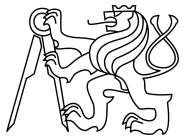
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Karkulka v gramatickém lese: Samoučící 2D hra generovaná gramatikou
Student:	Bc. Klára Hájková
Vedoucí:	Ing. Radek Richtr
Studijní program:	Informatika
Studijní obor:	Znalostní inženýrství
Katedra:	Katedra teoretické informatiky
Platnost zadání:	Do konce letního semestru 2017/18

Pokyny pro vypracování

Karkulka v gramatickém lese je jednoduchá 2D hra, jejíž prostředí (terén, plošiny, překážky, nepřátelé, ...) se vytváří dle pravidel kontextové gramatiky. Hra se na základě vytvořené metriky učí od uživatele (problematické překážky, problematičtí nepřátelé, častý způsob smrti, ...) a vytváří tak úpravou svých pravidel pro uživatele stále těžší herní výzvy.

1) Nastudujte problematiku procedurálního generování prostředí.
2) Navrhněte sadu pravidel kontextové gramatiky (model) vhodné pro vytváření prostředí hry.
3) Navrhněte vhodný způsob měření kvality hraní hráče (metrika) a její aplikaci na učení hry.
4) Navrženou hru implementujte (C++).
5) Vytvořenou hru podrobte vhodným testům (výkonnostním, uživatelským, ...) a vyhodnoťte dosažené výsledky.

Seznam odborné literatury

Dodá vedoucí práce.

doc. Ing. Jan Janoušek, Ph.D.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 22. ledna 2017

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra teoretické informatiky

Diplomová práce

Karkulka v gramatickém lese

Bc. Klára Hájková

Vedoucí práce: Ing. Radek Richtr

9. ledna 2018

Poděkování

Děkuji svému vedoucímu Ing. Radku Richtrovi za rady, ochotu, trpělivost a podnětné připomínky a své rodině za morální podporu. Také bych chtěla poděkovat všem lidem, kteří se podíleli na uživatelském testování. Speciálně děkuji Martinu Karabovi za grafiku Karkulky, kterou vytvořil pro tento projekt.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 9. ledna 2018 .....

Odkaz na tuto práci

Hájková, Klára. *Karkulka v gramatickém lese.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2018.

Abstrakt

Práce se věnuje procedurálnímu generování samoučící 2D hry typu platformer. Generování je prováděno stochastickou kontextovou gramatikou, jejíž pravidla se vytvářejí podle herních objektů definovaných uživatelem. Model řídící pravděpodobnostní rozdělení pravidel se za běhu hry učí a přizpůsobuje tak, aby danému hráči na míru připravoval stále těžší herní výzvy. Byl implementován prototyp hry a za pomoci uživatelského testování byly vybrány faktory vhodné k měření a učení modelu.

Klíčová slova kontextová gramatika, procedurální generování, samoučící hra, 2D platformer, Qt

Abstract

The thesis deals with procedural generation of a self-learning 2D game of the platformer type. Generation is conducted by stochastic context-sensitive grammar, rules of which are created based on game objects defined by the user. The model managing probability distribution of rules learns and adapts during the run of the game in such a way, so that the player is presented with custom-tailored and gradually more difficult challenges. Game prototype has been implemented and factors suitable for measuring and model's learning have been selected through user testing.

Keywords context sensitive grammar, procedural generation, self-learning game, 2D platformer, Qt

Obsah

Seznam obrázků

Seznam tabulek

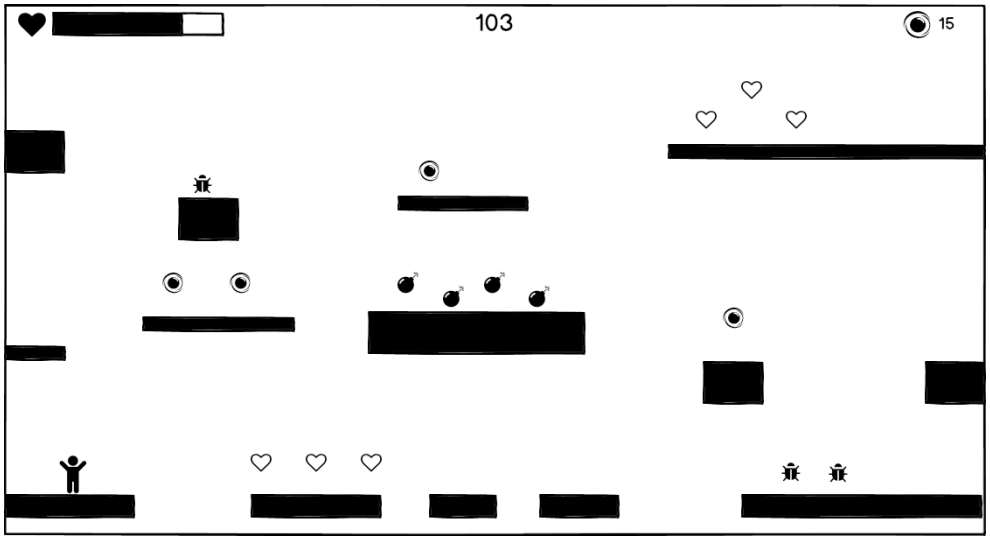
Úvod

Jako procedurální generování se označuje způsob vytváření obsahu pomocí algoritmů s minimálním vstupem od uživatele. Zapsání pouze pravidel, jež jsou následně použity k vygenerování obsahu, šetří paměťové nároky aplikací, což bylo potřebné především v dobách, kdy byla paměť počítačů značně omezená. V minulosti tak procedurální generování umožnilo vznik komplexních her, jež by se do jejich paměti nevešly.

Procedurální generování v herním průmyslu však nachází uplatnění i v dnešní době, kdy se velikost operačních pamětí počítá na gigabajty. Zavedení prvku náhody do generování umožňuje tvůrcům vytvářet nepřeberné množství objektů, map, herních možností, či dokonce zvuků.

Ačkoliv potenciál vygenerovaných prostředí je veliký sám o sobě, lze jej zkombinovat s učícími algoritmy a vytvořit tak obsah unikátní pro každého hráče. S tím souvisí i přizpůsobení se hráčovu chování – postupně hru ztěžovat tak, aby mu vytvářela stále nové herní výzvy na míru. Cílem této práce je vytvoření právě takové hry, která se bude umět přizpůsobit každému hráči a své prostředí bude generovat pomocí formální kontextové gramatiky.

Vzhledem k tomu, že vytvářená hra má náležet do žánru platformer, je vhodné si krátce představit jeho obvyklou podobu. Hlavními prvky na obrazovce tohoto žánru jsou plošinky (platformy). Chytré rozmístění plošinek nutí hráče k přesným skokům. Během průchodu úrovněmi se hráč potýká s různorodými monstry a sbírá bonusové předměty. Možnou podobu hry v tomto žánru lze vidět na následujícím obrázku:



Práce je rozčleněna do následujících kapitol:

Rešerše V kapitole rešerše jsou zevrubně rozebrány zdroje relevantní pro tuto práci. Lze mezi nimi nalézt odborné články vztahující se jak ke způsobu generování herního obsahu, tak i vyjádření odborníků z praxe vztahující se například k samotné architektuře her.

Analýza výpočetní inteligence Zde se řeší způsob použití kontextové gramatiky ke generování prostředí. Dále se kapitola zabývá analýzou a definicí pravidel a hledání vhodných parametrů pro učení.

Analýza a návrh hry Kapitola se zabývá analýzou a návrhem herního prototypu, jenž slouží nejprve k otestování parametrů vhodných k učení a poté k testování samotného generování prostředí a jeho validace.

Návrh výpočetní inteligence V kapitole návrhu je představen způsob převedení generování kontextovou gramatikou do podoby, jež se dá implementovat.

Realizace Kapitola se krátce věnuje detailům z implementace herního prototypu i generování prostředí.

Testování Poslední kapitola popisuje provedené testy a jejich výsledky.

Kapitola 1

Rešerše

Tato kapitola se zabývá nejprve rešerší procedurálního generování obecně. V další části se rozebírají myšlenky, možnosti a celkový přístup k procedurálnímu generování v plošinovkách. Poslední část rešerše řeší používané architektury v oblasti her a jejich výhody a nevýhody.

1.1 Procedurální generování

Do kategorie procedurálního generování patří vše, co se negeneruje manuálně, ale algoritmicky na základě minimálního (nebo žádného) vstupu od uživatele a definovaných pravidel.

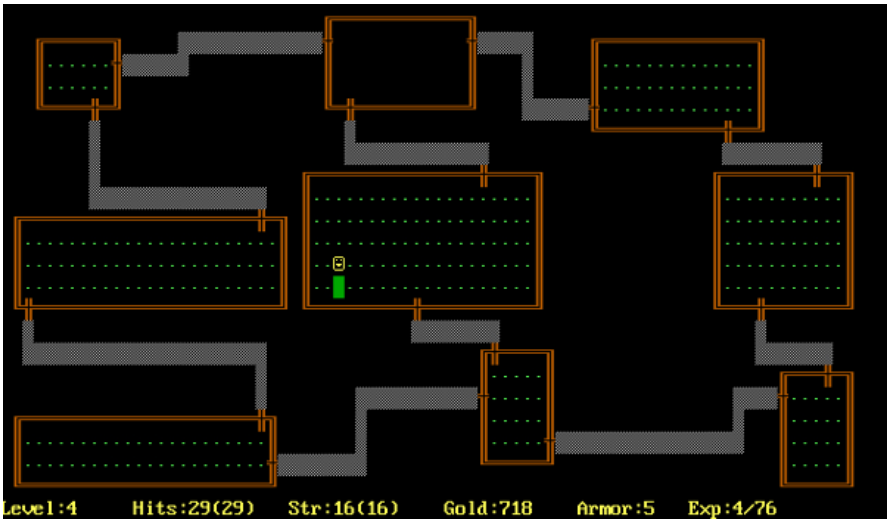
Historickým důvodem vzniku procedurálního generování bylo podle A. Amato[10] uspoření paměti – dříve neměly počítače dostatečnou paměť, aby se do ní dala nahrát veškerá data hry – například kompletní mapa, pozice objektů v ní, zvuky, či animace. Pravidla pro vytvoření mapy se tedy napsala do zdrojového kódu a mapa se vytvářela za běhu hry. Hra potom nebyla omezena paměťovými možnostmi, ale zbylým hardwarem. Podobný problém se znovu objevil s příchodem prvních mobilních her.

I v dnešní době se tento přístup zachoval a tvoří samostatnou subkulturu, která si říká demoscéna. Skupina tzv. scenerů, již obvykle tvoří hudebníci, grafici a programátoři, se snaží tvořit audiovizuální díla a z aktuálního hardwaru dostat maximum [11][1].

Procedurální generování lze rozdělit do dvou hlavních skupin:

- deterministické (vždy stejné – například výše zmíněná dema),
- stochastické (některé části mohou být náhodné – pokaždé nová úroveň ve hře).

[1] Dokonce se každý rok udílejí ceny Scene Awards mj. i v kategorii, kdy je velikost výsledného kódu omezena na 64 kibibajtů, případně i 4 kibibajty (více na webu `awards.scene.org`).



Obrázek 1.1: Jedna z prvních her generovaných procedurálně – Rogue[14] z roku 1980

1.2 Procedurální generování her

V herním prostředí můžeme (dle [12]) generovat různé úrovně, vlastnosti nepřátel, 3D mapy, bonusy, grafiku, dokonce i adaptivní hudbu. Jednou z prvních her, která své prostředí (konkrétně kobky) generovala procedurálně a stochasticky byla tahová hra Rogue z roku 1980 (viz. Obrázek 1.1). V této hře se vytvářely náhodné místnosti, které se iterativně propojovaly, dokud do každé místnosti nevedla alespoň jedna cesta.

Z některých dalších starších titulů zmiňuje R. Moss v [13] procedurální generování ve strategii Civilization, kde se s každou novou hrou vytvářela nová mapa k osídlení a prozkoumání. V Shadow of Mordor zase orkové měli různé vlastnosti vygenerované herním Nemesis Systémem. Obrovské otevřené světy ve Skyrimu, či kobky v Diablu nikdo nevytvářel ručně, ale byly také vygenerovány systémem pravidel.

Vzhledem k dnešním kapacitám operačních pamětí a velikosti diskového prostoru se zdá, že dnešní hry nepotřebují tolik šetřit pamětí jako dříve, ale stochastické procedurální generování přináší i jiné výhody. Nejzásadnější výhodou je znovuhratelnost – každá jednotlivá hra může být jiná, hráč může navštěvovat různé světy, poklady jsou schované na nových místech atd.

Generování ale šetří i čas a peníze – v momentě, kdy má hra obsahovat stovky či tisíce úrovní, zjevně levnější a rychlejší vytvořit generátor, než každou úroveň svěřit level designérovi a tvořit ji ručně. I proto se více indie vývojářů spoléhá na procedurální generování obsahu (viz Chao[15]).

Na druhou stranu, ani procedurálně generované moderní hry, nemusí hráčům zajistit stovky hodin zábavy, jelikož vytvořené prostředí je omezeno pravi-

dly generátoru. Takovým případem se v roce 2016 stala dlouho očekávaná hra *No Man's Sky*, která sice obsahovala 2^{64} planet s vlastní florou a faunou, ale herní možnosti se příliš nelišily, na což si stěžovala spousta hráčů, například:

> *The first planet you explore will take your breath away, but by the tenth, you'll have seen it all. There simply isn't enough to do.* – arugulaKhan, gog.com

> *Stereotyp je největší slabinou celého projektu. Je téměř bizarní něco takového prohlásit o hře s prakticky nekonečným počtem prozkoumatelných planet, ale No Man's Sky krutě trpí na nedostatek obsahu. Co tu vlastně dělat? Pokud vás nebaví čistě jen prozkoumávat podivuhodné krajiny, pak nemilá odpověď zní, že nic. Můžete běžet kupředu či létat okolo planet, jenže k čemu je bambilión světů, když na všech narazíte na pět totožných věcí?* – Jan Slavík, games.tiscali.cz

S tímto problémem úzce souvisí i další faktor – ručně vytvořené prostředí může lehce pamatovat na detaily, na malé hádanky k vyřešení a unikátnost každého kousku, čehož se v genericky vygenerovaném prostředí dosahuje velmi složitě. Takový systém musí dostatečně komplexní, aby jeho výstupy obsahovaly takové vlastnosti.

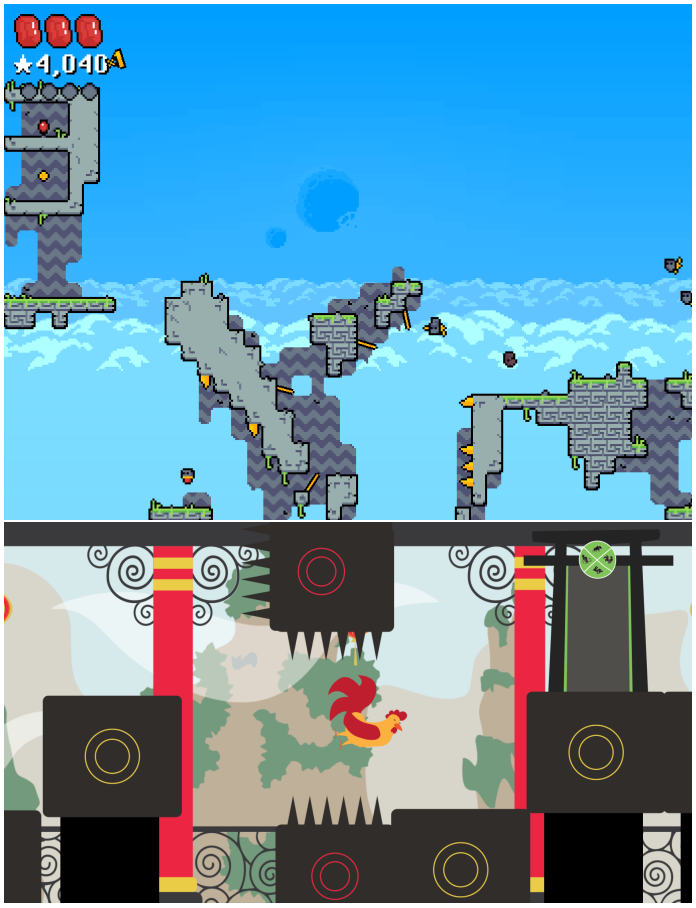
1.3 Generování prostředí v plošinovkách

Tato část se zaobírá procedurálním generováním v herním prostředí, přesněji v žánru plošinovek (platformer) a sidescroller platformer. Tento žánr je charakterizovaný překonáváním překážek, skákáním na plošinky (platformy), sbíráním bonusů a poražením monster. Podle [16] „Sidescroller" znamená, že obrazovka se posouvá zleva doprava spolu s tím, jak se pohybuje hráč. Zástupci v tomto žánru jsou například hry ze sérií Mario Bros a Sonic[16].

Průzkum platforem pro propagaci her a článků – akademických publikací, blogů i různých videonávodů odhalil, že o procedurální generování platformerů je sice velký zájem, ale většina řešení staví na velmi jednoduchých principech a nároky kladené na výsledek obvykle nejsou příliš vysoké.

1.3.1 Webové platformy pro propagaci her

Herní platforma *Steam* má ve svém obchodě pouze 15 titulů, které mají označení „2D", „platformer", nebo „procedural generation". Často jsou hry označeny jako „rogue-like" anebo „rogue-lite", což nejsou úplně přesně definované pojmy, ale vztahují se ke hře Rogue (viz sekci 1.2). Naneštěstí jen jedna z těchto her má uvedeno, jaké algoritmy jsou použity ke generování (kap. *Spelunky* 1.3.3).



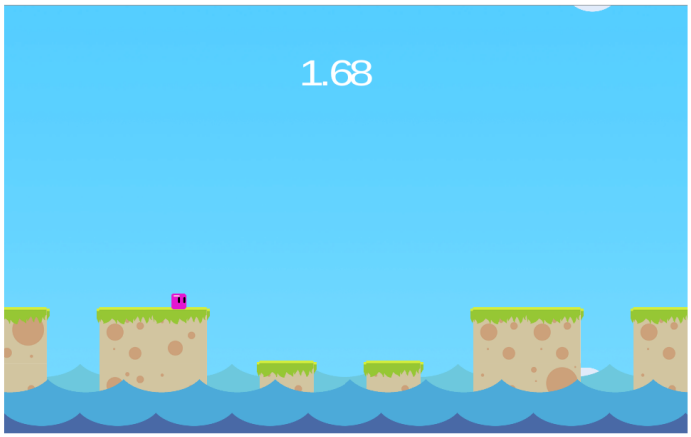
Obrázek 1.2: Dvě různé hry[1, 2] se stejným systémem procedurálního generování. Nahoře hra *The Bond of Stone*, dole *Samsara*.

Další platforma pro prezentaci a prodej indie her, *Itch.io*, obsahuje celkem 72 her splňující štítky výše. U některých her se přímo píše, jakým způsobem jsou vytvořené, u jiných lze poznat inspiraci *Spelunek* a u dalších jsou pravidla zjevná.

1.3.2 Hry se silným prvkem náhody

Hry *The Bond of Stone*[1] a *Samsara*[2] vypadají odlišně (viz. Obrázek 1.2), přesto se princip jejich procedurálního generování shoduje – jak autoři píší v popiscích her, všechny úrovně jsou náhodně poskládané z větších celků vytvořených člověkem.

Do kategorie her se silným prvkem náhody by se dala zařadit i jednoduchá hra (Obrázek 1.3), ve které je cílem hráče prostě přežít co nejdéle – *Jump'n* [3] je nekonečná hra, kde se podle popisku od autora náhodně generují



Obrázek 1.3: Hra Jump'n s velmi jednoduchým systémem generování plošin[3].

sloupy platforem různé výšky. Vzhledem k tomu, že postavička dokáže vyskočit do výšky celé obrazovky (a kamera se vertikálně nehýbe), na generování sloupů jsou z hlediska dosažitelnosti (doskoku) kladeny malé požadavky.

1.3.3 Spelunky

V roce 2008 Derek Yu vydal hru *Spelunky*, která spadá do „rogue-like" kategorie a zároveň se jedná o procedurálně generovanou plošinovku. Co se týče generování, má hra s Rogue společný první krok – vytváření a spojování místností. Ve Spelunkách ale místnosti nepůsobí tak odděleně ze dvou hlavních důvodů:

- hráč vidí jen malou část mapy,
- místnosti nejsou propojeny chodbami, ale tvoří mřížku 4×4, kde všechny sousední místnosti, které tvoří cestu úrovní, musejí být průchozí.

Darius Kazemi vytvořil webovou verzi[2], na níž demonstruje průběh prvních dvou fází generování hry.

A protože se spousta procedurálně generovaných plošinovek inspirovala právě tímto postupem, je vhodné si ho představit detailněji.

Každá úroveň sestává celkem z 16 místností, přičemž existují 4 základní druhy místností:

Typ 0: vedlejší místnost, která není na cestě úrovní,

Typ 1: místnost s levým a pravým průchodem,

[2] `http://tinysubversions.com/spelunkyGen/`

Typ 2: místnost s levým, pravým a spodním průchodem, pokud nad ní je další místnost tohoto typu, má průchod i v horní části,

Typ 3: místnost s levým, pravým a horním průchodem.

Algoritmus pro generování jedné úrovně pak pracuje následovně:

1. Nejprve se přidá startovní místnost typu 1 nebo 2 do prvního řádku mřížky.

2. Uniformně se vybere číslo od 1 do 5. Pro {1, 2} cesta pokračuje vlevo, pro {3, 4} vpravo a pro {5} dolů. Pokud narazí na konec mřížky, posune se dolů a změní směr vlevo / vpravo.

   - Při pohybu dolů je nutné zajistit průchodnost cesty. Aktuální místnost se musí změnit na typ 2 (průchod dolů) a následující místnost musí mít typ 2 nebo 3 (průchod shora).
   - Vzhledem k tomu, že typ 2 i 3 mají levé i pravé průchody, algoritmus pokračuje dalším uniformním výběrem.

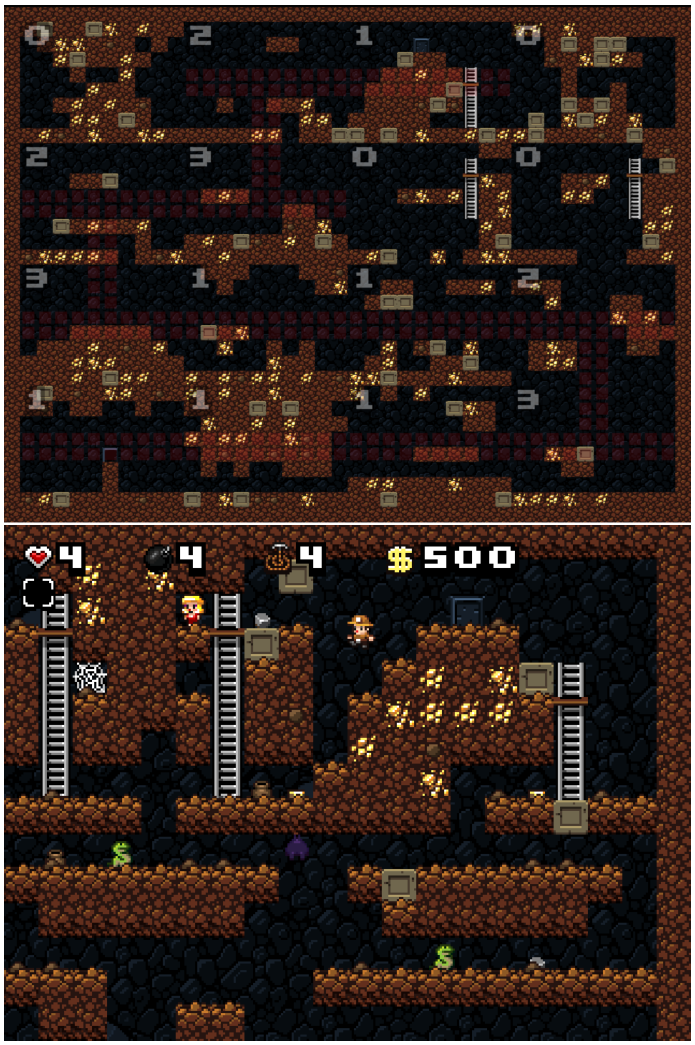
3. Algoritmus končí, když dostane příkaz pohnout se dolů (není kam) – přidá jen závěrečnou místnost a místnosti typu 0 všude, kde není cesta.

Jakmile existuje mřížka místností různých typů, pokračuje návrh úrovně druhou fází (druhá fáze podle [4]), ve které se použijí předpřipravené šablony pro každou místnost. Aby hra obsahovala více náhodných prvků, jsou některé části pevně dané a jiné se objeví s určitou pravděpodobností. Výsledná mřížka místností pak skutečně budí dojem nového levelu.

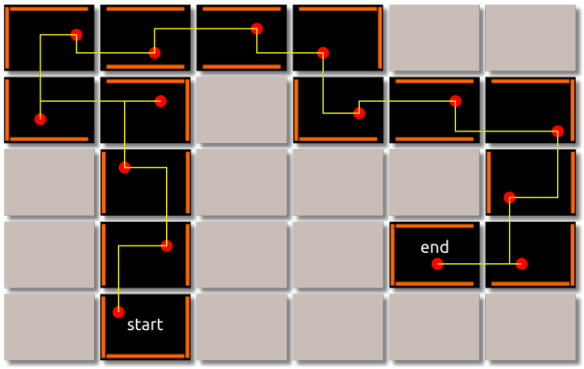
1.3.4 Jack Benoit

Hra *Jack Benoit* vypadá podobně jako *Spelunky* a i podobně generuje prostředí [6]. Generování úrovně opět začíná s mřížkou předem definovaných rozměrů, do které se následně umístí místnosti, které tvoří cestu úrovní. Pro každou místnost se vybere náhodný bod a sousední místnosti se propojí pomocí jedné velké platformy a jednoho žebříku (viz. Obrázek 1.5). Jakmile je zajištěna existence cesty a tedy dosažitelnost cíle, náhodně se přidávají další platformy tak, aby neblokovaly cestu. Následně se na každé platformě náhodně vytvoří počátek žebříku a v poslední kroku se všechny žebříky protáhnou tak, aby dosáhly na platformy pod sebou.

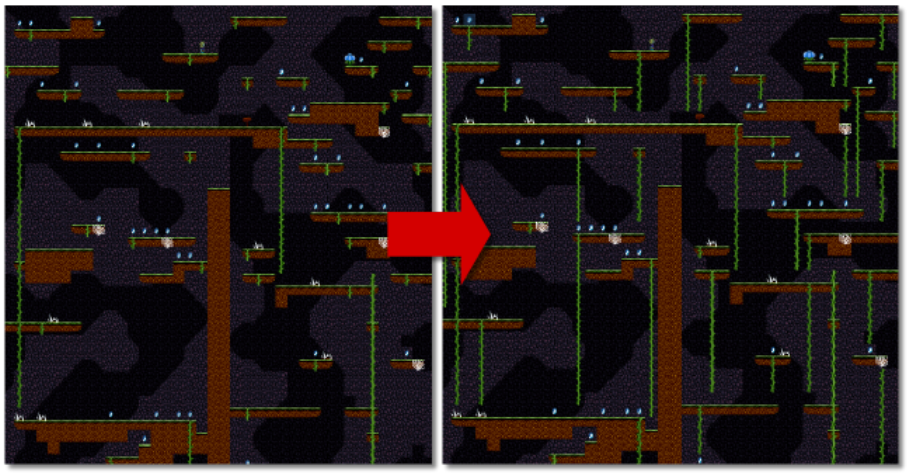
Tento přístup zní velmi jednoduše, ale jak sám autor dokládá přiloženými obrázky (viz. Obrázek 1.6), skutečná hlavní cesta je skryta v množství odboček. Zároveň ale přiznává[6], že přidávání platforem někdy způsobí neprůchodnost levelu.



Obrázek 1.4: Vygenerovaná cesta pro celou úroveň ve Spelunkách [4] a screenshot ze hry [5].



Obrázek 1.5: Vygenerované místnosti a jejich spojení v cestu [6].



Obrázek 1.6: Ukrytí hlavní cesty úrovní ve hře Jack Benoit [6].

1.3.5 Cloudberry Kingdom

V článku [17] J. Fisher, developer v Pwnee Studios, píše o svých zkušenostech z programování level design AI (dále jen LDAI) pro hry typu platformer. Základní principy, jež musí vygenerované úrovně podle Fishera splňovat:

- dají se projít,
- zajímavý design – bude to hráče bavit?
- dostatečná výzva.

Největší problémem uměle generovaných úrovní spočívá v tom, jestli se vůbec dají projít. Jak píše Fisher, pokud se generuje bludiště, které nemá

cestu, stačí bourat zdi, dokud se nějaká cesta nevytvoří. Co ale dělat v případě levelu platformera, který se nedá projít? Oprava takového řešení nemá žádné intuitivní řešení.

Jak tedy vytvořit proveditelný level? Jelikož se jedná o NP-úplný problém[17], je snadné řešení na správnost ověřit, ale je těžké nějaké řešení najít.

V Pwnee Studios si kromě LDAI pomohli Player AI. Každý další jednotlivý blok, který do úrovně dají ověří Player AI – postavička řízená umělou inteligencí, jež má všechny informace o fyzických vlastnostech hráče a dokáže tak simulovat jeho chování. Problém takového řešení je, že se zabývá kontextem celého levelu — pro každou novou část prochází AI celý level od začátku do místa, kam se nový blok přidal. Takový přístup se nedá použít v nekonečných levelech generovaných za běhu z důvodu časové náročnosti.

O druhém principu – zábavnosti se v článku sice píše, ale autor se mu příliš nevěnuje a omezuje se na tvrzení, že nastavení parametrů LDAI, aby úrovně byly dostatečně zábavné je *černá magie*.

Třetí princip, dostatečnou výzvu pro hráče, se autor pokoušel řešit genetickými algoritmy – pomocí evoluce nastavit stovky parametrů LDAI, avšak zjistil, že tento přístup nikam nevede – optimalizace nepřinášela dobré výsledky. Vhodné nastavení parametrů pak po celé tři roky vybírali beta-testeři.

Ačkoliv dva ze tří principů nejsou popsány algoritmicky, tento článek přináší jednu velmi zajímavou myšlenku – stačí změnit fyzická omezení hráče (např. výšku a délku skoků) a úroveň se vygeneruje na míru těmto změnám.

1.3.6 Generátor nekonečného platformera Dylana Wolfa

Generátor[3] vytváří nekonečný sidescroller platformer, který kromě obyčejného lineárního terénu země umí vytvářet i platformy a tvořit tak alternativní cesty.

Wolfův algoritmus je iterativní – vyplňuje mřížku specifické velikosti zdola nahoru:

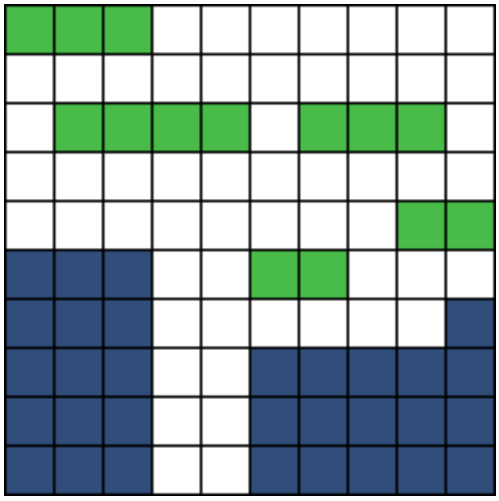
1. Začíná se s prázdnou mřížkou.
2. První vygenerovanou položkou je země – terén se vytváří zcela náhodně, musí jen dodržovat to, aby byl hráč schopen takový terén překonat.
3. V další iteraci se přidávají platformy:
   a) Prochází se mřížka sloupec po sloupci a s pravděpodobností 20 % na tomto místě bude začátek platformy. Poté se mřížka prochází shora dolů dokud nenarazí na první obsazené místo. Finální pozice začátku platformy je pak o dvě nebo tři políčka výše.
   b) Náhodně se zvolí délka platformy a iterativně se přidávají políčka platforem, dokud není celá platforma vygenerovaná nebo dokud

[3]Generátor nemá uvedené žádné jméno

nenarazí na nějakou překážku (pod každou platformou musí být minimálně jeden řádek volného místa pro hráče).

c) Dokud je místo, přidávají se další vrstvy platforem [7].

Výsledek algoritmu je vidět na obrázku 1.7.



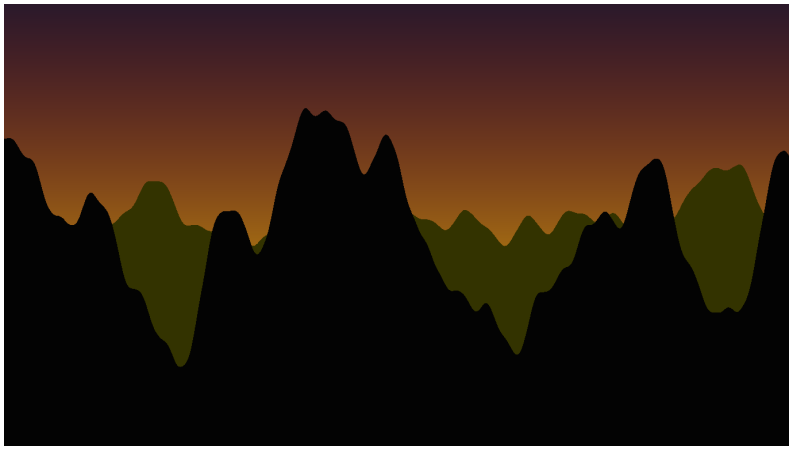
Obrázek 1.7: Výsledek generátoru Dylana Wolfa [7]. V první iteraci se vytvořila země (modrá) a v dalších vrstvy platforem.

1.3.7 Algoritmy generování terénu

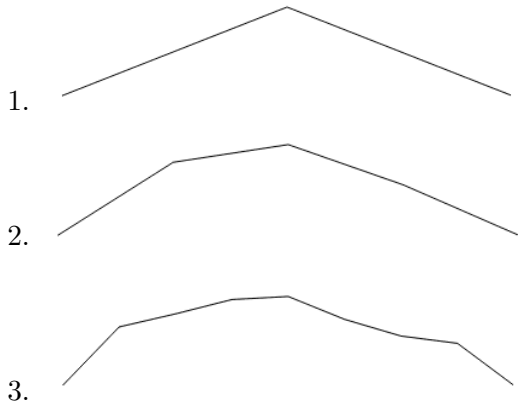
Jak již bylo vidět v sekci 1.3.2, generování jediné vrstvy terénu je v tomto žánru častý jev. Na rozdíl od již diskutovaných řešení, existují kromě náhodného generování výšky sloupců sofistikovanější algoritmy, které zachovávají návaznost a zároveň působí náhodně.

V různých amatérských článcích je často k vidění generování terénu pomocí tzv. Perlinova šumu (nebo vylepšeného, ale pro vyšší dimenze patentovananého Simplexového šumu), který používá skládání nelineárních funkcí [18]. Ačkoliv Ken Perlin vytvořil šum (publikovaný v roce 1985) k syntéze realisticky vypadajících textur kouře, vody, mramoru atd., uplatnění našel i v generování herního terénu[8]. Nemusí jít nutně o 2D terén, který je vidět na obrázku 1.8, Perlinův šum se dá rozšířit do libovolného počtu dimenzí.

Podobného efektu lze docílit algoritmem Midpoint Displacement, nebo jeho vylepšenou verzí Diamond Square. Základní myšlenka Midpoint Displacement algoritmu, jak je popsána v [9], je rekurzivní – začíná se jednou horizontální úsečkou, poté se nalezne střed úsečky (Midpoint), který se v ose Y přesune (Displacement) o náhodnou hodnotu. Rozsah náhodných hodnot se sníží a postup se opakuje pro každou úsečku.



Obrázek 1.8: Použití Perlinova šumu pro generování 2D terénu [8].



Obrázek 1.9: Postupné generování terénu algoritmem Midpoint Displacement [9].

1.4 Učení

Aby se mohlo generování přizpůsobit hráči, musí se naučit, co hráči dělá problémy a takové překážky mu pak s jistou pravděpodobností předkládat. Přístupů k učení a adaptaci herního prostředí na hráče je mnoho. Většina se principiálně nehodí pro okamžitou adaptaci hry, přesto v nich lze najít inspiraci. Například ve hře *Polymorph*[19] je hráči předloženo několik testovacích úrovní a poté je po něm požadováno subjektivní zhodnocení. Tento článek předkládá zajímavé příznaky, které jsou vhodné i pro učení s okamžitou adaptací:

- jak dlouho hráč stál na místě,
- vzdálenost, kterou šel opačným směrem,
- čas dokončení segmentu,
- počet sebraných mincí,
- jestli hráč zemřel, nebo segment dokončil.

Také se nehodí metody, které potřebují sesbírat velké množství dat k přizpůsobení se hráči. Výsledná hra v této práci se má adaptovat co nejdříve – a ušetřit tak hráče potencionálně nudných částí (úrovní), které se neshodují s jeho schopnostmi.

1.4.1 Přístup Daniela Wheata

Ve své dizertační práci [20] z roku 2013 Daniel Wheat zkoumal problém podobný této práci – dynamická adaptace obtížnosti procedurálně generované 2D plošinovky. Ačkoliv generuje jendotlivé úrovně iterativně, jeho přístup se dá zobecnit na jednu nekonečnou úroveň.

Každou úroveň při jeho přístupu tvoří 8 bloků, jejichž terén určuje graf parametrizované lineární nebo kvadratické funkce. Výslednou úroveň tvoří jedna cesta, s jedním druhem bonusu a jedním druhem monstra, což se příliš neliší od řešení diskutovaných výše. Zajímavé jsou ale parametry zmíněných funkcí:

`distanceFactor` – výšky kopců

`timeFactor` – šikmost grafů, čím vyšší číslo, tím strmější kopce, tento faktor také řídí pravděpodobnost na vytvoření platformy, avšak platformy netvoří samostatnou cestu,

`scoreFactor` – počet vygenerovaných bonusů

`challengeFactor` – počet monster

Před samotným učením musí hráč odehrát 5 úrovní, aby se zachytily jeho charakteristiky a vytvořili se podle nich *agenti*, kteří budou napodobovat hráčovo chování:

`random`pause time – čas, kdy se hráč nehýbe,

hurt pause time – čas, kdy se hráč nehýbe po zasažení monstrem,

pause frequency – kolikrát hráč zapauzoval hru

carrots collected – relativní počet sebraných bonusů (mrkví), oproti celkovému počtu v úrovni

response time – relativní počet kolikrát byl hráč zraněn vs. kolik monster bylo v úrovni vygenerováno

spring use – poměr aktivací pružiny a počet vygenerovaných pružin (pružiny jsou na místech, která jsou pro hráče příliš vysoko, aby je překonal obyčejným skokem)

random jumps – počet hráčových skoků−vygenerovaná vzdálenost∗100

Generování dalších úrovní řídí genetický algoritmus[20]. Genotypem algoritmu jsou 4 reálná čísla (`distanceFactor` atd.) – faktory popsané výše. Na počátku se vytvoří populace o 50 jedincích a pro každého jedince je vytvořena úroveň dle daných parametrů. Její průchod je pak simulován pomocí virtuálního agenta vytvořeného na parametrů získaných z analýzy hraní hráče. Fitness funkce jedince je definována jako:

$$\text{fitness} = 1.0 - \text{abs}(\text{Overall}_{\text{Difficulty}} - \text{Target}_{\text{difficulty}})$$

Kde $\text{Overall}_{\text{Difficulty}}$ je aritmetický průměr z relativních hodnot (vygenerované vs. výsledky agenta v intervalu $(0, 1)$): vzdálenost, čas, posbírané bonusy a střetnutí s monstry (počet kousnutí monstrem vs. počet vygenerovaných monster). Zatímco $\text{Target}_{\text{difficulty}}$ byla zjištěna empiricky jako hodnota 0.8 tedy, že úroveň se přizpůsobí schopnostem hráče na 80 %.

1.4.2 Model nezávislých překážek

V článku [21] autoři sestavili statistický model, který počítal všechny jednotlivé překážky (skoky, monstra) jako statisticky nezávislé. Do modelu také zahrnuli možnosti, že hráč bude danou úroveň opakovat při neúspěchu, nebo ji naopak vzdá. Obtížnost úrovně pak vyjadřovala pravděpodobnost hráčova úspěchu.

Jako nedostatek článku lze chápat fakt, že se v něm autoři zabývali hlavně statickými překážkami. Vlastnosti dynamických překážek pak jen simulovali upravením konstant v rovnicích pro pravděpodobnosti skoků – například horizontálně pohybující se platformu simulovali jako dva skoky (na platformu a z ní) plus upravili konstanty podle rychlosti platformy. Zároveň u skoků uvažovali zjednodušené rovnice projektilu pro určení rozpětí chyby (margin of error). Tyto rovnice poté aplikovali na několik levelů hry *Super Mario Bros*, u níž, jak autoři přiznali[21], neznají skutečné vlastnosti skoků.

1.5 Architektura počítačových her

Na závěr rešerše je vhodné krátce prozkoumat možnosti organizace kódu v herních systémech. Podle Roberta Nystroma v [22] proti sobě stojí dvě možnosti, jak organizovat entity v herním systému. První z nich je starší dědičnost, kdy všechny objekty dědí od hlavní třídy a hierarchie je velice hluboká. Takový přístup s sebou nese problém vysoké provázanosti mezi entitami a horší reakci na případné změny.

Druhý přístup upřednostňuje kompozici před dědičností. V systému existují komponenty, které se dají opakovaně používat, měnit a přidávat. Hlavní třída tak většinou neobsahuje více než nějaký identifikátor a kolekci komponent.

Ačkoliv je design založený na komponentách v poslední době používanější v herních systémech[4] [23], má i své nevýhody. Aby spolu komponenty mohly komunikovat, je třeba pečlivě navrhnout, jakým způsobem komunikaci provést. Komunikace přes stavy v hlavní třídě (která obsahuje kolekci komponent) se totiž příliš neliší od dědičnosti (hlavní třída je plná nesouvisejících stavů) a pokud konkrétní objekt nevyužije žádný ze stavů, zbytečně se plýtvá pamětí.

Druhým systémem komunikace může být systém, kde komponenty mají reference na ty, s nimiž potřebují komunikovat. Tím se uvolní místo v hlavní třídě, ale opět se přibližuje k dědičnosti a to vlastností velké provázanosti.

Třetí a nejsložitější systém na implementaci je podle Nystroma [22] systém posílání zpráv. Komponenty na sobě nejsou již tolik závislé – spojují je jen hodnoty (typy) posílaných zpráv a hlavní třída neobsahuje zbytečné stavy.

[4] Například herní engine Unity používá komponenty ve všech objektech herní scény `https://docs.unity3d.com/Manual/UsingComponents.html`. V omezenější míře používá kompozici před dědičností také Unreal Engine 4 `https://docs.unrealengine.com/latest/INT/Programming/UnrealArchitecture/Actors/Components/`

Kapitola 2

Analýza výpočetní inteligence

Výpočetní inteligence je v této práci myšlena jako prostředek procedurálního generování prostředí. Kapitola nejprve diskutuje výběr relevantních funkčních a nefunkčních požadavků, poté se zaměřuje na použití kontextové gramatiky ke generování prostředí včetně monster a bonusů. Následně se zabývá analýzou a definicí možných pravidel a řeší způsoby učení gramatiky k adaptaci obtížnosti pro hráče.

V této části také proběhlo testování na implementovaném prototypu hry, která bylo klíčové pro stanovení a otestování vhodného systému pro učení hry a k nastavení vhodných parametrů učení. Poslední část se krátce zmiňuje o myšlence zakomponování rytmu.

2.1 Funkční a nefunkční požadavky

Požadavky na výpočetní inteligenci:

- Generování světa by mělo být online – čili se obtížnost hry nemění po úrovních, ale co nejdříve. Vliv na následující část světa by tedy měly mít všechny předchozí akce.
- Hra by neměla mít žádné oddělené úrovně ve smyslu – vygenerovat úroveň, nechat ji hráče odehrát a následně vygenerovat další úroveň na základě toho, jak se hráči dařilo. Výsledná hra tedy musí mít jednu „nekonečnou“ úroveň.
- S předchozím požadavkem souvisí, že hra bude zpracovávat všechny akce co nejdříve.
- S online generováním přichází další požadavek, výpočetní inteligence musí poskytnout navazující prostředí na již vygenerované jakmile se hráč dostane na konec – tedy i při velmi rychlém průchodu předchozího vygenerovaného prostředí musí být další část k dispozici, hráč během hry

nesmí čekat na vygenerování (tento požadavek může být částečně zodpovědností prostředí hry, avšak výpočetní inteligence musí jednu obrazovku vygenerovat rychleji než hráč jednu obrazovku stihne projít).

- Ze zadání práce vyplývá, že interně se generování musí řídit nějakou formou kontextové gramatiky.

- Od začátku hry do poslední vygenerované části musí existovat minimálně jedna cesta.

Jak ukázala rešerše (kapitola 1), v nalezených řešení se procedurálně generovaný sidescroller platformer omezoval na jednu *hlavní* lineární cestu. Platformy byly přidávané náhodně a alternativní cesta byla k dispozici jen po krátký úsek.

Dalším požadavkem na výpočetní inteligenci tedy je

- existence alternativních cest,

- cesty se mohou s hlavní cestou prolínat, případně se prolínat i mezi sebou (musí být splněn požadavek existence minimálně jedné cesty v každém okamžiku hry).

V článku [17] Jordan Fisher píše, že se jejich LDAI dokáže přizpůsobit fyzickým vlastnostem hlavní postavy – pokud se např. změní výška skoku nebo samotné postavy, generátor úrovně musí zajistit, že hráč i s takovou postavou může projít do konce. Přirozeným požadavkem této práce tedy je, aby výpočetní inteligence dokázala totéž a navíc uměla vygenerovat úroveň i pokud se změní, přidá či smaže nějaký základní stavební prvek (platforma). Tento nárok se přidá k požadavku o konfigurovatelnosti herního prostředí.

Uživatel si může rozumně upravit velikosti a grafické vyobrazení platforem, přidat další platformy atd. Totéž platí pro bonusové předměty a monstra a samozřejmě jejich vlastnosti. Z čehož vyplývá, že generátor musí být dostatečně obecný, aby takové nároky zvládl zpracovat. Podobné nároky jsou kladeny na formát konfiguračního souboru.

2.2 Kontextová gramatika

Definice 2.2.1. Stochastická kontextová gramatika (Context-Sensitive Grammar) je v [24] definovaná jako pětice $G_s = (N, T, P, P_s, S)$, kde N je abeceda neterminálních symbolů, T je abeceda terminálních symbolů taková, že $N \cap T = \emptyset$. P je množina pravidel ve formátu $\alpha_1 A \alpha_2 \rightarrow \alpha_1 \eta \alpha_2$, kde $A \in N$,$\alpha_1, \alpha_2, \eta \in (N \cup T)$. Jinými slovy, přepsání neterminálu A závisí na jeho kontextu α_1 a α_2. P_s značí rozdělení pravděpodobnosti pro množinu pravidel. S je speciální neterminál vyjadřující startovní symbol.

Volba stochastické verze kontextové gramatiky souvisí s tím, že vygenerované úrovně by měly být pokaždé jiné a právě úpravami rozdělení pravděpodobnosti pro pravidla se bude gramatika *přizpůsobovat* hráči.

2.3 Kontext gramatiky a reprezentace

Protože gramatika v definici 2.2.1 pracuje s neterminálními symboly, které postupně rozvíjí pomocí odvozovacích pravidel, vzniká lineární řetězec terminálů. Takový řetězec ale nekoresponduje s nelineární podstatou 2D hry. Natož libovolným počtem vzájemně se prolínajících cest, jak vyžadují požadavky 2.1. Na druhou stranu definice nezakazuje dvě věci:

1. neterminál může být reprezentován jako 2D mřížka,
2. neterminální symboly gramatiky mohou být externě upraveny jinou instancí gramatiky.

Čímž se dostáváme blíže ke stavu, který je cílem této práce – více gramatik kooperujících na generování různých cest v mřížce. Stochastická verze gramatiky pomáhá řešit kolizní situace, které v takovém případě mohou nastat.

Jelikož se jedná o *nekonečné* online generování, musí být splněny takové vlastnosti, které zaručí, že se budou pravidla odvozovat tak, aby se neterminální symboly derivovaly zleva a generovaly tak terminální symboly, co nejdříve. Pokud totiž hráč dojde na konec aktuálně vygenerované části (jen terminální symboly), výpočetní inteligence by měla ihned navázat a zleva generovat terminály, které se dají vykreslit.

Takovou vlastností podle Shakera[12] (kapitola 5) disponují tzv. L-systémy. Jejich další vlastností je paralelní odvozování pravidel, což v této práci není žádoucí, jelikož gramatika bude pracovat na omezeném prostoru (obrazovka uživatele) a během generování se těmito omezeními musí řídit, což paralelní zpracování implicitně neřeší.

Do definice kontextové gramatiky v této práci tedy přidáme požadavek:

- všechny neterminální symboly se přepisují zleva – dokud je přítomen neterminální symbol, gramatika nesmí odvozovat neterminály v další části.

Externí úprava neterminálů jedné instance gramatiky druhou dává možnost generování do jediné mřížky – vygenerované cesty se budou moci křížit, případně se dočasně slučovat.

2.4 Definice pravidel gramatiky

K definici pravidel si potřebujeme zavést základní stavební prvky, které budou obecně reprezentovat všechny entity ve hře. Nejzákladnější je z počátku

Tabulka 2.1: Významy použitých barev

	terminál platformy
	cokoliv (pravidlo tato pole nezajímají)
	bonus
	monstrum
	neterminál nehmotné entity
	terminál průchozí části
	terminál platformy umístěný v aktuálním kroku do mřížky
	ilustrace neprůchozí části aktuální cesty
	ilustrace pevných entit (platforem) aktuální cesty
	prolnutí průchozích částí aktuální cesty (růžová) a již existující cesty (šedá)
	aktuální požadavek přidání platformy
	teleport

prázdná mřížka představující svět, do něhož se budou generovat cesty. Ostatní entity vyplňující políčka mřížky světa mají tyto vlastnosti:

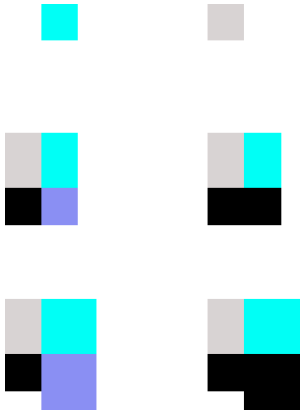
- pevná entita (země, platformy...) – nedají se projít žádným směrem,
- nehmotná prázdná entita (místo pro průchod hráče) – jsou plně průchozí,
- nehmotná plná entita (bonusy) – jsou plně průchozí a dají se případně sebrat.

Do mřížky se generují jak terminály (výše), tak neterminály přepisující se na konkrétní pohyby (chůze, skoky). Pravděpodobnost těchto neterminálů určuje model, jež bude ovlivňován akcemi hráče a aktuální situací v mřížce.

Vzhledem k tomu, že neterminály v tomto případě reprezentují jistou část 2D mřížky sestavující se do většího celku, bude intuitivnější pravidla zavést v jejich grafické podobě, významy jednotlivých barev vysvětluje tabulka 2.1.

2.4.1 Chůze

Aby mohla Karkulka jít rovně, potřebuje k tomu pevnou entitu pod sebou a nehmotnou entitu zajišťující její průchod. Základní tři pravidla jsou znázorněna na Obrázku 2.1, první z nich řeší jednoduchý přepis neterminálu pro nehmotnou prázdnou entitu na terminál. Druhý řádek značí pravidlo chůze – aby mohla být provedena jen chůze, musí být předchozí platforma (černá barva, terminál) průchozí (šedá barva, terminál) až do výšky Karkulky a po celé šířce



Obrázek 2.1: Grafické znázornění základních pravidel (vlevo) a jejich odvození (vpravo). Na prvním řádku je neterminál (tyrkysová), přepsaný na terminál znázorňující nehmotnou prázdnou entitu (šedá). Druhý je neterminál nové platformy (fialový) s kontextem průchozích entit nad ním. Třetí pouze rozšiřuje druhé pravidlo o jinou velikost platformy.

platformy. Pravidlo chůze nelze vygenerovat do vzduchu bez návaznosti, proto jsou terminály obsaženy v pravidle jako povinný kontext odvození.

Tyrkysová barva značí neterminální kontext nehmotné entity a fialová barva nutnost prázdné části mřížky – platformu nelze umístit jinam než na prázdné místo.

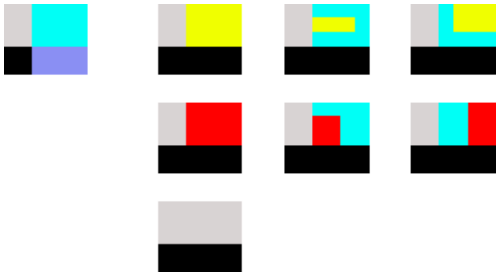
Poslední řádek znázorňuje stejné pravidlo chůze jako druhý řádek, ale upravené pro jinou velikost přidávané platformy. Protože platformy si může uživatel libovolně přidávat, konkrétní podoba pravidel se po načtení konfiguračního souboru přizpůsobí.

2.4.2 Monstra a bonusy

Jelikož gramatika generuje cestu zleva doprava a již se nevrací, je nutné do pravidel pro generování chůze (případně později i skoků) zakomponovat přidávání monster a bonusů.

Stejně jako platformy, i velikosti monster a bonusových předmětů se mohou měnit, tedy gramatika musí svá pravidla upravit natolik, aby takové entity dokázala generovat.

Na obrázku 2.2 jsou vidět odvození, jimiž se generují bonusy (žlutá barva, terminál) a monstra (červená barva, terminál). Objevila se otázka, zda-li monstra vyčlenit jako samotné pravidlo a zajistit tak, že hráč se přes ně dostane, např. pravidlem ošetřit místo na přeskočení. Takové pravidlo by bylo z principu dosažitelnosti konce nutné v případě, kdy by monstrum nebylo odstranitelnou překážkou (zabitelné) – tím by mohl být porušen požadavek na možnost vždy dojít do konce. S tím uváděný model nepočítá, proto zde není nutné kontrolo-



Obrázek 2.2: Grafické znázornění pravidel pro přidání chůze i s dalšími entitami. Vlevo: původní stav s plošinou a prázdným místem. Vpravo: možné další stavy pravidel. První řádek demonstruje přidávání bonusů, druhý řádek monster stojících na platformě, spodní řádek pak rozšíření stávající platformy.

vat výšku oblasti monstra (výška červené) vzhledem k fyzikálním vlastnostem avatara hráče (výška jeho skoku).

2.4.3 Skoky

Skok Karkulky musí pamatovat nejen na její rozměry (výška a šířka), ale také na její fyzické schopnosti – největší možná výška a délka skoku. Komplexnějším problémem k řešení by pak byl model beroucí v potaz další fyzikální veličiny, jako např. zrychlení či tření jako v [17].

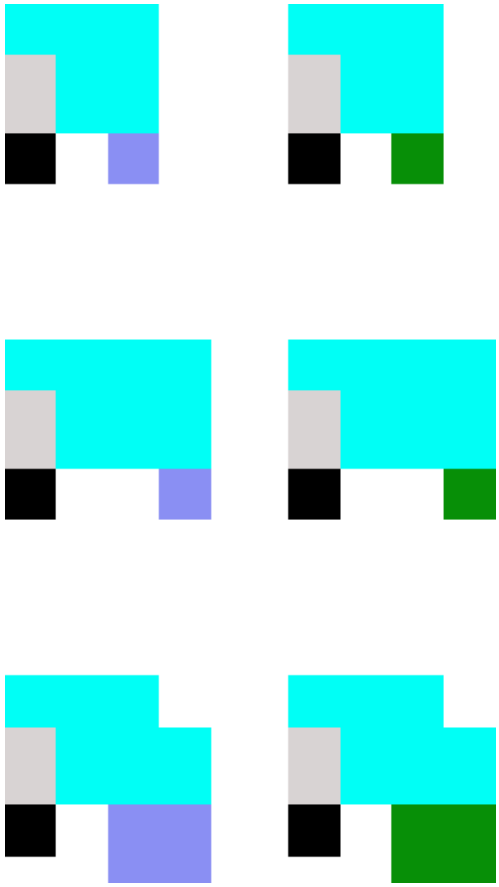
Skoky musí být generovány tak, aby je Karkulka dokázala překonat a aby ostatní cesty skok nenarušily – musí být zajištěn dostatečný prostor pro skok. Zároveň by skok neměl vyžadovat více prostoru než je pro něho nutné a gramatika by neměla generovat skoky menší než je šířka Karkulky – nejednalo by se o skok.

Podle směru byly rozlišeny tři hlavní varianty skoku:

1. skok přes překážku,
2. skok nahoru,
3. skok dolů

Skok přes překážku Pravidla pro skok přes překážku definují skok, kdy je další platforma přidána na souřadnice, jež se liší jen v ose x. Předpokladem (kontextem) takového skoku (viz obrázek 2.3) je existující průchozí (šedá, terminál) platforma (černá, terminál) a neterminální průchozí část (tyrkysová) umožňující skok. V prvním sloupci musí být přidán průchozí neterminál až do maximální výšky skoku nehledě na vzdálenost, kterou postava musí přeskočit (první a druhý řádek obrázku 2.3 zobrazují různě široké skoky).

Třetí řádek obrázku 2.3 zobrazuje pravidlo, které šetří místo potřebné pro skok, pokud je cílová platforma příliš dlouhá. Nad takovou platformu se přidá

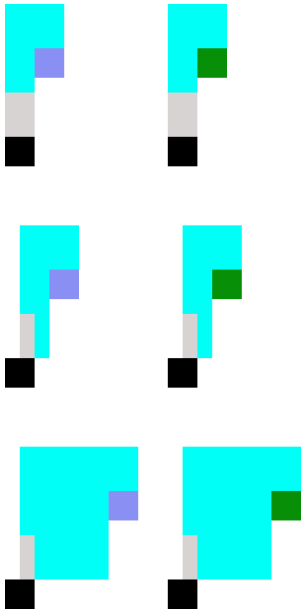


Obrázek 2.3: Grafické znázornění pravidel pro přidání skoku přes překážku.

průchozí neterminální část (tyrkysová) jen do prvního sloupce, zbylou část platformy pokryje průchozí část jen do výšky Karkulky.

Zajímavostí pravidla je, že ignoruje mezeru mezi platformami, kterou je nutné přeskočit. Tato definice napomáhá řešení kolizí a prolínání cest, jak je vidět na obrázku 2.7 a vysvětleno v sekci 2.5.

Skok nahoru Pravidla pro skok nahoru, graficky znázorněná na obrázku 2.4, jsou relativně intuitivní. První řádek obrázku zobrazuje nejtriviálnější skok nahoru – přidá se průchozí část jen do výšky cílové platformy plus výšky Karkulky, pro skok nahoru není nutné přidávat průchozí část až do maximální výšky skoku. Druhý a třetí řádek řeší pravidla, která se liší v šířce skoku. Aby neprůchozí část skoku nezabírala příliš místa, přidává se zleva až na úplný konec – hráči se nechává minimální prostor pro skok. Důvod, proč je ve třetím řádku průchozí část i ve spodní části skoku je ten, že hráč sice bude skákat nahoru, ale nedá se obecně (pro všechny velikosti Karkulky, pro všechny velikosti skoků a funkce řídících skok) říci, jaký prostor bude potřeba.



Obrázek 2.4: Grafické znázornění pravidel pro přidání skoku nahoru.

Skok dolů Pravidla pro jednoduchý skok dolů, znázorněná na obrázku 2.5 kladou na funkci řídící skok jediný nárok – skok nahoru i padání dolů musí být symetrické. Tj. Karkulka dokáže bez vyskočení nahoru seskočit na platformu, která je nejvýše ve vzdálenosti (na ose x) od startovní platformy max_šířka_skoku/2. Taková podmínka umožňuje pravidlu znázorněném na třetím řádku přidat průchozí část jen do spodní části.

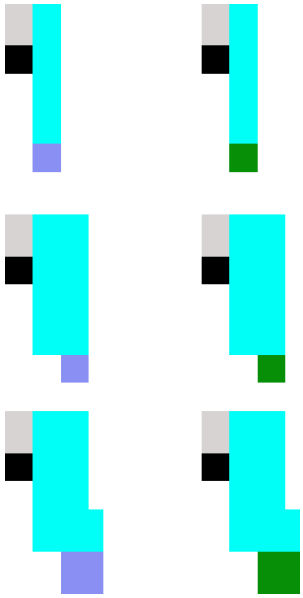
Pokud by skok dolů byl širší než výše uvedená podmínka, Karkulka musí nejprve vyskočit (je rezervována maximální výška skoku), aby šířku skoku překonala. Tato pravidla jsou na obrázku 2.6 (druhý řádek navíc řeší širší platformu). Stejně jako u skoku nahoru, i zde je průchozí prostor rezervován i v dolní části.

2.4.4 Rozlišení země a plošinek

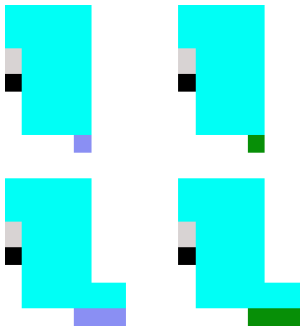
Aby herní obrazovka nepůsobila jako změť klikatících se cest, ale více patřila do žánru, měla by existovat nějaká hlavní cesta (země) oddělená typově i graficky. Pro cestu po zemi v této práci by mělo platit, že nebude obsahovat jiné skoky než přes překážku.

2.5 Řešení prolínání gramatik

Pravidla tak jak jsou nyní definována řeší prolínání cest částečně implicitně, konkrétní příklad lze vidět na obrázku 2.7. Již vygenerovaná černá cesta s de-



Obrázek 2.5: Grafické znázornění pravidel pro přidání skoku dolů.



Obrázek 2.6: Grafické znázornění pravidel pro přidání širokého skoku dolů.

finovanou průchodovou (šedou) částí a aktuálně generovaná růžová cesta se krátce prolnou. Generátor pro novou cestu vyhodnotil, že nastane skok dolů a umístil platformu do mezery mezi platformami vygenerované cesty – pravidla na skok přes překážku nekladou na mezeru žádné podmínky, tedy se tváří jako prázdná a využitelná pro jinou cestu.

2.5.1 Explicitní prolínání

Pravidla ale neumí implicitně řešit kolizi cest, která nastala na obrázku 2.8 vlevo. Stejně jako v předchozím případě, již v mřížce existuje vygenerovaná černá cesta, červená linka nahoře zobrazuje strop obrazovky (přes strop se generovat nesmí). A růžová cesta došla do stavu, kdy nemůže přidat žádný

Obrázek 2.7: Implicitní prolínání dvou cest

Obrázek 2.8: Explicitní prolínání dvou cest

neterminál a rozvinout jej ve vlastní akci. V mřížce je sice místo na hráče, ale už se pod něj nevejde platforma. Taková situace značí, že slovo do formálního jazyka popsaného gramatikou nepatří.

Možností, jak takovou situaci řešit prakticky, může být více, například backtracking – přejít o krok zpět a vydat se jinudy. Pro generátor, jež běží online může být backtracking důvod k zamrznutí aplikace (byť jen krátkodobému), což nikdy není pro uživatele příjemné. Případně se také může stát, že bude potřeba odmazat část řetězce, který už je vykreslený a hráč jej vidí. Pravděpodobnost takového stavu je vysoká, protože z požadavků vyplývá, že generátor má na hráčovo chování reagovat, co nejdříve, tedy část světa vygenerovaná napřed nebude příliš velká.

Dále se dá uvažovat, že pokud si cesty vzájemně překáží, řešením může být vymezit každé cestě prostor, na kterém se může generovat – v našem případě dvou cest tedy rozdělit obrazovku horizontálně na dvě části. Takové řešení ztrácí vlastnost, že jedna gramatika funguje jako kontext pro druhou a vzájemně se prolínají.

Nasnadě je prolnutí cest jiným způsobem než v prvním případě – gramatika generující růžovou cestu nalezne nejbližší platformu a zkusí na ni aplikovat některé z pravidel. Jak je vidět na prostřední části obrázku, lze aplikovat pravidlo seskoku dolů. Gramatika do mřížky vygeneruje terminály tak, jakoby platforma seskoku patřila k ní a poté pokračuje dál v generování – třetí část obrázku.

2.5.2 Neřešitelné prolnutí cest

Existují i situace, kdy dvě cesty nelze prolnout a aktuálně generovaná cesta nemá kudy pokračovat. Taková situace je znázorněna na horní části obrázku

Obrázek 2.9: Možné řešení kolize přidáním okrajů

Obrázek 2.10: Přidání teleportu

2.9 (červená značí hranice mřížky, obrazovky, či herního světa). I kdyby byl povolen backtracking (a nějak vyřešeno, aby se nešlo zpět až do vykreslené části), tak se může stát, že již existující černá cesta bude blokovat cestu, např. začne nahoře a vždy se vygeneruje skok o 1 dolů až k hranici, ani s takovou cestou se nelze prolnout.

Jedním řešením je zúžení prostoru, na kterém se může černá cesta generovat (světle červené hranice v dolní části obrázku 2.9). Aktuálně generovaná cesta (růžová) má pak vždy místo, kam generovat a po nějaké době najít vhodné místo na prolnutí.

Takové řešení vypadá dobře, je funkční, ale málo obecné. Co se stane, pokud budeme chtít přidat třetí cestu? Generování třetí cesty může uváznout v sevření dvou předchozích vygenerovaných cest. Zde jsou pomocné okraje k ničemu.

Vzhledem k tomu, že generátor slouží k vytváření herního prostředí, dá se uvažovat o jednodušším řešení, tedy o nespojité cestě. V takovém případě, pokud neexistuje žádná možná cesta, je vytvořeno pokračování cesty jež není přímo návazné (kouzelná brána, teleport atd.), viz obrázek 2.10.

2.6 Učení hráčova chování

Rešerše ukázala, že faktorů, jak charakterizovat jednotlivé hráče může být hodně. Jelikož se tato práce v některých podstatných vlastnostech liší, nelze

tyto faktory převzít a vyzkoušet je všechny. Dalším problémem může být jiný generovací algoritmus – většina nalezených prací použila nějakou verzi genetického algoritmu, zde se používá stochastický model vystavěný nad kontextovou gramatikou.

2.6.1 Faktory k měření

Některé faktory charakterizující hráče byly převzaty z rešerše a jejich smysluplnost byla testována na uživatelích. Testování probíhalo na funkční hře, jejíž obsah byl přidán staticky – ručně (platformy) nebo zcela náhodně (bonusy, monstra). Hráči byli požádáni, aby všechna svá rozhodnutí a jiné myšlenky o hře říkali nahlas.

Jelikož hra neobsahuje mince, či jiné předměty, jejichž sbíráním se zvyšuje skóre, motivace hráče spočívá v tom jak daleko se ve hře dostane, přičemž veškeré sbírání předmětů je jen prostředkem zjednodušení si (nebo umožnění) cesty dál a tady dosažení vyššího skóre.

Naopak se zde přidává faktor času – čím rychleji hráč překoná nějaký úsek, tím snazší pro něho byl. Jelikož nelze počítat čas po úrovních (existuje jen jedna), může se měření provádět po segmentech, například pro každou vygenerovanou obrazovku. Testování ukázalo, že faktor času zásadním způsobem korespondoval se subjektivní obtížností segmentu.

Dále se ukázalo, že pro různé hráče jsou obtížné jiné typy monster. Někomu dělal největší problémy kouzelník, jelikož se dá zabít jen bojem zblízka. Jinému vadil rychle se pohybující skřet a někoho ke smrtelnému propadu zemí dohnal vlk, který ho pronásledoval. Otázkou bylo, jaká interakce s monstrem se dá považovat za úspěch? Zabití monstra? Přeskočení monstra? Obojí značí úspěšné vyhnutí se monstru, neúspěchem tedy je zranění od monstra – další vhodný faktor k učení modelu.

Článek rozebraný v sekci 1.4.2 řešil skoky nikoliv jako čím delší tím těžší, ale náročnější skoky jsou takové, které se více odchylují od maximálního skoku. Jistě by bylo zajímavé mít nějaký faktor, který by určoval, jaké skoky hráči dělají potíže. Dalo by se říci, že skok, kterým hráč propadne je pro něho obtížný. Toto tvrzení bylo vyvráceno při testování. Často hráči seskakovali mezi cestami, aby se vyhnuli monstrům, dostali se k bonusům, nebo jim přišlo, že nižší cesta je snazší. Ani propady zemí nebyly nutně vinou konkrétních skoků, nýbrž kombinací několika jiných faktorů (pronásleduje mě monstrum, chtěl jsem seskočit na nižší cestu atd.). Nebyl nalezen jasný způsob, jak u hráče měřit neúspěch skoků.

Čas, kdy je hra pozastavena se též nedal použít, jelikož žádný z testerů hru nikdy nepozastavil kvůli tomu, aby si rozmyslel svůj další krok. Pauza byla použita jen, když chtěli upozornit na nějaký problém, tedy o klávese pauzy věděli.

Velmi podobně dopadla chůze doleva (hra se prochází zleva doprava), ve většině případů hráč v takový moment utíkal od monstra, které ho zra-

nilo.

Existence více cest znemožnila použití relativních koeficientů, například nelze použít poměr sebraných vs. vygenerovaných léčivých předmětů, počet provedených skoků a počet vygenerovaných skoků atd. Při jediné cestě by se dalo pozorovat, která posloupnost událostí měla na hráče negativní vliv. Teoreticky by se dalo uvažovat sledování hráče, který by v každý okamžik měl přiřazenou cestu. Problémem takové úvahy jsou pohybující se entity – monstrum patří na cestu A, ale došlo až k hráči na cestu B, ke které cestě patří? Podobně na tom jsou místa, kde se cesty prolínají – tento skok na cestě A byl extra jednoduchý, ale kvůli tomu, že cesta B vyplnila jeho mezeru. Analýza těchto možností by byla vhodným rozšířením této práce.

Z předchozích odstavců vyplývá, že hra musí generátoru posílat následující informace:

- zranění od monstra (okamžitě)
- čas, za který hráč prošel obrazovku

2.6.2 Neustálé přizpůsobování hráči

Výhodou hry v této práci je její rychlé přizpůsobení se hráči. Aby se mohla přizpůsobit, od prvních obrazovek už se musí učit. *Učit se* v tomto případě znamená upravovat pravděpodobnosti pravidel gramatiky. Jelikož gramatika generuje velmi odlišné entity (např. skoky a bonusy), pravděpodobnosti pro tyto entity se v gramatice učí jinými způsoby, které co nejvíce reflektují doménu entity.

2.6.2.1 Monstra

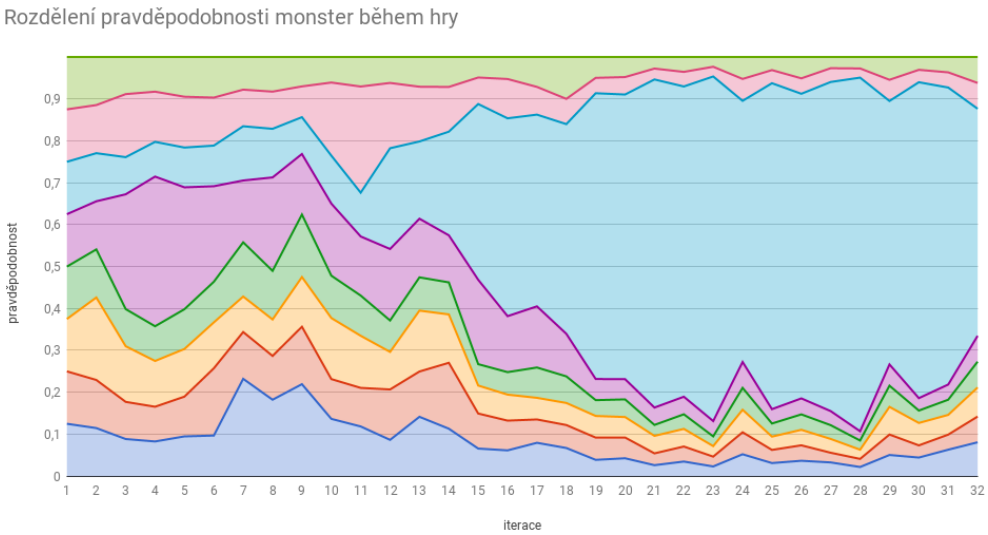
Všechna monstra začínají se stejnou pravděpodobností[5]. Ačkoliv monstra mají vlastní faktor, z něhož se přímo učí (zranění od monstra), učící algoritmus nespočívá v přímočarém zvyšování pravděpodobnosti daného monstra.

Pokud by každé kousnutí monstra M znamenalo zvýšení jeho pravděpodobnosti bez korekce (počítáme ale s normalizací, tj. pravděpodobnost všech monster dohromady je 100 %), po prvních obrazovkách by monstrum M zaplavilo všechny následující cesty. Pravděpodobnost vytvoření jiných monster by klesala k nule, čímž by se umocnila dominance monstra M a byla by naprosto ztracena diverzita. Hra by byla monotónnější a žádné jiné monstrum by nemělo šanci hráče potrápit. Výsledný algoritmus tedy musí brát v úvahu iterativní učení, zabránění přeučení a zachování diverzity všech monster.

V každé iteraci bude pravděpodobnost monstra před normalizací M následovná:

$$p_M = p_M * 2^{\min\{2,b\}} + t$$

[5]Případně se dá v některé z dalších verzí hry přidat příznak, který bude značit náročnější, vzácné monstrum.



Obrázek 2.11: Graf rozdělení pravděpodobnosti monster při zachování diverzity

kde b je počet, kolikrát byl hráč zraněn monstrem a t reprezentuje práh, který pomáhá zabezpečit diverzitu monster. Jeho hodnota je závislá na celkovém počtu monster n a spočítá se jako $t = n * 0.4$. Při $n = 8$ je $t = 5$ %.

2.6.2.2 Čas

Čas je další položka, která má vlastní učící faktor. Otázkou bylo, jak čas průchodu obrazovky interpretovat. Výsledný model pracuje s průměrným časem – podle času se řídí obecná obtížnost, která je odstupňována od 1 (nejsnazší) po 5 (nejtěžší). Obtížnost 0 reprezentuje speciální obtížnost určenou pouze pro začátek hry. Další mezistupně je možné přidat (mít nejtěžší např. 10), generování pracuje pouze s krajními hodnotami.

Obtížnosti řídí délku chůze – s vyšší obtížností bude platforem tvořící nepřerušenou linku méně. Řídí také poměr monster a bonusů, čím těžší úroveň, tím více monster a méně bonusů, na nejtěžší úrovni se bonus nevygeneruje žádný. Vychází se z jednoduchých rovnic:

$$\text{monsters} = \left\lfloor \frac{\text{current}_{\text{difficulty}}}{\text{max}_{\text{difficulty}}} \right\rfloor$$

$$\text{bonuses} = 1 - \left\lfloor \frac{\text{current}_{\text{difficulty}}}{\text{max}_{\text{difficulty}}} \right\rfloor$$

Podle těchto pravděpodobností se na platformy přidávají monstra a bonusy. Jelikož se jedná o iterativní generování (cesta dopředu neví bude-li mít

dostatek místa vygenerovat celou chůzi) Pokud se např. vygeneruje monstrum, jeho pravděpodobnost se sníží, v dalším kroku může mít vyšší šanci bonus.

Pokud hráč prošel obrazovku stejně rychle jako je jeho průměr, úroveň se zvýší stejně jako kdyby ji prošel o něco rychleji. Čímž se pomalu zvyšuje obtížnost hry. Zde se ukázala výhoda použití průměru – průměr vyhladí extrémy a zabrání hráčům podvádět. Pokud by hráč prošel jednu náročnou obrazovku a potřeboval více léčivých předmětů, stačilo aby před koncem jedné obrazovky chvíli počkat a obtížnost by se razantně snížila – vygenerovalo by se více bonusů a méně monster.

2.6.2.3 Skoky

Vzhledem k tomu, že skoky nemají vlastní učící faktor, bylo vhodné nějak aplikovat faktor času na pravděpodobnosti jednotlivých skoků. Dá se předpokládat, že čas, za který hráč projde obrazovku ovlivnily značnou měrou nejen monstra, ale také různé vzdálenosti skoků. Proto v momentě, kdy se úroveň obtížnosti ztíží, za daný úsek se také zvýší pravděpodobnost zobrazeným skokům.

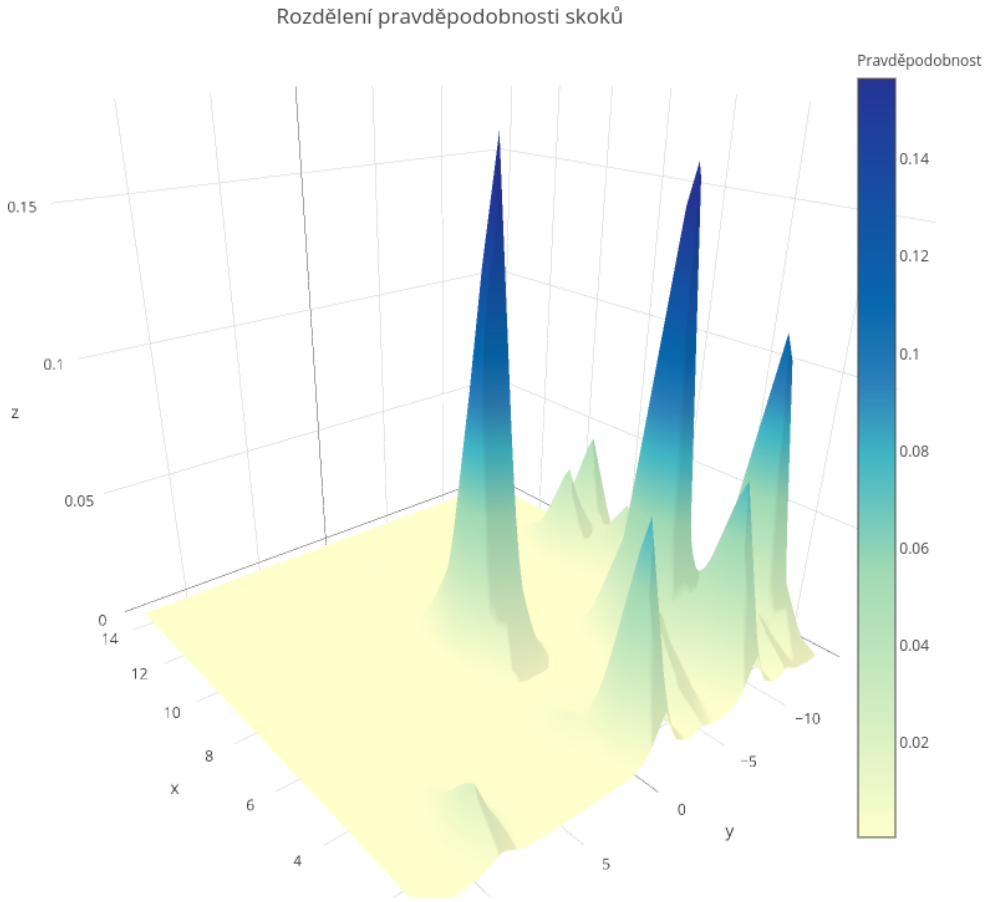
Skoky představují matici vzdáleností J, prvek $J_{(3,-4)}$ značí skok 3 jednotky vpravo a 4 jednotky dolů. Při upravě pravidel pak nedochází jen ke zvýšení četnosti skoku těchto přesných rozměrů, ale v určité míře (dané rozměrem konvolučního gaussovského jádra) i skokům rozměrů velmi podobných. Matice pravděpodobností vygenerování skoku daných rozměrů po několika iteracích učení je pak vidět na obrázku Obrázku 2.12, vysoké hodnoty značí rozměry skoků, jež byly přítomny na obrazovce při úmrtí Karkulky během testování, přičemž vyšší hodnoty reprezentují opakovaný a tedy znatelně obtížnější jev. Je pravděpodobné, že skok, jež je hodnocen jako obtížný bude obtížný i při drobné změně velikosti. Zvýšení pravděpodobnosti obdobných kroků tak simuluje vícenásobné selhání hráče a snižuje tak čas potřebný k naučení modelu.

To, že se pravděpodobnosti zvyšují všem skokům v úseku, nemusí nutně znamenat problém. Stejně jako u monster i zde se zachovává jistá diverzita a dá se očekávat, že problematické skoky budou mít stále vyšší pravděpodobnost – úseky v nichž se objeví budou pro hráče časově náročnější a tedy opět zvýší svou pravděpodobnost, ale tentokrát s jinými velikostmi skoků.

2.7 Herní rytmus

Podle autorů článku [25] je v plošinkovkách důležitý rytmus – opakování správně načasovaných pohybů, které hráč musí vykonat, aby překonal jistou sadu překážek. Bylo by vhodné rytmus v nějakém smyslu do generátoru také zakomponovat.

Vyžadování toho, aby byly veškeré generované prvky součástí definovaného rytmu (například střídající se rytmus úseků chůze a skoku) může působit vysoké množství kolizí především v případě prolínání několika cest. Tuto mož-



Obrázek 2.12: Graf rozdělení pravděpodobnosti skoků

nost je přesto možné využit. Rytmus muže být generován pouze v případě, kdy na něj bude v mřížce místo.

Kapitola 3

Analýza a návrh hry

Kapitola se zabývá analýzou a návrhem herního prostředí (bez výpočetní inteligence), na němž by se pozdější generování mělo testovat. Návrh by měl být obecně nezávislý na použité technologii, nicméně kvůli použití velkého množství Qt modulů je lepší držet se principů použitých ve frameworku Qt, jelikož jejich nedodržení by v implementaci nutilo k nepřehledným a zbytečně složitým řešením.

3.1 Funkční a nefunkční požadavky

Hra, stejně jako její vývoj, by měla být rozdělena na dva hlavní celky, které si budou pouze vyměňovat informace. Jedná se o část hry jako takové (postavu, prostředí, zvuky...) a část výpočetní inteligence, tj. gramatika její pravidla, úprava používání pravidel a reakce na hráčovo rozhodování. Tyto celky by na sobě neměly být závislé více, než že hra předá informace o hráčových akcích a jejich důsledku (např. hráč byl zraněn monstrem X) a výpočetní inteligence dodá prostředí navazující na již vygenerované.

Požadavky na hru nejsou nijak vysoké, jelikož hra slouží pouze jako prototyp, na němž se testuje výpočetní inteligence. Na druhou stranu by takový prototyp měl být snadno dopracovatelný na plnohodnotnou hru.

Požadavky na obsah hry:

- 2D svět,
- žánr platformer sidescroller – plošinky různých výšek, pokud hráč propadne „zemí“, zemře,
- několik typů monster ovládaných jednoduchou umělou inteligencí,
- nebezpečné předměty / překážky
- léčivé předměty,

- munice (sbírání)

Nefunkční požadavky na hru

- ovládání přes klávesnici, do budoucna rozšiřitelné o jiný druh ovládání (konkrétně např. Leap Motion[6]),
- kamera by měla hráči vždy umožnit, aby viděl, kam skáče,
- kamera by se neměla hýbat po více než jedné ose, pokud hráč zůstane po celou dobu pohybu viditelný,
- hlavní postavu nesmí zranit něco, co nevidí (např. monstrum o obrazovku dál než hráč),
- hlavní postava musí mít čas zareagovat, než dojde k útoku na ni,
- nesmí se čekat na dokončení animace spritu, aby mohla být provedena další akce,
- pozadí hry musí jít jednoznačně rozlišit od aktivních prvků,
- kolize a veškeré interakce by měly korespondovat s grafickým vyobrazením,
- herní prvky by se měly dát konfigurovat beze změny zdrojového kódu (např. výše zranění způsobené monstrem, počet životů, atp.).

Jelikož akce hráče jsou to, čím hráč komunikuje s hrou, je důležité, aby všechny akce měly smysl a hra k nim byla od začátku uzpůsobena. Např. nemá smysl stavět doskočitelné platformy, pokud postava dokáže létat atd. Sestavení těchto požadavků klade konkrétní podmínky na grafickou složku, definovaná pravidla a další vlastnosti hry.

Požadavky na akce hlavní postavy:

- konstantní chůze – s možností přidání zrychlení v některé další verzi,
- skok (nahoru, do stran) – s variabilní výškou skoku, dle toho, jak dlouho hráč drží klávesu skoku,
- přikrčení,
- útok na blízko,
- útok na dálku.

[6] `https://www.leapmotion.com/`

3.2 Průběh hry

Před samtným návrhem je potřeba stanovit, jak bude hra probíhat a vymezit nějaké její obecné vlastnosti.

- Hra by měla obsahovat nějakou úvodní obrazovku, která se zobrazuje jen po nezbytně nutnou dobu a poté by měl následovat interaktivní herní svět.
- Svět se skládá z jediné nekonečné úrovně, kterou hráč prochází – iterativně probíhá učení a generování prostředí.
- Jakmile hráč přijde o všechny životy, má možnost ve hře pokračovat za penalizaci poloviny skóre. Hra zůstává naučená.
- Kdykoliv během hry má hráč možnost hru pozastavit nebo ukončit.

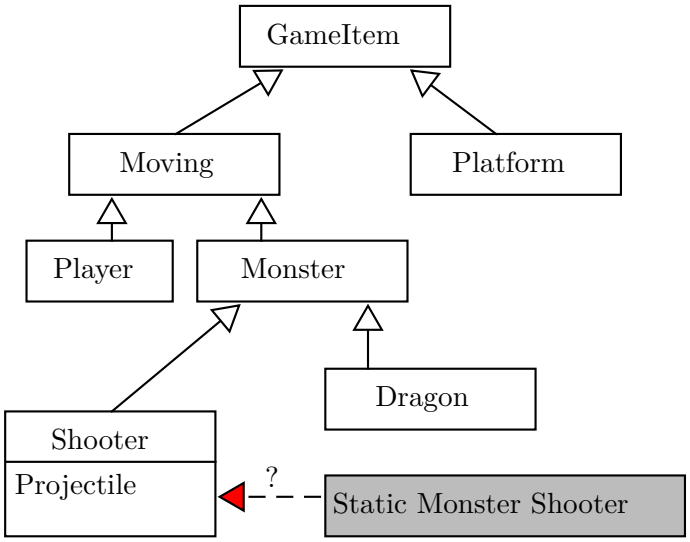
3.3 Architektura hry

Hra je postavena především na architektuře ECS (Entity-component-system), která je pro vývoj her v poslední době poměrně preferován[22] a dopad jejích pozitivních vlastností je stále předmětem výzkumu[26]. ECS se striktně drží principu preference kompozice před dědičností.

Dědičnost se v herní architektuře příliš nevyplatí, jelikož není nakloněna ke změnám. Jako příklad se můžeme podívat na obrázek 3.1. Veškeré herní entity dědí od obecné třídy `GameItem`, všechny pohybující se entity zase dědí od `Moving` (případně implementují rozhraní `Movable`). Doposud byla všechna monstra, včetně shootera, pohybující se entity. Pokud bychom chtěli přidat monstrum, které střílí a nehýbe se, máme s takovou architekturou problém. Nebo pokud bychom chtěli, aby `Player` uměl střílet – všechna logika, která řeší vystřelení nějakého projektilu je zanořena v jiné třídě, kam ji dát? Bylo by nutné významnou část architektury přepracovat.

ECS se tedy jeví jako výrazně výhodnější systém, jak uspořádat třídy. Základem ECS je stále nějaký hlavní `GameItem`, od něhož se ale další třídy neodvozují – `GameItem` slouží především jako kolekce komponent. Každá komponenta přidává nějakou funkcionalitu, speciální vlastnosti, nebo nové chování. V tomto typu architektury tedy neexistuje samostatná třída `Player` nebo `Monster`. Postava hráče je definována použitými komponentami, například komponenty: vstupu z klávesnice, přehrávání zvuku, života, animovaného spritu, povolených akcí, atd. Jak by vypadalo střílející monstrum sestavené z komponent lze vidět na obrázku 3.2. Pokud bychom chtěli monstrum upravit, aby se nehýbalo, jednoduše stačí odstranit `MovingComponent`.

Tato architektura pomáhá naplnit funkční požadavek na externí konfigurovatelnost hry. Pokud bychom v externím (textovém) souboru měli definované



Obrázek 3.1: Zásadní nevýhoda použití dědičnosti – provedení změn je složité a pracné.

Shooter Monster

GameItem
ID GraphicsComponent MovingComponent ShooterComponent DamageOnTouchComponent LivingComponent

Obrázek 3.2: Shooter Monster v architektuře ECS.

herní entity jako kolekce komponent s různými parametry, je velice snadné provádět rozsáhlé změny bez zásahu do zdrojového kódu.

3.3.1 Rozdělení zodpovědností

Na začátku návrhu je potřeba vymezit základní oblasti a těm předat zodpovědnost za určité části programu.

- Průběh hry – celek, který by měl řídit celou hru, její vnitřní stavy a komunikaci s uživatelem.
- Oddělená by měla být grafická složka, zobrazování herního okna, vykreslování scén.

- Ze zvolené architektury ECS vyplývá existence nějaké třídy (zde pojmenovaná jako `GameItem`), která bude obsahovat kolekci komponent reprezentující objekty ve hře. Tato třída bude zároveň sloužit jako směrovač komunikace (zpráv) mezi jednotlivými komponentami – změna rozhraní nějaké komponenty by neměla ovlivnit jinou, ale pouze třídu `GameItem`.

- Objekty ve hře existují v nějakém světě, kde spolu interagují. Další položkou je tedy správa světa.

- Správa zvukových a grafických zdrojů by taktéž měla být oddělena.

Zodpovědnost za to, aby hráč nemusel čekat na vygenerované řešení přebírá hra. Jejím úkolem bude po generátoru vyžadovat dostatečné množství obrazovek a to s předstihem vykreslit do scény.

3.4 Implementační jazyk a knihovny

Jako implementační jazyk byl zadán C++. Použití frameworku Qt bylo zvoleno. Qt je rozsáhlý framework, který podporuje všechny oblasti potřebné pro implementaci hry.

- Grafické rozhraní – Graphics View Framework

- Paralelní komunikace mezi vlákny – Qt Event System

- Posílání zpráv mezi objekty – Meta-Object System

- Práce s obrázky – třídy QPixmap a QImage

- Přehrávání zvuků – Qt Multimedia

- Vstup z klávesnice – Qt Virtual Keyboard

- Spousta vlastních kontejnerů kompatibilních s ostatními Qt objekty – QVector, QList, QHash...

Obsáhlý seznam důvodů, proč je Qt vhodné pro implementaci hry je v [27]. Dalším argumentem je herní engine V-Play vystavěný právě nad frameworkem Qt [28].

Použití dalších knihoven by nemělo být nutné, fyzika hráče i monster ve 2D platformeru se bude řešit jednoduše prací s obdélníky různých velikostí. Na nevhodnosti použití frameworku na fyzikální výpočty ve 2D platformeru se shoduje několik zdrojů[29, 30].

3.5 Platformy

Vzhledem k požadavku ovládání přes klávesnici se výběr platforem velmi zužuje. Zároveň není předem jasné, jakou výpočetní sílu bude potřebovat zpracovávání a generování prostředí, tedy předpokládanou platformou je PC.

Hra se bude vyvíjet pod operačním systémem Linux (konkrétně distribuce Debian Jessie) a na tomto systému proběhne i testování. Ačkoliv použitý framework to umožňuje, účelem této práce není vytvořit multiplatformní aplikaci, ale funkční prototyp.

3.6 Herní smyčka nebo události

Psát hru ve frameworku, který byl vytvořen pro vývoj aplikací s grafickým rozhraním, nikoliv pro hry, s sebou nese úskalí s tradičním přístupem k aktualizaci objektů. V Qt jednotlivé třídy pomocí systému signal-slot reagují na události, čili nejsou aktualizovány periodicky.

Periodická aktualizace, neboli v tomto kontextu herní smyčka, je přístup používaný v herních systémech[22], kdy se provádí v cyklu následující příkazy:

1. zpracuj vstup,
2. aktualizuj vnitřní stavy,
3. vykresli výsledek.

Výsledný přístup kombinuje oba přístupy, svět a objekty v něm se periodicky aktualizují v herní smyčce. Zatímco rendering a zpracování uživatelských vstupů řeší Qt ve svém vlastním event loopu. Zjednodušeně řečeno vše, co řeší framework pracuje s událostmi, kdežto jádro hry pracuje periodicky.

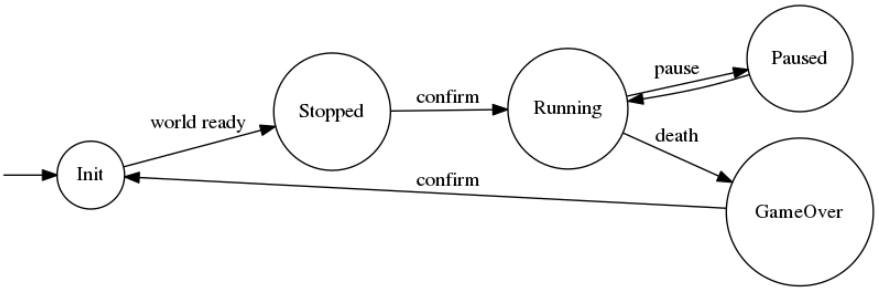
3.7 Hra

Průběh hry má na starosti jediná třída jménem `Game`, diagram na obrázku 3.4. Obsahuje základní herní stavy (viz automat na obrázku 3.3), podle nichž se řídí, co uživatel vidí na monitoru (samotnou hru, úvodní obrazovku…).

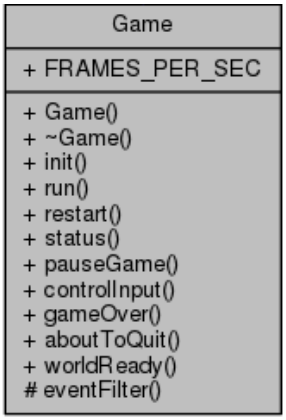
Zde se volá periodická aktualizace herního světa (který ji propaguje na obsažené objekty). Třída se také stará o iniciální načtení konfiguračního souboru.

3.8 Grafika

Grafika byla rozdělena na dvě třídy, z nichž `Graphics`se stará o zobrazení maximalizovaného herního okna na obrazovce a škálování svého obsahu. Na škálování spoléhá zbytek systému, který se tak nemusí starat o pozice entit v různých rozlišení.



Obrázek 3.3: Stavy hry



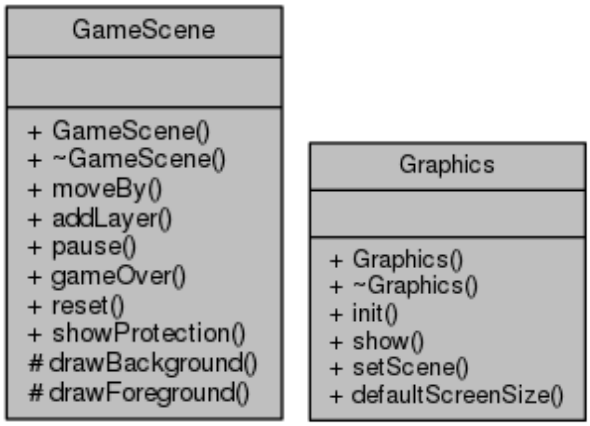
Obrázek 3.4: Diagram třídy Game.

Druhou třídou je `GameScene`, která vykresluje pozadí (viz níže) a informace o hře, jmenovitě zdraví hráče, skóre a počet dostupné munice. Scéna nevykresluje herní objekty, o to se starají jejich grafické komponenty.

Parallax pozadí Pozadí hry je tvořeno vrstvami, které se pohybují relativně k pohybu kamery. Aby pozadí budilo dojem hloubky, pohybuje se každá vrstva jinak rychle. Vrstvy v popředí mají relativní koeficient vyšší (pohybují se skoro stejně rychle jako kamera) než vrstvy vzadu. O správu vrstev a jejich pohyb se stará třída herní scény.

3.9 Svět a propojení interakcí

Všechny herní objekty (platformy, hráč, monstra atd.) existují a interagují ve světě hry, jenž funguje jako propojovací místo ostatních tříd. Zde probíhá hlavní komunikace mezi herními objekty (`GameItem`) a také se herní objekty dotazují této třídy na aktuální stav světa, například stojí-li objekt na pevné zemi. Akce, které ovlivňují jiné objekty ve světě než ten, který jej vyvolal také



Obrázek 3.5: Diagramy tříd GameScene a Graphics.

řeší třída světa (`World`, diagram na obrázku 3.6) – například pokud hráč útočí na blízko, třída zajistí, že všechny živé objekty v dosahu obdrží zranění.

Mezi další zodpovědnosti třídy `World` patří přidávání projektilů do scény a pokud obsahují komponentu *Nebezpečí na dotek* (viz sekci 3.12.5), nastaví jí vlastníka.

Svět je třídou `Game` periodicky aktualizován – což v důsledku znamená aktualizaci objektů ve světě. Grafickým znázorněním světa je scéna (`GameScene`), do níž se jednotlivé objekty vykreslují.

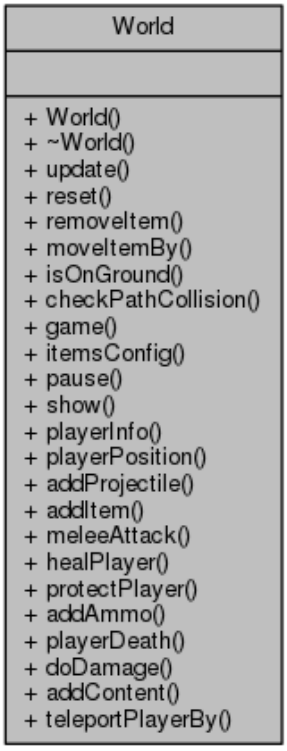
Jelikož třída `World` obsahuje všechny herní objekty a informace o jejich pozici, je vhodné, aby hlídala, kdy bude potřeba po generátoru vyžadovat další část herního světa. Pouze tato třída tedy bude komunikovat s generátorem – předávat mu informace k učení a získávat od něho vygenerovaný výsledek.

Pohyblivá kamera byla, jak je tomu v žánru zvykem, což potvrzuje například i Jonkers[31], navržena tak, aby udržovala hráče uprostřed obrazovky. Tento způsob se jak pro zpracování tak pro uživatele jeví jako ideální, neb dovoluje neustálý přehled o okolí hráče v obou směrech.

3.9.1 Zjišťování a řešení kolizí

Kolize je možné buď jen zjišťovat, nebo je zjišťovat a řešit. Samotná lokalizace kolize se dá využít v momentě, kdy samotná informace o interakci postačuje (hráč vstoupil do oblasti, hráč může sebrat předmět, který není překážkou,...), kolize jsou řešeny jen v případě, kdy je to vzhledem k pohybu objektu potřeba.

Vzhledem k tomu, že všechny pohyby probíhají sekvenčně, kolize se vždy řeší pro aktuálně aktualizovaný objekt vs. všechny ostatní objekty ve scéně. Avšak objekt se může hýbat a detekce kolize musí probíhat na celé vzdálenosti od výchozí do cílové pozice.



Obrázek 3.6: Diagram třídy World.

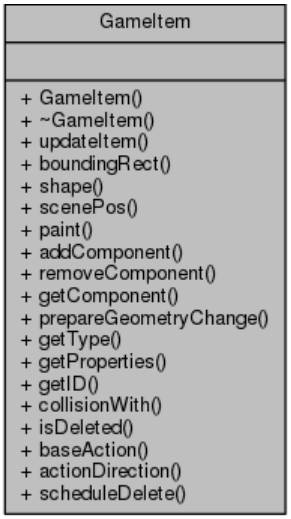
Pro zjištění možné kolize objektu e se používá následující postup:

- Získej ohraničující obdélník ϵ (bounding rectangle, zkráceně BR) pro e.
- Pokud se e resp. ϵ má při další aktualizaci pohnout o vektor $v = (x, y)$, zkopíruj jej a proveď na něm translaci pozice o v, čím vznikne BR ϵ^v. Sjednocení $\epsilon \cup \epsilon^v$ pak vstupuje do dalšího kroku.
- S pomocí BSP (Binary Space Partitioning) stromu[7] zjisti, které entity leží (celé či částečně) v $\epsilon \cup \epsilon^v$.

Kdyby se kolidující objekty nehýbaly, není zcela jasné jak situaci obecně řešit — který z objektů by měl změnit pozici? Co když tato úprava pozice způsobí řetězovou reakci? Musí se zajistit, že nikdy nebude vyžadováno řešení (detekce ano) statické kolize.

Oproti tomu, dynamická kolize, tedy ta, při níž se aktualizovaný objekt lineárně hýbe, řešení má. Jedinou podmínkou je, aby výchozí pozice nebyla kolizní. Pak lze všem dalším kolizím (i vícenásobným) předejít deterministickým způsobem použitím algoritmu SweptAABB (více v [32]).

[7]Použití BSP má dva důvody, jedním z nich je, že jej implementuje přímo framework Qt pro objekty ve scéně a druhý, že při tom používá kolizní tvar (shape).



Obrázek 3.7: Diagram třídy reprezentující herní entity.

3.10 Herní entity

Entity představují objekty, které se přidávají do světa a nějakým způsobem spolu interagují (monstra, hráč, platformy,...). Entity, které reprezentuje třída `GameItem` (diagram na obrázku 3.7), se skládají z komponent, jež je definují a určují jejich chování. Zodpovědností každé entity je správa svých komponent jednotlivě a jak již bylo řečeno výše, slouží také k výměně informací mezi jednotlivými komponentami. Všechna komunikace mezi komponentami musí probíhat přes třídu `GameItem`, jelikož změna jedné komponenty by neměla ovlivnit jiné komponenty.

Z diagramu 3.7 je vidět, že `GameItem` skutečně slouží jen jako kolekce komponent – každá entita má vlastní ID, podle kterého si v konfiguračním souboru přečte seznam komponent a vlastností. Kromě jednoznačného identifikátoru mají entity ještě typ, který určuje jejich kategorii a roli ve hře:

- hráč,
- monstrum,
- platforma,
- bonus.

Typy se používají například u bonusů, kdy se při kolizi každý bonus ptá na typ entity s níž koliduje a pokud se nejedná o hráče, na kolizi nijak nereaguje. Také se používají při boji na blízko, útočící monstrum by nemělo způsobovat zranění jiným monstrům atd. Hra ale nespoléhá jen na definované typy, pokud

si uživatel přidá vlastní, některé vlastnosti zůstanou zachovány. Například vystřelený projektil nikdy nezraní typ vlastníka.

3.11 Vlastnost nebo komponenta

Aby se mohly herní entity rozdělit na komponenty, je potřeba vyčlenit takové části, které nemusejí nutně mít vlastní komponentu, ale lze je reprezentovat nějakým příznakem.

Například „nebezpečný na dotek“, takovou vlastnost by mohla mít některá monstra a jednalo by se o rozumný příznak. Ale nebezpečí nejde reprezentovat jen tím, jestli je nebo není (potřebujeme binární příznak). Nějakým dalším parametrem je potřeba vyjádřit, jak moc je daná entita nebezpečná, tedy přiřadit jí nějaký počet zranění, který způsobí objektu, který se jí dotkne.

Specifikace vlastností entit Binárním příznakem se dá reprezentovat vlastnost pevnosti, tedy dá-li se daný objekt projít. Další příznak je antigravitační, například u bonusů nechceme, aby padaly. Poslední vlastností zvolenou pro tuto hru je „odstranění na dotek“ (remove on touch), která se dá také použít u bonusů.

Komponenty mohou vlastnosti používat různým způsobem, bonusy by například neměla odstranit kolize s monstrem a naopak výboj monstra se nemusí zastavit o nepevné monstrum.

Vlastnosti vzájemně nejsou v rozporu, tedy by se měly dát kombinovat. Vhodným implementačním řešením je reprezentace pomocí bitových masek.

3.12 Komponenty

Komponenty jsou v ECS základním stavebním kamenem. Jednotlivé entity jsou definovány složením svých komponent. V této části se řeší způsob jejich komunikace a také jsou zde jednotlivé komponenty podrobně rozebrány.

3.12.1 Komunikace

Rozdělení herních entit na komponenty s sebou nese jednu zásadní otázku k vyřešení. Zatímco v hierarchii tříd probíhá komunikace velmi intuitivně, v ECS nastává problém s tím, že komponenty o sobě vzájemně nevědí. V dědičnosti, pokud hráč vyskočil a sebral lektvar, který ho vyléčí, zavolala by se ve třídě `Player` metoda `healPlayer(amount)`. V ECS taková třída ani metoda není, tudíž komunikace musí probíhat jinak.

Jakmile nastane kolize dvou herních entit (hráče a lektvaru), obě entity jsou na tuto skutečnost upozorněny a záleží na jejich komponentách, jak situaci vyřeší. Ale ne všechny jejich komponenty musí zajímat kolize. V této chvíli nachází uplatnění návrhový vzor Observer – komponenty se přihlásí o akce,

jež je zajímají a dokáží na ně reagovat. V našem příkladě se tedy bonusová komponenta přihlásí, že chce dostávat informace o kolizích.

Způsob komunikace bude implementován systémem posílání zpráv, který byl popsán v poslední části rešerše (viz sekci 1.5).

3.12.2 Abstraktní třída Component

Periodicky se každá herní entita aktualizuje – volá metodu `update()` na každou svou komponentu. Právě tato metoda slouží jako základní rozhraní, jež všechny komponenty musejí mít. Třída `Component`obsahuje obecný typ dané komponenty a metoda na jeho zjištění.

3.12.3 Grafická komponenta

Grafickou komponentu (`GraphicsComponent`) mají všechny herní entity v návrhu, ačkoliv není bezpodmínečně nutná. Tato komponenta se stará o vykreslování svého obsahu na scénu.

Jedním z principů Qt, jež byl do komponenty zahrnut, ačkoliv sémanticky příliš nesouvisí s grafickou stránku entity, je metoda `shape`. Metoda se používá pro zjištění tvaru entity, který je nutný při detekci kolizí. Na druhou stranu by kolizní tvar alespoň nějakým způsobem měl odpovídat tomu, jak entita vypadá (např. postava hráče by neměla kolidovat s platformou, pokud to graficky není vidět). Pro každou grafickou entitu tak lze nastavit offset od levého horního okraje obrázku a také velikost kolizního tvaru (obdélníku), čímž se dá docílit přesnějšího systému kolizí. Původní bounding rectangle obrázku a kolizní tvar je vykreslený na obrázku 3.9.

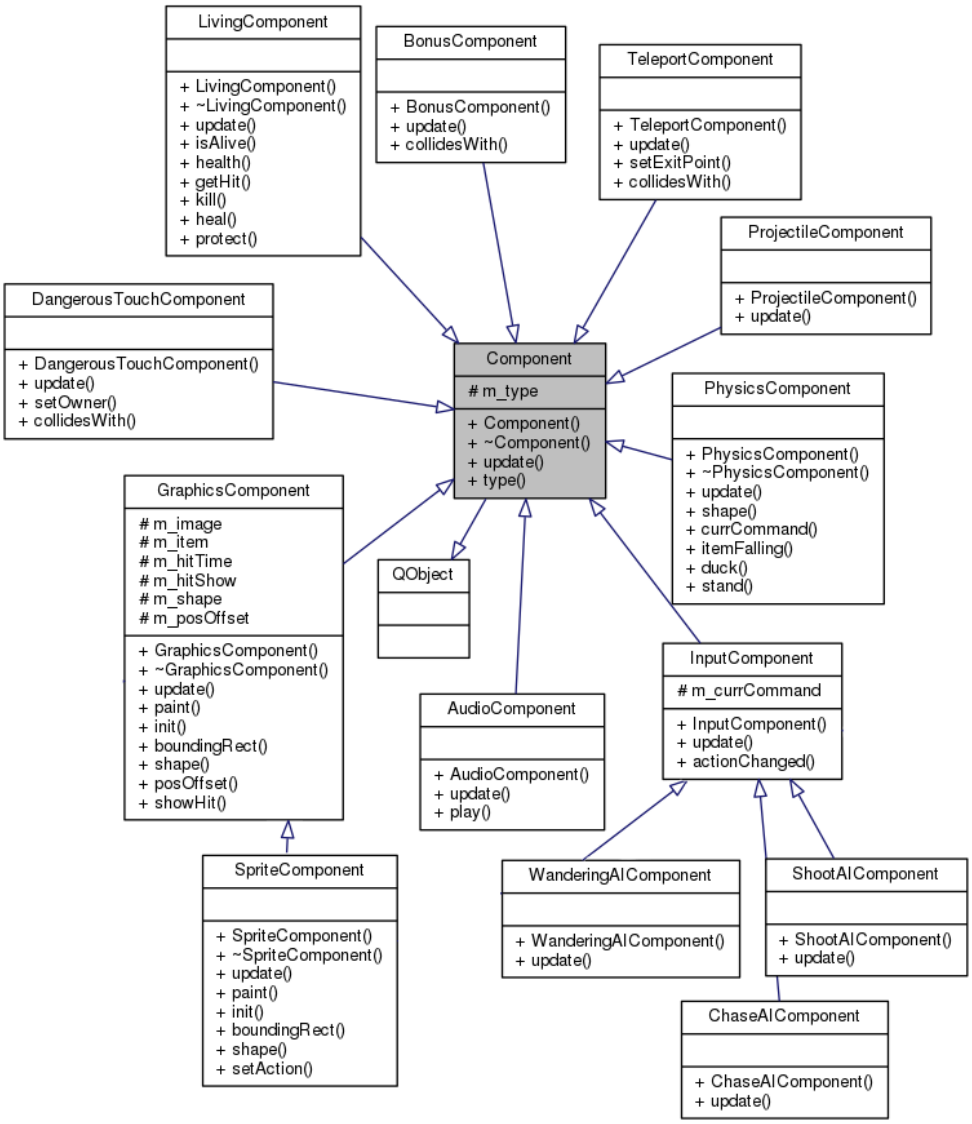
Komponenta dokáže vykreslit pouze statické obrázky, ale speciální vlastností je reakce na zranění. Vizuálně se v této hře zranění vyjadřuje „blikáním“ grafické složky.

3.12.4 Animovaná grafická komponenta

`SpriteComponent`, je odvozena od grafické komponenty a obsahuje animovanou grafickou reprezentaci entity. Mohlo by se zdát, že používat dědičnost v komponentách ECS architektuře porušuje její základní myšlenku, nicméně Sprite skutečně je grafickou komponentou, jen ji rozšiřuje o animovanou část.

Dále komponenta reaguje na změnu akce pomocí slotu `setAction`. Např. když hráč změní akci z chůze na útok, grafická komponenta tuto změnu reflektuje bez znalosti toho, jakým způsobem entita akci změnila (vstup z klávesnice, AI, ...). Navíc je změna provedena okamžitě, nečeká se dokončení animace.

Aby bylo zajištěno, že grafická reprezentace koresponduje s kolizním systémem, komponenta klade na obrázek s grafikou požadavek ohledně velikostí jednotlivých snímků animace – všechny by měly být zhruba stejně velké (malé odchylky by hráč nemusel pozorovat). Jedinou výjimkou je akce přikrčení (angl. duck).



Obrázek 3.8: Návrh komponent

Obrázek 3.9: Bounding rectangle obrázku (zelená) a tvar (shape) použitý pro kolize (červená).

Komponenta také šetří paměť, jelikož si udržuje informaci o směru entity a tedy jí stačí mít uložené animace otočené pouze vpravo. Vykreslení opačného směru řeší vertikální zrcadlení.

3.12.5 Komponenta zranění na dotek

Původně byla tato komponenta zamýšlena jako vlastnost (property), ale nebyla možnost uložit, kolik zranění má způsobovat. Další výhodou samostatné komponenty je fakt, že v reakci na kolizi se dá vyřešit, komu má komponenta způsobovat zranění. Aktuální verze počítá s tím, že nelze zranit nehmotný objekt a také že si monstra nemohou ubližovat navzájem.

Tato komponenta využívá vlastnosti REMOVE_ON_TOUCH k dotvoření chování typického pro entity projektilu – vystřelený projektil se po kolizi odstraní. Kvůli správnému chování projektilu je v komponentě i typ a identifikátor vlastníka – pokud by měl projektil nastavenou nižší rychlost letu než např. pohybující se hráč, který bude střílet během skoku či chůze, střílející tak zraní sám sebe, což není chování, které se očekává.

3.12.6 Komponenta vstupu

Aby entita mohla reagovat na prostředí, potřebuje být kontrolována příkazy. K tomu slouží třída `InputComponent`, jež v základní variantě přijímá příkazy[8] ze vstupu (zde z klávesnice) a přeposílá je přes události dál. Všechny ostatní příkazové komponenty – především jednoduchá umělá inteligence monster – dědí od této komponenty a taktéž vystřelují aktuální příkazy jako události.

[8] Jedná se skutečně již o herní příkazy, stisknuté klávesy na příkazy zpracovává InputHandle.

3.12.7 Definice jednoduchých AI monster

Každému monstru (obecně vzato čemukoliv) lze přiřadit AI, kterou se bude ve hře řídit. Každému definovanému monstru lze přiřadit pouze jeden typ AI, což je vhodné pro hráče. Pokud by totiž exemplář jednoho monstra hráče ignorovat a druhý jej pronásledoval, bylo by to pro něho matoucí.

Hlídkování Nejjednodušší forma AI, požadované příkazy se omezují jen na jdi vlevo a jdi vpravo nezávisle na pozici hráče. V konfiguračním souboru lze nastavit kolik kroků má být vlevo a vpravo (délku kroku upravuje komponenta fyziky). AI se po aktivaci vydá náhodným směrem, ujde počet kroků definovaných pro tento směr, poté se otočí a jde opačným směrem.

Pronásledování Monstrum může hráče pronásledovat buď jen chozením za hráčem, nebo i (podle nastavení fyziky monstra) rychlejšími skoky. Komponenta má jako parametry vzdálenost v pixelech, odkdy má začít reagovat. Například když je hráč blízko x pixelů v horizontálním směru, monstrum se vydá k hráči chůzí a když je blízko i y pixelů ve vertikálním směru, monstrum se k hráči přibližuje skoky.

Střílení Tato AI byla přidána po testování schopnosti generátoru vytvořit průchozí hru. Střílení monster, ke kterým se hráč nedostane, vytvářelo nepřekonatelné situace, tudíž byla vytvořena tato AI. Pracuje na podobném principu jako pronásledování. Pokud je hráč v podobné výšce (určeno v konfiguračním souboru) jako monstrum, monstrum začne střílet.

3.12.8 Komponenta bonusu

Vzhledem k tomu, že všechny bonusové předměty sbírá pouze hráč, nemělo smysl každý bonusový item (nebo jeho kategorii) vytvářet jako speciální komponentu. `BonusComponent` tedy sdružuje všechny bonusy, kterých jsou v tomto návrhu tři druhy[9]:

- léčivé předměty – kolik HP (health points) vyléčí,
- munice – kolik munice přidá,
- ochrana – kolik milisekund hráč nemůže dostat zranění.

[9] V této komponentě lze vidět, že ECS má obrovské možnosti variability. Není zde specifikováno, jestli léčivý předmět je lektvar, který vyléčí 15 HP (zdraví, health points), nebo srdíčko, které vyléčí 100 HP. Tyto vlastnosti se dají jednoduše specifikovat v externím souboru.

3.12.9 Komponenta života

Všechny entity, které lze zranit musí někde uchovávat počet životů (HP). `LivingComponent` kromě číselného parametru ještě hlídá tzv. čas ochrany (protection time), jež je specifikován jako parametr a jeho význam spočívá v tom, že událost kolize s nebezpečným předmětem se emituje častěji, než je hráč schopen reagovat – po tuto dobu tedy nelze entitu zranit znovu.

Komponenta kromě možnosti entitu zranit a vyléčit ještě obsahuje metodu `kill`. Metoda najde jasné uplatnění v případě, kdy účinek na entitu má být okamžitý bez ohledu na jeho aktuální stav (například v případě vypadnutí z hrací plochy)

3.12.10 Komponenta zvuku

Nejen hráč může vydávat nějaké zvukové efekty – k jejich přehrávání slouží `AudioComponent`. Komponenta funguje na podobném principu jako komponenta fyziky (viz kapitolu 3.12.12) – pomocí slotu `play` zachytává typy zvuků, jež má za úkol přehrát. Díky sdílení zdrojů viz 3.16 tak může být stejný zvuk použit nejen pro různé entity, ale i pro různé typy akcí (např. stejný zvuk pro doskočení hráče a pro útok monstra).

3.12.11 Komponenta projektilu

Hozené jablíčko Karkulky nebo nebezpečný magický výboj monstra mají jednu společnou vlastnost – jedná se o projektily. Pohyb projektilu se v tomto žánru obvykle řeší dvěma způsoby – pohyb je buď pouze horizontální, nebo využívá zjednodušený fyzikální model.

V případě přímého pohybu na něj pak nemá vliv gravitace a projektil pokračuje, dokud nenarazí na překážku. Takový přístup by byl přijatelný pokud by hra měla předgenerované úrovně a projektil by se několik obrazovek dopředu o něco zastavil (platformu, monstrum...). Vzhledem k tomu, že generování probíhá s předstihem jen několika málo obrazovek, není možné předpovědět chování projektilu za nimi.

Druhý způsob imituje realističtější chování – hozený předmět opíše oblouk a spadne na zem. Takový způsob zbytečně frustruje hráče – musejí se na (často pohybující se) monstrum obloukem trefit.

Konečné řešení tak kombinuje oba přístupy. Na začátku pohybu má projektil tzv. antigravitační čas (konfigurovatelný) během něhož se pohybuje jen horizontálně a po jeho uplynutí gravitace začne působit a projektil postupně padá (parametry konfigurovatelné) k zemi.

3.12.12 Komponenta fyziky

`PhysicsComponent` spravuje všechny akce dané entity a přechody mezi nimi jako stavový automat. Aktuální příkaz k akci získává komponenta přes slot

Obrázek 3.10: Vstupní (nahoře) a výstupní teleport.

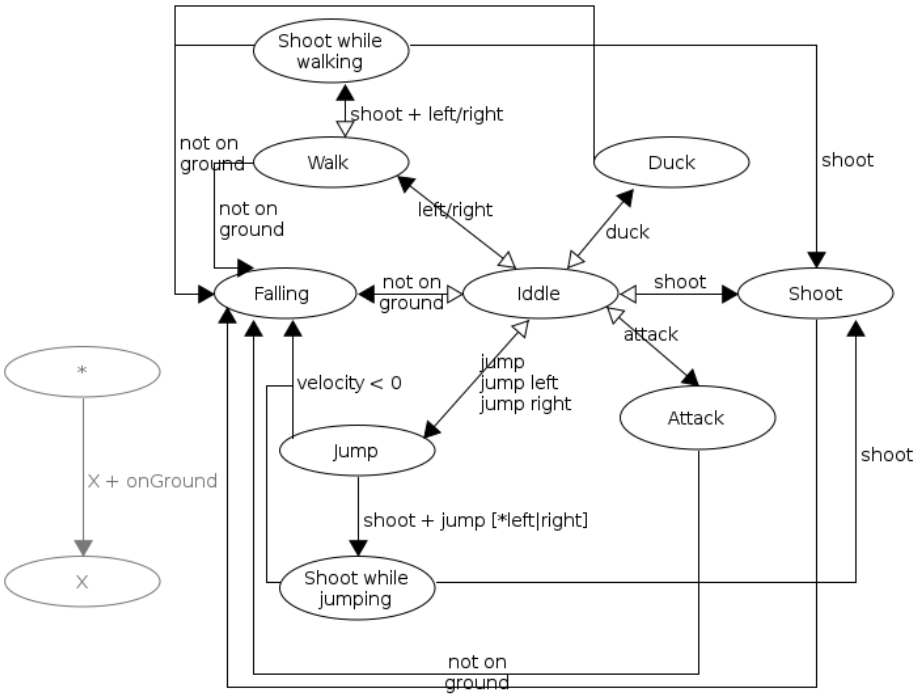
`currCommand`, na který reaguje v metodě `update`. Pokud je daný příkaz pro entitu definovaný, předá jej aktuální akci – např. aktuální akcí je skok vlevo a nový příkaz je přikrčení. Pokud entita podporuje přikrčení, do stavu „skok vlevo“ se tento příkaz pošle a podle interních pravidel akce se do stavu přikrčení buď přejde nebo nikoliv. Pokud se do stavu přejde, komponenta dá změnu vědět událostí `actionChanged`.

Stavový automat akcí (Obrázek 3.11, akce vysvětlené níže v sekci 3.13) zůstává pro všechny entity stejný nehledě na to, jaké akce jsou pro entitu definovány. Komponenta fyziky zodpovídá za to, že přechody proběhnou pouze do definovaných akcí.

Tento přístup opět podporuje požadavek na externí konfigurovatelnost – pro přidání akce (např. monstrum umí útočit na dálku) stačí do konfiguračního souboru přidat akci střílení do komponenty fyziky daného monstra. Všechny přechody a pravidla tak jsou již definovány.

3.12.13 Komponenta teleportu

Tato komponenta byla přidána dodatečně po analýze generátoru a situace nerešitelných kolizí (viz 2.5.2). Jediným specifikem komponenty je výstupní bod teleportu, který podle pokračování cesty nastavuje generátor. Obrázek 3.10 zobrazuje teleport ve hře.



Obrázek 3.11: Automat akcí

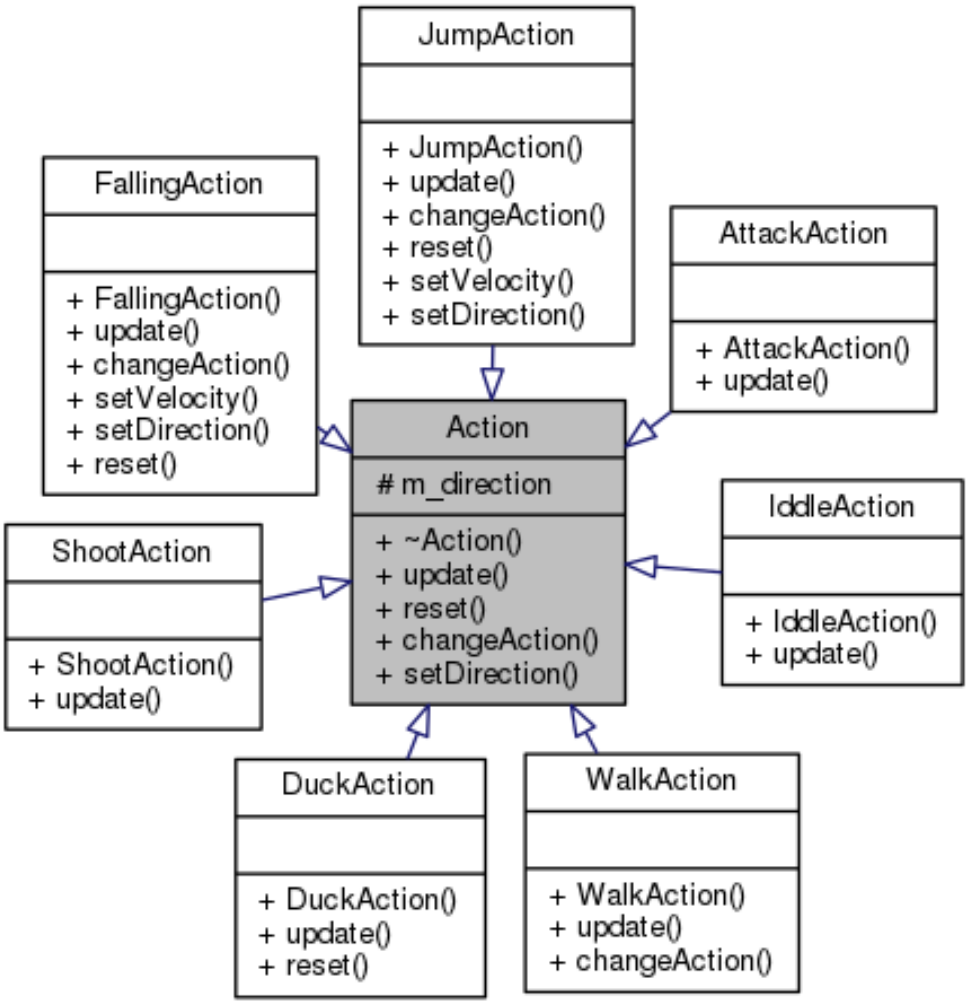
3.13 Akce

Akce definované pro všechny herní entity (automat na obrázku 3.11, model na obrázku 3.12) spravuje Komponenta fyziky (3.12.12). Důvod, proč byly všechny akce a přechody mezi nimi definovány obecně pro všechny souvisí s externí konfigurovatelností. Pokud by chtěl uživatel přidat například nové monstrum, musel by pro něho definovat nejen fyzické vlastnosti, ale také přechody mezi stavy a podmínky, za kterých se přechody mohou uskutečnit. Takové řešení je složité a zbytečně přenáší zátěž na uživatele.

V této sekci budou jednotlivé akce podrobně rozebrány, stejně jako jejich přechody a reakce na vlastnosti entit.

Klíčovou vlastností všech akcí (kromě skoku a pádu) je kontrola, zda-li entita stojí na pevné zemi. Pokud ne, tak padá. Pádu se dá zabránit jedině nastavením antigravitačního příznaku, případně kompletního odstranění komponenty fyziky a všech akcí.

Dále mají všechny akce definováno, že pokud stojí na pevné zemi a přijde požadavek na změnu stavu, do nového stavu se přejde, což značí šedá část obrázku 3.11.



Obrázek 3.12: Návrh akcí

3.13.1 Chůze

Chůze pohybuje entitou po ose x. Délka kroku (v pixelech) je definována v konfiguračním souboru. Pokud se entita před uskutečněním kroku nenachází na pevném podloží a zároveň nemá nastavenou antigravitační vlastnost, mění se akce na pád. Z tohoto vyplývá, že entita, která má definovanou chůzi, musí mít definovanou buď akci pádu nebo antigravitační vlastnost.

Chůze na přechod do jiného stavu (pokud entita nepadá) neklade žádné podmínky. Dokonce lze během chůze útočit na dálku viz sekci 3.13.5.

Tabulka 3.1: Příklady parametrů pro skok

	hráč	monstrum
T	3,5s	2,5s
x	15	7
y	$(-60, 60)$	$(-20, 20)$
p	25/1000	

Obrázek 3.13: Vykreslení rovnice skoku a pádu

3.13.2 Skok a pád

Skok i pád jsou samostatné akce ačkoliv sdílejí stejné fyzikální vlastnosti. Důvod rozdělení těchto akcí je ten, že entita nemusí mít nastavenou antigravitační vlastnost, ale zároveň neumí skákat. Tedy pokud se vygeneruje nad platformu, dopadne na ni a dále už se po ose y nepohybuje. Kdyby akce byly spojené, entita by musela umět skákat, ačkoliv je taková dovednost pro ni naprosto zbytečná.

Kroky skoku i pádu se řídí následující rovnicí:

$$\Delta_{(t_{n+1})}(x, y) = [x, y_n - 2 * (T - p^2)],$$

kde Δ reprezentuje, jak se entita v prostoru posune. Krok po ose x je konstantní a y_n je krok na ose y v předchozí iteraci. T je celkový čas skoku a p představuje uplynulý čas od začátku skoku.

Průběh skoku a tedy i vykreslení rovnice pro hodnoty z tabulky 3.1 je na obrázku 3.13. Jedná se o aktuálně nastavené parametry skoku pro hráče a jednoho z monster.

Akce skoku není kvantována, ale je možné ovlivnit jeho fyzikální parametry i během skoku samotného. Při konstantním příkazu skoku (ať už AI, nebo stálým stisknutím klávesy) se provede skok předně dle parametrů rovnice výše a entita tedy skočí do maximální výšky. V případě přerušení příkazu (např.

puštění klávesy) a pokud je entita stále ve vzduchu, skok se okamžitě nepřeruší, ale entita ztrácí zrychlení (hodnota y) s každou aktualizací komponenty fyziky o $\frac{1}{5}$.

Je možné během skoku měnit jeho směr. Skok také není výsadní akci, ale lze jej kombinovat s dalšími akcemi, například útokem na dálku.

3.13.3 Přikrčení

Přikrčení je speciální akce, při níž se mění kolizní tvar entity. O kolik (v procentech) se má tvar snížit upravuje konfigurační soubor, aby kolize mohla korespondovat s grafickou složkou zadanou uživatel v souboru. Během přikrčení nejsou povoleny žádné jiné akce kromě otočení. Entita se nemůže pohybovat ani útočit, tedy jediným přechodem, kromě pádu, zůstává změna do stavu nečinnosti postavením se.

3.13.4 Útok na blízko

Parametry útoku na blízko jsou:

- způsobené zranění,
- dosah (na ose x),
- zpoždění.

Kolizní systém útok zblízka řeší tím, že útok vede od kolizního tvaru po ose x až do hodnoty dosahu.

Hodnota zpoždění udává pauzu v milisekundách mezi jednotlivými útoky. Pokud by zpoždění nebylo nastaveno, útok by probíhal během každé aktualizace.

3.13.5 Útok na dálku

Útok na dálku je speciální kompozitní akce, která umožňuje střílet během chůze, skoku či pádu, ale také existuje i sama o sobě.

Parametry akce jsou:

- čas nabití,
- ID projektilu.

Čas nabití má stejný význam jako zpoždění u útoku na blízko, hodnota v milisekundách určuje kadenci střelby. ID projektilu je identifikátor entity, která bude přidána do scény. Parametry akce nemohou ovlivnit fyzikální vlastnosti projektilu. Což nemusí nutně znamenat nevýhodu akce se tak neomezuje pouze na entity obsahující komponentu projektilu, v některé z budoucích verzí tak například snadno může některý z bossů „střílet“ monstra na hráče. Či může

(vhodnou kombinací komponent) existovat platforma periodicky emitující bonusy.

3.14 Konfigurační soubor

Formátem konfiguračního souboru byl zvolen JSON. Je pro účely konfiguračního souboru pro neprogramující uživatele dostatečně přívětivý – existují pro něho různé editory s kontrolou syntaxe a zároveň je dobře strojově zpracovatelný (framework Qt obsahuje přímo knihovny na práci s tímto formátem).

3.15 Grafické zdroje

Aby se grafické zdroje v paměti zbytečně neduplikovaly, jsou spravovány statickou třídou `GraphicsResources`. Komponenty statické a animované grafiky tak udržují pouze reference na zdroje.

Zatímco statická grafika se vzhledem k malému datovému objemu načítá přímo do třídy reprezentující obrázek (a následně je možné ji snadno předat referencí), v případě o poznání datově objemnější animace je situace poněkud složitější.

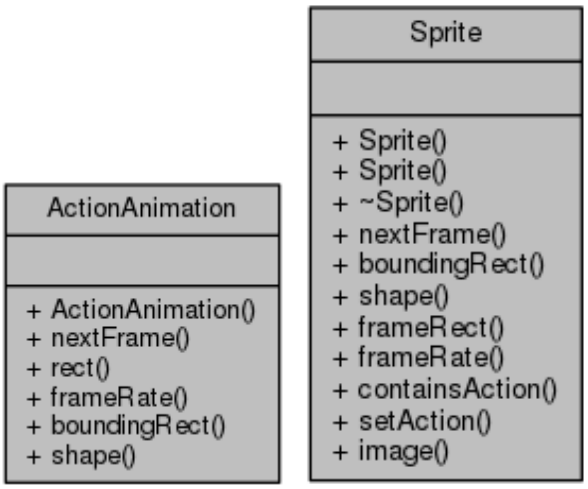
Jelikož ve hře může být velké množství animované grafiky, je žádoucí, aby nejen obrázky obsahující animace byly v paměti obsaženy jen jednou, ale také informace o animaci. Mezi takové informace patří posloupnosti akcí a lokace jednotlivých snímků animace akce v obrázku. Animace jednotlivých akcí spravuje třída `ActionAnimation`,diagram na obrázku 3.14, což jsou konkrétní obdélníky snímků v obrázku a jejich obnovovací frekvence (frame rate). Třída je připravena na situaci, kdy bude možné měnit kolizní tvar entity v různých akcích, tedy obsahuje i funkce informace o kolizním tvaru konkrétní akce.

Jednotlivé akce sdružuje třída `Sprite`, diagram viz obrázek 3.14, která také obsahuje referenci na obrázek animací a třída sama pak funguje jako prostředník, který se sdílí mezi Animačními komponentami jednotlivých entit.

3.16 Audio zdroje

Na rozdíl od grafických zdrojů, zvuky je vhodnější nechávat v paměti po celou dobu hry. Hlavním důvodem je, že zvukové soubory jsou obvykle větší než obrázky a pokud ve hře dojde k akci, která vyžaduje přehrání zvuku, jeho načtení by mohlo trvat tak dlouho, že hráč by zpoždění zaregistroval.

Zároveň se zvuky nenačítají zbytečně – jejich inicializace proběhne v momentě, kdy se do světa vkládá entita, která daným zvukem disponuje. Je to dostatečně brzy na přehrání efektu bez zpoždění (entity se vkládají několik obrazovek dopředu) a dostatečně pozdě, aby hra od začátku neměla v paměti celou kolekci zvuků, čímž by se mohl zpomalit její start.



Obrázek 3.14: Diagramy pomocných tříd pro práci s grafickými zdroji

O správu souborů zvuků se stará statická třída `AudioResources`. Kdykoliv se do světa přidá entita přehrávající zvuk pomocí Audio komponenty, je jí předána reference na `QSoundEffect`. Soubory jednotlivých zvukových efektů jsou udržovány v hashovací tabulce dle jejich jména souboru. Tím je umožněna i nezávislost zvuku na konkrétní akci.

Kapitola 4

Návrh výpočetní inteligence

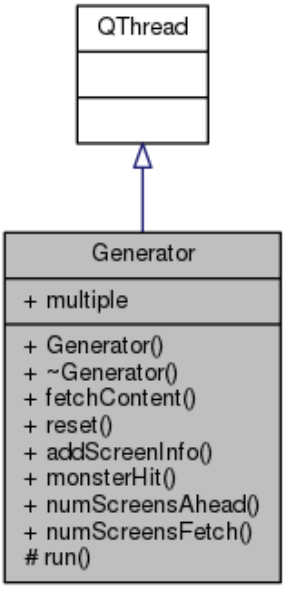
Hlavní třídou výpočetní inteligence je `Generator` (diagram 4.1), který spravuje všechny instance gramatik generujících cesty. Provádí adaptaci prostředí a komunikuje s prostředím hry.

Generátor pracuje ve vlastním vláknu a s prostředím hry komunikuje pouze prostřednictvím třídy `World`, která mu periodicky po každé překonané obrazovce předá jak dlouho hráči úsek trval. Zranění od monstra předává okamžitě.

Generátor si všechny tyto informace agreguje a zpracovává je jakmile je hotov s generováním aktuálního požadavku. Práce generátoru po přijetí požadavku se provádí v cyklu:

1. Je konec? Ukonči cyklus.
2. Restart? Smaž mřížku, posuň gramatiky a nastav výchozí obtížnost.
3. Gramatiky sekvenčně generují jeden úsek.
4. Posuň mřížku a gramatiky (vygenerovaná data již generátor nepotřebuje v těchto strukturách uchovávat).
5. Předej nové prostředí do hry.
6. Nauč se na nových datech.

Parametrizace počtu cest Jelikož se v analýze neuvádí přesný počet cest, které budou gramatiky generovat, bude vhodné počet parametrizovat. Kolize se řeší pouze na malém okolí aktuální pozice gramatiky, tedy nezáleží, jestli gramatika koliduje s jednou nebo více cestami. Počet cest by ale měl být úměrný velikosti hrací plochy (obrazovky).



Obrázek 4.1: Diagram generátoru

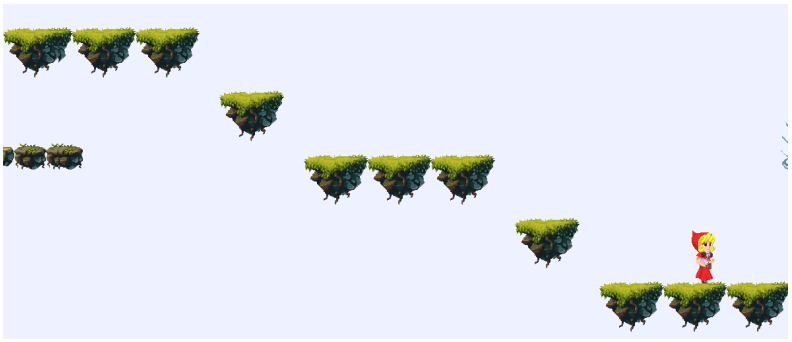
4.1 Gramatiky

Rozhraní stochastické kontextové gramatiky ve třídě `GrammarPath` (diagram na obrázku 4.3) se omezuje pouze na tři základní funkce:

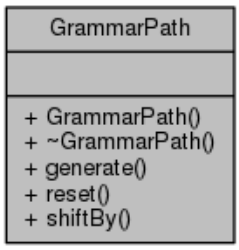
- samotné generování cesty,
- posunutí,
- restart

Gramatika pracuje s myšlenkou rytmu. Každý rytmus se skládá z jednoho konkrétního typu skoku a délky chůze, které se střídají v určitém počtu opakování (jak lze vidět na obrázku 4.2). Také si rytmus udržuje jedinou grafickou podobu platformy (detaily v sekci o generátoru platforem 4.6.1). Toto řešení pomáhá vytvořit dojem, že prostředí není generováno zcela náhodně. To, kolik jednotek cesty má gramatika vygenerovat získává parametrem. Poté se řídí rytmem a střídá skoky s chůzí.

Gramatika má také určený index (pořadí v generování), gramatika s indexem 0 je speciálně vyhrazena pro generování země. Specifikem této cesty je, že neobsahuje jiné skoky než přes překážku. Dále má cesta po zemi i speciální grafickou reprezentaci (detaily v generátoru platforem 4.6.1). Generátor také zajišťuje, že cesta po zemi bude vždy generována jako první a zajišťuje jí místo tím, že při každém generování má právě tato cesta možnost vygenerovat delší úsek.



Obrázek 4.2: Výsledná podoba rytmu ve hře



Obrázek 4.3: Diagram gramatiky

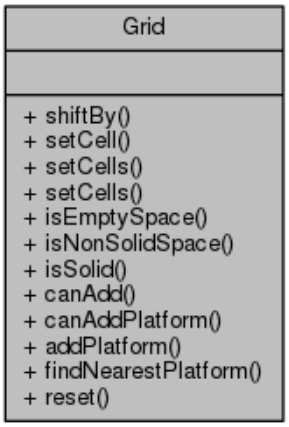
4.2 Měřítko

Změna měřítka se dá použít, pokud jsou kolizní tvary všech entit násobkem nějakého celého čísla. V této práci byla použita grafika s rozměry dělitelnými 16. Měřítko je tedy 16× menší, stejně tak mřížka z čehož plyne, že operace na ní jsou výrazně rychlejší. Se škálovanými jednotkami (políčky v mřížce) poté pracuje celá výpočetní inteligence.

4.3 Mřížka

Mřížka (diagram 4.4) udržuje informace o prostředí, do kterého se generují cesty, tedy by měla být sdílená mezi všemi gramatikami. Její buňky smí obsahovat pouze typy vyjmenované v seznamu 2.4. Každé pravidlo se před přidáním obsahu do mřížky dotazuje, je-li možné obsah přidat. Mřížka k tomuto účelu obsahuje několik dotazovacích funkcí, včetně dotazu na přidání plošinky.

Při prolínání cest je potřeba v mřížce vyhledat nejbližší platformu, na níž se generování napojí. Vyhledávání probíhá po sloupcích až do vzdálenosti maximální šířky skoku hráče (dál se hráč nemá šanci fyzickými schopnostmi dostat). Zároveň bylo potřeba ošetřit případ, kdy nalezená platforma bude začínat vlevo od současné pozice – to není možné, gramatika nedokáže generovat



Obrázek 4.4: Diagram mřížky

např. skok doleva. Nejbližší platforma se tedy počítá od jejího levého horního rohu Euklidovskou vzdáleností od aktuální pozice.

4.4 Implementace pravidel

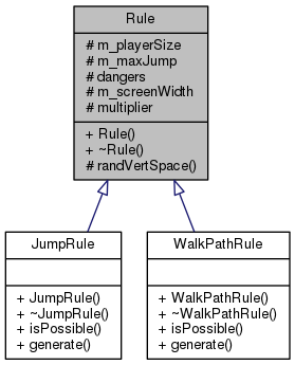
Každé pravidlo (diagram na obrázku 4.5) pracuje ve dvou fázích. Gramatika se nejprve pravidla dotazuje, zda-li je možné jej z neterminálu odvodit za aktuálního stavu mřížky. V této fázi se uplatňují definice jednotlivých pravidel (viz 2.4). Pravidlo skoku si interně rozděluje skoky přes překážku, skoky nahoru a dolů, poté se předávají konkrétní dotazy na mřížku, např. je-li možné na pozici (x, y) přidat obdélník velikosti (w, h) typu pevné entity.

První fáze má navíc příznak, který značí, že se cesta snaží navázat na již existující platformu, v takovém případě se pravidlo nedotazuje mřížky na možnost přidání pevné entity (platformy).

Druhá fáze už je ryze generativní, kdy se do mřížky vkládají informace o cestě. Pravidlo chůze ve druhé fázi generuje nejen první platformu, ale podle aktuální obtížnosti také monstra a bonusy. Pravděpodobnost jejich generování se řídí rovnicemi vysvětlenými v sekci 2.6.2.2.

Důvod, proč je generování pravidla chůze ve druhé fázi vícenásobné je ten, že by se mohlo stát, že několik přidaných platforem by mělo šířku menší než všechny bonusy a monstra, tudíž by nebylo možné tyto entity vůbec přidat. Vícenásobné generování toto řeší a díky iterativnímu dotazování na možnost přidání každé další platformy je zajištěno, že cesta bude korektní a dle pravidel.

Učení se obtížnosti Každé pravidlo dostává třídu `Dangers`, která obsahuje skoky, monstra a příslušnost k segmentu ve hře, jež pravidlo vygenerovalo. Díky této třídě je možné aplikovat učení na základě času, tedy se všem skokům a monstrům v daném segmentu (zde obrazovce) zvýší pravděpodobnost.



Obrázek 4.5: Diagram pravidel

4.5 Uchovávání času

Obtížnost se určuje podle průměrného času, za který hráč zdolal jeden úsek. Uchovávat časy všech obrazovek není paměťově optimální, stačí proto mít pouze průměrný čas p a počet úseků $n > 1$. Další čas t pro $n + 1$ obrazovku se přidá následovně:

$$p = p * \frac{(n-1)}{n} + \frac{t}{n}$$

4.6 Jednotlivé generátory

Generování monster, platforem, bonusů a skoků bylo odděleno do samostatných tříd (diagramy na obrázku 4.6), které starají o generování, načítání a případně i o učení.

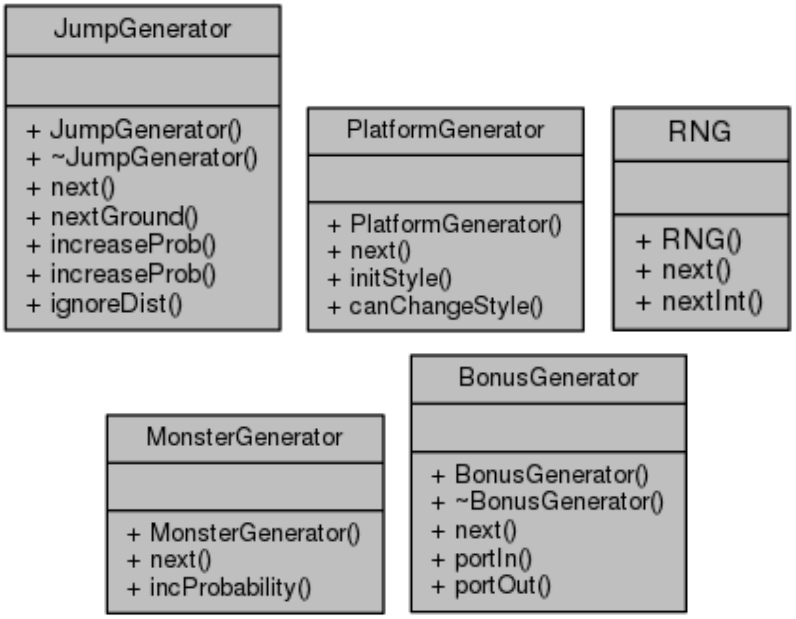
Generování monster, platforem a bonusů probíhá tak, že se z generátoru náhodných čísel dostane číslo c v intervalu $(0, 1)$ a poté se postupně načítají pravděpodobnosti jednotlivých entit, dokud se nedosáhne hodnoty čísla c.

Jelikož se generátory často předávají jako parametry funkcí, bylo jejich rozhraní sjednoceno do třídy `Generators` (diagram na obrázku 4.6), která představuje kompaktní verzi.

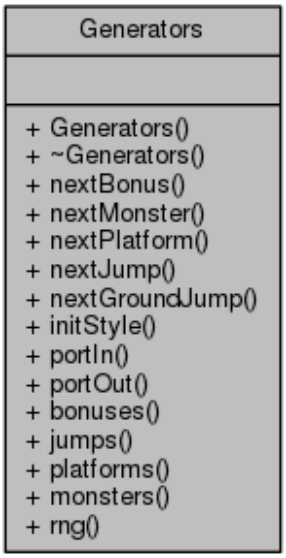
4.6.1 Platformy

Generování platforem spravuje třída `PlatformGenerator`. Načítání platforem z konfiguračního souboru probíhá tak, že inicializační funkce projde entity a načte všechny s typem `PLATFORM`. Každá platforma by v souboru měla mít uvedený styl. Styly od 0 do 2 (včetně) jsou vyhrazeny pro platformy země, vyšší čísla stylů mohou generovat ostatní cesty.

Dále mají platformy uvedenou váhu, proto, aby něčím výrazné grafické prvky nebyly ve hře příliš často. Váha je poté převedena na pravděpodobnost



Obrázek 4.6: Diagramy generátorů



Obrázek 4.7: Diagram pomocných generátorů

v rámci stylu. Použití váhy v souboru je lepší než použití přímé pravděpodobnosti, jelikož když uživatel přidá další platformu, nastaví jí váhu a o zbytek se postará program. Kdyby přidal pravděpodobnost, musel by přepočítat pravděpodobnosti všech ostatních platforem.

4.6.2 Skoky

O skoky se stará třída `JumpGenerator`. Skoky jsou reprezentované maticí J, kde prvek $J_{2,3}$ určuje pravděpodobnost skoku doprava o 2 pole a o tři pole dolů.

Generování není ovlivněno pouze samotnou pravděpodobností, ale částečně také pozicí hráče v obrazovce. Pokud je hráč příliš vysoko (téměř u stropu), je vyšší šance, že se vygeneruje skok dolů. Aby bylo možné toto implementovat, při normalizaci matice není součet pravděpodobností 1.0 ale 2.0. Podle pozice hráče se tak začíná načítat pravděpodobnost od skoků dolů, nebo od skoků nahoru.

Učení skoků se řídí Gaussovským kernelem velikosti 3 a $\sigma = 1.0$. Má-li být pravděpodobnost daného skoku zvýšena výrazně (skok v náročnosti, který řídí generátor), použije se přičtení hodnoty v matici k pravděpodobnosti skoku a jeho okolí. Nižší zvýšení pravděpodobnosti přičítá jen polovinu uvedených hodnot Gaussovského kernelu:

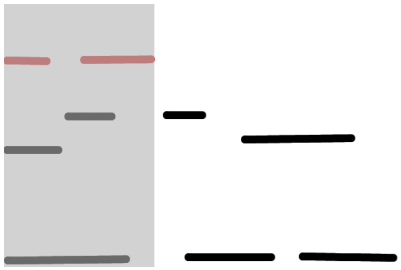
$$\begin{pmatrix} 0.024879 & 0.107973 & 0.024879 \\ 0.107973 & 0.468592 & 0.107973 \\ 0.024879 & 0.107973 & 0.024879 \end{pmatrix}$$

4.6.3 Monstra

Generování a adaptaci monster zařizuje třída `MonsterGenerator`. Inicializační funkce podobně jako u platforem prochází všechny entity a načítá jen ty, které mají uvedený typ `MONSTER`. Žádné další informace generátor nepotřebuje. Jak již bylo řečeno v analýze, všechna monstra začínají se stejnou pravděpodobností. Navržený algoritmus z analýzy nevyžaduje žádné implementační úpravy, tedy je možné jej přímo použít.

4.7 Navazování cest

Aby mohla gramatika navázat na předchozí vygenerovaný úsek, udržuje si svou aktuální pozici. Zároveň bylo potřeba vyřešit, jak pracovat s mřížkou, aby v ní po každém vygenerování nezůstávalo v paměti zbytečné množství starých dat. Proto se posouvají gramatiky i mřížka o nejkratší vygenerovaný úsek (viz obrázek). Informaci o tom, kde se generování nachází předává generátor při každém požadavku.



Obrázek 4.8: Posouvání gramatik a mřížky

4.8 Kolizí

Zjištění kolize dvou cest probíhá při generování gramatikou. Pokud pravidlo s aktuálním rytmem nemůže pokračovat, pokusí se místo skoku přidat chůzi a opačně. Pokud toto řešení nepřinese pokračování cesty, nastaví se zcela nový rytmus a pokus se opakuje.

Pozorováním generování bylo zjištěno, že gramatika je schopna zotavit se z kolize po 150 iteracích. Pokud je překročeno toto množství iterací, gramatika přechází do stavu, kdy se snaží kolizi explicitně vyřešit.

První pokus o vyřešení je nalezení nejbližší platformy, na níž se gramatika zkusí napojit. Pokud se na tuto platformu napojit nelze (např. není-li dostatek místa na skok), nebo nejbližší platforma nebyla nalezena (ve vzdálenosti, kterou je hráč schopen na jeden krok urazit nebyla nalezena žádná platforma), přidává se brána teleportu.

4.8.1 Přidávání teleportů

Teleport je poslední možnost jak lze zajistit pokračování cesty, tedy musí nalézt nějaké místo pro cestu. V takovém případě se nejprve hledá místo ve stejném sloupci, aby byl průchod teleportem pro hráče, co nejméně matoucí – pokud by byl teleport přidán příliš daleko ve směru osy x, trhnutí kamery (jež hráče udržuje vždy uprostřed) by bylo pro hráče zbytečně nepříjemné.

Jakmile selžou pokusy o přidání teleportu ve stejném sloupci, pokračuje se dalšími sloupci vpravo – přidávání veškerých entit se drží pravidla přidávat jen vpravo od aktuální pozice.

Informace o přidání teleportu se přidá do třídy, která se posílá třídě `World`. Vstupní bráně se také nastaví vektor posunu hráče, nikoliv absolutní pozice po teleportu.

Kapitola 5

Realizace

Kapitola krátce shrnuje realizaci hry i generátoru, jež byly vytvořena dle návrhu výše. Vzhledem k malým odlišnostem od návrhu jsou v kapitole zmíněny jen některé klíčové části implementace a způsobu práce s mediálními daty.

5.1 Umístění zdrojů

Jednou z důležitých otázek bylo, jakým způsobem s hrou dodávat použité multimediální zdroje a konfigurační soubory. Díky použití Qt se nabízí dvě možnosti:

1. nechat všechny zdroje v nějaké separátní složce,

2. zabalit je v binární podobně do spustitelného souboru .exe s pomocí Qt Resource System (QRS)[10].

Bylo vybráno použití QRS. Díky zabalení do binární podoby je velice nízká šance, že by zdroje někdo přímo upravil, změnil k nim přístupová práva, nechtěně je přesunul, nebo dokonce smazal. Další výhoda spočívá v tom, že ve zdrojovém kódu se nemusí rozlišovat platforma a formát cest – v Unixových systémech se používá jiné lomítko než na systémech Windows. QRS má jednotný formát cest na všech platformách, např. `:/images/grass.png`. Výhodou ale částečně i nevýhodou může být fakt, že zabalené zdroje zpomalují spuštění hry a po celou dobu hraní zůstávají v paměti – zabírají místo, ale zároveň jsou rychle k dispozici. V tomto případě se na moderních počítačích jedná spíše o výhodu, jelikož zdroje nejsou nijak velké. Zjevná nevýhoda spočívá ve složitější úpravě zdrojů – při každé změně se částečně mění i spustitelný soubor.

[10] `http://doc.qt.io/qt-5/resources.html`

5.2 Propagace vstupu

Ve frameworku Qt se pro zpracování vstupu určitou třídou musí nainstalovat tzv. `eventFilter`. Ten zařídí, že třída bude informována o všech událostech. Třída, do níž se filtr instaluje je `Game`, ta posílá třídě `InputHandler` stisknuté klávesy. `InputHandler` klávesy přeloží na herní příkazy a pokud se jedná o řídící příkaz (pauza, ukončení atd.) zpracuje jej samotná `Game`. Příkazy pro herní postavu třída `Game` propaguje do světa, kde se o ně mohou přihlásit herní entity – třída `World` předává příkazy pomocí událostí.

5.3 Paralelní zpracování

Generátor pracuje paralelně ke hře. K tomu, aby spolu tyto dva celky mohly efektivně (a asynchronně) komunikovat se používá Signal-Slot mechanizmus frameworku Qt v kombinaci s jeho systémem metatypů[11].

5.4 Rozlišení

Jako výchozí rozlišení, vůči kterému se bude na ostatních platformách hra škálovat bylo zvoleno 1920×1080, tedy strany v poměru 16:9. Podle statistik uvedených v [33] dvě nejčastější rozlišení na stolních počítačích jsou v tomto poměru a dohromady jsou pak zastoupeny v 46,25 % všech obrazovek.

5.5 Grafické podklady a zvuky

Všechny grafické podklady (platformy, pozadí, monstra, bonusy atd.) a zvuky využívají možností volně přístupných kolekcí, jejichž autoři dílo zpřístupnili pod licencí, která umožňuje nekomerční užití. Ke všem dílům jsou uvedeni autoři i když to třeba jejich licence nevyžadovala. Seznam licencí je na přiloženém CD.

Výjimkou z tohoto pravidla je sprite samotné Karkulky, jež byl vytvořen speciálně pro tuto hru. Autorem spritu je Milan Karaba.

5.6 Odmazávání části světa

Protože hra obsahuje jedinou nekonečnou úroveň, musel být zvolen práh, kdy se staré části budou odmazávat. Cokoliv je od hráče vlevo a vzdáleno více než 2 obrazovky, bude v další aktualizaci smazáno. Tento práh se ukázal jako dostačující, hráči se nikdy tak daleko nepotřebovali vracet a výkonnostní potíže se též neobjevily.

[11] `http://doc.qt.io/qt-5/qmetatype.html`

Kapitola 6

Testování

6.1 Statická analýza kódu

Statická analýza kódu byla prováděna během všech iterací. U tříd jako například `Grid`, na níž závisí spousta dalších tříd byla prováděna opakovaně spolu s jinými druhy testů. Statická analýza odhalila problémy s mezními hodnotami a fungovala jako prvotní nástroj pro řešení potíží s pamětí.

6.2 Manuální testování

Každá funkce byla podrobena testováním na korektnost ihned po implementaci, větší části kódu – např. spolupráce komponent či komunikace komponent byla testována za běhu aplikace na správné chování, správné výstupy a pomocí nástrojů Valgrind[12] a Address Sanitizer[13] na korektní práci s pamětí.

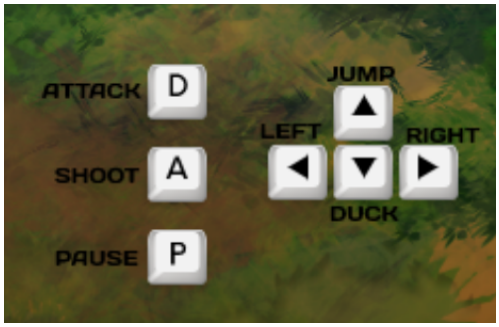
Valgrind odhalil špatnou práci s pamětí (čtení neinicializované hodnoty) a také neuvolněnou paměť v Qt Frameworku, konkrétně ve třídě `QMediaPlayer`, která přehrává hudbu v pozadí (chyba je nahlášena[14]).

6.3 Uživatelské testování

Uživatelské testování probíhalo od počátku, kdy se v herním prostředí začala Karkulka pohybovat a všechny entity (několik obrazovek) byly přidány ručně. Od první iterace se mj. testovaly její fyzické vlastnosti – nastavení parametrů v konfiguračním souboru. Například aktuální parametry pro skok vyžadovaly hodně iterací s každým hráčem.

Všechny testy probíhaly osobně, jelikož žádný z testerů neměl OS Linux.

[12] `http://valgrind.org/`
[13] `https://github.com/google/sanitizers`
[14] `https://bugreports.qt.io/browse/QTBUG-56819`



Obrázek 6.1: Ovládání na úvodní obrazovce

6.3.1 Ovládání

Základním nedostatkem byla absence nějaké nápovědy k ovládání. Žádný z testovaných subjektů nepřišel na všechny klávesy sám (někteří zkoušeli používat i myš). Obrazová nápověda byla přidána hned na úvodní obrazovku, viz obrázek 6.1.

6.3.2 Začátek a první obrazovka

V momentě, kdy se generovalo více cest se hráčům nelíbilo, že od první chvíle nemají možnost dostat se na cestu, která byla nejvýše. Také jim přišlo neférové, že existuje šance propadnutí když Karkulka na začátku hry padá shora.

První řešení zaručovalo, že všechny cesty na první obrazovce budou mít zakázané jakékoliv skoky. Tedy při třech cestách vypadala první obrazovka jako tři horizontální linky, takové řešení se hráčům vizuálně nelíbilo.

Finální řešení tedy na první obrazovce nastaví všem cestám stejnou výchozí pozici – z vlastnosti slučování cest (žádná cesta nemohla obsahovat skok) generátoru pak první obrazovka obsahuje jedinou cestu a poté se všechny cesty oddělí a pokračují nezávisle. Toto řešení zároveň zaručuje, že se hráč dostane ke všem cestám.

6.3.3 Strop

Při testování bylo vypozorováno, že nejvyšší cesta je všemi testery preferovaná – všechna monstra se dala přeskočit tím, že Karkulka skokem nahoru částečně zmizela stropem obrazovky, kde nebyla žádná překážka, jelikož generátor nic nad pomyslný strop nevložil.

Aby nebyla nejvyšší cesta natolik zvýhodněná (propadnutí skokem v nejvyšší cestě na rozdíl např. od země nemusí znamenat okamžitou smrt), byl implementován strop – Karkulka už se nedostane mimo hranice obrazovky.

6.3.4 Neprůchozí část

Jedno z pravidel – generování skoku nahoru muselo být upraveno, aby bylo vždy průchozí. Pokud se vygenerovala cesta nad cílovou platformou skoku a ponechala hráči minimální možný vertikální prostor (výška Karkulky), hráč neměl dostatek místa, aby se mohl na cílovou platformu dostat.

Testování taktéž ukázalo, že přidání jedné jednotky generátoru (16 px) k výšce průchozí části stačí k vyřešení tohoto problému.

6.3.5 Nevyhnutelné monstrum

Po přidání monstra, které mělo jako svou jedinou akci střílení jedním směrem vznikla nepříjemná situace. Mohlo se stát, že monstrum stálo na cestě o úroveň výš než hráč a jeho projektily propadaly nějakou mezerou pro skok přímo na hráče. Pokud se takových monster vygenerovalo více na složitějších místech, hráč se jim nedokázal vyhnout – sice nezemřel, ale způsobená zranění by se též měla počítat do vlastností *průchozí části*, ten nejlepší hráč by měl teoreticky procházet bez zranění.

Jedním řešením by bylo zakázat střílející monstra, ale byla by škoda se takové vlastnosti zbavovat. Druhé řešení bylo vytvoření další jednoduché umělé inteligence, která povoluje střílení monster jen pokud je hráč na podobné vertikální úrovni.

6.3.6 Mizení teleportů

Kdykoliv hráč prošel teleportem, teleport zmizel. Pokud se hráč dostal zpět na původní cestu, už se dál nedokázal pohybovat. V jedné z iterací vývoje tedy byly teleporty (vstupní i výstupní) ponechány i s možností opakované ochrany od výstupního teleportu.

6.4 Výkonnostní testy

Hra i generátor byly vyvíjeny a testovány na počítači s následující konfigurací:

CPU INTEL Core i7-6700,
32GB DDR4-2800MHZ RAM, SSD,
ASUS STRIX-GTX950-DC2-2GD5-GAMING.

Přičemž během celého uživatelského testování nebyly zaznamenány výkonnostní problémy s aktuálním nastavením. Další výkonnostní testy proběhly na výrazně slabším notebooku HP ProBook 4340s s konfigurací:

INTEL Core i5 3230M Ivy Bridge,
4GB RAM, HDD (5 400RPM),
Intel HD Graphics 4000.

V průměrném případě pak hra vytížila CPU na 56,4 % a využila 119,2 MB paměti[15].

[15] Zjištěno nástrojem smem `https://www.selenic.com/smem/`

Závěr

V této práci byla provedena rešerše existujících řešení v oblasti procedurálně generovaných 2D plošinovek a také v oblasti adaptivní změny obtížnosti. Bylo zjištěno, že procedurálně generované plošinovky se většinou omezují na vytvoření jediného průchodu úrovní, tedy se opomíjí existence alternativních cest – nebyl nalezen žádný článek, který by toto řešil. Proto byla možnost alternativních cest zařazena do analýzy výpočetní inteligence, jenž se stará o generování prostředí. Podrobná analýza a návrh výpočetní inteligence je těžištěm této práce.

Myšlenka prezentována v [17] o tom, že se prostředí má adaptovat na fyzické možnosti herní postavy byla v této práci rozvinuta na konfigurovatelnost všech herních objektů. Hráč si tak může bez znalosti programovacích jazyků vytvořit herní postavu s různými vlastnostmi, vlastní grafickou složkou či specifickými zvuky. Totéž lze aplikovat na tvorbu bonusů (typy bonusů a např. léčivé účinky) a monster (jejich chování a další parametry). Obdobně lze konfigurovat také stavební prvky úrovní – samotné plošinky. S různými kombinacemi vlastností dokáže výpočetní inteligence v podobě stochastické kontextové gramatiky pracovat a vytvořit s těmito objekty nové herní prostředí.

Od prvních obrazovek se podle chování hráče adaptuje obtížnost, např. při vyšší obtížnosti se generuje více monster, jež dělají konkrétnímu hráči největší potíže. Adaptace se provádí učením modelu, který určuje pravděpodobnostní rozdělení jednotlivých pravidel gramatiky. Faktory, které měly potenciál na to se účastnit adaptace (učení) byly otestovány v rámci uživatelského testování a měření. Pro toto testování a ověření realizace popsaných postupů byl vytvořen funkční prototyp hry postavený na architektuře *Entity Component System*.

Možná další vylepšení Herní prostředí se dá považovat za dostatečné ačkoliv obsahuje jen omezenou množinu pravidel. Vhodným rozšířením by bylo přidat více pravidel, více unikátních herních prvků, například pohybující se platormy, žebříky, kývající se lana atd. Dále by se do pravidel daly vložit

kombinace, kdy monstrum slouží jako platforma, na níž má hráč vyskočit, aby překonal nějakou propast. Zranění hráče zabrání vygenerování bonusu ochrany před propast.

K takovému generování pravidel by generátor potřeboval mít více informací o objektech – aby jako platformu nevygeneroval monstrum, na kterém nelze stát (není pevné). V současné verzi pouze ví, že objekt je monstrum.

Implementace rytmů se omezuje pouze na opakování skoků a chůze. Jistě by se do rytmů pomocí detailní analýzy dalo zakomponovat načasování boje s monstry a sbírání bonusů.

Jak bylo řečeno v analýze, učení by mohlo probíhat na základě nikoliv izolovaných částí, ale sledovat posloupnosti, které hráče vedly k chybným krokům.

Literatura

[1] Liotti, D.; Dickinson, W.; Butler, C.; aj.: The Bond of Stone [online]. září 2017, [cit. 2017-12-16]. Dostupné z: https://kortuga.itch.io/tbos

[2] Skistad, K.: Samsara [online]. červen 2017, [cit. 2017-12-16]. Dostupné z: https://sparkadegames.itch.io/samsara

[3] Brown, S.; Mitchell, A.; Navidson, J.: Samsara [online]. květen 2017, [cit. 2017-12-16]. Dostupné z: https://muzuka.itch.io/jumpn

[4] Kazemi, D.: Spelunky Generator Lessons [online]. [cit. 2017-12-16]. Dostupné z: http://tinysubversions.com/spelunkyGen/

[5] Mossmouth, LLC: Spelunky World [online]. [cit. 2017-12-16]. Dostupné z: http://www.spelunkyworld.com/whatis.html

[6] Benoit-Koch, F.: Procedural Level Generation for a 2D Platformer [online]. květen 2014, [cit. 2017-12-17]. Dostupné z: http://fbksoft.com/procedural-level-generation-for-a-2d-platformer/

[7] Wolf, D.: Building a Procedurally Generated Platformer [online]. květen 2016, [cit. 2017-12-19]. Dostupné z: http://www.dylanwolf.com/2016/05/09/building-a-procedurally-generated-platformer-in-unity/

[8] nicebyte: Using Perlin Noise to Generate 2D Terrain and Water [online]. [cit. 2017-12-16]. Dostupné z: https://gpfault.net/posts/perlin-noise.txt.html

[9] Martz, P.: Generating Random Fractal Terrain [online]. [cit. 2017-12-16]. Dostupné z: http://www.gameprogrammer.com/fractal.html

[10] Amato, A.: Procedural Content Generation in the Game Industry. *V: Game Dynamics – Best Practices in Procedural and Dynamic Game*

Content Generation [online], 2017, [cit. 2017-12-14]. Dostupné z: http://www.springer.com/cda/content/document/cda_downloaddocument/9783319530871-c2.pdf?SGWID=0-0-45-1604224-p180609223

[11] Scene.org: Scene.org Awards - About [online]. [Cited 2017-12-14]. Dostupné z: http://awards.scene.org/info.php

[12] Shaker, N.; Togelius, J.; Nelson, M. J.: *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer, 2016.

[13] Moss, R.: 7 uses of procedural generation that all developers should study [online]. leden 2016, [cit. 2017-12-16]. Dostupné z: https://www.gamasutra.com/view/news/262869/7_uses_of_procedural_generation_that_all_developers_should_study.php

[14] King, A.: The Key Design Elements of Roguelikes [online]. duben 2015, [cit. 2017-12-12]. Dostupné z: https://gamedevelopment.tutsplus.com/articles/the-key-design-elements-of-roguelikes--cms-23510

[15] Chao, F.; Schockaert, S.; Zhang, Q.: *Advances in Computational Intelligence Systems: Contributions Presented at the 17th UK Workshop on Computational Intelligence, September 6-8, 2017, Cardiff, UK*. Advances in Intelligent Systems and Computing, Springer International Publishing, 2017, ISBN 9783319669397. Dostupné z: https://books.google.cz/books?id=VBo0DwAAQBAJ

[16] Klappenbach, M.: What is a Platform Game? [online]. listopad 2017, [cit. 2017-12-20]. Dostupné z: https://www.lifewire.com/what-is-a-platform-game-812371

[17] Fisher, J.: How to Make Insane, Procedural Platformer Levels [online]. květen 2012, [cit. 2017-12-16]. Dostupné z: https://www.gamasutra.com/view/feature/170049/how_to_make_insane_procedural_.php

[18] Perlin, K.: An image synthesizer. *SIGGRAPH '85 Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, 1985: s. 287–296.

[19] Jennings-Teats, M.; Smith, G.; Wardrip-Fruin, N.: Polymorph: Dynamic Difficulty Adjustment Through Level Generation. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, PCGames '10, New York, NY, USA: ACM, 2010, ISBN 978-1-4503-0023-0, s. 11:1–11:4, doi:10.1145/1814256.1814267. Dostupné z: http://doi.acm.org/10.1145/1814256.1814267

[20] Wheat, D.: Dynamically adjusting game-play in 2D platformers using procedural level generation [online]. 2013, [cit. 2017-12-16]. Dostupné z: http://ro.ecu.edu.au/theses_hons/96

[21] Mourato, F.; Santos, M. P. d.: Measuring difficulty in platform videogames. In *4. ª Conferência Nacional Interacção humano-computador*, 2010.

[22] Nystrom, R.: *Game Programming Patterns*. Genever Benning, 2014, ISBN 0990582906.

[23] Wiebusch, D.: Decoupling the Entity-Component-System Pattern using Semantic Traits for Reusable Realtime Interactive Systems. březen 2015.

[24] Huang, J.; Schonfeld, D.; Krishnamurthy, V.: A new context-sensitive grammars learning algorithm and its application in trajectory classification. In *2012 19th IEEE International Conference on Image Processing*, 2012, ISSN 1522-4880, s. 3093–3096, doi:10.1109/ICIP.2012.6467554.

[25] Smith, G.; Treanor, M.; Whitehead, J.; aj.: Rhythm-based Level Generation for 2D Platformers. In *Proceedings of the 4th International Conference on Foundations of Digital Games*, FDG '09, New York, NY, USA: ACM, 2009, ISBN 978-1-60558-437-9, s. 175–182, doi: 10.1145/1536513.1536548. Dostupné z: http://doi.acm.org/10.1145/1536513.1536548

[26] Folmer, E.: *Component Based Game Development – A Solution to Escalating Costs and Expanding Deadlines?* Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, ISBN 978-3-540-73551-9, s. 66–73, doi:10.1007/978-3-540-73551-9_5. Dostupné z: https://doi.org/10.1007/978-3-540-73551-9_5

[27] Rolland, L.: Will Qt be a good choice for a game? [online]. únor 2015, [cit. 2017-12-10]. Dostupné z: https://stackoverflow.com/questions/4905222/will-qt-be-a-good-choice-for-a-game

[28] V-Play GmbH: Hi - V-Play Engine [online]. [cit. 2017-12-20]. Dostupné z: https://v-play.net/developers/hi/

[29] Itterheim, S.: Why Using A Physics Engine For A 2D Platformer Is A Terrible Idea [online]. srpen 2013, [cit. 2017-12-20]. Dostupné z: http://www.learn-cocos2d.com/2013/08/physics-engine-platformer-terrible-idea/

[30] Halford, C.: Year 2, Semester 2 - Generating Box2D bodies to cover an arbitrary tile map in Oh God Why [online]. květen 2014, [cit. 2017-12-20]. Dostupné z: http://codetrip.weebly.com/blog/year-2-semester-2-generating-box2d-bodies-to-cover-an-arbitrary-tile-map-in-oh-god-why

[31] Jonkers, D.: 13 More Tips for Making a Fun Platformer [online]. červenec 2012, [cit. 2017-12-24]. Dostupné z: http://devmag.org.za/2012/07/19/13-more-tips-for-making-a-fun-platformer/

[32] BrendanL.K: Swept AABB Collision Detection and Response [online]. duben 2013, [cit. 2017-12-24]. Dostupné z: https://www.gamedev.net/articles/programming/general-and-gameplay-programming/swept-aabb-collision-detection-and-response-r3084/

[33] StatCounter: Desktop Screen Resolution Stats Worldwide [online]. prosinec 2017, [cit. 2017-12-24]. Dostupné z: http://gs.statcounter.com/screen-resolution-stats/desktop/worldwide

Příloha A

Seznam použitých zkratek

AI Artificial Intelligence

BR Bounding rectangle

BSP Binary Space Partitioning

ECS Entity Component System

GUI Graphical user interface

HP Health Points

JSON JavaScript Object Notation

LDAI Level Design Artificial Intelligence

QRS Qt Resource System

Příloha B

Uživatelská příručka

B.1 Potřebné knihovny

Kromě C++ kompilátoru je nutné jen Qt verze 5.10, které se dá stáhnout z `qt.io`. Pokud by na linuxu nefungoval zvukový výstup, chybí GStreamer:

```
sudo apt-get install gstreamer1.0
```

B.2 Kompilace

Kompilace v systému Linux. Obdobný přístup lze použít i na platformě Windows, ale nebyl testován.

1. Rozbalte archiv se zdrojovými kódy
2. Ve složce se zdrojovými kódy vytvořte složku a přejděte do ní:

   ```
   mkdir build; cd build
   ```

3. Spusťte qmake a make:

   ```
   qmake ../Draklia.pro -spec linux-g++ && make
   ```

4. Pro spuštění zkompilované hry napište:

   ```
   ./Draklia
   ```

Pokud se vyskytne problém s neexistencí `qmake` – buď přidejte cestu do proměnné `\$PATH`, nebo spusťte `qmake` s absolutní cestou. Měl by se v závislosti na architektuře počítače a verze Qt nacházet v : `složka-s-QT/5.10.0/gcc_64/bin/qmake`.

B.3 Ovládání

Ovládání je uvedeno také ve hře.

Klávesa	Akce
Enter	Zahájení hry
Escape	Vypne hru
P	Pauza
A	Útok na dálku
D	Útok na blízko
↓	Přikrčení
↑	Skok
←	Chůze vlevo
→	Chůze vpravo

B.4 Formát konfiguračního souboru

Konfigurační soubor je ve formátu JSON. Každá entita musí mít následující podobu:

```
"[int] unikátní id entity":
    {
        "type":[int] typ,
        "properties": [int] bitová maska vlastností,
        *"probability": [int] váha pravděpodobnosti
        "components":
        [
            [object]* pole objektů komponent
        ]
    }
```

Následují parametry jednotlivých komponent:
Statická grafika:

```
    {
        "id": 5,
        "shape": [ [int] x-offset, [int] y-offset,
                   [int] šířka, [int] výška ]
        "imagePath": "[string] QRC cesta k obrázku"
        "style":[int] zařazení do stylu
    }
```

Sprite:

```
{
    "id": 6,
    "spriteConfig":[string]QRC cesta k textovému souboru spritu",
    "shape": [ [int] x-offset, [int] y-offset,
               [int] šířka, [int] výška ]
 }
```

Život:

```
{
    "id": 7,
    "health": [int] počet životů,
    "protectionTime": [int] délka ochrany v milisekunách
}
```

Vstup z klávesnice

```
{
    "id": 9,
    "type": 1
}
```

Projektil:

```
{
    "id": 8,
    "velocity": [int],
    "minVelocity": [int],
    "maxTime": [real],
    "step": [int],
    "antiGravityTime": [int] v milisekundách
}
```

Nebezpečí na dotek:

```
{
    "id": 11,
    "damage": [int] zranění
    "ownerType": [int] typ entity
}
```

Bonus:

```
{
    "id": 10,
    "type": [int] typ bonusu
    "amount": [int] množství (čas ochrany v ms)
}
```

Hlídkovací AI:

```
{
    "id": 9,
    "type": 2,
    "left": [int] počet kroků vlevo
    "right": [int] počet kroků vpravo
}
```

Teleport

```
{
    "id": 12,
    "type": [0/1] vstupní / výstupní brána teleportu
}
```

Pronásledovací AI:

```
{
    "id": 9,
    "type": 3,
    "width": [int] vzdálenost v pixelech na ose x
    "height": [int] vzdálenost v pixelech na ose y
}
```

Střílející AI

```
{
    "id": 9,
    "type": 4,
    "distance": [int] vzdálenost v pixelech na ose y
}
```

Zvukové efekty:

```
{
    "id": 1,
    "soundEffects":
        [
            {
                "type": [int] typu zvuku,
                "path": "[string] qrc cesta ke zvuk. souboru"
            }+
        ]
},
```

Fyzika a akce:

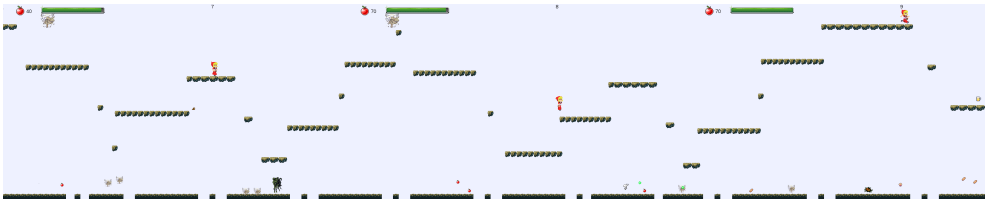
```
{
    "id": 4,
    "initAction": [int] ID výchozí akce,
    "actions":
    [
        {
            "id": 1 (nečinnost)
        },
        {
            "id": 2, (chůze vpravo)
            "stepSize": [int] velikost kroku v px
        },
        {
            "id": 4, (chůze vlevo)
            "stepSize": [int] velikost kroku v px
        },
        {
            "id": 8, (přikrčení)
            "ratio": [0-100] procentuální zmenšení
        },
        {
            "id": 16, (útok na blízko)
            "range": [int] dosah v px na ose x
            "damage": [int] zranění
            "delay": [int] délka pauzy v ms mezi útoky
        },
        {
            "id": 32, (skok-je přidán i pád)
            "velocity": [int] iniciální zrychlení
            "stepSize": [int] velikost kroku v ose x
```

```
                "time": [real] čas skoku
            },
            {
                "id": 64, (útok na dálku)
                "reloadTime": [int] délka v ms pauzy mezi útoky
                "projectileID": [int] ID entity projektilu
            },
            {
                "id": 128, (pád, nepřidávat existuje-li skok)
                "velocity": [-int] iniciální zrychlení
                "stepSize": [int] velikost kroku v ose x
                "time": [real] čas pádu
            }
        ]
    }
```

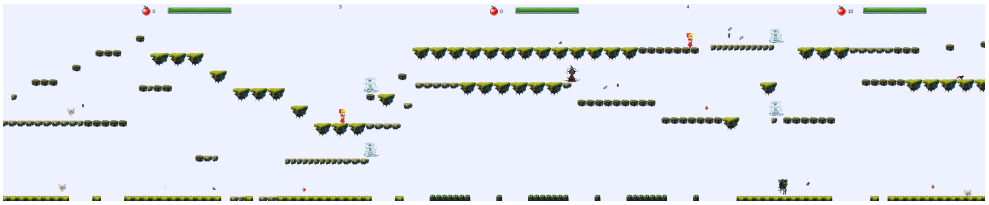
Příloha C

Ilustrace specifických situací ve hře

Obrázek C.1: Screenshot aktuální verze hry

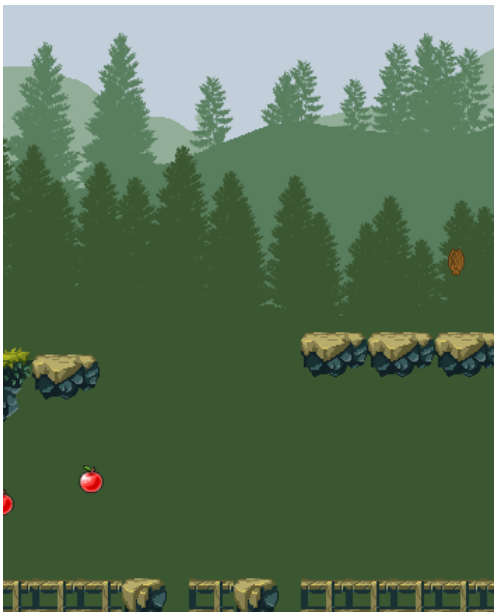


Obrázek C.2: Vygenerovaná část levelu za použití dvou cest



Obrázek C.3: Vygenerovaná část levelu za použití tří cest

Obrázek C.4: Detail na generování pěti cest



Obrázek C.5: Implicitní prolnutí dvou cest – země (most) a alternativní cesty (kamenité platformy)

Obrázek C.6: Zachování diverzity monster předchází tomuto stavu – přeučení na jeden typ

Příloha D

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF