

Střední průmyslová škola strojní   
a elektrotechnická a Vyšší odborná škola, Liberec 1, Masarykova 3

2D logická hra v Unity

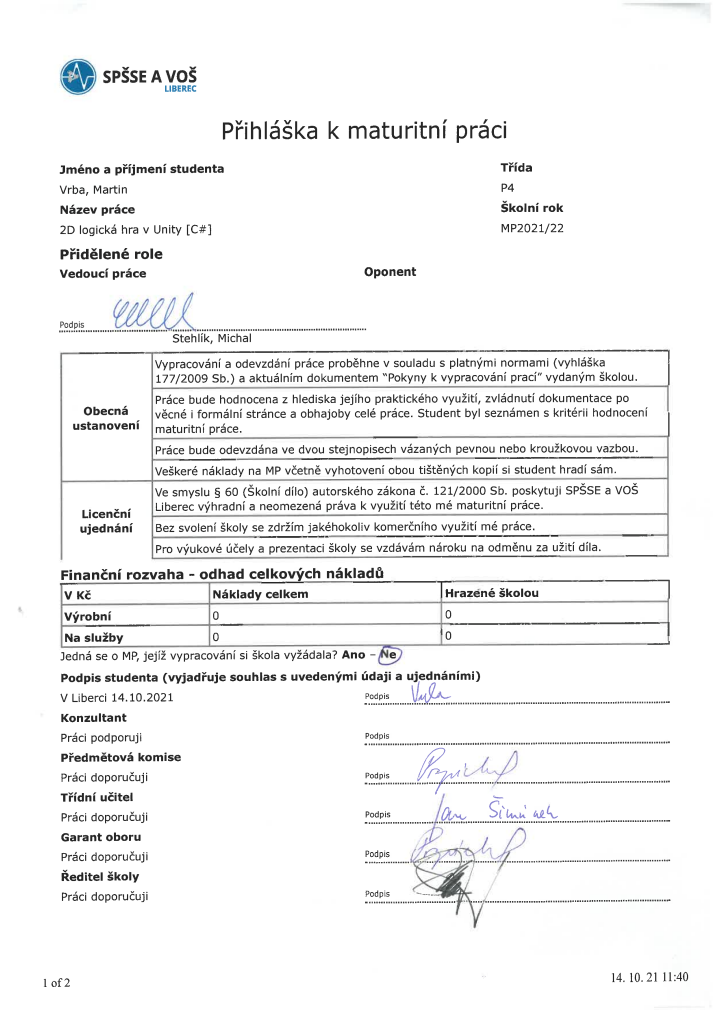
Maturitní práce

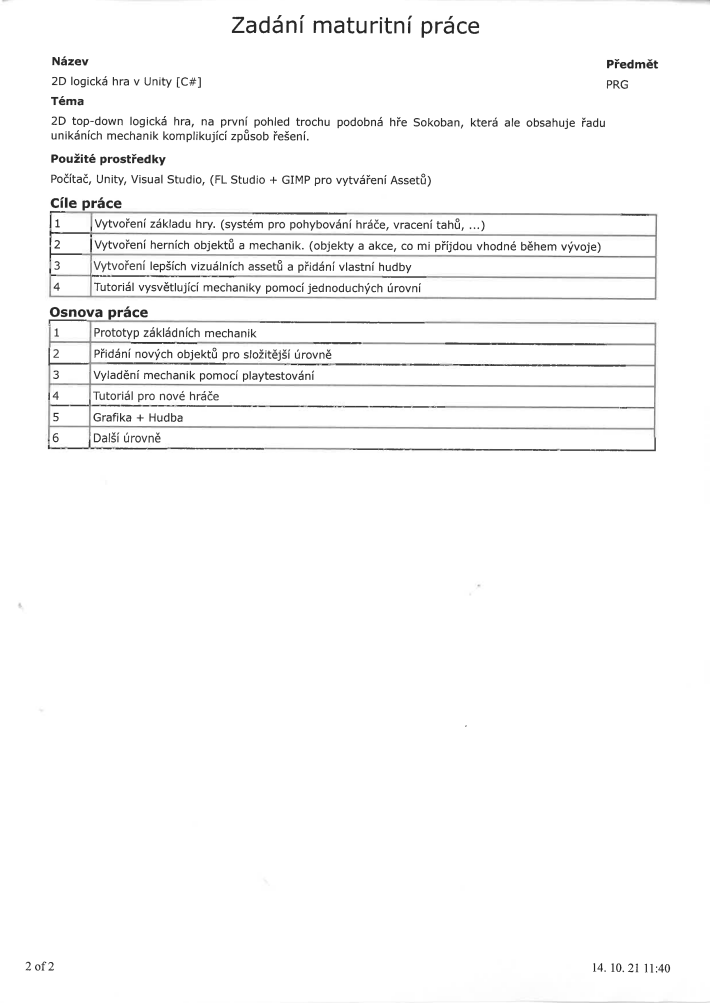
Autor **Martin Vrba**

Obor **Informační technologie**

Vedoucí práce **Mgr. Michal Stehlík**

Školní rok **2021/2022**





Anotace

Práce se zabývá vývojem logické hry založené na japonské hře Sokoban. Popisuje vnitřní fungovaní samotné hry, vývoj vlastního nástroje pro vlastní úrovně, a možné využití v budoucnu.

Summary

This work deals with development of a puzzle game based on Japanese videogame Sokoban. It describes internal functions of the game itself, custom tool for generating own levels, and possible future use and expansion.

Čestné prohlášení

Prohlašuji, že jsem předkládanou maturitní práci vypracoval sám a uvedl jsem veškerou použitou literaturu a bibliografické citace.

V Liberci dne

Martin Vrba

Obsah

[Úvod 1](#_Toc97988835)

[1 Použité technologie a nástroje 2](#_Toc97988836)

[1.1 Herní engine 2](#_Toc97988837)

[1.1.1 Rozhraní Unity 3](#_Toc97988838)

[1.2 Kód 2](#_Toc97988839)

[1.2.1 Programovací jazyky 3](#_Toc97988840)

[1.2.2 Vývojové prostředí 3](#_Toc97988841)

[1.3 Audio 2](#_Toc97988842)

[1.3.1 Sfxr 3](#_Toc97988843)

[1.3.2 FL Studio 3](#_Toc97988844)

[1.3.3 Audacity 3](#_Toc97988845)

[1.4 Grafika 2](#_Toc97988846)

[2 Princip hry 2](#_Toc97988847)

[2.1 Sokoban 2](#_Toc97988848)

[2.2 Vlastní rozšíření 2](#_Toc97988849)

[2.3 Cíl vývoje 3](#_Toc97988850)

[3 Vývoj 4](#_Toc97988851)

[3.1 Datová struktura 4](#_Toc97988852)

[3.1.1 Vlastní datové typy 4](#_Toc97988853)

[3.1.2 Entity 5](#_Toc97988854)

[3.1.3 Rozhraní 6](#_Toc97988855)

[3.1.4 Levely 7](#_Toc97988856)

[3.2 Editor 9](#_Toc97988857)

[3.3 Cyklus hry 10](#_Toc97988858)

[3.3.1 Načtení 10](#_Toc97988859)

[3.3.2 Vykreslování 11](#_Toc97988860)

[3.3.3 Reakce na vstup 13](#_Toc97988861)

[Závěr 18](#_Toc97988862)

[Seznam zkratek a odborných výrazů 20](#_Toc97988863)

[Seznam obrázků 21](#_Toc97988864)

[Použité zdroje 22](#_Toc97988865)

[A. Seznam přiložených souborů I](#_Toc97988866)

Úvod

Programování bylo mým koníčkem už od třetí třídy základní školy, kdy jsem se poprvé setkal s roboty Lego Mindstorms, na kterých jsem poprvé pochopil koncept proměnných, smyček a podmínek. Od té doby jsem se sám učil základům jazyků jako Javascript, PHP, nebo Python. Sice jsem za tu dobu vytvořil mnoho miniaturních projektů, většina sloužila pouze jako experiment pro získání více zkušeností. Teprve když jsem začal studovat na Prumyslové škole v Liberci, učil jsem se dobrým návykům, organizaci kódu, a obecně konceptu objektově orientovaného programování. Díky tomu jsem byl schopen rozšířit své znalosti i do vývoje her v enginu Unity, který mi do té doby nedával tolik smysl. Cílem této práce je tedy vytvořit hru, která bude jednodušše rozšiřitelná o nové mechaniky a úrovně. Nejedná se tedy o finální verzi hry, ale spíš základ, který má úrovně uložené v externích souborech.

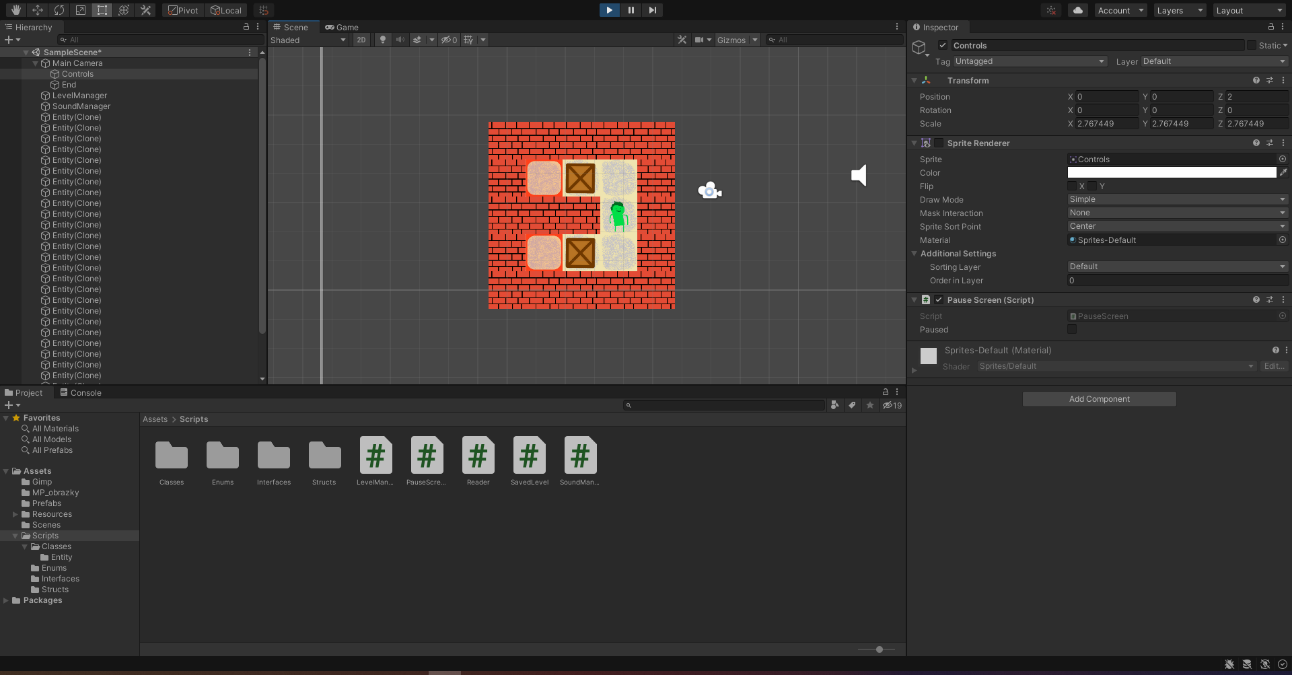
# Použité technologie a nástroje

V této kapitole jsou všechny programy a jazyky, které jsem k vytvoření této práce použil.

## Herní engine

Jedna z nejvíc důležitých rozhodnutí pro vývoj této hry. Pro mě byl engine Unity lákavým kandidátem už od první chvile, kdy jsem se začal učit objektově orientované programování a jazyk C#. V minulosti jsem si trochu vyzkoušel práci i v Unreal Engine, Godot, a Phaser. Myslím si, že jsem se rozhodl správně pro Unity, protože jsem v něm měl již více zkušeností a jazyk C# je ideální pro organizování rozšiřitelného kódu

### Rozhraní Unity



Obrázek 1 Unity3D

Unity je 3D herní engine poprvé vydaný v roce 2005 firmou Unity Technologies. Od té doby byl až do dnes neustále vyvýjen a za ten čas v něm bylo vytvořeno mnoho známých titulů v herním průmyslu jako například:

* Monument Valley
* Ori and the Blind Forest
* Pokémon Go
* The Long Dark
* Inside
* Gone Home
* Furi

Přestože je Unity ve své podstatě engine pro trojrozměrné hry, má velmi dobrou podporu i pro 2D grafiku a mechaniky.

Unity pracuje s takzvanými hernímy objekty (GameObject). Každý má svoje vlastnosti (například název, pozici v prostoru, rotaci, nebo zvětšení) a komponenty (zvuky, definice hran pro kolize, simulaci fyzikálního tělesa)

## Kód

Vzhledem k tomu, že je má hra hlavně o funkčnosti herních mechanik, je kód velmi důležitou součástí vývoje a budoucího rozšiřování.

### Programovací jazyky

Základní jazyk v Unity je C# (C Sharp). Je objektově orientovaný a syntaxí je podobný jazykům jako Java nebo C++. C# se běžně využívá pro vytváření desktopových, mobilních, i webových aplikací. Unity má svojí vlastní knihovnu pro ovládání prvků jménem UnityEngine, která obsahuje mnoho užitečných funkcí a metod. Dá se pomocí ní dotazovat na různé podmínky běžící hry jako třeba velikost okna, pozice myši, čas mezi vykreslenými snímky hry a mnoho dalších informací, které budou ještě zmíněny. V jazyce C# je psaná veškerá logika hry a všechny skripty pracující se soubory a hernímy objekty.

Přestože vše v Unity se programuje v jazyce C#, zahrnuje moje práce i webový editor úrovní, který byl vytvořen v kombinaci HTML, CSS a Javascript. Je to statická stránka, která lze otevřít offline ve webovém prohlížeči. Na předávání informací o levelu se používá datový formát JSON (JavaScript Object Notation), takže lze úrověň převést z Javascriptu do C# bez problémů z kompatibilitou. Konkrétní provedení aplikace bude ještě podrobněji vysvětleno.

### Vývojové prostředí

#### Visual Studio

Visual Studio je IDE od firmy Microsoft, určené k vyvýjení programů v jazycích jako C#, C++, Python a další. Pro mé účely funguje jako editor C# skriptů v Unity. Má automatické doplňování klíčových slov a přehledné UI pro opravu chyb v syntaxi. Dělá mnoho pracných činností za programátora. Například při používání externích metod importuje potřebné knihovny pomocí jednoho kliknutí myši.

#### VS Code

Visual Studio Code je

## Audio

K vytvoření veškerého audia jsem použil programy sfxr, FL Studio a Audacity, každý k jinému účelu.

### Sfxr

Sfxr je malý opensource program vytvořený Tomasem Petterssonem, který umožňuje generování 8-bitových zvukových efektů. Lze v něm upravovat a mutovat parametry, takže je možné se postupně dopracovat k žádoucímu zvuku. V sfxr byly vytvořeny zvukové efekty.

### FL Studio

Původně zvané FruityLoops, FL Studio je DAW vyvýjené belgickou společností Image-Line. Používá se převážně na vytváření hudby. Má 4 edice lišící se v ceně a v některých vlastnostech a funkcí. Existuje sice bezplatná zkušební verze, ale já jsem měl program zakoupený, takže jsem toho využil. V FL Studiu byla vytvořena hudba na pozadí.

### Audacity

Audacity je známý opensource audio editor. Umožňuje aplikovat velkou škálu efektů na audio klipy a zároveň je mixovat či stříhat. Obvykle využívám Audacity k více účelům, ale zde jsem pouze upravoval hlasitosti jednotlivých zvuků a odstraňoval šum.

## Grafika

Všechny assety byly vytvořeny v programu GIMP (GNU Image Manipulation Program). DOPLNIT

# Princip hry

## Sokoban

Základní mechaniky hry jsou založené na starší japonské hře Sokoban (česky Skladník). V její nejjednodušší formě se jedná o 2D hru v mřížce, kde hráč pohybuje jednou postavou. Jeho úkolem je dostat všechny bedny na předem určená místa. Bedny může pouze tlačit před sebou, a to maximálně jednu. Nikdy nejde tlačit dvě a více beden, nebo naopak bedny přitahovat.

A screenshot of a computer screen

Description automatically generated with medium confidence

Obrázek 2 Originální Sokoban (1984)

## Vlastní rozšíření

Původní plán byl vytvořit hru, která se bude Sokobanem pouze inspirovat, ale bude mít mnoho prvků navíc, které by zásadně rozšířily možnosti kreativity při tvoření levelů. Mezi takové prvky by mohlo patřit třeba:

* Led, který omezí volný pohyb. Místo toho bude pokračovat ve stejném směru, dokud se nedostane na druhou stranu. Led by ovlivnil i ostatní prvky.
* Dveře, a k nim propojené tlačítka a klíče. Dveře blokují hráči a objektům průchod, dokud nejsou odemčeny buď stáním na tlačítku, nebo přímým odemčením pomocí klíče. Samozřejmě by šly i další způsoby jak některé z dveří ovládat.
* Skleněné bedny a případně jiné objekty, které blokují tlačení beden, a hráč je musí rozbít přímým dotykem.
* Nestabilní podlaha. Když se jakýkoliv objekt ocitne na tomto poli, aktivuje nestabilní podlahu. V moment kdy všechny objekty opustí aktivní nestabilní podlahu, vytvoří se na původním místě díra, která blokuje průchod hráče. Bylo by možné přidat ještě další mechaniku, kde přebytečné bedny lze zatlačit na pole díry, a tím vytvořit jakýsi most. Do té doby by nestabilní podlaha fungovala jako jednosměrná překážka.
* Robot chodící ze strany na stranu a tlačí vše před ním. Udělá krok po každém kroku hráče. Pokud hráč počká na místě, robot se stále posune. Robot by mohl na rozdíl od hráče posouvat více beden na jednou. Mezi různá řešení úrovní by mohlo patřit i načasování si tahů tak, aby robot tlačil bedny ve vhodný okamžik přes led, a hráč je pak dostal na místo, kam by to sám nezvládl.

Určitě během budoucího rozšiřování hry narazím na vhodné úpravy těchto nápadů a nebo dokonce i jiné zatím nezmíněné nápady.

## Cíl vývoje

Směr, kterým jsem se vydal krátce po založení projektu, není udělat konkrétní hru obsahující víše zmíněné prvky a mechaniky. Je tím naopak vytvořit jakýsi engine, do kterého lze vložit vygenerované levely, co automaticky poskládájí GameObjecty do ovladatelné hry.

# Vývoj

## Datová struktura

Původní plán byl mít zakódované herní prvky pomocí čísel v dvourozměrném poli. Toto pole by se po každém tahu aktualizovalo a následně promítlo na obrazovku pomocí pozicovaných obrázků. Velice brzo jsem ale přišel na problém, který se ukázal až kritickým. V tomto systému by bylo obtížné zakódovat více objektů na jednom místě (v tomto bodě bylo pro mě ještě pořád důležité udělat plnou hru s více komplexními mechanikami, a mezi takové by mohly patřit i objekty které mohou být naskládány na sobě). Další problém, který na mě pomalu dopadal byla neschopnost tohoto systému zakódovat některé vlastnosti objektů, jako je třeba směr. Rozhodl jsem se tedy vymyslet lepší způsob, jak uchovávat informace o herním poli a jeho obsahu. Více efektivní a rozšiřitelný postup by byl využít objektově orientovaného programování a vytvořit si systém dědičnosti a rozhrání.

### Vlastní datové typy

Aby se mi v kódu lehce pracovalo s daty, vytvořil jsem si několik pomocných datových typů a naprogramoval i vlastní operátor pro zjednodušení zápisu.

#### TileType

Tento enum je zde hlavně pro konvertování z JSON souboru na samostatné třídy. Obsahuje výčet všech možných entit.

public enum TileType

{

Floor,

Wall,

Player,

Box,

Storage

}

#### Direction

Direction je enum pro značení směru. Mezi 4 strany je zde ještě pátá možnost pro žádný směr.

public enum Direction

{

None = 0,

Up = 1,

Down = 2,

Left = 3,

Right = 4

}

#### Coordinates

Pro zapisování souřadnic jsem použil struct o dvou proměnných (pozice X a Y). Mimo to je zde i vlastní operátor pro „přičítání“ směru k souřadnicím. Funguje to tak, že pokud by se k souřadnici měl přičíst směr nahoru, vrátí operátor novou souřadnici,

### Entity

Entitou se v mé práci myslí jakákoliv věc, která bude mít svůj vlastní sprite. Samotná třída Entity neměla na začátku vývoje žádné vlastnosti kromě souřadnic a odkazu na GameObjekt, který k ní byl vytvořen a přiřazen.

public abstract class Entity

{

public GameObject MappedObject;

public Coordinates Position;

protected LevelState state;

protected SpriteRenderer renderer;

protected Dictionary<string, Sprite> sprites;

//... metody pro ovladani sprite

Z této třídy by se následně vytvářely dědící třídy, každá pro jinou entitu.

### Rozhraní

Abych si udržel pořádek v tom, která entita dělá jaké činnosti, vytvořil jsem si rozhrání pro vlastnosti, které jednotlivé entity mohou mít. Mám v plánu v budoucnosti přidat alespoň některé z rozšiřujících mechanik, takže následující list je pouze pro objekty v původním Sokobanovi.

#### IDirectionFacing

Všechny entity, které mají schopnost se otáčet, nebo dívat na konkretní stranu, mají rozhraní IDirectionFacing. Obsahuje metody pro získání a modifikování směru a zároveň souřadnici, na kterou se entita právě „dívá“. V tuto chvíli je jedinout entitou s tímto rozhraním samotný hráč.

public interface IDirectionFacingEntity

{

public Direction GetDirection();

public void SetDirection(Direction newDir);

public Coordinates LookingAt();

}

#### IMovingEntity

Cokoliv, co se může pohybovat (ať už samo, nebo pomocí jiné entity) má rozhraní IMoving. Obsahuje pouze metodu pro posunutí na nové souřadnice.

public interface IMovingEntity

{

public void Move(Coordinates destination);

}

#### IObstacle

IObstacle pozůstatek z původního nápadu na dveře otevíratelné tlačítky. Obsahuje možnost získat a kontrolovat, jestli je entita s IObstacle „otevřená“. Nyní má toto rozhraní pouze Wall a je konstantně zavřená.

public interface IObstacle

{

public bool Opened { get; }

public void Open(bool open);

}

#### IPushable

Jediná entita, kterou může hráč momentálně posouvat, je Box. Přišlo mi lepší tuto vlastnost zobecnit, kdybych v budoucnosti chtěl přidat jiné posouvatelné prvky.

Metoda Push je typu bool proto, aby mohla dávat zpětnou vazbu, jestli akce posunutí proběhla, nebo entitě něco brání.

public interface IPushable

{

bool Push(Direction direction);

}

### Levely

Už od začátku jsem věděl, že chci levely mít nějak zakódované do souborů, aby se s nima dalo jednoduše zacházet a importovat. Vytvořil jsem si tedy třídu SavedLevel, která je serializovatelná (tj. dá se převést z instance třídy na JSON a i naopak) a obsahuje pole EntityConstructor, ze čehož se dá vytvořit každá entita v levelu.

[System.Serializable]

public class SavedLevel

{

public string Name;

public EntityConstructor[] Entities;

}

EntityConstructor je také serializovatelná třída, obsahující vše informace o entitě, která bude vytvořena.

[System.Serializable]

public class EntityConstructor

{

//Entity

public TileType T; //Type

public int CoordinateX;

public int CoordinateY;

//IDirectionFacing

public Direction Direction;

//IObstacle

public bool Opened;

}

Se SavedLevel pracuje asi nejduležítější třída v celé práci, LevelManager, která má na starost všechnu režii hry, vnímá vstupy hráče, načítá nové levely, a ovládá všechny entity.

public class LevelManager : MonoBehaviour

{

//vlastnosti a promenne

// ...

void Start() {…}

void ReadAllLevels() {…}

bool LoadLevel(int id) {…}

void Update() {…}

void Undo() {…}

void Restart() {…}

void Tick(Direction pressed) {…}

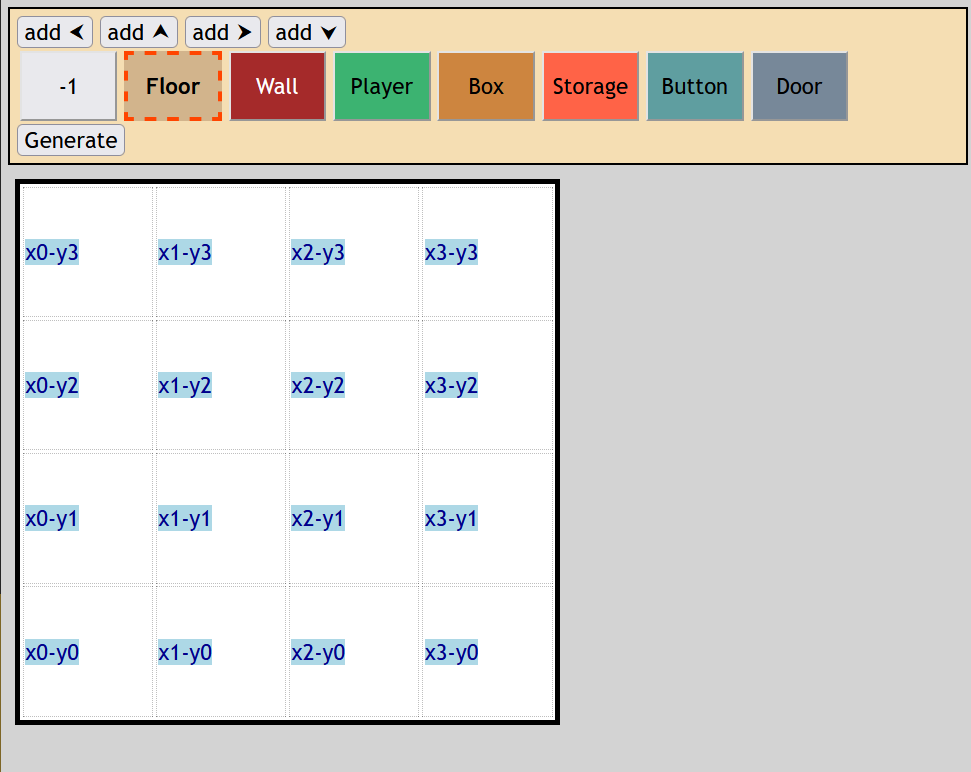
void Render() {…}

void RenderTile(Entity entity, int z) {…}

}

## Editor

Abych nemusel levely ručně psát do souborů, vytvořil jsem si vlastní editor pomocí HTML a JavaScript. K dokonalosti má sice daleko, ale jako nástroj pro tvoření vlastních úrovní postačí.



Obrázek 3 Rozhraní editoru

Nynější verze editoru umožňuje následující:

* Zvětšování plochy mřížky
* Možnost vybrat typ entity a následně jej přidat na políčka
* Vyplnit souvislou prázdnou plochu podlahou
* Vygenerovat JSON soubor

## Cyklus hry

### Načtení

Po spuštění programu se do paměti uloží odkazy na všechny JSON soubory v adresáři určeném pro levely.

//LevelManager.cs

void ReadAllLevels()

{

levelTexts = Resources.LoadAll<TextAsset>(path);

}

Následuje samotné načtení první úrovně. LevelManager si pamatuje, kolikátý level je právě načten, takže lze jednoduše postupovat v řadě. Při každém tahu se zkontroluje, jestli je již úroveň splněna. Pokud ano, tak se právě provede načtení následujícího JSON souboru. Pokud již není žádný další level, zobrazí se statický obrázek značící konec hry.

A picture containing graphical user interface

Description automatically generated

Obrázek 4 Konec hry

Po úspěšném zvolení souboru pro načtení se daný JSON zparsuje do SavedLevel. Pro každý EntityConstructor se vytvoří nová entita a k ní se automaticky přiřadí nová instance GameObjectu, aby se dala pozicovat a měnit sprite.

for (int i = 0; i < savedLevel.Entities.Length; i++)

{

EntityConstructor constructor = savedLevel.Entities[i];

GameObject newObject = Instantiate(prefabEntity);

Entity newEntity = null;

switch (constructor.T)

{

case TileType.Floor:

newEntity = new Floor(constructor, newObject);

break;

case TileType.Wall:

newEntity = new Wall(constructor, newObject);

break;

// ...

}

Environment.Add(newObject);

levelState.Add(newEntity);

}

### Vykreslování

Když je hotový LevelState se všemi používanými entitami, je potřeba úroveň vykreslit. Na to je v LevelManageru metoda Render, která projde entitami, a upraví podle jejich dat propojené GameObjecty. Některé objekty se musí vykreslit nad jinými, takže se zde používá proměnná Z, co ovlivňuje vzdálenost od kamery

void Render()

{

int z = 0; //floor level

foreach (Floor floor in level.Floors)

{

RenderTile(floor, z);

}

z = -1; //object level

RenderTile(level.Player, z);

foreach (Wall wall in level.Walls)

{

RenderTile(wall, z);

}

foreach (Box box in level.Boxes)

{

RenderTile(box, z);

}

foreach (Storage storage in level.Storages)

{

RenderTile(storage, z);

}

}

RenderTile pouze aktualizuje sprite objektu a nastaví jeho pozici.

void RenderTile(Entity entity, int z)

{

entity.MappedObject.transform.position = new

Vector3(entity.Position.x, entity.Position.y, z)

\* tileSize;

entity.UpdateSprite();

}

### Reakce na vstup

Metoda Update se v Unity automaticky provádí mezi každým vykresleným snímkem. Pomocí třídy Input z knihovny UnityEngine lze zjistit, jestli od posledního snímku uživatel stiskl nějakou klávesu a případně kterou. V každém cyklu Update se tedy ptám, jestli byl zaznamenán nějaký vstup, který by měl ovlivnit stav hry. Mezi takové vstupy patří:

* Pohyb hráče na 4 strany
* Čekání na místě (pro případné budoucí využití v entitách, které se aktivně samy pohybují)
* Vracení tahů
* Zobrazení pomocného obrázku s ovládáním (to se ale provádí v jiné části kódu)

Pokud je zadán vstup pro pohyb nebo čekání, LevelManager provede Tick, což je provedení vnitřní logiky všech entit. Třída Entity má totiž virtuální (přepsatelnou) metodu PerformTick, kterou si každá děděná entita implementuje sama.

//Entity.cs

public virtual void PerformTick(Direction input)

{

throw new NotImplementedException();

}

//Player.cs

public override void PerformTick(Direction input)

{

if (input != Direction.None)

{

///change direction

SetDirection(input);

Tile destination = state[LookingAt()];

if (destination.Opened)

{

if (destination.Box != null)

{

Box box = destination.Box;

if (box.Push(direction))

{

Move(Position + direction);

sound.Play("push");

}

}

else

{

Move(Position + direction);

sound.Play("step");

}

}

}

else

{

Debug.Log("input is none");

}

}

Mnoho entit ve své implementaci PerformTick používá třídu Tile, což je jákýsi zprostředkovatel mezi entitami a třídou LevelState. Tile obsahuje všechny potřebné informace o daném poli.

public class Tile

{

private LevelState state;

public Tile(LevelState state)

{

this.state = state;

Entities = new List<Entity>();

}

public IList<Entity> Entities { get; private set; }

public void Add(Entity entity)

{

Entities.Add(entity);

}

public void Update(Entity entity)

{

state[entity.Position].Add(entity);

Entities.Remove(entity);

}

public bool Opened

{

get

{

bool hasClosedObstacle = false;

bool hasFloor = false;

foreach (Entity entity in Entities)

{

if (entity is Floor)

hasFloor = true;

if (entity is IObstacle)

{

if (!(entity as IObstacle).Opened)

{

hasClosedObstacle = true;

}

}

}

return hasFloor && !hasClosedObstacle;

}

}

public Box Box

{

get

{

foreach (Entity entity in Entities)

{

if (entity is Box)

{

return entity as Box;

}

}

return null;

}

}

}

Závěr

Když jsem poprvé dostal myšlenku tohoto tématu maturitní práce, nikdy by mě nenapadlo, kolik se toho vlastně musí vytvořit a zvládnout, než se spustí samotný vývoj obsahu hry. Musel jsem si vytvořit vlastní systém pro ukládání dat, interakce mezi jednotlivými prvky hry, a způsob jak je vykreslovat při každé změně. Určitě by to šlo vyřešit jednodušeji, ale jak už bylo několikrát řečeno, to nakonec nebyl můj cíl. Mým účelem bylo vytvořit solidní základ, do kterého bude možné přidávat všelijaké nové nápady a mechaniky.

Kdyby výsledkem mé práce byla nerozšiřitelná kopie hry Sokoban, nemělo by to pro mě žádný smysl. Takto jsem se naopak naučil mnoho zkušeností o organizaci práce, nebo o přemýšlení nad budoucím vývojem. Hlavní pozitivní stránka mého finálního cíle je to, že mám stále motivaci na této práci pokračovat. Snad se mi i podaří implementovat některé z uvedených plánovaných rozšíření. To je něco, co by s původním přístupem nebylo vůbec možné.

Sice mám základ hotový, ale stále je do mé představy dokončené hry daleko. Zde jsou tedy některé ze změn, na kterých mám v plánu ještě pracovat:

* Lepší editor levelů. Momentální verze je rychlá verze co jsem si vytvořil pouze pro svoje účely během několika hodin. Po každém přidání mechaniky do hry se musí editor ručně aktualizovat. Další důležitá vlastnost, co zde chybí je editování již existujících souborů. Zatím lze levely pouze generovat z prázdné plochy. Pravděpodobně přepíšu celý backend editoru do C#, aby se vytvořila kompatibilita s daty ve hře.
* Rozšíření hry. Jeden z hlavních důvodů, proč jsem vytvořil vlastní třídy a rozhraní pro entity, je rozšiřitelnost mechanik. S tímto systémem lze přidávat prvky tak, aby zapadaly do již existujícího a fungujícího mechanismu.
* Možnost importování souborů po kompilaci, případně i za běhu. Nyní lze levely importovat pouze do nezkompilovaného projektu Unity. Po zkompilování se soubory zakódují a nelze je jednoduše měnit. Nový systém by tedy měl umožnit otevírát a načítat soubory s úrovněmi rovnou při spuštění, nebo později i za běhu. Viděl jsem podobné mechaniky u jiných her vytvořených v Unity. Jen jsem se k vlastní implementaci ještě nedostal.
* Podpora vlastních barevných variací a obrázků pro jednotlivé entity. Případně i jiné aspekty jako vlastní zvuky a hudba. Pomáhalo by to mému cílu vytvořit univerzální Sokoban engine, a ne jen jednu izolovanou hru.

Rozhodně nechci na tomto projektu přestat pracovat. Mám pro něj plán a vizi, kterou bych rád naplnil. Je to v podstatě zatím nejvíc dokončená programátorská práce, jakou jsem kdy udělal, a nerad bych zahodil tuto příležitost dotáhnout to do úplného konce.

Seznam zkratek a odborných výrazů

JSON

JavaScript Object Notation – formát dat

Sprite

obrázek reprezentující herní objekt

IDE

Integrated Development Environment – program určený pro vývoj software

Seznam obrázků

[Obrázek 1 Unity3D 2](#_Toc97990412)

[Obrázek 2 Originální Sokoban (1984) 6](#_Toc97990413)

[Obrázek 3 Rozhraní editoru 13](#_Toc97990414)

[Obrázek 4 Konec hry 14](#_Toc97990415)

Použité zdroje

1. **Pettersson, Tomas.** sfxr. *DrPetter's homepage.* [Online] https://www.drpetter.se/project\_sfxr.html.

2. **Image-Line.** FL Studio. [Online] https://www.image-line.com/fl-studio/.

3. **Unity Technologies.** *Unity.* [Online] https://unity.com/our-company.

4. **Domènech, Jordi.** Sokoban 30th anniversary. [Online] http://sokoban-jd.blogspot.com/2012/10/sokoban-30th-anniversary-1982-2012.html.

5. **Microsoft.** Visual Studio Code. [Online] https://code.visualstudio.com/docs.

6. **The GIMP Team.** Gimp. [Online] https://www.gimp.org/.

7. **Crockford, Douglas.** JSON. [Online] https://www.json.org/json-en.html.

8. **Audacity.** Audacity. [Online] https://www.audacityteam.org/.

9. **Huber, Thomas Claudius.** [Online] https://www.thomasclaudiushuber.com/2021/02/25/c-9-0-pattern-matching-in-switch-expressions/.

10. **Wikipedia.** Sokoban. [Online] https://en.wikipedia.org/wiki/Sokoban.

1. Seznam přiložených souborů

Na přiloženém datovém nosiči se nacházejí následující soubory a složky:

* **MP2010-Vrba-Martin-P4-2D\_logická\_hra\_v\_Unity.docx** – editovatelná verze dokumentace maturitní práce
* **MP2010-Vrba-Martin-P4-2D\_logická\_hra\_v\_Unity.pdf** – tisknutelná verze dokumentace maturitní práce
* **Unity Projekt.zip** – nezkompilovaný projekt pro Unity 2020.3.12f1
* **Hra.zip** – zkompilovaná a spustitelná hra