

public void UpdateVolumeSettings()

{

soundEffectsChannel1.volume

= soundEffectsChannel2.volume

= ambientChannel.volume

= (float)Settings.SoundVolume / 100;

}

}

using UnityEngine;

using UnityEngine.SceneManagement;

public class ManagementController : MonoBehaviour

{

[SerializeField] private GameObject settingsManagerPrefab;

[SerializeField] private GameObject scoreManagerPrefab;

[SerializeField] private GameObject soundManagerPrefab;

private void Start()

{

Instantiate(settingsManagerPrefab);

Instantiate(scoreManagerPrefab);

Instantiate(soundManagerPrefab);

SceneManager.LoadScene("Menu");

}

}