# Základní pojmy a příprava projektu

V této kapitole si vysvětlíme, co znamenají některé základní pojmy a nainstalujeme si nejdůležitější pomůcky, které nám pomohou při tvorbě našeho projektu.

## Příkazová řádka

Jako první bych se měl zmínit o příkazové řádce neboli command line. Je to uživatelské rozhraní, díky kterému může uživatel komunikovat s programy nebo operačním systémem pomocí příkazů. Příkazová řádka se na různých operačních systémech spouští odlišně.

Například v Microsoftu Windows ji spustíme pomocí kláves Windows + R, poté stačí do nově otevřeného okénka napsat zkratku „cmd“ (command), u Linuxu je třeba stisknout klávesy ALT + F2 a následně do nového okénka napsat „konsole“.

## Python

Python je programovací jazyk (nástroj pro zápis algoritmů do počítače), který se objevil poprvé v roce 1991 a jeho autorem je Guido van Rossum. Python je jedním z nejjednodušších programovacích jazyků hlavně díky jednoduchosti syntaxe, a proto je velice vhodný pro začínající programátory. Python je zdarma dostupný pro většinu platforem (Microsoft Windows, Unix, macOS). Většina Linuxů obsahuje Python hned při základní instalaci.

Python má široké možnosti využití, umožňuje tvorbu rozsáhlých aplikací nebo také tvorbu grafického uživatelského rozhraní. Práce v pythonu je velmi produktivní v závislosti na rychlosti psaní a projevuje se hlavně v stručnosti zápisu kódu a snadné použitelnosti většiny typů knihoven. Python sice patří mezi pomalejší programovací jazyky, ovšem rychlost aplikací napsaných v něm je dobrá. jelikož hlavním problémem v tomto hledisku jsou pomalu fungující knihovny, které jsou implementovány v programovacím jazyce C, se kterým python skvěle spolupracuje.

Python lze najít například v 3D grafickém programu Blender, v kancelářském balíku Apache OpenOffice nebo v počitačové hře Civilization IV.

### Instalace Pythonu

Instalace pythonu je na každé počítačové platformě jiná, jak už víme, Linux většinou python obsahuje, je ale vhodné si to zkontrolovat, a to v příkazové řádce pomocí příkazu „**python3 --version**“. Pokud se nám poté objeví „Python“ a číslo některé verze (např. 3.4.7), tak máme python nainstalovaný. Pakliže se objeví „bash: python: command not found“, znamená to, že si musíme python nainstalovat sami. Například tím, že si ho stáhneme na oficiální internetové stránce Pythonu „https://www.python.org/downloads/“.

Microsoft Windows neobsahuje python, a tak bychom měli si ho měli stáhnout ze stejné internetové adresy, jako při případné instalaci na Linuxu.

## Instalace viruálního prostředí

Teď bychom si měli nainstalovat virtuální prostředí. Což je prostředí, které nám zajišťuje, že se v něm všechny počítače budou chovat stejně, díky tomu nebudeme potřebovat instrukce zvlášť pro každý operační systém (Microsoft Windows, Linux). Další výhodou je, že knihovny, které si do našeho virtuálního prostředí nainstalujeme, nebudou zasahovat do našeho operačního systému. Instalace virtuálního prostředí probíhá zpravidla v příkazové řádce. Ovšem při instalaci budeme muset použít příslušné instrukce pro operační systém, který používáme.

V příkazové řádce bychom se měli nejprve přesunout do našeho projektu, kde chceme naše virtuální prostředí (v našem případě Virtualenv) využívat. U Linuxu budeme potřebovat pro instalaci příkazy uvedené níže na obrázku.

Obrázek č. 1 Instalace virtuálního prostředí na Linuxu



Zdroj: Vlastní zpracování

U Microsoftu Windows se instalace liší použitými příkazy. Pip je správce balíčků pythonu, který nám umožňuje nainstalovat jak Virtualenv, tak některé knihovny pythonu.

Obrázek č. 2 Instalace virtuálního prostředí na Windows



Zdroj: Vlastní zpracování

Je nutné dodat, že po ukončení naší práce ve Virtualenvu, bychom toto prostředí měli deaktivovat, a to už stejným příkazem pro všechny operační systémy „deactivate“.

## Knihovna Pyglet

Pyglet je knihovna, která plně podporuje práci s rastrovými obrázky, které sloužit buď ve funkci spritů (pohybující se objekty ve scéně s volitelnou velikostí, rotací a průhledností) nebo je možné rastrové obrázky použít pro vytvoření textur nanášených na trojrozměrná tělesa. Pyglet běží na platformách Microsoft Windows, Linux a macOS.[[1]](#footnote-1)

### Instalace knihovny Pyglet

Instalace knihovny Pyglet probíhá v příkazové řádce, konkrétně ve virtuálním prostředí (Virtualenv), k tomu nám poslouží příkaz „**python -m install pyglet**“. Na konci instalace by nám měla příkazová řádka napsat „Successfully installed pyglet – číslo verze knihovny“.

# Github

Github je webová služba, která podporuje vývoj softwaru díky verzovacímu nástroji Git. Hlavní výhodou je bezplatný webhosting (pronájem prostoru pro webové stránky na cizím serveru) pro open source (otevřené) projekty a možnost spolupráce několika programátorů na stejném projektu, aniž by musely sdílet společnou síť. Mezi další důležité vlastnosti patří například diskuze nad kódem, notifikace o změnách a dokumentace ve formě README (čtimě), která je ve stejné složce jako samotný program. Github byl poprvé spuštěn v roce 2008 a jeho zakladateli byli Tom Preston-Werner, Chris Wanstrath a PJ Hyett. V současné době Github hostuje něco okolo 11 miliónů repozitářů (datová úložiště).

Pří tvorbě jakéhokoliv projektu je vždy vhodné ukládat nové změny do veřejného repozitáře na Githubu, jelikož se vždy budeme moci vrátit k některé ze starších verzí projektu, i když nám nejdřív mohla připadat špatná. Github najdeme na internetové adrese www.github.com.

## Git

Git je nástroj pro správu verzí, to znamená, že zaznamenává veškeré změny souboru v čase, které uživatelé provedou. To je dobré třeba, když potřebujeme zjistit, kdo jako poslední upravil soubor, který náhle přestal vykonávat svou funkci. Git je velice rychlý, a proto je efektivní i při práci s velkými projekty. Oproti ostatním systémům správi verzí, Git ukládá data ve formě snímků projektu, vždy když uživatel uloží stav projektu, tak Git vyfotí všechny soubory v daném čase a uloží odkaz na tento snímek. Veškeré příkazy, ať už pro instalaci Gitu nebo pro samotnou práci s ním, budeme psát do příkazové řádky.

Git používá pro spravování souborů tři základní části: adresář systému Git (Git directory), oblast připravených změn (staging area) a pracovní adresář (working directory).[[2]](#footnote-2)



Obrázek č. 3 Části systému Git

Zdroj: https://git-scm.com/book/cs/v1/Úvod-Základy-systému-Git

Adresář Git si uschovává data a databázi veškerých objektů projektu, je to vlastně adresář souborů, který se zkopíruje, když klonujeme repozitář z jiného počítače.

Oblast připravených změn (staging area) je soubor, který se většinou nachází v adresáři Gitu a obsahuje informace o tom, co bude obsahovat příští revize.

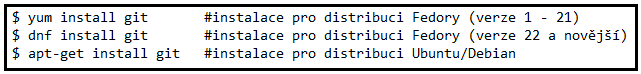
Pracovní adresář obsahuje lokální kopii jedné verze projektu. Tyto soubory jsou staženy ze zkomprimované databáze v adresáři Git a umístěny na disk, kde je může uživatel používat nebo upravovat.[[3]](#footnote-3)

### Instalace Gitu

Instalace Gitu probíhá na každém operačním systému odlišně.

Instalace Gitu v Linuxu probíhá pomocí binárního instalátoru, většinou tak může uživatel učinit pomocí základního nástroje pro správu balíčků, který je součástí dané distribuce.

Obrázek č. 4 Instalace Gitu na Linux



Zdroj: Vlastní zpracování

U instalace na systému Mac lze využít buď grafický instalátor Git, který je volně dostupný na adrese „http://sourceforge.net/projects/git-osx-installer/“, anebo pokud máme nainstalovaný systém MacPorts, tak nám postačí jediný příkaz viz obrázek č. 5.



Obrázek č. 5 Instalace Gitu na Mac

Zdroj: Vlastní zpracování

U Microsoftu Windows je instalace nejjednodušší, jelikož stačí jenom navštívit internetovou adresu „http://msysgit.github.io“ stáhnout si instalační soubor a spustit ho.[[4]](#footnote-4)

### Práce s vlastním Gitovým repozitářem

Projekt v systému Git lze zahájit dvěma základními způsoby. Buď vezmeme existující projekt nebo adresář a importujeme ho do systému Git, nebo můžeme naklonovat již existující repozitář Git z jiného serveru.

Obrázek č. 6 Zahájení práce v Gitu



Zdroj: Vlastní zpracování

Nyní už víme, jak si můžeme vytvořit vlastní gitový repozitář, do kterého budeme moci přidávat nové soubory nebo upravovat ty stávající.

Každý soubor v našem pracovním adresáři může být ve dvou různých stavech: sledován (tracked) a nesledován (untracked). Za sledované jsou označovány soubory, které byly součástí posledního snímku. Mohou být ve stavu změněn (modified), nezměněn (unmodified) nebo připraven k zapsání (staged). Nesledované soubory jsou všechny ostatní, tedy veškeré soubory v našem pracovním adresáři, které nebyly obsaženy ve vašem posledním snímku a nejsou v oblasti připravených změn. Po úvodním klonování repozitáře budou všechny naše soubory sledované a nezměněné, protože jsme právě provedli jejich kontrolu a dosud jsme neudělali žádné změny.[[5]](#footnote-5)

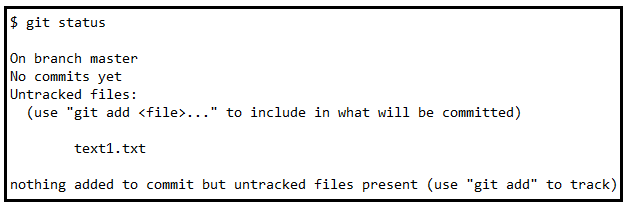
Obrázek č. 7 Stavy souborů v repozitáři



Zdroj:https://git-scm.com/book/cs/v1/Základy-práce-se-systémem-Git-Nahrávání-změn-do-repozitáře

Když se tedy rozhodneme, že do našeho nového prázdného repozitáře přidáme nový soubor a budeme ho chtít zapsat, tak ho nejprve musíme dát sledovat. Nejprve se, ovšem podíváme, jak vypadá náš repozitář teď.

Obrázek č. 8 Git status

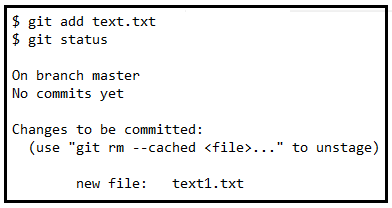


Zdroj: Vlastní zpracování

Tady nám Git říká, že je ve větvi „master“, dále také tvrdí, že zatím nemá žádné snímky, ale vidí jeden nesledovaný (Untracked) soubor konkrétně „text1.txt“. V závorkách nám dokonce radí, jak nechat soubor sledovat (pomocí git add).

Když tak tedy provedeme a poté se znovu podíváme, jak repozitář vypadá, uvidíme, že soubor „text1.txt“ je připraven na revizi (commit). Při práci s Gitem je nutné napsat do příkazové řádky nejen jméno souboru, ale i příponu.

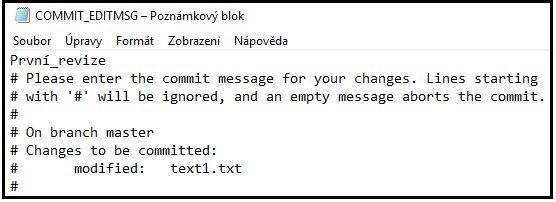
Obrázek č. 9 Git add



Zdroj: Vlastní zpracování

Nyní tedy můžeme provést revizi souboru „text1.txt“, je ovšem důležité podotknou, že revize (commit) uloží snímek souboru v té podobě, v jaké byl, když jsme ho naposledy přidali mezi sledované soubory. To znamená, že kdybychom udělali v souboru další změnu a neprovedli znovu příkaz „git add“, zapsala by se jen ta první změna.

Poté co napíšeme „git commit“ a zmáčkneme Enter, na nás vyskočí okno textového editoru, které bude vypadat podobně jako na obrázku č. 10. Na první řádek editoru napíšeme název revize a uložíme soubor. A revize je hotová.

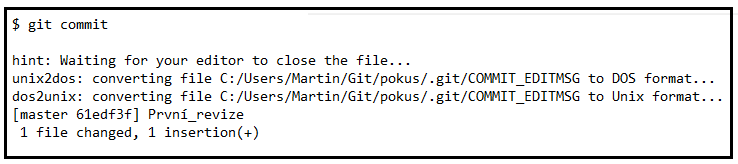


Obrázek č. 10 První revize

Zdroj: Vlastní zpracování

Na konci revize nám Git vypíše, do jaké větve zapsal revizi, v tomto případě do větve „master“. Jaký má revize kontrolní součet SHA-1 (61edf3f), SHA-1 je rozšířená hashovací funkce (funkce, jenž převádí vstupní data na malá čísla). Dále kolik souborů bylo změněno souborů a počet přidaných, či odstraněných řádků.

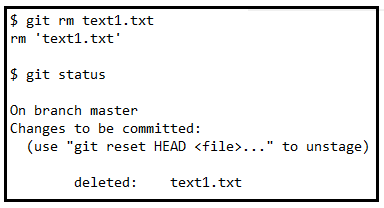
Obrázek č. 11 Výpis revize



Zdroj: Vlastní zpracování

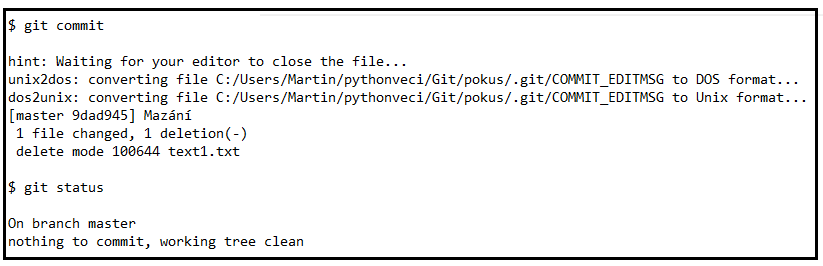
Chceme-li soubor z Gitu odstranit, musíme ho odstranit ze sledovaných souborů (přesněji řečeno, musíme ho odstranit z oblasti připravených změn) a poté zapsat revizi. Je k tomu určen příkaz „git rm“, který soubor odstraní také z našeho pracovního adresáře.[[6]](#footnote-6)

Obrázek č. 12 Mazání souboru



Zdroj: Vlastní zpracování

Po příští revizi (commit) už tam soubor neuvidíme a nebude se ani sledovat.

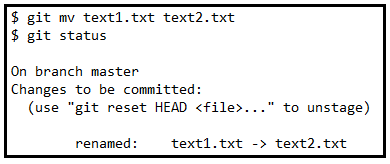


Obrázek č. 13 Kontrola vymazání souboru

Zdroj: Vlastní zpracování

Další důležitou funkcí je přejmenování souboru, to se provádí pomocí příkazu „git mv“. Za tento příkaz napíšeme nejprve staré jméno a po něm i to nové.

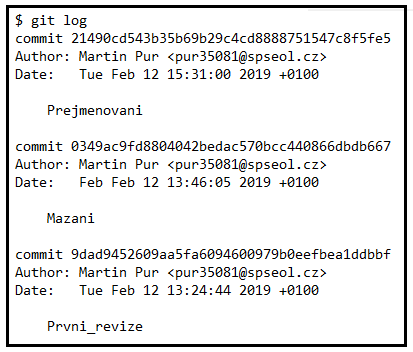
Obrázek č. 14 Přejmenování souboru



Zdroj: Vlastní zpracování

Po vytvoření několika revizí, nebo pokud jste naklonovali repozitář s existující historií revizí, můžete chtít nahlédnout do historie projektu. Nejzákladnějším a nejmocnějším nástrojem je v tomto případě příkaz „git log“, který nám ukáže všechny provedené revize projektu.

Obrázek č. 15 Výpis revizí



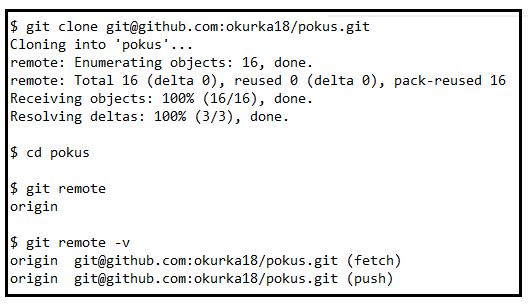
Zdroj: Vlastní zpracování

Git nám vypíše revize v obráceném chronologickém pořadí (poslední revize se zobrazí na začátku). Můžeme vidět, že tento příkaz vypíše všechny revize s jejich kontrolním součtem SHA-1, jménem a e-mailem autora, datem zápisu a zprávou o revizi. [[7]](#footnote-7)

### Práce se vzdáleným Gitovým repozitářem

Teď máme za sebou základní práci s vlastním repozitářem a je tedy čas, naučit se pracovat i s tím vzdáleným. Využijeme to hlavně při spolupráci s více lidmi na libovolném projektu. Už víme, že k naklonování vzdáleného repozitáře nám poslouží příkaz „git clone“. Kdybychom si nebyli jisti, jestli jsme už některý repozitář nenaklonovali, tak se to dá jednoduše zjistit díky příkazu „git remote“, ten nám vypíše jména, která jsme přidělili naklonovaným repozitářům, pakliže jsme žádné jméno nepřidělili měl by vypsat „origin“. Lepší je ovšem k příkazu „git remote“ přidat „ -v“, poté bychom už měli vidět i celou URL adresu.

Obrázek č. 16 Klonování vzdáleného repozitáře

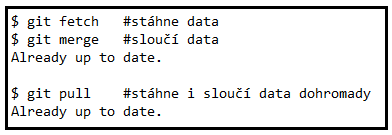


Zdroj: Vlastní zpracování

Příkaz „clone“ automaticky přidá vzdálený repozitář origin za nás. Teď si ukážeme, jak můžeme přidat nový vzdálený repozitář explicitně. Chceme-li přidat nový vzdálený gitový repozitář a zadat zkrácený název, přes který se můžete snadno odkazovat, spusťte příkaz „git remote add <zkrácený název> <url>“.

Nyní bychom se měli zaměřit na vyzvedávání změn ze vzdáleného repozitáře. Když některý z našich kolegů udělá změnu v našem projektu a bude po nás chtít, abychom si ji od něj vyzvedli a podívali se na ni, tak toho docílíme pomocí příkazu „git fetch“.

Měli bychom zmínit, že příkaz „git fetch“ jen stáhne data do našeho lokálního repozitáře. Neprovede ale automatické sloučení (merge) s naší prací, ani nezmění nic z toho, na čem právě pracujete. Sloučení s vaší prací musíme udělat ručně, až to uznáme za vhodné. Druhou možností je použít příkat „git pull“, ten vyzvedne (fetch) data ze serveru, z něhož jsme původně klonovali a automaticky se pokusí začlenit je (merge) do kódu, na němž právě pracujeme.



Obrázek č. 17 Ukládání změn ze vzdáleného repozitáře

Zdroj: Vlastní zpracování

Teď už jen zbývá naučit se posílat vlastně provedené změny do vzdáleného repozitáře. S tím nám pomůže příkaz „git push“. Tento příkaz bude funkční, pouze pokud jsme klonovali ze serveru, k němuž máme oprávnění pro zápis, a pokud tam mezi tím nikdo nic neodeslal. Pokud ve stejnou dobu obsah naklonuje ještě někdo jiný a svou práci odešle na společný server, naše později odesílaná práce bude oprávněně odmítnuta. Nejdříve musíte jeho práci vyzvednout (fetch), zahrnout ji do vaší a teprve potom budete moci vše odeslat. [[8]](#footnote-8)



Obrázek č. 18 Posílání změn do vzdáleného repozitáře

Zdroj: Vlastní zpracování

Dále je nutné říci, že když budeme posílat nějakou změnu do vzdáleného repozitáře, tak tam vlastně pošleme revizi (commit) daného souboru. Musíme tedy před odesláním změn vytvořit jejich commit.

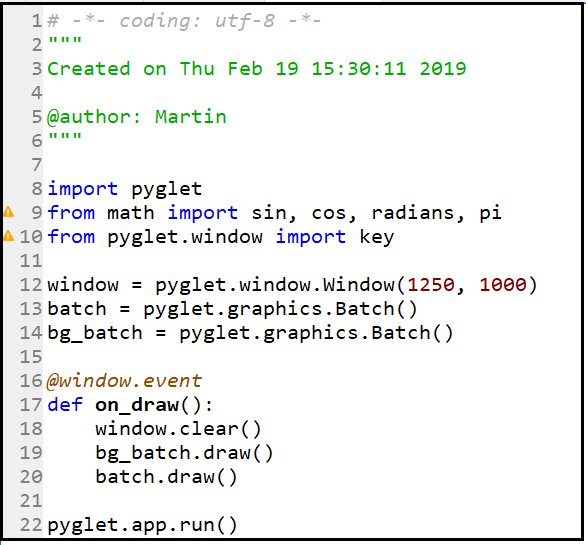
# Hra

Jako hru, kterou budu programovat, jsem si zvolil hru Asteroids, hlavně z důvodu toho, že ji většina lidí zná a ví, jak funguje.

## Asteroids

Ve hře Asteroids budeme používat tři typy objektů. Prvním je raketa, pomocí které budeme mířit na meteority. Druhým typem je meteorit, jenž bude jakýmsi terčem, do něhož se budeme snažit trefit laserem. Poslední typ je laser, ten bude ničit meteority.

Jako první bychom měli připravit okno, v němž se bude všechno odehrávat. To znamená, že si naimportujeme knihovnu pyglet a zní rovnou i klávesy, které budeme potřebovat na ovládání rakety. Rovněž naimportujeme z knihovny math funkce sinus, cosinus, radians a konstantu π, tyto funkce budeme potřebovat při výpočtu souřadnic objektů**.** Dále si zvolíme velikost okna, přidáme metodu pro vykreslování objektů a funkci pro vykreslení pozadí (bg\_batch) celého okna, k němu se dostaneme později. Na konci nesmí chybět příkaz pro spuštění aplikace.



Obrázek č. 19 Nastavení okna

Zdroj: Vlastní zpracování

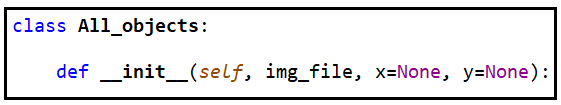
### Třída

Teď je na čase, vytvořit si třídu, v níž budou všechny objekty spolu. Třída je část kódu, ve které mají všechny objekty odlišná data, ovšem metody jsou pro všechny objekty daného typu stejné. Třídy se značí příkazem „class“.

U objektů, vytvořených ve vlastních třídách, je možnost nadefinovat vlastní atributy (vlastnosti), které se uloží k danému objektu třídy. Atributy se značí tak, že se napíše tečka mezi hodnotu a jméno atributu.

Na začátku třídy vždy použijeme inicializační metodu „\_\_init\_\_“, kterou přidáme na začátek třídy a pomocí níž potřebné atributy objektu zavedeme. Této metodě se říká konstruktor.

Obrázek č. 20 Třída All\_objects



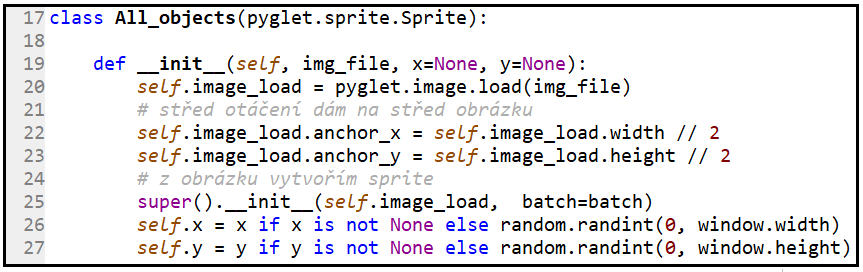
Zdroj: Vlastní zpracování

V naší třídě All\_objects jsou v konstruktoru 4 parametry, první parametr self je povinný u každé funkce ve třídě. Ostatní parametry zajišťují, že každý objekt třídy bude mít atributy img\_file, x a y. Atribut img\_file bude sloužit pro přidělení obrázku objektům. Atributy x a y budou udávat polohu objektů, nyní mají hodnotu „None“, a to proto, že v této třídě budou všechny objekty dohromady a my nechceme, aby všechny byli na stejném místě. Tím pádem si pro každý typ objektu vytvoříme ještě jeho vlastní třídu (podtřídu třídy All\_objects). Ale před tím bychom si měli říci něco o dědičnosti tříd.

### Dědičnost tříd

Dědičnost je jev, který nám umožňuje podědit některé společné atributy (vlastnosti) objektů ze společné třídy v našem případě to bude All\_objects. Když si to převedeme do našeho případu, tak můžeme říci, že raketa, meteority i laser budou mít většinu vlastností odlišnou, ovšem každý objekt bude mít nějakou polohu a vzhled, a to jsou jejich společné vlastnosti. Abychom si ulehčili práci s vypisováním, tak si právě musíme vytvořit třídu, v níž si tyto vlastnosti předdefinujeme. Takové třídě se také říká předek. A třídě, která z předka čerpá informace se říká potomek. U názvu potomka musíme dát na konec, do závorky jméno předka.

Pokud vytváříme nového potomka a potřebujeme přepsat atributy předka, tak k tomu slouží příkaz „super()“. Pozor na to, že volané metodě musíme dát všechny argumenty, které potřebuje kromě self, ten se doplní automaticky.



Obrázek č. 21 Nadefinovaný předek

Zdroj: Vlastní zpracování

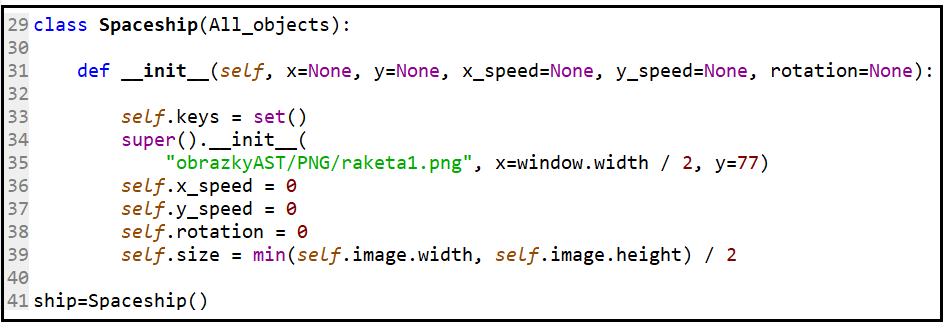
Jak jste si asi všimli, za název třídy All\_objects jsem ještě přidal závorku s textem. Knihovna pyglet využívá při přidělování vzhledu objektům příkaz sprite, což je 2D objekt v pygletu s polohou a obrázkem. Abychom si ještě více ulehčili práci, tak můžeme tuto funkci (pyglet.sprite.Sprite) pygletu podědit do hlavní třídy All\_objects, tím pádem nebudeme muset psát při určování polohy objektu psát dva příkazy (poloha objektu, poloha spritu), ale jen jeden (poloha objektu), jelikož tím pygletu říkáme, že každý objekt bude mít sprite.

Na řádku 20 je rovněž zjednodušení, a to v té formě, že do obecného příkazu pro přiřazeni obrázku vložíme místo cesty k libovolnému obrázku proměnnou img\_file, jíž máme jako atribut konstruktoru třídy. Tudíž si při přidělování obrázků vystačíme s voláním předka pomocí příkazu „super()“.

Na řádcích 26 a 27 je vidět, že když nejsou zadány souřadnice x a y, tak je náhodně vygenerujeme, pomocí knihovny random, kterou bychom si tedy měli naimportovat.

### Raketa

Teď už můžeme přejít k samotné třídě rakety. U konstruktoru rakety budou kromě souřadnic x a y i parametry x\_speed, y\_speed. Ty nám reprezentují posun objektu po obou osách. Nemůže chybět parametr rotation, který značí náklon rakety. Dále si uvnitř konstruktoru uložíme hodnotu velikosti rakety a schováme stisknuté klávesy do množiny „set“, díky nim se bude raketa pohybovat. Na závěr zavoláme třídu Spaceship a tím vytvoříme raketu.



Obrázek č. 22 Třída Spaceship

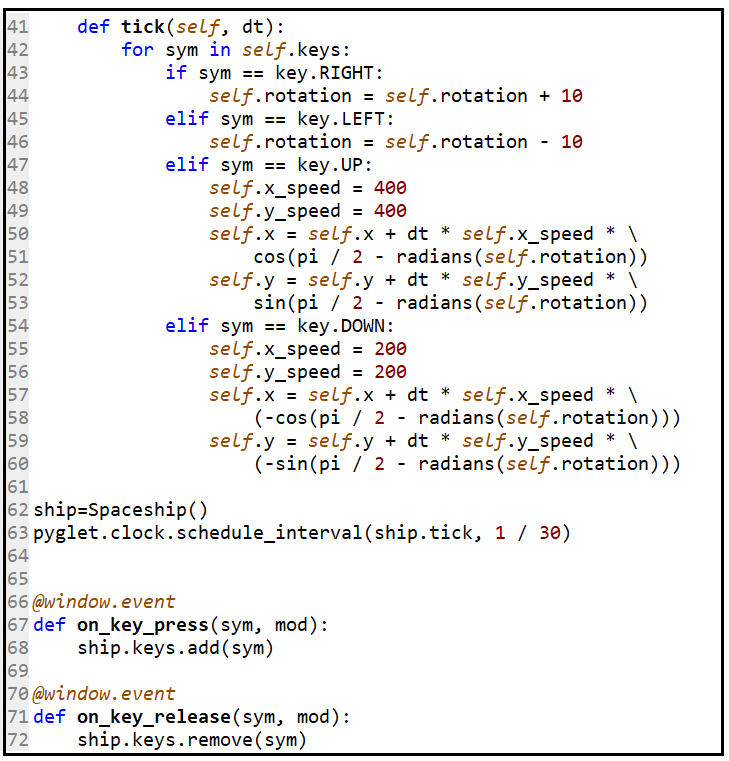
Zdroj: Vlastní zpracování

Teď už máme vytvořenou raketu, ovšem ta ještě není ovladatelná. S tím nám pomůže metoda tick, která obstará veškerou mechaniku lodi. Před tím, než začneme psát kód této metody, bychom měli říci pygletu, že budeme chtít klávesnici používat. Na to existují tzv. „window eventy“, které programu říkají, že budeme klávesy používat.

Zpátky k metodě tick, v ní budeme říkat co se má stát, když se zmáčkne jakákoliv ze 4 šipek na klávesnici. Bude mít dva parametry self a dt, dt bude čas od posledního ticknutí (kontroly zmáčknutí) Otáčení doprava a doleva je velice jednoduché, pakliže zmáčkneme šipku doprava, zvětšíme raketě náklon (rotation) o 10, jestliže ovšem zmáčkneme šipku doleva, tak rotation zmenšíme o 10. Pohyb dopředu a dozadu bude složitější. Jak doufám víte z fyziky dráha je součinem času a rychlosti. K původní poloze na obou souřadnicích připočteme novou dráhu a vynásobíme vzorcem pro směr letu, který využívá rotation.

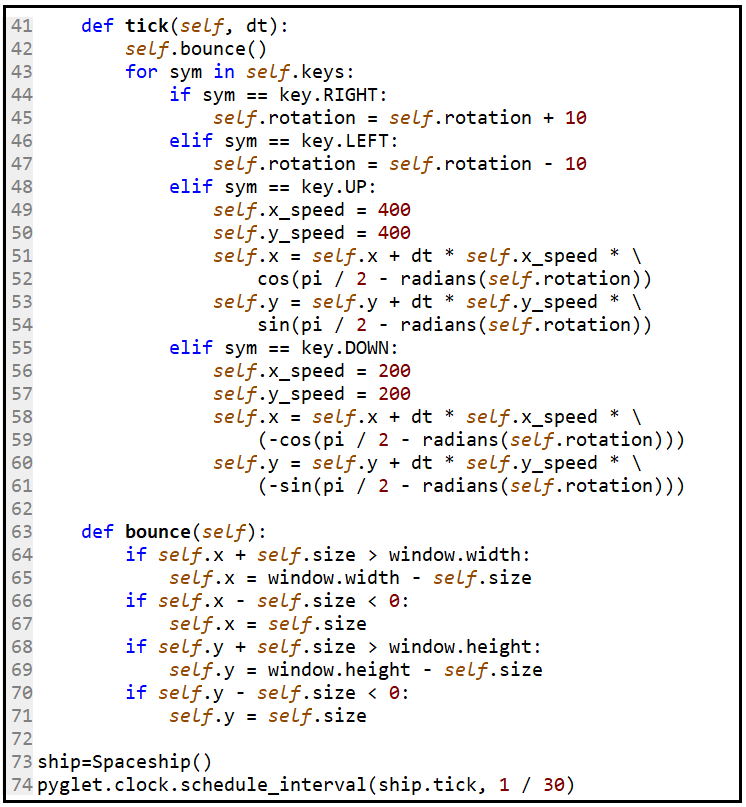
Pod metodu tick nám zbývá dát časovač, který bude kontrolovat, zda jsme zmáčkli některou z kláves a zároveň poslouží i jako čas pro výpočet nové polohy.

Obrázek č. 23 Metoda tick u třídy Spaceship



Zdroj: Vlastní zpracování

Když to spojíme s dříve vytvořenými částmi a spustíme program, tak už budeme moci raketu ovládat. Je tu ještě jeden problém, pokud necháme raketu letět pořád jedním směrem, zjistíme, že může vyletět z okna. Tento problém vyřešíme funkcí bounce. V této funkci bude, že poloha rakety na ose x nesmí být větší než šířka okna a zároveň nesmí být menší než nula. Podobně to bude i s osou y, jen s tím rozdílem, že nebudeme porovnávat polohu rakety s šířkou okna, ale s výškou. Při těchto výpočtech budeme používat i rozměr rakety, jelikož jinak by jedna její polovina byla mimo okno a druhá v okně. Teď už jen stačí zavolat funkci bounce v ticku rakety. Nyní zbývá to jen poslat na github.



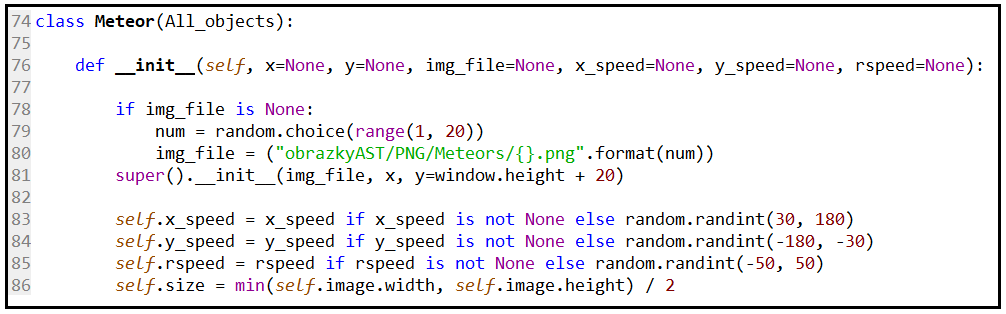
Obrázek č. 24 Funkce bounce u třídy Spaceship

Zdroj: Vlastní zpracování

### Meteority

Další v pořadí jsou meteority, ty budou přilétávat z horní části okna a jejich třída bude rovněž potomkem All\_objects. Meteority budou mít stejně jako raketa parametry x, y, x\_speed, y\_speed a navíc bude mít parametr rspeed, který značí rychlost rotace meteoritu. Jelikož v našem okně nebudou létat pořád stejné meteority, tak si jejich vzhled musíme dát jako další parametr konstruktoru. Nejprve budeme losovat mezi čísly 1-16 a poté přidělíme meteoritu vzhled odpovídající danému číslu ve složce s obrázky. Teprve poté zavoláme předka a přepíšeme spawnovací polohu (místo, kde se meteorit objeví jako první) a dále musíme doplnit parametr img\_file.

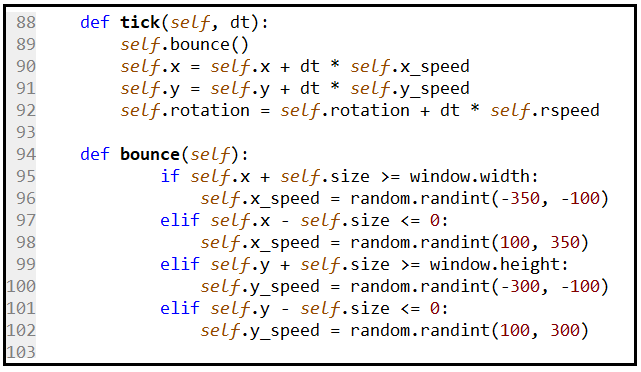
Následně pomocí knihovny random přidělíme každému meteoritu vlastní rychlost na obou osách. I u meteoritů si uložíme jejich velikost, jíž využijeme při zjišťování polohy mezi raketou a meteoritem. Díky knihovně random můžeme každému meteoritu přidělit i rychlost rotace (otáčení). Třídu Meteor nebudeme přímo volat, na to si později uděláme funkci.



Obrázek č. 25 Třída Meteor

Zdroj: Vlastní zpracování

Aby se i meteority mohli pohybovat, tak musí mít také metodu tick. Počítání rychlosti zde bude jednodušší než u rakety, poněvadž meteority nemají, žádný bod, z něhož bychom něco stříleli. U meteoritů nám hrozí, že mohou vyletět z okna, to nám až, tak nevadí, jelikož pro ně můžeme použít jejich metodu bouce. Když meteorit narazí do některé ze stran okna, tak jeho rychlost na příslušné ose změníme.



Obrázek č. 26 Metody tick a bounce u třídy Meteor

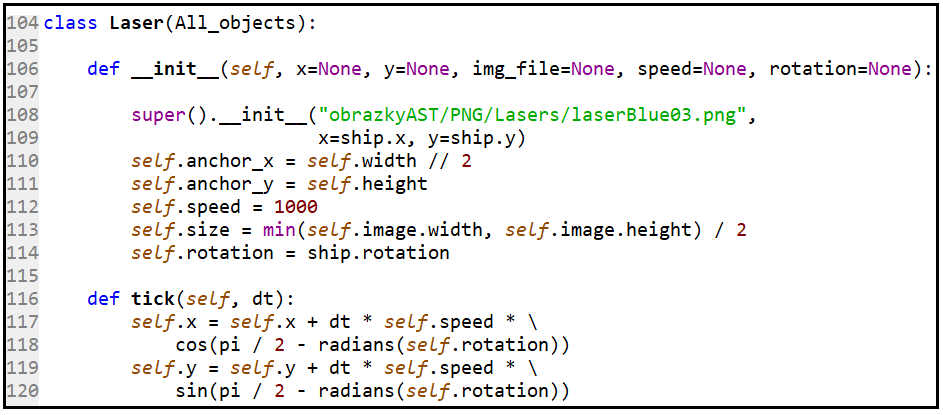
Zdroj: Vlastní zpracování

### Laser

Jako poslední typ objektu nám zbývá laser, jimž budeme ničit meteority. Laser je také potomkem třídy All\_objects a jeho konstruktor bude mít atributy x, y, img\_file a rotation. Pomocí předka laseru přidělíme obrázek a pozici na obou osách soustavy. Dále nastavíme jeho rychlost na 1000. Kotvu (anchor) laseru nastavíme na střed jeho šířky a na hodnotu jeho celkové výšky. Stejně jako u předešlých typů objektů si uložíme jeho velikost.

Jelikož se laser vystřeluje z rakety, bude mít stejné souřadnice a rotation jako ona. Laser bude tedy mít i stejný výpočet pro novou pozici v jeho metodě tick jako má raketa.

Obrázek č. 27 Třída Laser



Zdroj: Vlastní zpracování

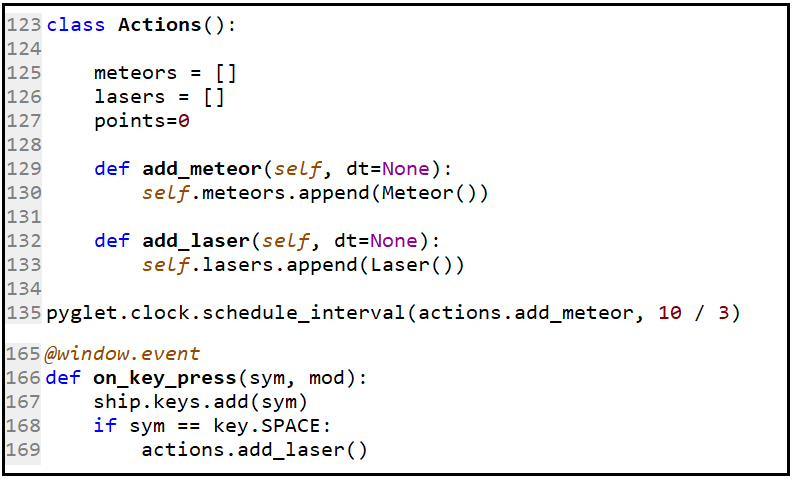
Přidávání laseru bude stejně jako u meteoritů probíhat v nově vytvořené funkci, k ní se dostaneme později.

### Třída Actions

Třída Actions nám bude sloužit, pro většinu událostí, které se v oknu mohou stát například srážka rakety s meteoritem, přidávání meteoritů a laserů do okna, ničení meteoritů pomocí laseru. Tato třída nebude potomkem All\_objects, tudíž není na ničem závislá.

Měli bychom začít tím, že si vytvoříme dva seznamy pro meteority (meteors) a lasery (lasers). Ke každému z nich vytvoříme funkci, která bude přidělovat tyto typy objektů do příslušných seznamů. Ještě si zde nastavíme počítání dosažených bodů na nulu. Funkci add\_meteor budeme opakovaně volat přes časovač na konci programu a funkci add\_laser voláme pouze tehdy, zmáčkne-li uživatel na klávesnici mezerník.

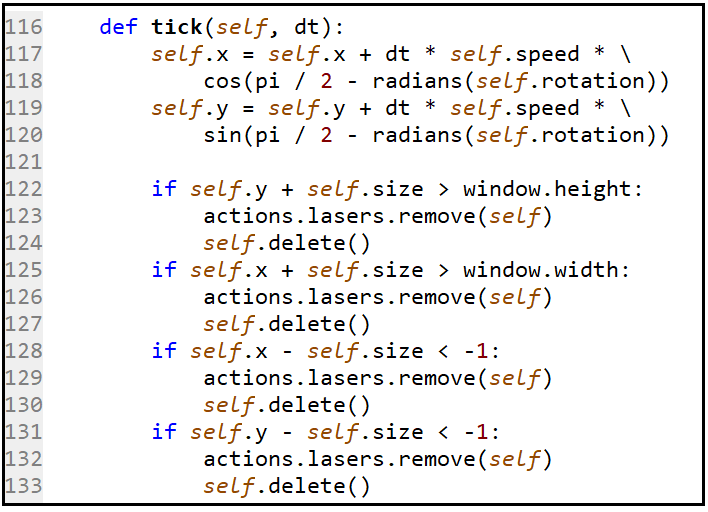
Obrázek č. 28 Třída Actions



Zdroj: Vlastní zpracování

Abychom nezatěžovali program, tak budeme mazat lasery, které vyletí z okna a to tak, že je odebereme ze seznamu lasers a následně objekt smažeme. Kontrolovat to budeme u ticku laseru.

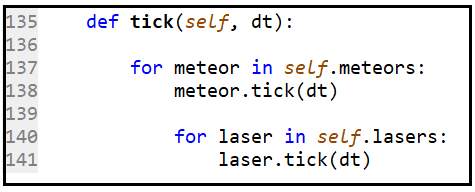
Obrázek č. 29 Mazání nepotřebných laserů



Zdroj: Vlastní zpracování

Dále abychom nemuseli u tříd (Meteor a Laser) volat jejich metody tick na různých místech a k nim přidávat časovače, tak si vytvoříme v této třídě velkou metodu tick, v níž se budou volat jednotlivé metody tick pro obě třídy.

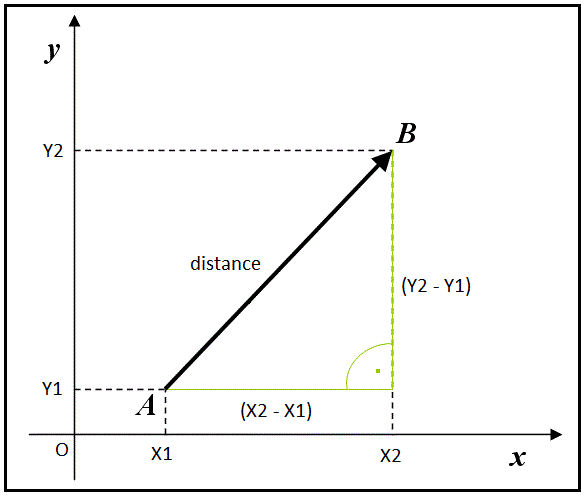
Obrázek č. 30 Metoda tick u třídy Actions



Zdroj: Vlastní zpracování

Měli bychom si také nadefinovat co se má stát, když raketa narazí do meteoritu. Bude toho více, a tak si na to raději vytvoříme funkci colision. V této funkci musíme zastavit actions.tick, ship.tick a samotnou raketu. Raketu zastavíme tak, že nastavíme její x\_speed a y\_speed na nulu.

Až budeme mít funkci hotovou, tak začneme procházet jednotlivé meteority v seznamu meteors a zjišťovat, zda do některého z nich nenarazila raketa. Vzdálenost získáme pomocí Pythagorovi věty.

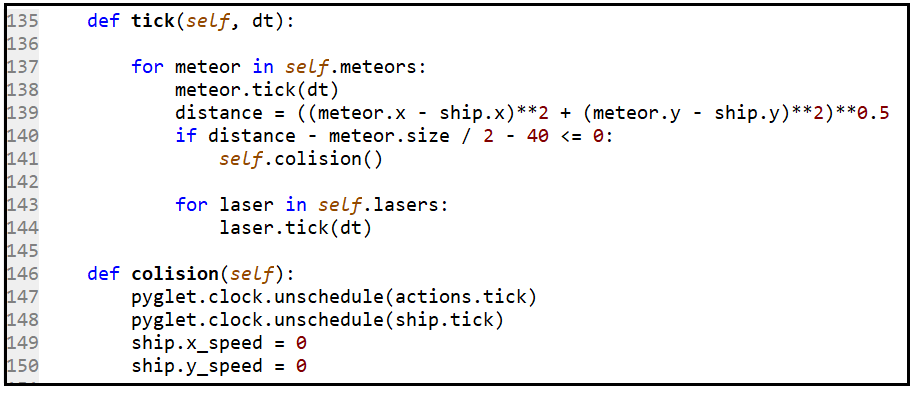


Obrázek č. 31 Výpočet vzdálenosti mezi objekty

Zdroj: Vlastní zpracování

Jen místo hodnot x1, x2, y1, y2 použijeme meteor.x, ship.x, meteor.y a ship.y. Od této vzdálenosti odečteme poloviční velikost meteoritu a konstantu (45). Pokud bude výsledek menší nebo roven nule, tak zavoláme funkci colision.

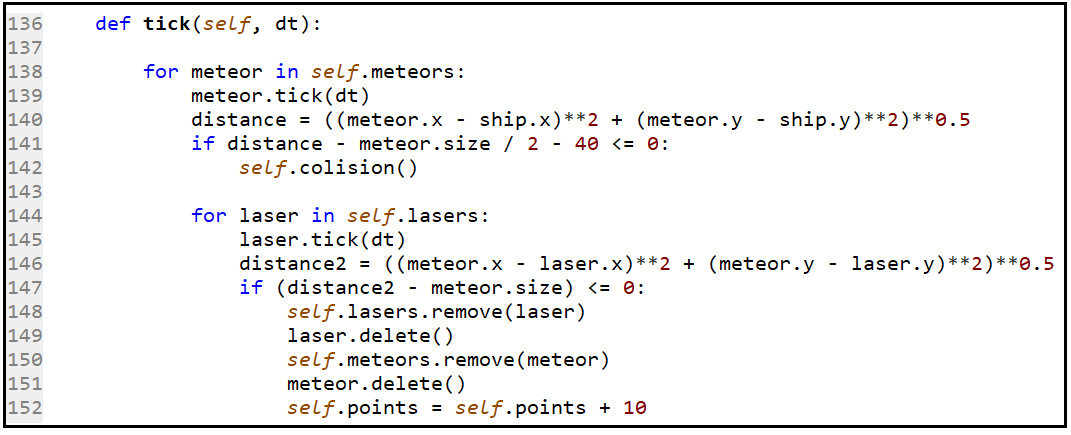
Obrázek č. 32 Kolize meteoritu a rakety



Zdroj: Vlastní zpracování

Velice podobně vytvoříme vzorec pro ničení meteoritů laserem, jen místo souřadnic rakety použijeme souřadnice laseru. Když laser narazí do meteoritu, tak oba objekty smažeme a přičteme hráči 10 bodů (za každý sestřelený meteorit). Abychom mohli meteorit a laser vymazat musíme je nejdříve odebrat z jejich seznamů. Výsledný tick, proto bude vypadat jako na obrázku č. 33.

Obrázek č. 33 Mazání meteoritů laserem

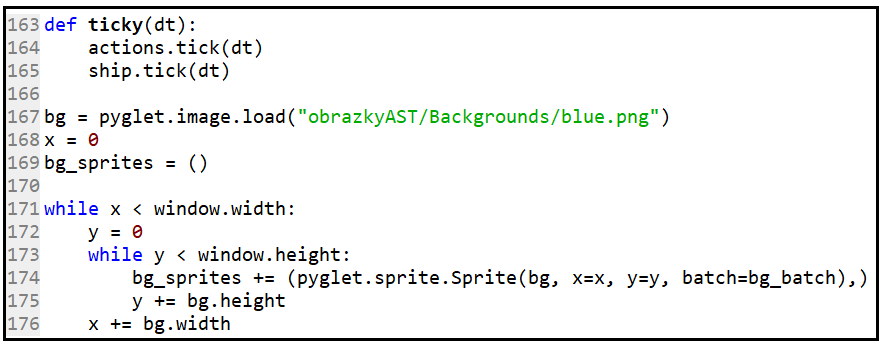


Zdroj: Vlastní zpracování

Abychom v našem programu nemuseli volat zvlášť actions.tick a ship.tick, tak si je zaobalíme do společné metody, která se bude jmenovat „ticky“ a v níž budou volány obě zmíněné metody tick.

Aby se děj hry neodehrával na okně s černým pozadím, tak nastavíme našemu oknu vlastní vzhled. K tomu nám poslouží funkce knihovny pyglet „pyglet.image.load“, poté nastavíme souřadnici obrázku na ose x na nulu. Dále budeme přidávat sprite obrázku za sebe, dokud se nedostaneme na x-ovou souřadnici okna, pak provedeme stejnou věc s osou y (přidávat obrázky na sebe, dokud nedosáhnou výšky okna).

Obrázek č. 34 Metoda ticky a vzhled pozadí



Zdroj: Vlastní zpracování

Veškeré obrázky, které jsem ve hře použil, nakreslil Kenney Vleugels a jsou volně dostupné na internetové adrese „https://opengameart.org/content/space-shooter-redux“.

1. https://mojefedora.cz/multimedialni-knihovna-pyglet-prace-s-rastrovymi-obrazky-a-sprity/ [online] [cit. 2019-01-15] [↑](#footnote-ref-1)
2. https://git-scm.com/book/cs/v1/Úvod-Základy-systému-Git [online] [cit. 2019-03-02] [↑](#footnote-ref-2)
3. https://git-scm.com/book/cs/v1/Úvod-Základy-systému-Git [online] [cit. 2019-03-02] [↑](#footnote-ref-3)
4. https://git-scm.com/book/cs/v1/Úvod-Instalace-systému-Git [online] [cit. 2019-03-03] [↑](#footnote-ref-4)
5. https://git-scm.com/book/cs/v1/Základy-práce-se-systémem-Git-Nahrávání-změn-do-repozitáře [online] [cit. 2019-03-03] [↑](#footnote-ref-5)
6. https://git-scm.com/book/cs/v1/Základy-práce-se-systémem-Git-Nahrávání-změn-do-repozitáře [online] [cit. 2019-03-08] [↑](#footnote-ref-6)
7. https://git-scm.com/book/cs/v2/Základy-práce-se-systémem-Git-Zobrazení-historie-revizí [online] [cit. 2019-03-08] [↑](#footnote-ref-7)
8. https://git-scm.com/book/v2/Základy-práce-se-systémem-Git-Práce-se-vzdálenými-repozitáři [online] [cit. 2019-03-09] [↑](#footnote-ref-8)