# Czech-Man
# Doprovodný dokument

Autor: Jan Hartman
Garant: RNDr. Tomáš Holan, Ph.D.
Studium: bakalářské, 1. ročník, letní semestr 2022/2023
Předmět: Programování 2

2. března 2025

## Obsah

# 1 Úvodní část

## 1.1 Anotace

Program Czech-Man je napodobeninou klasické hry Pac-Man z roku 1980. Neklade si za cíl být naprosto přesnou kopií hry, hlavní funkčnosti by však měly být zachovány. Hra je vyvíjena v programovacím jazyce C#.

## 1.2 Formát tohoto dokumentu

Pokud se v tomto dokumentu budu vyjadřovat k nějaké třídě nebo objektu jménem stejným jako ve zdrojovém kódu, tak budu psát odpovídající název **tučně**. Jako třeba pokud zmiňuji třídu **GameManager**.

Naopak pokud zmiňuji nějaké klíčové slovo jazyka C# nebo nějakou built-in třídu, tak budu používat formát `verbatim`. Například `char` nebo `Bitmap`.

## 1.3 Zvolené vývojové prostředí

Prvotním úmyslem bylo hru vyvíjet ve vývojovém prostředí Unity, které je přímo určeno k vytváření počítačových her. S Unity jsem však neměl moc velké předchozí zkušenosti a po krátké snaze naučit se základy fungování tohoto prostředí jsem jeho užití zavrhl, jelikož naučit se pracovat v tomto prostředí vyžaduje mnoho specifických znalostí. Zároveň mi prostředí Unity nevyhovovalo v jeho zaměření na přesné fyzikální simulace a také fakt, že se nedá toto zaměření plně „odstínit“. Chtěl jsem hru vytvářet v takovém prostředí, ve kterém budu mít nad svým herním světem co možná největší kontrolu, a tak jsem se nakonec rozhodl pro složitější postup, který mi však umožnil mít více kontroly nad herní mechanikou.

Touto volbou bylo pracovat v prostředí Windows Forms (dále už jen WinForms). To má na druhou stranu nevýhodu v tom, že není primárně určeno pro tvorbu počítačových her, ale spíš k vytváření převážně statických okenních aplikací. Tím jsem se však nenechal odradit, jelikož žádné jiné prostředí podporující grafické prvky jsem neuměl používat a trvalo by mi dost času se s jakýmkoliv jiným naučit. Byla to tedy čistě pragmatická volba.

# 2 Programátorská část

## 2.1 Hlavní principy sledované při tvorbě programu

### 2.1.1 Obecnost a přenositelnost kódu do budoucna

Během psaní kódu jsem se snažil především dbát na to, aby byl program psán co nejvíce obecně a tudíž abych měl co největší prostor program do budoucna rozvíjet jakýmikoliv směry. Chtěl jsem si také nechat otevřenou možnost v budoucnu přejít na jiné vývojové prostředí než WinForms a dbal jsem tedy na to, ať je na WinForms všechna logika hry kromě samotného vykreslování co nejvíce nezávislá. Toto mě vedlo například k úplnému odstranění vizuálního designeru formy, jelikož jsem chtěl co nejvíce funkčnosti obsažené v samotném kódu, ze kterého by se v budoucnu dalo vyčíst, jak program potenciálně převést do jiného grafického prostředí lépe, než by se dalo vyčíst z interních nastavení WinForms.

Také mě to vedlo k implementaci herní smyčky jiným způsobem než užitím třídy `Timer` poskytovanou prostředím WinForms (tento `Timer` údajně není pro herní smyčky tak přesný). Zároveň je to další prvek spojený přímo s WinForms a bylo tedy lepší ho nahradit užitím C# třídy `Stopwatch` a `TimeSpan` k spočtení, jestli se má hra občerstvit. Tím jsem se zbavil další závislosti na WinForms.

### 2.1.2 Objektová orientace

Snažil jsem se také programovat co možná nejvíce objektově. Dával jsem tedy důraz na to, aby jakákoliv věc figurující ve hře byla reprezentována nějakým objektem. Zde jsem možná zaběhl do přílišných extrémů a můj kód je proto delší než by mohl být. Například tím, že jako objekty reprezentuji i prázdná políčka (což má i nějaké výhody). Upřednostňuji však často i delší kód, pokud je to za cenu lepší přehlednosti kódu. Tím, že téměř každou věc reprezentuji objektem si také otevírám dveře k tomu, abych v budoucnu mohl těmto věcem přiřadit nějaké chování, což určitě není k zahození.

### 2.1.3 Čitelný kód

Dalším principem, kterého jsem se snažil držet byl důraz na tzv. „samokomentující“ kód. To znamená, že jsem dbal velmi na to, aby jména proměnných byla co nejvíce výstižná a opravdu popisovala, co dané objekty, proměnné nebo funkce znamenají či dělají. Nebál jsem se proto klidně i delších jmen pokud mi přišlo, že by to pomohlo budoucímu čtenáři se v kódu lépe zorientovat. Zde jsem se však přistihl, že možná také zabíhám do přílišných extrémů.

Během tvorby programu jsem si také uvědomil, že nemá smysl ve jménu proměnné slepě opakovat typ proměnné, ale snažit se trefným pojmenováním objektu v aktuálním kontextu přidat informační hodnotu navíc. Příkladem může být třeba příkaz:

```
List<StaticLayerBlankSpace> adjacentExits = new List<StaticLayerBlankSpace>();
```

Zde by nemělo smysl pojmenovávat seznam jako **adjacentBlankSpaces** jelikož fakt, že jde o prázdné políčko už vyplývá z typu proměnné.

S čitelností kódu souvisí také využití privátních metod, které jsou volány jen jednou a mohou se tedy zdát zbytečné. Mají však dle mého názoru roli podobnou, jako by měl komentář popisující celou takovou část kódu. Krásně se tím vystihne funkce daného úseku jako logického celku a umožní do budoucna třeba využít tento logický celek někde jinde v kódu. Dobrým příkladem budiž třeba metoda **SetAllGhostsToFrightenedIfPossible()** třídy **GameManager**. (Ano, délka identifikátoru je opravdu trochu extrémní.)

Snažil jsem se také chytře využít klíčová slova jako `private`, `public` nebo `protected` k tomu, aby bylo zřejmé, jaký má daná funkce či atribut charakter tzn. jestli má být využit pouze interně a jde o jakýsi pomocný prvek nebo je to klíčový element, který slouží ke komunikaci mezi jednotlivými

objekty programu. Podobným způsobem jsem oddělil data, která se během života objektu mění od těch, která naopak zůstávají neměnná klíčovým slovem `readonly`. Tyto data jsem také označil jako veřejná, jelikož mi nedávalo smysl pro ně zavádět „gettery“, když stejně vrací stále to samé a nemusí se nic dopočítávat. Dobrým příkladem jsou například atributy objektu **Map**, které souvisí s neměnnou velikostí mapy (třeba **gridWidth** nebo **cellSize**).

### 2.1.4 Zapouzdření privátních dat

V neposlední řadě jsem pak usiloval o to, aby každý objekt zpřístupňoval navenek jen ty informace, které jsou potřeba a více ne. To se vyplácí nejen z důvodu lepší čitelnosti, jak zmiňuji výše, ale také to usnadňuje práci s našeptávačem, který mi zbytečně nenabízí ty funkce, které stejně nebudu chtít na daném objektu nikdy volat.

## 2.2 Chronologický postup tvorby

Zde se pokusím popsat, jak se program postupně vyvíjel a jak na sobě jednotlivé důležité části programu časově navazují. Mám pocit, že je to užitečná informace, která slouží k lepšímu pochopení, proč jsou různé funkcionality programu řešeny tak, jak jsou řešeny a poskytne lepší přehled o tom, jaké části programu jsou závislé na kterých ostatních.

### 2.2.1 Načítání vstupních dat

Nejprve bylo nutné vůbec se rozhodnout, jak budu reprezentovat vstupní data. Rozhodl jsem se, že nejjednodušší bude zadávat herní mapu jako textový soubor, ve kterém jednotlivé symboly budou znamenat různé herní objekty. Rozhodl jsem se nejprve zadávat soubory jako Resources, což umožňuje Visual Studio a poskytuje to možnost jednotlivé zdroje, jako třeba obrázky přímo adresovat v kódu svým jménem jako by to byla klasická proměnná. To jsem však později změnil, jelikož mi přišlo příliš krkolomné přidávat tímto způsobem nové obrázky. (Bylo potřeba otevřít soubor s příponou .resx a proklikat se několika volbami.) Je jednodušší mít jednu složku, ze které akorát přečtu soubor s daným jménem, které jednoduše zadám do proměnné jako `string`.

Jinak jsem se snažil vstupní data co nejvíce oddělit od zbytku programu a založil jsem tedy samostatnou statickou třídu **GamePresets**, která se stará o vše týkající se načítání a předzpracování vstupních dat. Tím je myšleno například vytvoření hotového objektu **Map** z přečteného souboru **map.txt**, který obsahuje textové zadání herního plánku. Nebo třeba vytvoření objektů typu `Bitmap`, které pak všechny předávám jednotlivým instancím třídy **GameObject** jako jejich obrazové reprezentace (neboli „sprity“).

V prvotních fázích programu si objekty tyto svoje interní informace načítaly přímo od třídy **GamePresets**, ale toto „šahání jiné třídě do zelí“ se mi později znelíbilo a zvolil jsem raději předávání těchto informací jako parametry konstruktoru. To byl také případ objektu **Map**, jehož seznam parametrů však na druhou stranu touto úpravou nabobtnal na dobrých devět prvků!

To, že si později každý objekt pamatoval všechna data jenž se ho týkala, vedlo i k tomu, že jsem nemusel ve třídě **Painter** provádět složité kontrolování pomocí `if`ů toho, jaký objekt se to vlastně teď snažím nakreslit, jak popisuji v následující sekci.

### 2.2.2 Základní grafika

Po vyřešení reprezentace vstupních dat bylo nutné se postarat o to, abych dokázal vykreslit herní pole se všemi objekty které v něm žijí. Bez toho by vůbec nemělo smysl implementovat chování jednotlivých objektů. Zde jsem se inspiroval kódem, který jsme využívali na cvičeních z Programování 2. (konkrétně při modifikaci hry s padajícími balvany a diamanty) a nejprve jsem tedy kreslení různých objektů rozhodoval přímo ve třídě **Painter**, která obsahovala také všechny možné obrázky jednotlivých objektů. Když pak bylo potřeba nějaký objekt vykreslit, tak jsem se zeptal

na jeho typ a dle toho jsem vykreslil odpovídající bitmapu. To však vedlo k nutnosti mít v kódu přehršel **if**ů a celé to bylo velmi nepřehledné

Rozhodl jsem se tedy později pro větší obecnost a tudíž místo toho, aby si informace o tom, jaké obrázky je potřeba vykreslit držela třída **Painter**, tak má každý herní objekt metodu **GetImageToDraw()**, která vrátí jeho aktuální bitmapu (tedy objekt třídy `Bitmap`) k vykreslení. To mi umožní, aby objekt sám mohl podle svého stavu vykreslit bitmapu kterou potřebuje, a samotný **Painter** vůbec nemusí vědět nic o tom, který konkrétně objekt aktuálně vykresluje. To se mi konkrétně hodilo při vykreslování duchů v závislosti na jejich aktuálním módu (více o módech v sekci 2.2.13). Pokud duchové byli vyděšení a měli prchat, tak stačilo přepsat rodičovskou metodu **GetImageToDraw()** a rozhodnout se, jakou bitmapu vrátit na základě aktuálního nastaveného módu (**currentMode**).

### 2.2.3 Vykreslování pohybujících se objektů

Poté, co jsem zprovoznil vykreslování nehybných objektů, přišlo na řadu vyzkoušet, jak si program povede s vykreslováním pohybujících se objektů. Zde jsem v první řadě zapomněl vždy vyčistit komponentu `Graphics` a tudíž se mi kreslily všechny další vrstvy přes sebe. Nabízelo se snadné řešení: Před každým dalším vykreslením všech objektů smazat celé plátno. S tím se však objevil nový problém, jelikož celá obrazovka při každém přepsání blikala. Zde mi hodně pomohla ChatGPT, která mi poradila, že to nejspíš bude tím, že program nestíhá tak rychle občerstvovat plátno a že bych měl kreslit jednotlivé obrázky do bufferu, (reprezentován objekty **bufferBitmap** a **bufferGraphics**) a teprve až budou všechny připravené, tak výsledek překreslit na plátno. (K tomu slouží příkaz `formGraphics.DrawImageUnscaled(bufferBitmap,0,0)`, kde **formGraphics** je `Graphics` komponenta našeho herního okénka vytvořená příkazem `CreateGraphics()`.) O vyčištění bufferu před dalším malováním se stará funkce **ClearBuffer()** a o překopírování bufferu do herního okénka pak **WriteBuffer()**.

### 2.2.4 Rozdělení projektu do více souborů

V tomto momentu, kdy jsem se pro cokoliv, co tomu alespoň trochu nahrávalo, rozhodl vytvořit objekt, už jsem měl opravdu mnoho kódu. Uvědomil jsem si, že mi poměrně velké množství času zabere samotné ježdění nahoru a dolu po mém zdrojovém souboru a hledání té správné třídy, jejíž kód jsem zrovna potřeboval upravit. To byla hlavní motivace k tomu, založit si pro můj projekt více souborů, které by logicky odpovídaly různým částem programu. Tak jsem se tedy rozhodl pro to, mít pět hlavních zdrojových souborů. Těmi byly následující:

- **GameForm.cs**
- **GameManager.cs**
- **GameObjects.cs**
- **HelperClasses.cs**
- **GamePresets.cs**

Soubor **GameForm.cs** obsahuje kód nutný k inicializaci okénka hry, a také kód pro časování herní smyčky, která je implementovaná voláním metod **Update()** a **Render()** na objektu **gameManager** stejnojmenné třídy (více o tom v sekci 2.2.10).

Třída **GameManager** žije ve vlastním souboru **GameManager.cs** a stará se o samotnou logiku hry. Rozhoduje o tom, co se má vykreslovat na základě toho, v jakém je hra aktuálně stavu, implementuje logiku herní smyčky a přijímá od Windows Forms stisknuté klávesy od uživatele, podle kterých se pak rozhoduje, co dělat. V neposlední řadě si třída **GameManager** drží reference na dva objekty klíčové pro běh hry, jimiž jsou **Map** a **Painter**.

Objekty **Map** a **Painter** se nacházejí v samostatném souboru jménem **HelperClasses.cs** společně se strukturou **Direction** (viz 2.2.7). Jsou to v podstatě třídy, které mi neseděly do jiného souboru. V budoucnu možná ještě tyto třídy nějak logicky rozčlením.

Dalšími souborem je **GameObjects.cs**, který obsahuje kód všech objektů, které žijí ve hře, dají se nějak vykreslit (většinou), dá se s nimi nějak interagovat, nebo do nich narážet a nebo se samy pohybují. Tyto objekty všechny dědí od společné rodičovské třídy **Game Object**. Ta poskytuje abstrakci nad všemi objekty a vynucuje si například, aby třídy, které od ní dědí, obsahovaly implementaci metod **GetGridX()** a **GetGridY()**, které slouží k vrácení aktuální souřadnic v mřížce (a to i pokud se objekt zrovna pohybuje mezi dvěmi buňkami mřížky!). Také si vynucuje, aby každý objekt o sobě uměl říci, jestli je vykreslitelný (metoda **IsDrawable()**) nebo aby uměl vrátit svou bitmapu, jak jsem již zmiňoval v sekci 2.2.2.

Posledním souborem, který ještě zbývá zmínit je soubor **GamePresets**, který se stará o předvolby programu jako jsou třeba rychlost hráče, velikost jedné buňky herního pole, nebo samotné čtení zadání herní mapy ze souboru **map.txt** či čtení obrázků ze souborů ve formátu PNG. Stará se tedy o předzpracování tohoto vstupu, aby si ho všechny objekty mohly jednoduše přečíst a nemusely se starat o nic dalšího. Tímto rozdělením se dá snadno všechny předvolby programu nastavovat pěkně na jednom místě a není třeba prohledávat kódy jednotlivých tříd.

Druhým důvodem, proč vytvořit samostatnou třídu pro veškeré (téměř veškeré až na přednastavení textu a ostatních věcí souvisejících více s WindowsForms než s mechanikou hry) obstarávání vstupu byl hypotetický případ, kdybych se někdy v budoucnu rozhodl vstupní data načítat jiným způsobem. Pak bude stačit jednoduše upravit pouze tento soubor a nikde jinde nebude potřeba dělat téměř žádné změny.

### 2.2.5 Pohyb objektů mimo mřížku

Velikým zádrhelem, který jsem zprvu naivně úplně opomenul byl fakt, že i přesto, že hra Pac-Man se vlastně celá odehrává na 2D mřížce, tak se jednotlivé objekty, jako samotný Pac-Man nebo duchové pohybují i mimo mřížku a jednotlivé buňky mřížky jim slouží pouze jako jakési „křižovatky“, na kterých občas mohou zatočit. Zde jsem nevymyslel žádné chytřejší řešení, než si spočítat velikost buňky v pixelech a dát si vždy pozor na to, aby rychlost objektu dělila tuto velikost beze zbytku. (Třída, od které objekty mající tuto vlastnost dědí se jmenuje **TweeningObject**.) Tím jsem zaručil, že po určitém počtu snímků se mi postavička vždy objeví přesně na další buňce a můžu tedy v kódu ošetřit, co dělat dál. Jestli například sníst nějakou kuličku (**Pellet**), nebo třeba zatočit někam jinam, či si zkontrolovat, jestli vůbec mohu pokračovat dál (**CanGoInDirection()**).

Zde jsem využil také skryté funkčnosti, která může být pro někoho trochu překvapivá a tím je, že pokud nastavíme postavičce rychlost na nějakou, která by nevyhovovala požadavku dělitelnosti zmíněného výše, tak bude automaticky snížena na první takovou, která už bude velikost buňky dělit beze zbytku. To může být trochu překvapující (až matoucí), ale rozhodl jsem se tak pro to, abych mohl rychle testovat různé rychlosti a nemusel jsem vždy v hlavě počítat, jestli to zrovna bude fungovat nebo ne. Konkrétně se o tuto funkčnost stará metoda **SetTweenSpeed()**.

Na druhou stranu je tato redukce rychlosti provedena další pomocnou funkcí **ReduceToDivisibleWithoutRemainder()** a v kódu volající funkce je tedy tato volaná funkce dobře vidět. To mi umožňuje, abych ji mohl v budoucnu jednoduše najít a odstranit, pokud by to bylo žádoucí.

### 2.2.6 Přednastavování hráčova směru

Dále jsem chtěl zprovoznit herní mechaniku, která hráči umožňuje přednastavit, kterým směrem se hráčova postavička vydá na příští odbočce, hned jak to bude možné, i když aktuálně to zrovna nelze. Pokud tedy například stisknu šipku nahoru, tak si postavička tento směr zapamatuje a pokusí se v nejbližší možné době tímto směrem vydat. To však nebylo nijak složité. Stačilo přidat atribut **nextDirection**, který umožňuje si daný směr do příště zapamatovat a využít to pokud možno při testování, kudy by se měla postavička z aktuální buňky pohnout dále. (To se děje uvnitř metody **StartNextMovementCycle()** patřící třídě **Hero**.)

### 2.2.7 Práce se směry

Bylo by v tomto momentu dobré zmínit, jak se tento směr v programu reprezentuje. Rozhodl jsem se pro vytvoření vlastní struktury jménem **Direction**, která má dvě datové položky **X** a **Y**, reprezentující jednotlivé složky 2D vektoru. Dohodou se sebou samým jsem si stanovil, že takovému vektoru budu přiřazovat pouze pět možných hodnot, tedy vektory jednotkové délky do čtyř světových stran a nulový vektor. Využil jsem také možnosti zavést si `properties` k tomu, abych například voláním **Direction.Left** mohl získat referenci na vektor (-1,0).

Každý pohybující objekt tedy musí obsahovat atribut **direction**, který se pak využívá k určení, kterým směrem bude pokračovat v pohybu a o kolik se má přesně pohnout. (Konkrétně se tento výpočet provádí v metodě **ContinueMoving()** objektu **TweeningObject**.) Atribut **direction** objektu **Hero** se také hojně využívá při nastavování cílů duchů, jak popisuji v sekci 2.2.15.

Struktura **Direction** také umožňuje rotovat daný směr doprava voláním funkce **RotateRight()**, čehož se využívá jak při testování všech sousedních políček v těle metody **GetAdjacentBlankCells()** objektu **Map** (více o jejím užití v sekci 2.2.11), tak při otáčení duchů nazad.

### 2.2.8 Detekce kolizí s duchy

Implementace detekce kolizí s duchy mě trochu děsila. Nevěděl jsem, zdali mi v tom skutečnost, že se jak hráč, tak i duchové pohybují mimo mřížku nebude dělat paseku. Nakonec jsem si však uvědomil, že stačilo jednoduše zjistit, zdali se jak x-ová, tak y-ová souřadnice (v pixelech!) obě liší maximálně o velikost jedné buňky. Pokud ano, tak se objekty dotýkají, jinak ne. Tento výpočet provádí funkce **IsTouchingTweeningObject()**. V hlavní smyčce hry tedy jednoduše zavolám funkci **IsTouchingAnyGhost**(), která na vstupu přijme seznam všech duchů ve hře a pomocí výše zmíněné funkce otestuje, jestli se některého z nich hráč dotýká nebo ne.

### 2.2.9 Implementace tzv. „wraparound“ a problémy s tím spojené

Pokud se postavička pokusí opustit herní pole, tak bylo potřeba, aby se objevila přesně na druhé straně mapy. To nebylo pro pohyb hráče tak složité. Bylo však nutné hráčovy souřadnice neměnit po přetečení nebo podtečení slepě přesně na nulu nebo na maximální možnou hodnotu, ale vždy tuto hodnotu snížit nebo zvýšit o nějaký násobek velikosti jedné buňky. Jinak by se totiž mohlo stát, že by postavička ukončila jeden svůj cyklus pohybu mimo střed nějaké buňky. To by mohlo mít za následek, že by pak postavička skončila pohyb uvnitř zdi nebo jinou podobnou prekérní situaci. O toto „zaobalování“ souřadnic se stará funkce **GetWrappedPixelLocation()**, kterou poskytuje třída **Map**.

Toto fungovalo nějakou dobu uspokojivě, ale později se ukázalo, že duchové, kteří pro své fungování potřebují znát mřížkovou lokaci poslední navštívené buňky při přechodu na druhý okraj mapy neregistrovali krajní políčka mapy. To vedlo k tomu, že si mysleli, že toto políčko ještě nenavštívili a otáčeli se zpět. Toto jsem bohužel musel opravit jako speciální případ (nenašel jsem lepší způsob) a řeším ho v metodě **WraparoundIfOutOfBounds()** zavoláním metody **UpdateLastOccupied()**, kterou bych ve standardním případě volal jen před začátkem dalšího cyklu pohybu mezi dvěmi sousedními buňkami. Nyní tedy tuto metodu volám i v případě, když duch přejde na opačnou stranu mapy, aby si ihned zapamatoval tuto novou pozici jako navštívenou a neměl pak tendenci se vracet v příštím kroku zpět.

Aby toho nebylo málo, tak se objevil další nečekaný zádrhel. Duchové totiž při hledání cesty, kterou se mají bludištěm navigovat kontrolují sousední políčka a koukají se, která z nich jsou prázdná. Co když jsou však duchové na kraji mapy? Nabízí se jednoduše „zaobalit“ souřadnici souseda při kontrole všech sousedních políček abychom získali políčko, které je sice na druhé straně mapy, ale prakticky je sousední. Zde jsem však zkolaboval na předpokladu, že poté co si duch zvolí sousední políčko na které bude pokračovat, stačí odečíst jeho aktuální pozici od pozici souseda a získám tím požadovaný vektor příštího pohybu. Výsledkem byl duch, jehož vektor **direction** (To je proměnná typu **Direction** viz 2.2.7) měl místo klasické jednotkové délky délku rovnou velikosti mapy (tedy buď šířky nebo výšky). Hodnota privátní proměnné **direction** se však využívá uvnitř těla metody

**ContinueMoving()** objektu **TweeningObject** k výpočtu toho, o kolik pixelů se v daném snímku hry má duch posunout. Tak se stalo, že mi duchové lítali z jedné strany mapy na druhou bleskovou rychlostí dokud je nezastavila zákeřná `IndexOutOfBoundsException`. Řešením byl bohužel již druhý speciální případ, který jsem musel umístit dovnitř metody **SetDirectionTowardsExit()**, kterou poskytuje třída **MovingObject**. Konkrétně dle toho, kterým směrem se vektor protáhl, lze určit správný směr, kterým by se měl duch vydat. Toť směr k tomuto přerostlému směru přesně opačný avšak délky jednotkové. Toto jednoduché řešení poskytuje privátní metoda **GetFixedOutOfBoundsCoordinate()**, která upraví zvlášť každou složku takto zdivočelého vektoru.

### 2.2.10 Přidání hacku na rychlejší a ortodoxnější herní smyčku ve WinForms

Dále jsem měl pocit, že hra běží na mém počítači poněkud pomalu a občas se zadrhává, což bylo z velké části nějspíš i tím, že jsem k práci s WinForms musel použít virtuální stroj k emulaci operačního systému Windows. Rozhodl jsem se tedy probádat internet, zdali někdo nemá řešení, jak program ve WinForms zrychlit.

Našel jsem způsob, který využívá jakýsi hack, kdy program nečeká, dokud se nespustí nějaký jeho event, ale namísto toho běží neustále, dokud je tzv. „windows message queue“ prázdná. Tento způsob je kromě (údajné) vyšší rychlosti a přesnosti herní smyčky nezávislý na tikání WinFormsového timeru (objekt typu `Timer`), který se údajně občas trochu zpožďuje a není tak přesný.

Musím se však přiznat, že jsem nezaznamenal žádné očividné zlepšení avšak ani zhoršení. Rozhodl jsem se tedy tuto změnu v programu ponechat. Částečně také z toho důvodu, že nová implementace se více podobá klasické herní smyčce, která se běžně užívá v počítačových hrách a navíc je závislá na objektu typu `Stopwatch`, který by měl poskytovat to zmíněné pravidelnější tikání.

### 2.2.11 Duchovský pathfinding

Dále jsem se rozhodl konečně uskutečnit zárodky umělé inteligence duchů. Tomu předcházelo, že jsem si přečetl jeden pěkný článek o tom, jak se implementovala tato umělá inteligence v originálním Pac-Manovi. Zjistil jsem, že jádro chování jednotlivých duchů je výpočetně jednodušší, než jsem se původně bál a nebudu tedy muset programovat žádné prohledávání do šířky na každé křižovatce a vystačím si s jednoduchým počítáním vzdáleností po způsobu starého známého Euklida.

Algoritmus hledání cesty bludištěm totiž spočívá v tom, že každý duch má v každém čase programu nějaký svůj cíl (v programu ho reprezentuji proměnnou **target** typu `Point`), na který se aktuálně snaží dostat (jeho poloha se neustále mění v čase). Pro volbu, kterou cestou se vydá na příští křižovatce pak duch volí jednoduchý hladový přístup, který spočívá v tom, že se koukne na všechna sousední políčka na křižovatce (Referenci na ně získá zavoláním metody **GetAdjacentBlankCells()** objektu **Map**.) a vybere si takové sousední políčko, které je od jím zvoleného cíle (**target**u) nejblíže vzdušnou čarou. A to nehledě na skutečný počet buněk, které bude muset duch během sledování svého cíle překonat. O toto nalezení nejbližšího sousedního políčka se stará metoda **FindExitClosestToTarget()**, která ve svém těle volá pomocnou funkci **GetDistanceToCell()**, aby určila jednotlivé vzdálenosti. Kupodivu tento přístup celkem pěkně funguje (pokud předpokládáme klasickou Pac-Manovskou herní mapu).

Ještě by bylo příhodné dodat, že duch si nikdy nevybírá východ z křižovatky, ze kterého právě přišel (což se dá hezky využít i v situaci, kdy jsou východy právě dva). Kvůli tomu potřebujeme zavést pomocný atribut **LastOccupiedCell**, díky kterému si duch pamatuje, jakou buňku posledně navštívit. Tento atribut je umístěn v objektu **TweeningObject** (viz sekce 2.2.5), i když původně se nacházel přímo v objektu **Ghost**. K této změně mě donutil právě problém se „zaobalováním“ souřadnic zmíněný v třetím odstavci sekce 2.2.9.

### 2.2.12 Stavy hry

Abych mohl pracovat na složitějším chování duchů, tak bylo nejprve potřeba přidat základní herní stavy. Tedy počáteční obrazovku, běžící hru a konec hry. Chtěl jsem totiž mít jednoduchou možnost

otestovat chování duchů a nemuset hru stále dokola spouštět manuálně přes Visual Studio. Místo toho jsem chtěl mít možnost jednoduše zmáčknout „enter" a hrát znovu. Dalším důvodem bylo ladění obtížnosti. To se také dělá těžko, pokud hra ještě nejde vyhrát. Takže bylo nutné tyto stavy přidat.

Samotná implementace stavů není složitá. Uvažoval jsem, že bych použil tzv. „strategy pattern" a využil `interface` a vytvořil tzv. „state machine". Nakonec mi toto však přišlo moc složité a zvolil jsem jednodušší způsob vytvořením výčtového typu **GameState**, který může nabývat hodnot různých stavů hry (tedy **StartSceen**, **Running** a **GameOverScreen**). Co přesně se má v jakém stavu dělat, pak kontroluji pomocí řídící struktury `switch` uvnitř veřejné metody **Update()**, která je volána z hlavní herní smyčky implementované uvnitř kódu třídy **GameForm**.

Stav **GameOverScreen** odpovídá jak vítězné obrazovce, tak i té poražené. Zde jsem uvažoval tyto dvě možnosti rozdělit do dvou separátních stavů, ale nakonec jsem se rozhodl přidat boolovskou proměnnou **gameLost**, která určuje, která ze dvou obrazovek se má vypsat.

Ze stavu **Running** do stavu **GameOverScreen** lze přejít dvěma způsoby. Buď hráče chytil duch, což kontroluje metoda **CheckGhostCollisions()** nebo hráč snědl všechny interaktivní objekty **Pellet**s a **Energizer**s a to zas kontroluje metoda **CheckGameWon()**.

### 2.2.13 Módy duchů

Jakmile jsem měl možnost přecházet mezi stavy hry, tak jsem se pustil do psaní kódu, který by duchům umožní přecházet mezi čtyřmi důležitými módy, které určují jejich komplexní chování. Tyto módy jsem chtěl implementovat v této fázi programu, jelikož na nich závisí, jakým způsobem si jednotliví duchové vybírají svůj cíl. Abych tedy mohl naprogramovat složitější chování duchů, tak bylo příhodné mít možnost mezi těmito módy nějak přecházet.

Módy jsou reprezentovány dalším výčtovým typem jménem **GhostMode** a nabývá hodnot **Preparing**, **Chase**, **Scatter** a **Frightened**. Každý duch má interně uložený svůj aktuální mód, ve kterém se v daném momentu nachází, v privátní proměnné **currentMode**. Zároveň, jelikož mezi módy **Chase**, **Scatter** a **Frightened** se duchové přepínají po signálu ze vnějšku, tak reprezentuji také jakýsi globální mód uvnitř objektu **GameManager**, který určuje, ve kterém stavu se duchové právě nachází (pokud zrovna nejsou v domečku ve stavu **Preparing**, potom je jejich mód na tom globálním nezávislý). Tento globální mód je uložen v proměnné **currentGhostMode**.

### 2.2.14 Obecný význam jednotlivých módů

Nyní bych chtěl krátce uvést, jaké chování mají jednotlivé módy duchů obecně reprezentovat a později bych se věnoval tomu, jak jsou řešené každým duchem zvlášť. To je totiž unikátní vlastnost hry Pac-Man, že každý duch má svůj charakter a chová se tedy trochu jinak, než ti ostatní. Jedinou výjimkou je mód **Frightened**, ve kterém se všichni duchové chovají úplně stejně.

Prvním módem duchů v mé implementaci hry je mód **Preparing**, kterému v originálním Pac-Manovi však žádný mód neodpovídá. Chtěl jsem tímto módem reprezentovat stav, kdy je duch ještě v domečku a čeká, až ho bude moci opustit. Domeček je totiž ohrazen objekty typu **Fence**, které jsou pro ducha nejprve neprostupné a po uplynutí času (ten si duch interně pamatuje v proměnné **prepareDuration**) mu je umožněno domeček opustit. Duchovi se v závislosti na jeho módu a na tom, jestli už opustil domeček (to zas kontroluje boolovská proměnná **leftHouse**) musí měnit skutečnost, zdali je pro něj plot dosažitelný nebo ne. To je zajištěno přepsáním metody **IsReachableCell()** rodičovské třídy **GameObject**.

Druhým módem je mód **Chase**, ve kterém se každý duch chová velmi jinak. Detaily popíši v sekci 2.2.15. Obecně to však znamená, že duch nějakým svým vlastním způsobem zvolí cíl tak, aby nějak zkřížil hráči cestu. Většinou si tedy volí svůj **target** (viz sekce 2.2.11) v závislosti na poloze hráče. V těle třídy **Ghost** je vyhrazena metoda **SetTargetToChaseTarget()**, která je abstraktní a je určená k tomu, aby si ji každý konkrétní duch přepsal dle své potřeby.

Třetím módem je mód **Scatter**, který je implementován velmi jednoduše. Jde o to, že duch si

nastaví svůj cíl do nějakého rohu mapy, což má za následek to, že duch pak v tomto rohu mapy krouží pořád dokola. Stejně jako je tomu u předchozího módu má třída **Ghost** pro jednotlivé duchy vyhrazenou abstraktní metodu **SetTargetToScatterTarget()**. Tento mód dává hráči možnost si na malou chvíli oddychnout od neúprosného pronásledování duchy během výše zmíněného módu.

Posledním avšak neméně důležitým a zajímavým módem je mód **Frightened**. Do tohoto módu se duchové nepřepínají v závislosti na čase, jako je tomu u zbylých módu, ale v závislosti na tom, zdali hráč sní power-up reprezentovaný objektem typu **Energizer**. To se kontroluje v metodě **TryEat()**, která je volána z hlavní herní smyčky pokud se hra nachází ve stavu **Running**. Pokud hráč **Energizer** sní, tak se **GameManager** všem duchům tento mód pokusí nastavit zavoláním metody **SetAllGhostsToFrightenedIfPossible()**, která se nachází právě uvnitř těla metody **TryEat()**. Metoda **SetAllGhostsFrightenedIfPossible()** projde seznamem všech duchů ve hře a zavolá na nich metodu **SetModeIfValid()** s parametrem **GhostMode.Frightened**. Tato metoda se pak postará o to, aby se duch nenastavoval do módu **Frightened** pokud je například stále ještě v domečku a zároveň ve stavu **Preparing**. (Pozor! Pokud je v domečku, ale zrovna přešel do jiného módu a chystá se opustit domeček, tak se stejné může vyděsit a zmodrat.)

V módu **Frightened** všichni duchové zmodrají a pohybují se po herní mapě náhodně (na každé křižovatce si zvolí náhodnou pozici **target**u. Hráči je umožněno v tomto módu duchy sníst a získat za ně bodový bonus. Poté, co je nějaký duch sněden, tak se objeví zpět na svém domovském políčku a je nastaven zpět do módu **Preparing** s časovačem nastaveným zpět na nulu.

### 2.2.15 Konkrétní implementace módů konkrétními duchy

Charakteristická chování jednotlivých duchů v módech **Scatter** a **Chase** jsem se rozhodl řešit tím, že jsem pro každého ducha zavedl jeho vlastní třídu. Při volání metod na nastavení cíle v daném módu je pak využito polymorfizmu díky tomu, že každý z konkrétních duchů dědí od třídy **Ghost**. Třídy jednotlivých duchů jsou postupně **RedGhost**, **PinkGhost**, **BlueGhost** a **OrangeGhost**.

**RedGhost** byl na implementaci vůbec nejjednodušší. Jeho způsob výběru cíle v módu **Chase** je takový, že ho na každé křižovatce umístí vždy přesně na hráče. Dále pak použije hladový způsob hledání cesty ze sekce 2.2.11. Umístění **target**u přímo na hráče obstarává metoda **SetTargetAheadOfHero()** zavolaná s argumentem 0 pro parametr **tilesAhead**, který udává, o kolik políček před hráčem si má duch nastavit svůj **target**. Touto obecnou metodou jsem si zkrátil kód, jelikož ji pak mohu využít i pro ostatní duchy. (Popravdě ji nějak využívá v módu **Chase** každý z nich.) V módu **Scatter** chodí v pravém horním rohu.

**PinkGhost** má velmi podobný algoritmus jako předchozí duch s jediným rozdílem, že místo toho, aby **target** umístil přesně na hráče, tak ho umísťuje přesně čtyři políčka v jeho aktuálním směru před něj. V módu **Scatter** si pak cíl nastavuje do levého horního rohu.

**BlueGhost** má ze všech duchů vůbec nejsložitější a nejméně předvídatelné chování. V módu **Chase** si nejprve nastaví cíl dvě políčka před hráče a poté se koukne, kde je **RedGhost** a posune své cílové políčko přesně o vzdálenost mezi **RedGhost**em a nově nastaveným cílem ve stejném směru. Během módu **Scatter** krouží v pravém dolním rohu.

**OrangeGhost** posledním z duchů je duch oranžový, který má stejně jako modrý duch poměrně zvláštní chování. V módu **Chase** se rozhoduje podle vzdálenosti od hráče. Pokud je o hráče vzdálen o méně než osm políček, tak se snaží prchnout do svého rohu, který je pro něj stejný jako během módu **Scatter**. Naopak pokud je od hráče dále, tak si nastavuje **target** přímo na hráče stejným způsobem, jako to dělá první z duchů. Pro zjišťování vzdálenosti od hráče využívá metody **GetDistanceToCell()**, kterou zmiňuji v sekci 2.2.11 ve druhém odstavci.

### 2.2.16 Časování globálních módů duchů

Jak jsem již zmínil, tak mezi některými módy je nutné přepínat v závislosti na čase nebo jiných podmínkách globálně, aby byli všichni duchové co možná nejvíce synchronizovaní. Nyní, když už jsem měl zhotovené konkrétní chování pro jednotlivé duchy, tak jsem mohl toto časování pěkně testovat. Jelikož toto přepínání je řízeno samotnou hrou, bylo mi hned jasné, že bude potřeba

upravit kód třídy **GameManager**. Konkrétně bylo nutné přidat této třídě časovač, kterými by mohla zkoumat, kolik času uběhlo od poslední globální změny duchovského módu. Tento čas je uložen jako atribut **lastModeChange** typu `DateTime`. O případnou změnu módu pokud nastane čas se stará metoda **ChangeModeIfTime()**, která pomocí **switch**e přes aktuální globální mód duchů (**currentGhostMode**) rozhodne, jaký časový interval má zrovna testovat. Testuje se vždy jeden z **scatterModeDuration**, **chaseModeDuration** nebo **frightenedModeDuration**, které jsou všechny typu `TimeSpan`.

#### 2.2.17 Implementace zmodrání duchů ve frightened módu

Hra po dokončení mechanismů zmiňovaných v předchozí sekci již pěkně fungovala. Samozřejmě jsem ještě opomenul počítání skóre, avšak to bylo tak triviální, že by bylo zbytečné mu věnovat samostatnou sekci. Zbývalo tedy už jen způsobit, aby duchové pokud jsou v módu **Frightened** přeměnili svůj obrázek na modrý, čímž hráči jasně signalizují, že je možné je sníst.

Rozhodl jsem se vyřešit toto chování tím, že jsem do třídy **Ghost** přidal navíc atribut typu **Bitmap**, který duchům umožní držet si odkaz na modrý obrázek, který pak v případě zavolání metody **GetImageToDraw()** vrátí místo svého klasického, pokud jsou právě v módu **Frightened**. Zbytek pak delegují na objekt třídy **Painter** která je popsána blíže v sekci 2.2.2.

#### 2.2.18 Kosmetické úpravy a komentování kódu

V této fázi už byla hra hratelná a tedy víceméně hotová. Rád bych samozřejmě přidával další vylepšení jako třeba více životů hráče, další úrovně či animování postaviček, ale bohužel mě již tlačil čas (a stále tlačí i během psaní této dokumentace).

I přesto jsem však věnoval vlastně jeden celý den komentování kódu, přejmenovávání a dalším činnostem s tím souvisejícím. Tolik času mi to trvalo i z toho důvodu, že pokud já začnu svůj kód komentovat, tak to má za následky, že zpozoruji, co vše by se dalo napsat lépe a nakonec třeba ještě nějaký kód pozměním. Výsledkem však je čitelnější a občas i optimalizovanější kód.

Takto jsem například už hotové hře přepsal tělo metody **StartNextMovementCycle()** tak, aby byla přehlednější. Původně jsem se spokojil s tím, že to „nějak fungovalo“ a přehlednost jsem tolik neřešil (čímž jsem porušil svůj zmiňovaný princip, ale jak říkám, už mě začal tlačit čas a chtěl jsem vidět rychlé výsledky). Když jsem však začal tuto metodu komentovat, tak jsem si uvědomil, že je pro mě opravdu takovým blackboxem a vlastně jsem ji částečně zprovoznil metodou pokus omyl. Během její analýzy jsem narazil na chybu, při které se duchové v případě, že zašli do slepé uličky (což se na klasickém herním poli hry Pac-Man nemůže stát), tak pokračovali dále a procházeli zdí. Tento bug jsem rychle opravil a do budoucna je díky tomu moje hra robustnější, pokud bych se rozhodl nějak pozměnit rozvržení herní mapy.

# 3 Závěr

## 3.1 Co by se dalo vylepšit

### 3.1.1 Životy hráče

Jak jsem již zmínil, tak by bylo hezké. Kdybych do hry naimplementoval ubírání životů hráče. To by nemělo být příliš složité. Potřeboval bych přidat funkci, která by resetovala duchy na své místo v domečku a všechny časovače, ale zároveň ponechala všechny objekty, které může hráč sníst a ještě zbývají na herním poli. Také by bylo nutné přidat stav, ve kterém hra dává chvilku času hráči se rozkoukat, pokud byl zrovna polapen. Tento čas by se také dal zavést přímo do stavu **Running** přidáním dodatečné podmínky s časovačem.

### 3.1.2 Přidání dalších levelů a počítání highscore

Aby hra byla trochu kompetitivnější, tak by se hodilo přidat jak další levely, ve kterých by se duchové například mohli zrychlovat, tak i přidat pamatování highscore. Přidat další levely by mohlo být časově celkem náročné. Nevím, jak přesně bych naipmlementoval různé zvětšování rychlostí duchů a zvyšování obtížnosti hry. Přidat highscore by naopak složité nebylo. Stačilo by zavést nový textový soubor, do kterého bych si toto highscore zapisoval nebo z něj četl.

### 3.1.3 Animace otevírání pusy hráče a očí duchů

Dalším vylepšením by mohlo být přidat animace pohybujících se postaviček. Zde by se dalo například poskytnout každému objektu namísto jedné bitmapy (nebo dvou) celé pole bitmap. Podle svého aktuálního směru by se pak postavička rozhodla, jakou bitmapu má při zavolání metody **GetImageToDraw()** vrátit.

U otevírání pusy hráče by pak chtělo ještě přidat objektu **Hero** nějaký časovač, který by umožnil volit snímky také v závislosti na aktuálním čase.

Uvažoval jsem také, že by nebylo nutné přidávat pro různé směry otočení hráče nové snímky, a místo toho prostě otáčet samotné bitmapy, ale po krátké analýze jsem zjistil, že toto otáčení bitmap v C# je poměrně náročný proces.

## 3.2 Závěrečný povzdech

Tvorba této hry pro mě byla něčím úplně novým. Sám moc počítačové hry nehraji, ale dostal jsem chuť konečně vytvořit nějaký grafický interaktivní program, čehož je počítačová hra dobrým příkladem. Nakonec bylo vtipné si uvědomit, že se mi občas hodily poznatky z kroužku tvorby počítačových her ve Scratchi, který jsem minulý rok vedl. Chtěl jsem totiž dětem nějak zábavně naučit základy programování a tak jsem zvolil téma počítačových her, které mě samotnému není až tak blízké. To mě však vedlo k tomu, že jsem si načetl mnoho materiálů o herních smyčkách, renderování obrázků atd. což mi pak pomohlo se v těchto termínech během tvorby této hry lépe zorientovat.

Zprvu jsem si myslel, že využiji nějakého vývojového prostředí určeného přímo pro vývoj her, ale nakonec převládl nedostatek času se s nějakým takovým učit a také to, že jsem chtěl mít co největší kontrolu nad tím, jak si hru naprogramuji a nebýt závislý na nějakém herním enginu.

Během programování hry jsem si také zamiloval práci s GitHubem. Líbí se mi možnost, vidět chronologicky všechny své změny provedené v programu a také mít tu možnost se k nim kdykoliv vrátit. Člověk má také klid v duši když ví, že může cokoliv ve hře porouchat a má možnost se jednoduše vrátit zpět. Také mi však Git pomohl k psaní této dokumentace, jelikož umožňuje koukat se, jak se projekt vyvíjel v čase a tedy vidět logické návaznosti na sebe, jak se na sebe jednotlivé komponenty programu postupně skládaly jako dílky stavebnice.

GitHub mě oslnil natolik, že jsem se rozhodl verzovat si i tento doprovodný dokument, takže můžu hezky vidět, jak se mi tato dokumentace měnila v čase a vidět drobné úpravy, které jsem na ní dělal. Skutečnost, že Git si drží časy jednotlivých commitů mi také prospěla v tom, že jsem si lépe uvědomil, kolik času čím trávím. Je zajímavé mít tyto informace k dispozici.

Samotného mě překvapilo, jak moc mě samotná dokumentace programu pohltila. Měl jsem radost z toho, uhlazovat svůj program a odplevelovat ho od zbytečného kódu, který tam neměl co dělat a třeba se dal zjednodušit. (Tyto nápady totiž často přicházely právě během psaní dokumentace.) Skutečnost, že mám teď jakousi verzi dokumentace sepsanou mi také otevřela dveře, abych ve volném času během prázdnil mohl hru ještě doladit o nějaké nové mechanismy.

Na závěr bych chtěl hlavně projevit obdiv nad tím, že někdo tak propracovanou hru jako Pac-Man vytvořil už před čtyřiceti lety. Je úžasné, být schopný takovéto dílo vytvořit bez všech těch vymožeností, které nám dnes pomáhají psát kód, jako třeba skákání na reference či definice v kódu. Chytré přejmenovávání proměnných, foreach cykly a mnoho dalších zjednodušení, které mi už

přijdou naprosto přirozené. No a ta umělá inteligence, která je vytvořena algoritmicky tak jednoduše, ale zároveň vede k velmi komplexně vypadajícímu chování je také úžasná. Opravdu musím říci, že původní hra Pac-Man je krásné programátorské i herně designové dílo.