Katedra informatiky
Přírodovědecká fakulta
Univerzita Palackého v Olomouci

BAKALÁŘSKÁ PRÁCE

Počítačová hra - Skákací hra pomocí Pygame

2023

Ondřej Nečesaný

Vedoucí práce:
Mgr. Jan Laštovička, Ph.D.

Studijní program: Informační technologie,
prezenční forma

Bibliografické údaje

Autor:	Ondřej Nečesaný
Název práce:	Počítačová hra - Skákací hra pomocí Pygame
Typ práce:	bakalářská práce
Pracoviště:	Katedra informatiky, Přírodovědecká fakulta, Univerzita Palackého v Olomouci
Rok obhajoby:	2023
Studijní program:	Informační technologie, prezenční forma
Vedoucí práce:	Mgr. Jan Laštovička, Ph.D.
Počet stran:	26
Přílohy:	elektronická data v úložišti katedry informatiky
Jazyk práce:	český

Bibliographic info

Author:	Ondřej Nečesaný
Title:	Computer game - Platformer using Pygame
Thesis type:	bachelor thesis
Department:	Department of Computer Science, Faculty of Science, Palacký University Olomouc
Year of defense:	2023
Study program:	Information Technologies, full-time form
Supervisor:	Mgr. Jan Laštovička, Ph.D.
Page count:	26
Supplements:	electronic data in the storage of department of computer science
Thesis language:	Czech

Anotace

Cílem této bakalářské práce bylo naprogramovat skákací hru v jazyce python za pomocí externí knihovny pygame, a to s použitím funkcionálního principu pro pohyb hráče ve světe. Hra má tři režimy. V prvním režimu je svět generován pomocí perlin noise algoritmu, ve druhém si hráč vytvoří vlastní mapu a v neposlední řadě ve třetím režimu je pro hráče již mapa připravena.

Synopsis

The goal of this bachelor thesis was to program a platformer in python using the pygame library. Player movement is done using functional principles. The game has three game modes, first where the world is generated using the perlin noise algorithm, second where the player creates their own map and lastly the third, where the map is already prepared for the player.

Klíčová slova: hra, python, pygame, perlin noise, funkcionální princip

Keywords: game, python, pygame, perlin noise, functional principle

Chci poděkovat vedoucímu práce Mgr. Janu Laštovičkovi, Ph.D. za skvělý přístup při vedení práce a za přínosné připomínky a nápady při konzultacích.

Odevzdáním tohoto textu jeho autor/ka místopřísežně prohlašuje, že celou práci včetně příloh vypracoval/a samostatně a za použití pouze zdrojů citovaných v textu práce a uvedených v seznamu literatury.

Obsah

1 Úvod 6

2 Technologie 6
- 2.1 Python . . . 6
- 2.2 Pygame . . . 6

3 Teorie 7
- 3.1 Funkcionální paradigma . . . 7
- 3.2 Platformové hry . . . 7
  - 3.2.1 Scroll . . . 7
  - 3.2.2 Historie . . . 8

4 Pohyb pomocí funkcionálního programování 8
- 4.1 Herní pohyb . . . 8
- 4.2 Kolizní modely . . . 8
- 4.3 Pohyb a kolize ve 2D hrách . . . 9
  - 4.3.1 Funkcionální řešení pohybu a kolize . . . 10
  - 4.3.2 Speciální kolize . . . 13
- 4.4 Konkrétní implementace . . . 13
  - 4.4.1 Speciální případy . . . 15

5 Procedurální generace mapy 15
- 5.1 Perlin noise . . . 15
- 5.2 Konkrétní implementace generování . . . 16

6 Uživatelská příručka 17
- 6.1 Ovládání . . . 17
- 6.2 Herní režimy . . . 17
  - 6.2.1 Generovaná mapa . . . 18
  - 6.2.2 Map creator . . . 18
  - 6.2.3 Story mode . . . 18

7 Implementace hry 19
- 7.1 Možné zlepšení . . . 21

Závěr 23

Conclusions 24

Seznam zkratek 25

Literatura 26

1 Úvod

Cílem práce je implementace dvourozměrné skákací hry za pomocí jazyka python a knihovny pygame. Pohyb a řešení kolize jsou kritické části vývoje hry. V této práci se zaměřuji na implementaci těchto prvků pomocí čistě funkcionálních principů. Ve hře má hráč na výběr ze tří režimů, které si liší způsobem vytvoření herního prostředí.

Záměrem je zhodnocení využitelnosti funkcionálního paradigmatu v herním vývoji, dopad na výkon při kombinaci s imperativním programováním a praktickou využitelnost. V práci se zaměřuji na řešení problémů spojených s pohybem ve dvoudimenzionálním prostoru a kolizními modely v herním vývoji.Jazyk python a knihovnu pygame jsem zvolil záměrně, protože mi umožňují jednoduše demonstrovat tyto principy. Vizuální aspekt hry je reprezentován pixelovými texturami po vzoru Minecraftu nebo Terrárie. Samotný princip hry je inspirován klasickou skákací hrou Mario.

2 Technologie

2.1 Python

Python je multiparadigmatický vysokoúrovňový programovací jazyk navržený v roce 1991. Zvolil jsem si jej cíleně díky možnosti hybridní implementace, což mi umožňuje ukázat funkcionální princip v herním vývoji, a přitom použít vybranou knihovnu pro herní prvky. Navzdory reputaci jazyka ohledně jeho výkonu program funguje plynule, a to i s využitím funkcionálních principů pro pohyb. Informace jsou čerpány z dokumentace jazyka [1].

2.2 Pygame

Pygame je knihovna pro jazyk python, která implementuje funkcionalitu, která značně usnadňuje vývoj her a jiných multimediálních programů. Hlavní funkce v této knihovně používají C a Assembly kód z důvodu optimalizace výkonu programů. Knihovna je velice modulární a části se dají volat separátně, jako například grafické či zvukové rozhraní se dají nahradit vlastní implementací nebo jinou knihovnou. Informace jsou čerpány z dokumentace knihovny [2].

Pro moji implementaci je nejdůležitější struktura obdélníků pro kolizi, ty jsou definovány pomocí souřadnice počátečního bodu, což je levý horní roh obdélníku, a s pomocí šířky a výšky obdélníku. Obdélníky je také možné graficky vykreslovat namísto používání systému texturování objektů, což umožňuje jednoduchou vizualizaci kolizního modelu.

3 Teorie

3.1 Funkcionální paradigma

Jedná se o deklarativní programovací paradigma, kde základní myšlenkou je, program skládající se z funkcí, které vrací hodnotu na základě vstupních parametrů a nemění stav programu. Existuje několik základních konceptů funkcionálního programování, které paradigma definují.

Funkce vyšších řádů jsou funkce, které lze použít jako hodnoty a může se s nimi i tak zacházet, například je možné ji předat jako vstupní hodnotu jiné funkci.

Čisté funkce jsou funkce jejiž výstup závisí pouze na vstupních hodnotách a vnitřním algoritmu. Ve funkci nedochází k vedlejšímu efektu.

rekurze je ve funkcionálním programování způsob pro opakování, kde funkce volá sama sebe.

Definice parafrázovaný z knihy [3].

3.2 Platformové hry

Jedná se o herní žánr, kde hlavním cílem pro hráče je překonat překážky na mapě pomocí skákání, lezení a běhání. Skákání je pro žánr klíčové a jeho implementace je v platformových hrách rozdílná na základě hratelnosti. Skákání může být fixní, nebo se může během skoku měnit, či může být různé na základě toho, jak dlouho podrží hráč klávesu. Skákání samozřejmě potřebuje implementovanou gravitaci. Je potřeba mít nějaké zrychlení, které se bude lišit pro každdou hru. Gravitace a její změna může být samotným herním prvkem přidávající další hratelnost. Na základě pádů je také potřeba implementovat co se stane po kontaktu se zemí, jako například ubrat životy, odrazit se a podobné efekty.

3.2.1 Scroll

První hry v žánru platformových her používaly princip jedné obrazovky, a to takovým způsobem, že úroveň se celá vykreslila na obrazovku a obraz nijak nereagoval na pohyb hráče. Tento způsob vývoje se využíval buď z důvodů omezených zdrojů, nebo jednodušší implementace. *Scrolling* je princip, kdy se obraz posouvá na základě hráčského pohybu. To vytváří efekt mnohem větších mapy, protože není potřeba mít načtenou celou mapu. Hry jako Jump bug nebo Moon Patrol byly jedny z prvních, které tento princip využily, ale popularizoval to z větší části až Super Mario. *Scrolling* je možné implementovat jako posun všech objektů proti hráčskému pohybu, tedy pokud se hráč hýbe doprava, objekty se posouvají doleva. Informace čerpány z článku [4].

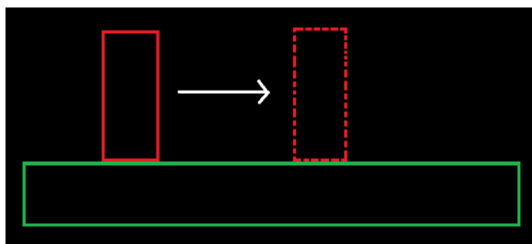
3.2.2 Historie

Za první hru v tomto žánru se považuje Space Panic z roku 1980, ale o popularizaci se zasadily až hry jako Donkey kong nebo Super Mario Bros. V době největší popularity platformových her mezi roky 1980 a 1990 tvořily čtvrtinu až třetinu vyvíjených her. Zmíněný Super Mario je dnes snad nejznámější platformová hra se skoro 60 miliony prodaných kopií. Tato hra a její společnost Nintendo značně ovlivnila herní průmysl. Informace čerpány z článku [4].

4 Pohyb pomocí funkcionálního programování

4.1 Herní pohyb

Řešení pohybu je jednou z nejdůležitějších částí herního vývoje a kvalita implementace má vliv na kritickou odezvu hry. Faktory jako plynulý pohyb postavy, přesná kolize, či jak rychle postava reaguje na stisknutí klávesy, značně ovlivňují vnímanou kvalitu řešení. Implementace se bude lišit v závislosti na žánru hry a na typu herního prostředí. Konrétní zaměření této práce je na dvoudimenzionální prostředí v žánru platformových her.



Obrázek 1: vizualizace pohybu

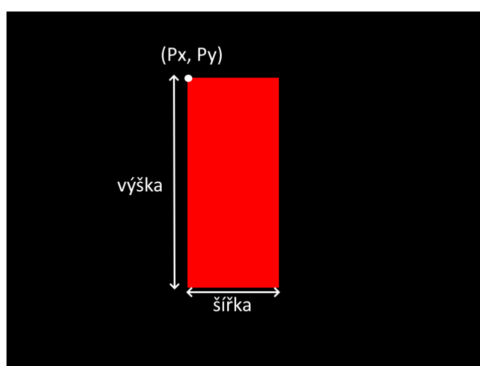
4.2 Kolizní modely

Před implementací pohybu je nutné si zvolit vhodný model kolize pro danou hru. Výběr modelu závisí na faktorech jako jsou herní mechaniky, výkon, tvar objektů ve hře a potřebná přesnost kolize. Existuje několik modelů kolize a každý má své výhody a nevýhody. Obdélníková kolize není náročná na výkon, ale pro složitější tvary nenabízí přesnou kolizi. Polygonová kolize je poněkud náročnější na výpočet, ale nabízí přesnější kolizi pro složité tvary. *Raycasting* je typ kolize pomocí přímky, ta se vytvoří v počátečním bodě a koncový bod bude objekt, se kterým došlo ke kolizi. Pokud je zvolen nevhodný typ kolize, bude to mít negativní dopad na zážitek hráče. Na obrázku 2 můžeme vidět rozdílné implementace ve stejné hře.

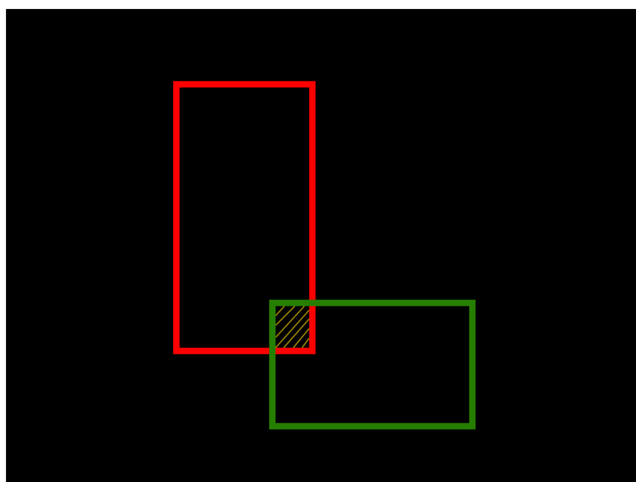
Obrázek 2: Dvě možné implementace kolize ve stejné hře. Obrázky převzaté z [5]

4.3 Pohyb a kolize ve 2D hrách

Ve dvou dimenzionálních hrách většinou není potřeba implementovat složitější kolize a stačí klasický model kolizních obdélníků. Ty mohou být definované například pomocí dvou párů souřadnic levého horního rohu a pravého dolního rohu, nebo pomocí jednoho rohu a šířky s výškou obdélníku. Používat budu druhou variantu, kterou lze vidět na obrázku 3. Pohyb je změna pozice těchto obdélníků podél x-ové a y-ové osy.Nová pozice se dostane tak, že se k původní pozici přičtou odpovídající složky vektoru, viz. obr. 5. Kolize nastává, když se dva či více objektů překrývají, tedy dochází k neprázdnému průniku, viz. obrázek 4. Konkrétní řešení pohybu a kolize závisí na použitém paradigmatu a herních mechanikách.



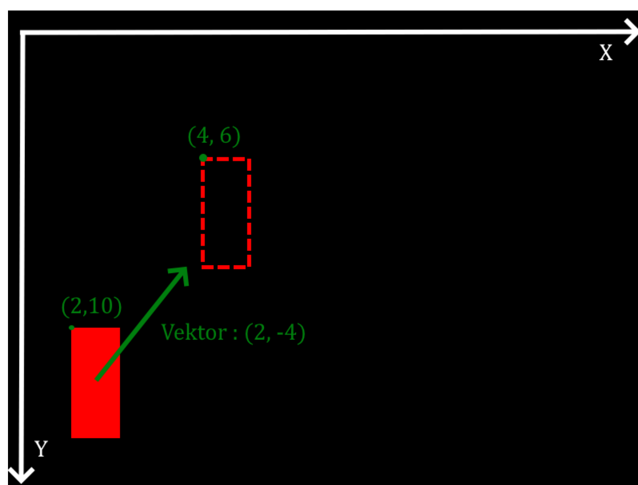
Obrázek 3: Zvolený způsob implementace obdélníku



Obrázek 4: vizualizace kolize

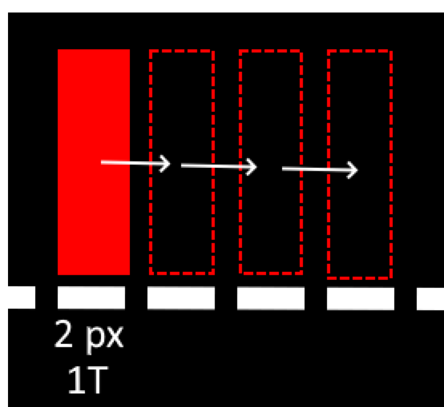
4.3.1 Funkcionální řešení pohybu a kolize

Ve funkcionálním programování není možné přímo měnit stav objektu, takže pro pohyb je potřeba napsat funkci, která vypočítá novou pozici. Na vstupu této funkce bude aktuální pozice objektu a vektor pohybu. Pozice objektu je reprezentována jako (p_x, p_y) viz. obr. 3. Složka vektoru se přičte k odpovídající souřadnici objektu a tento výpočet se vrátí jako nová pozice objektu. Matematický zápis pohybu: $(p_x + v_x, p_y + v_y)$ viz. obr. 5. *Směr pohybu* určuje znaménko složky vektoru. Pro příklad uvedeme vektor $(2, -4)$, na x-ové ose se pohybujeme směrem doprava, protože první složka vektoru je kladná a na y-ové ose se pohybujeme nahoru, protože druhá složka je záporná.



Obrázek 5: Vizualizace konkrétního pohybu

Program pracuje v diskrétním čase, takže pohyb nastává v okamžiku před překreslením obrazu, dle obnovovací frekvence viz. obr. 6. Na pohyb postavy má vliv pouze vstup od hráče a gravitace. Hráč může s postavou pohnout doprava, doleva a nebo může vyskočit. Na x-ové ose je vstup od hráče nesetrvačný, takže ve chvíli, kdy hráč pustí klávesu je pohyb na x-ové ose zastaven. Na y-ové ose může hráč pouze vyskočit, což je nastavení gravitační hodnoty na negativní hodnotu.

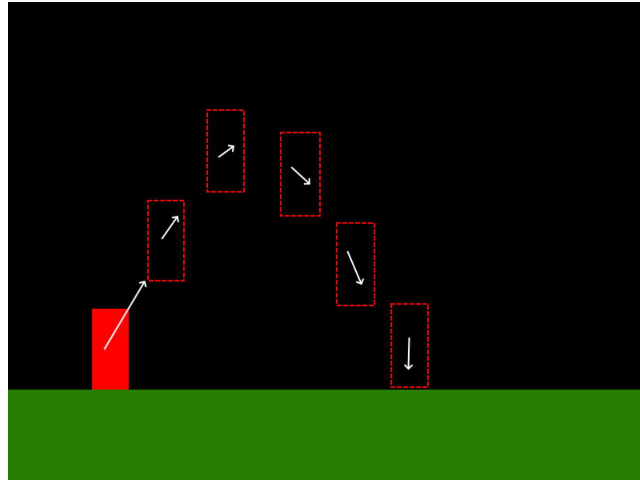


Obrázek 6: Vizualizace pohybu v jednom okamžiku

Gravitace je simulována pomocí speciální hodnoty, která se vždy přičítá k y-ové složce pohybového vektoru. Máme gravitační hodnotu g, y-ovou složku vektoru Py a konstantu zrychlení z. Při každém obnovení se hodnoty počítají takto: $Py' = Py + g$ a $g' = min(g + z, 3)$. Uveďme příklad, kdy maximální hodnota gravitace je číslo 3 a hodnota zvětšení za jeden okamžik je 0.5. V momentě, co hráč zmáčkne tlačítko pro skok se gravitační hodnota nastaví na -5 a postava se začne pohybovat směrem nahoru. V prvním okamžiku se pohne o 5 pixelů nahoru, v dalším o 4.5 a tak dál, dokud se hodnota nedostane na nulu. V tu chvíli se směr pohybu otočí a postava se začne pohybovat směrem dolů. Tyto hodnoty ovlivňují jak moc daleko může hráč skočit a jak dlouho bude trvat než dopadne. Vizualizace skoku viz. obr. 7.

Hráč může skákat pouze pokud postava stojí na obdélníku, takže je potřeba v implementaci kontrolovat, jestli došlo ke spodní kolizi(viz. dále). Pokud došlo ke spodní kolizi je možné skočit. Také je potřeba kontrolovat, že nedošlo k horní kolizi, pokud ano, je nutné v daný okamžik vynulovat y-ovou složku pohybového vektoru.

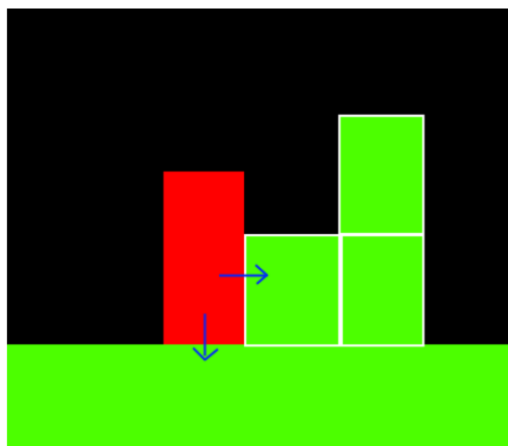
Pohybu musí předcházet kontrola kolize. Funkce nemůže vrátit takovou pozici, která by znamenala, že se budou dva či více objektů překrývat viz. obr. 4. Je tedy potřeba, aby na vstupu funkce řešící pohyb byly všechny obdélníky, se kterými může dojít ke kolizi. Seznam těchto obdélníků se bude nazývat *mapa*. V případě, že by mělo dojít ke kolizi, je potřeba zjistit, se kterým obdélníkem na mapě dochází ke kolizi a o jaký typ kolize se jedná. Na dvourozměrné ploše máme čtyři typy kolizí, jednu pro každou hranu hráčského objektu.



Obrázek 7: Vizualizace skoku

Jak již bylo zmíněno, *směr pohybu* určuje znaménko složky vektoru. Kladné hodnoty znamenají pohyb doprava a dolů, a záporné hodnoty pohyb doleva a nahoru. Pro pohyb po y-ové ose se kontroluje horní a spodní kolize, například pokud se hráčská postava pohybuje směrem dolů je možné, aby na y-ové ose nastala pouze spodní kolize. Pro x-ovou osu se jedná o pravý, nebo levý typ kolize, například pro pohyb směrem doprava může nastat pouze pravá kolize. Vizualizace na obrázku 8.

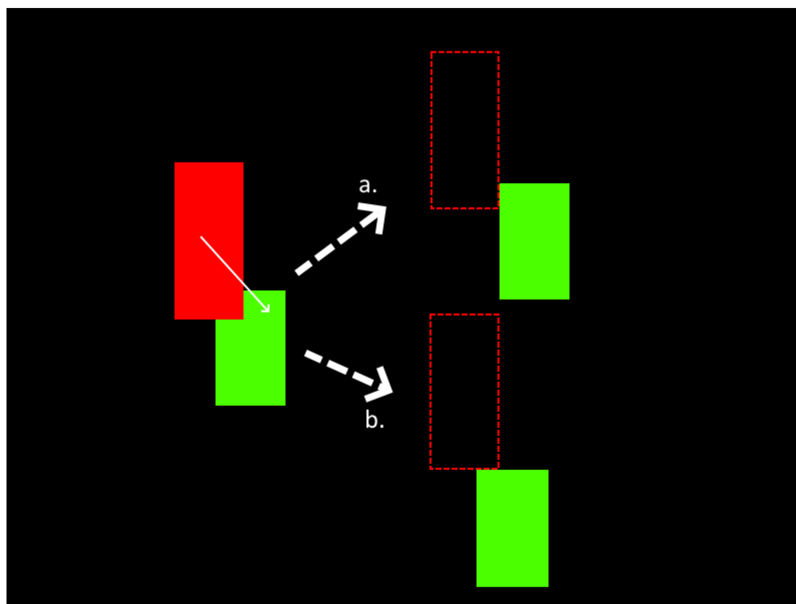
Kolize se řeší tak, že pro daný typ kolize se vypočítá taková pozice, ve které nedochází k překrytí s kolidovaným objektem. Například při pravé kolizi je potřeba vypočítat x-ovou souřadnici tak že, pravá hrana hráčského objektu odpovídá levé hraně kolidovaného obdélníku. To se vypočítá pomocí x-ové souřadnice kolidovaného obdélníku, od které se odečte šířka hráčského objektu. Takto vypočítaná hodnota se vrátí místo původní kolidující složky vektoru.



Obrázek 8: Situace kdy dochází k pravé a dolní kolizi

4.3.2 Speciální kolize

S výjimkou těchto základních typů kolize existují i speciální případy, které je potřeba brát v úvahu. Kolize rohů je případ, kdy dochází ke kolizi na rozích objektů. Pro příklad, kdy hráčský objekt narazí pravým dolním rohem obdélníku na levý horní roh obdélníku kolidovaného objektu. Toto může být způsobeno tím, že výpočet souřadnice jedné osy nebere ohled na osu druhou viz. obr. 9.



Obrázek 9: Kolize rohu a její dva možné výsledky

Rubberbanding je efekt, kdy hráčský objekt je přemístěn na předchozí pozice navzdory pohybovému vektoru. Toto může být způsobeno pořadím, ve kterém se počítají x-ové a y-ové souřadnice.

Přeskočení kolize je problém, kdy je jeden či více obdélníků přeskočeno. To může být způsobeno velikostí vektoru pohybu, pokud je pohyb za jedno obnovení obrazovky větší než je objekt, se kterým by mělo dojít ke kolizi bude objekt přeskočen a kolize nenastane.

Všechny tyto speciální typy v této práci řeším pomocí přiřazení priority y-ové osy. Ta se počítá první a vypočítaná hodnota je předána výpočtu x-ové osy, takže kolize na y-ové ose mají přednost a jsou vyřešeny první.

4.4 Konkrétní implementace

V mé implementaci jsou souřadnice počítány zvlášť pro obě osy. Dvě pomocné funkce, `solve_x_col` a `solve_y_col`, které nám vrací nové x-ové a y-ové souřadnice. První věc je potřeba vytvořit seznam kolidovaných objektů pomocí funkce `collision_test`. Detekce je preemptivní, tedy funkce `collision_test` je volána na objekt, ke kterému je již přičten požadovaný pohybový vektor.

Funkce vrací generátor, který iteruje přes každý objekt, se který je možné kolidovat a vrací pouze ty, se kterými dochází ke kolizi. Ze seznamu je vytvořen druhý seznam obsahující pouze relevantní vlastnosti pro výpočet nové pozice, pro funkci na výpočet x-ové souřadnice je to x-ová souřadnice objektu a šířka objektu. V případě, že ke kolizi nedochází je přičtena hodnota pohybového vektoru k x-ové souřadnici a tato hodnota vrácena jako nová x-ová souřadnice. Pokud byla detekovaná kolize, je nová souřadnice zjištěna na základě pohybového vektoru. Pokud je kladný, znamená to, že se hráčský objekt pohybuje doprava, takže nová x-ová souřadnice se získá pomocí odečtení šířky hráčského objektu od nejmenší x-ové souřadnice kolidovaných objektů. V opačném případě se vrací součet šířky a x-ové souřadnice kolidovaného objektu. Y-ová souřadnice je řešena podobně, s rozdílem, že je zde potřeba také vracet dvě logické hodnoty pro zjištění, zda hráč může provést skok a pro zjištění, zda hráč narazil na objekt horní hranou, aby bylo možné vyrušit jeho momentum.

```
1  def solve_x_col(rect, tiles, dx, new_y):
2    hit_list = collision_test(pygame.Rect(rect.x + dx, new_y, rect.w,
         rect.h), tiles)
3    x_col = [[tile.left, tile.w] for tile in hit_list]
4
5    if x_col:
6      if dx > 0:
7        return min([x[0] for x in x_col]) - rect.w
8
9      elif dx < 0:
10       return max([x[0] for x in x_col]) + max([w[1] for w in x_col])
11
12     elif dx == 0:
13       return rect.x
14
15   else:
16     return rect.x + dx
```

Code Listing 1: Kód pro výpočet x-ové souřadnice

4.4.1 Speciální případy

Jak již bylo popsáno výše, existují speciální případy kolize, které je potřeba nějakým způsobem řešit. V této implementaci se jednalo hlavně o kolizi v rozích a o zmíněný *rubberbanding* efekt. Původní implementace počítala nové souřadnice nezávisle na sobě, což způsobovalo, že zmíněné speciální případy kolize měly reálně dvě řešení. Pro příklad uvedu situaci, kde hráčský objekt dopadl na roh objektu, zatímco se pohyboval doprava. V tento moment dochází k oběma kolizím, tedy ke kolizi na ose x a na ose y zároveň pro jeden objekt, takže hráčský objekt přeskakuje mezi dvěma hodnotami, dokud hráč neukončí pohyb doprava. Kolize přeskakují každou šedesátinu vteřiny a podle toho, která byla vyřešená v moment ukončení pohybu je určeno, kam se hráčský objekt posune.

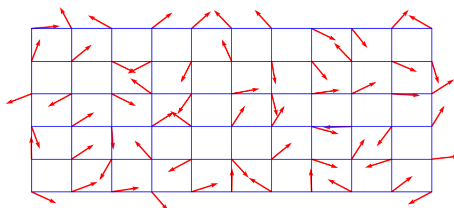
Druhý případ speciální kolize způsoboval to, že v případě, že hráč dopadl na více objektů, zatímco se pohyboval na ose x, byl vrácen na x-ovou souřadnici prvního objektu, na který dopadl. Tyto speciální kolize jsou v mé implementaci řešeny pomocí priority y-ové osy tak, že se počítá první a následně je nová y-ová hodnota předána funkci pro řešení pohybu a kolize x-ové osy, takže například pro kolizi v rozích objektů má řešení funkce `solve_y_col` prioritu a druhé řešení nemůže nastat.

5 Procedurální generace mapy

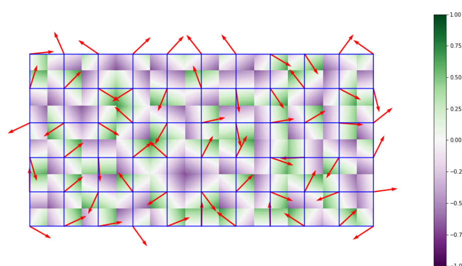
Jedná se o metodu vytváření dat pro mapu pomocí algoritmu a předem stanovených pravidel. To umožňuje vytvářet nekonečné mapy s náhodnou variací. V herním vývoji se jedná o velice populární techniku, zejména při vývoji her s otevřeným světem. Asi nejznámějším příkladem je hra Minecraft, kde celý svět je procedurálně generován. Existuje mnoho specializovaných algoritmů pro generaci herního obsahu, jako například pro jeskyně, města, řeky a jiné. Pro můj program jsem pro implementaci generování mapy zvolil algoritmus Perlin noise.

5.1 Perlin noise

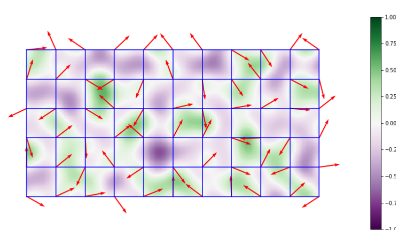
Perlin noise je algoritmus využívaný ke generování šumu, který má různé využití. Mimo procedurální generování v herním vývoji se dá například využít pro generování počítačové grafiky. Princip spočívá v tom, že je nejdříve vytvořena mřížka stejně velkých čtverců, následně se do každého rohu čtverce umístí náhodný vektor gradientu viz. obr. 10. Pro libovolný bod se identifikují vrcholy buňky, do které bod spadá a pro každý vrchol se spočítá posunový vektor. Dále se udělá skalární součin mezi tímto vektorem a vektorem gradientu, pokud se bod nachází přesně ve vrcholu, je součin nulový viz. obr. 11. Nakonec se pomocí interpolace vypočítá průměr skalárních součinů pro každý bod, což zajistí přirozenější výsledek viz. obr. 12. Parafrázováno z webové stránky: [6].



Obrázek 10: Mřížka s vektory gradientu. Obrázek převzatý z [7].



Obrázek 11: Skalární součin vektorů. Obrázek převzatý z [7].



Obrázek 12: Vizualizace Perlin noise algoritmu. Obrázek převzatý z [7].

5.2 Konkrétní implementace generování

V mé implementaci používám pro samotný algoritmus knihovnu `OpenSimplex`. Generování je implementováno pomocí *chunků*, což je část mapy, konkrétně $8 * 8$ bloků. Jako první věc se určí kolik *chunků* se má generovat najednou, čím více se jich generuje najednou, tím náročnější na výkon generování je. Vzhledem k implementaci není nutné generovat více jak $4 * 4$ *chunků*. Spočítá se x-ová a y-ová pozice *chunku* a následně se vygeneruje *chunk* pomocí funkce `generate_chunk`. Funkce pro každý *chunk* vygeneruje bloky dle šumu. Pro zajištění plynulejšího terénu se kontroluje y-ová pozice *chunku*, pokud je větší než $8 - noise$, vygenerují se pevné bloky. To zajistí, že od určité hranice se budou generovat pouze bloky, a ne prázdná místa, což vytváří efekt realističtější mapy.

```
def generate_chunk(x, y, seed=None):
  if seed is not None:
    random.seed(seed)
    noise = OpenSimplex(random.randint(0, 10000))

  def generate_tile(x_axis, y_axis):
    chunk_x = x * 8 + x_axis
    chunk_y = y * 8 + y_axis
    my_noise = int(noise.noise2(chunk_x * 0.1512, chunk_y * 0.1512) *
        8)


    if chunk_y > 8 - my_noise:
      return [[chunk_x, chunk_y], 1] # dirt
    else:
      return None # air

  tile_rows = [[generate_tile(x_axis, y_axis) for x_axis in range(8)
      ] for y_axis in range(8)]
  chunk_tiles = [tile for row in tile_rows for tile in row if tile
      is not None]

  return chunk_tiles
```

Code Listing 2: Generování chunků

6 Uživatelská příručka

Hru není potřeba instalovat, stačí spustit její exe soubor. Úvodní menu dává hráči na výběr mezi herními režimy, které si může zvolit pomocí kliknutí na odpovídající tlačítka. Tlačítka změní barvu pro signalizaci, že se ukazatel nachází na tlačítku. Po kliknutí hra ihned začne.

6.1 Ovládání

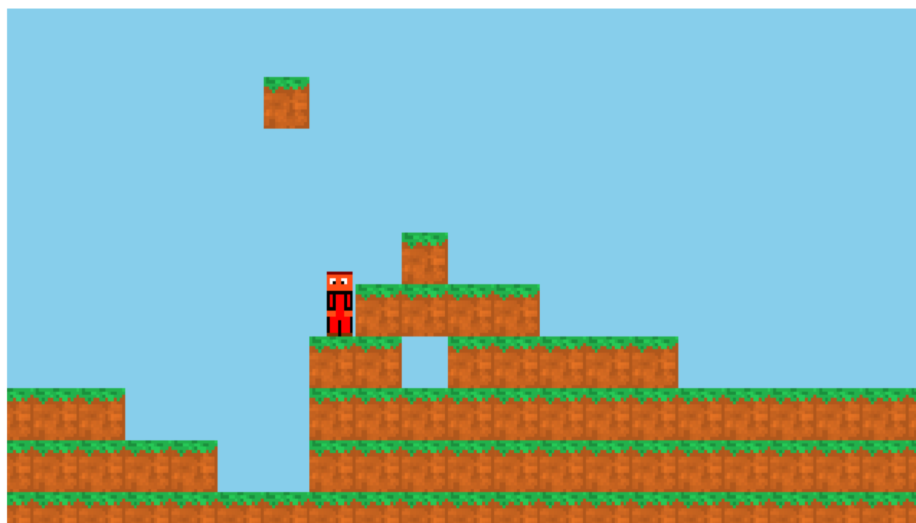
Uživatelské prvky jako výběr a kliknutí na tlačítko se ovládají pomocí myši. Ukončení nebo pauza hry může být vyvolána zmáčknutím tlačítka ESC. Herní postava se ovládá buď pomocí klasického W,A,S,D, nebo pomocí směrových šipek. Hráč má kontrolu nad postavou i v případě skoku, takže se stále může pohybovat, ale nemůže znovu skočit. Pohyb na y-ové ose se nevyruší změnou pohybu na x-ové ose, takže je možné například přeskočit překážku nad postavou.

6.2 Herní režimy

Hra obsahuje tři herní režimy, mezi kterými si může hráč vybrat. Všechny režimy nabízí stejnou hratelnost, a co se liší je způsob vytvoření mapy. Režim je možné zvolit pouze na hlavní obrazovce.

6.2.1 Generovaná mapa

V tomto herním režimu se mapa procedurálně generuje a cílem hráče je dostat se co nejdál. Hráč by měl ihned po začátku jít směrem doprava, protože z leva postupuje temnota, které když se hráč dotkne tak, hra končí. Je potřeba poznamenat, že generace nezaručuje, že se dá postoupit dále, takže tento herní režim je částečně založen na štěstí generace. Není implementována žádná metoda, kterou by bylo možné hru vyhrát, takže jediný způsob, jak hru ukončit je manuálně nebo prohrou.



Obrázek 13: Ukázka vygenerováného terénu

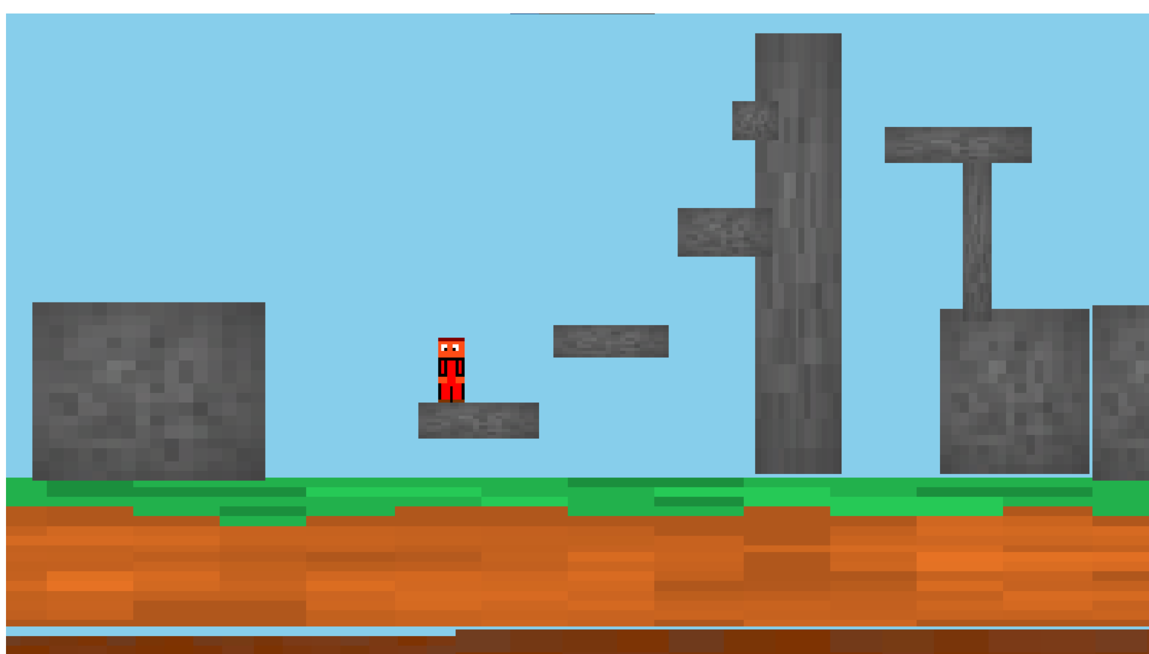
6.2.2 Map creator

V tomto režimu si hráč vytváří vlastní mapu a to tak, že myší podrží levé tlačítko a začne se vytvářet obdélník. Jakmile hráč pustí levé tlačítko je obdélník vytvořen. Režim obsahuje volbu textur pro tvořené obdélníky, stačí najet ukazatelem myši do pravé části obrazovky a otevře se menu s texturami. Zároveň toto menu obsahuje možnost mazání, pokud je tlačítko mazání zakliknuto, je možné stejným způsobem jako při vytváření zvolit obdélníky, které chce hráč smazat. Mazání se vypne opětovným kliknutím na stejné tlačítko. Mapu, respektive kameru je možné posouvat pomocí směrových šipek. Nachází se zde červený čtverec, který není možné smazat. Ten ukazuje, kde se ocitne hráčská postava. Ke spuštění hry s touto mapou musí hráč zmáčknout tlačítko ENTER.

6.2.3 Story mode

Tento herní režim je předem vytvořená mapa, kde se hráč musí dostat na konec mapy do dveří. Jedná se o jediný režim, který je možný dohrát.

Obrázek 14: Ukázka vytváření mapy



Obrázek 15: Spuštění vytvořené mapy

7 Implementace hry

Vytváření herní smyčky pomocí Pygame knihovny se dělá pomocí `while` cyklu, kde se vykonají všechny změny dat, kontrola vstupů od hráče, či vizualizace generované mapy. Na konci smyčky je potřeba překreslit obrazovku. O to se

postará knihovna pomocí příkazu `pygame.display.update()`. Vstupy od uživatele se řeší pomocí `pygame.event.get()`, což je funkce, která nám vrátí stisknuté klávesy. Jsou zde dva typy. `KEYDOWN` je stisknutí klávesy a `KEYUP` je puštění. Dále je možnost specifikovat konkrétní klávesy. Myš má vlastní typ `MOUSEBUTTONDOWN` a funkce, které vrací tlačítka na myši, které byly zmáčknuty a kde se nachází kurzor myši.

```
1  while running:
2   for event in pygame.event.get():
3 
4   if event.type == KEYDOWN:
5    if event.key == K_ESCAPE:
6     pause_menu()
7 
8    if event.key == K_RIGHT:
9     player_movement = [2, player_movement[1]]
10    movement = True
11 
12 
13   if event.key == K_LEFT:
14    player_movement = [-2, player_movement[1]]
15    movement = True
16 
17   if event.key == K_UP:
18    if can_jump:
19     player_movement = [player_movement[0], -5]
20     can_jump = False
21 
22  pygame.display.update()
23  clock.tick(60)
```

Code Listing 3: Kód pro vstup hráče v pygame.

Pro herní režimy, kde je hráčská postava je vhodné mít stejnou smyčku. Jediné, co se mění je mapa a několik prvků, ale ovládání a pohyb zůstávají stejné. Ostatní režimy jako například menu, pauza či vytváření mapy, je třeba oddělit. V každém z těchto režimů je potřeba jiné vykreslování a ovládání, a slučovat je by vyžadovalo kontrolu, která by zbytečně dělala kód nepřehledným. Navíc toto přináší výhodu toho, že pokud přeskočíme z jednoho cyklu do druhého, není potřeba ukládat pozici hráče či mapu.

Nejunikátnější smyčka je ta pro vytváření herní mapy, zde je hned několik problémů, které bylo potřeba vyřešit. Prvním byl způsob, jak je definován obdélníkový objekt v knihovně Pygame a to pomocí počáteční souřadnice, která reprezentuje levý horní roh obdélníku, což zabraňovalo tvoření obdélníků jiným směrem než doprava a dolů. Toto se vyřešilo tak, že pro x-ovou a y-ovou počáteční souřadnici se bere minimální hodnota z aktuální pozice a počáteční pozice. Pro šířku a výšku se počítá absolutní hodnota z aktuální pozice mínus z původní.

Dalším problémem u vytváření herní mapy bylo ukládání a mazaní obdélníků a jejich odpovídající textury, při spuštění hry je na obdélníky aplikovaná textura, kterou si hráč zvolil při vytvoření, tedy je potřeba uložit, která textura patří ke kterému obdélníku. Bohužel knihovna tuto možnost nenabízí, a proto je potřeba ukládat textury v seznamu se stejnou délkou, který je potom předán spolu se seznamem obdélníků. Toto dělalo problém u mazání obdélníků. Je možné smazat kterékoliv obdélníky zároveň, a tak bylo potřeba najít odpovídající texturu pomocí indexu hodnoty a tu následně mazat ze seznamu.

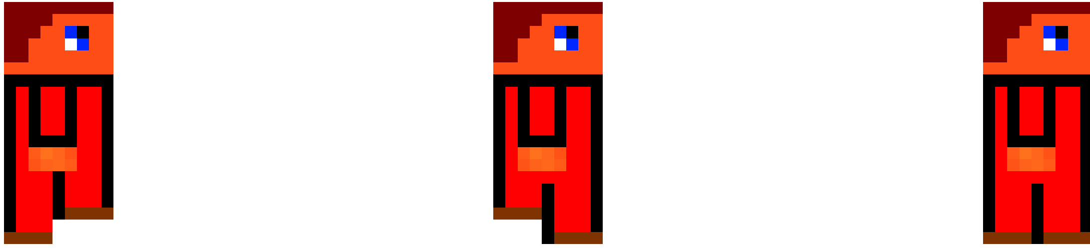
Scrolling představoval problém ve dvou případech. Byla potřeba ho implementovat jak pro pohyb postavy, tak i pro pohyb při vytváření mapy. V případě pohybu herní postavy nevypadal *scrolling* přirozeně, což nevytvářelo dobrý pocit z pohybu, proto jsem implementoval určité opoždění, které vytváří mnohem přirozenější pohyb kamery.

V případě *scrollingu* při vytváření mapy toto nebylo zapotřebí, ale bylo potřeba zajistit to, že *scrolling* neovlivnil reálnou pozici obdélníků. Bylo potřeba si udržovat absolutní hodnotu, o kterou se kamera posunula. Tato hodnota byla odečtena od pozice obdélníku, a tím se uložila reálná pozice vzhledem k startovací pozici.

Animace jsou řešeny pomocí slovníku obrázků. Pokud je postava v pohybu, je postupně přičítána hodnota k animační proměnné. Tato hodnota je zaokrouhlena na celé číslo, které odpovídá obrázku, a to vzhledem k obnovovací frekvenci vytváří dojem pohybu postavy. pokud se postava pohybuje na druhou stranu, jsou obrázky jednoduše otočeny po dané ose.

7.1 Možné zlepšení

Zaměření práce bylo na optimalizaci funkcionálního pohybu, generace a vytváření mapy, ale nestihl jsem implementovat zábavnější elementy hry. Jedno z možných zlepšení by bylo přidání zábavnějších elementů, jako například sběratelské ob-



Obrázek 16: Obrázky animace herní postavy

jekty, které by dávaly skóre. Další by byli schopnosti hráče, které by přidávaly variaci pro pohyb. Nepřátelské entity by zlepšili hratelnost a obtížnost pro hráče a přidaly další vrstvu pro pohybovou implementaci, protože by bylo potřeba reagovat jinak na kolizi s takovými entitami.

Grafická část programu taky potřebuje vylepšení. Všechny textury jsou ručně kresleny, ale nejsem profesionální grafik. Systém animací je značně zjednodušený a při přidání dalších entit by bylo potřeba implementovat obecný způsob pro animace.

Závěr

Výsledkem práce je platformová skákací hra, která demonstruje využití funkcionálních paradigmat v herním vývoji. Ačkoliv čistě funkcionální paradigma není vhodné pro herní vývoj, v kombinaci s imperativním programováním může vést k menší chybovosti, mnohem lepšímu škálování a přehlednosti kódu.

Za celou dobu vývoje jsem nezaznamenal pokles ve výkonu programu a program vždy probíhal plynule. Paradigma značně usnadnilo testování a řešení chyb. Čistě funkcionální části programu se velice snadno opravovali a z výsledku funkce bylo patrné, kde vzniká chyba.

Hra je hlavně pro demonstrativní účely. Z tohoto důvodů zde chybí zábavnější prvky, o kterých píšu v sekci možné zlepšení. Hra není určena k distribuci, a proto by tyto prvky přidávaly nadbytečný kód, což by bylo zbytečné pro tuto práci.

Conclusions

The result of this thesis is a platformer game that demonstrates the use of purely functional paradigms in game development. While purely functional paradigms aren't suitable for game development, in combination with imperative programming, they can lead to better scaling, code clarity, and fewer errors.

During the whole development of this program, I haven't noticed any decrease in performance, and the program is always executing steadily. Purely functional paradigm simplified testing and error fixing. These parts of the program were very easy to fix, and the outcome of the functions always highlighted where the error had occurred.

The game is for demonstration purposes. Because of this, more enjoyable elements are not implemented, I talked about these in the "possible improvements" section. The game is not meant for actual play, and these elements would add unnecessary code that would be useless for the purpose of the thesis.

Literatura

[1] Foundation, The Python Software. *Python*. Dostupný také z: ⟨https://www.python.org/doc/⟩.

[2] Community, Pygame. *Pygame*. Dostupný také z: ⟨https://www.pygame.org/docs/⟩.

[3] Alexander, Alvin. *Functional Programming, Simplified: (Scala Edition)*. First. USA: CreateSpace Independent Publishing Platform, 2007-12-7. 728 s. ISBN 1979788782.

[4] Gaming, Nodwin. *The evolution of platform games in 9 steps*. Dostupný také z: ⟨https://www.redbull.com/in-en/evolution-of-platformers⟩.

[5] Corporation, Valve. *Hitbox*. Dostupný také z: ⟨https://counterstrike.fandom.com/wiki/Hitbox⟩.

[6] Touti, Raouf. *Perlin Noise: A Procedural Generation Algorithm*. Dostupný také z: ⟨https://rtouti.github.io/graphics/perlin-noise-algorithm⟩.

[7] Matthewslf. *Visualisation of Perlin noise*. Dostupný také z: ⟨https://en.wikipedia.org/wiki/Perlin_noise⟩.