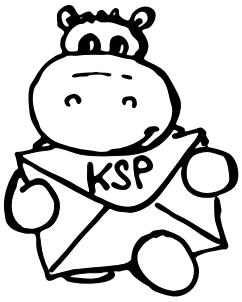
# Korespondenční Seminář z Programování

**33. ročník** **KSP** **Duben 2021**

## *Milí řešitelé, řešitelky a řešitelčata!*




Právě máte před sebou pátou sérii 33. ročníku KSP-H, která uzavírá tento ročník. Jinými slovy právě máte před sebou poslední možnost zisku bodů, které vám mohou pomoci dostat se na **podzimní soustředění** počátkem září.

Naleznete zde **4 normální úlohy**, **netradiční bonusovou úlohu** a pak také pokračování **seriálu o počítačové grafice** a k němu ještě přibalený **text o integrálech** z naší Encyklopedie. I předchozí díly seriálu můžete **odevzdávat až do konce této série**, takže vůbec nevadí, že jste je dosud nestihli. Detaily naleznete u zadání seriálu. Připomínáme, že se do výsledkové listiny započítávají **všechny úlohy mimo bonusové** a body se již nepřepočítávají podle počtu vyřešených sérií.

**Odměny & na Matfyz bez přijímaček**

Za úspěšné řešení KSP můžete být **přijati na MFF UK bez přijímacích zkoušek**. Úspěšným řešitelem se stává ten, kdo získá za celý ročník (této kategorie) alespoň 50 % bodů. Také každému řešiteli, který v tomto ročníku z každé série dostane alespoň 5 bodů, darujeme KSP propisku, blok, placku a možná i další překvapení.

**Termín série:** **neděle 30. května 2021 ve 32:00 (tedy další ráno v 8:00)**

**Odevzdávání:** Přes web na adrese *https://ksp.mff.cuni.cz/submit/*

**Značky úloh:** Lehčí úloha (či její část) vhodná pro začátečníky; Praktická open-data úloha; Úloha, u které doporučujeme začíst se do kuchařky; Seriálová úloha

**Odměna série:** Každému, kdo z této série **získá alespoň 42 bodů**, pošleme sladkou odměnu.

## Pátá série třicátého třetího ročníku KSP

### 33-5-1 Šotek a obrazy — 11 bodů

Ve věhlasné *Galerii uspořádaného umění* jsou k vidění různorodá umělecká díla v tom přesně správném pořadí, v jakém by je měl návštěvník vidět. Kurátor muzea je zvláště pyšný na řadu obrazů v hlavní místnosti, které po mnoha bezesných nocích uspořádal do dokonalého pořadí. Aby bylo pořadí jasné, tak navíc ještě obrazy označil štítky s celými čísly od 1 do $N$.

V noci ale do galerie přišel šotek a čísla na obrazech prohnal svoji oblíbenou funkcí $f$ – štítek, kde původně bylo číslo $k$, přepsal na $f(k)$. A aby kurátor nemohl čísla zase lehce přepsat zpátky, tak pak šotek obrazy ještě setřídil podle změněných štítků.

Ráno zděšený kurátor vyslechl nočního hlídače, kterému se naštěstí povedlo najít papírek, na který si šotek svoji oblíbenou funkci napsal. Byl to polynom pátého stupně (jako třeba $f(x) = -2x^5 + 4x^4 + x^2 + 256x - 7$).

Nyní by od vás potřeboval kurátor pomoci. Vymyslete algoritmus, který dostane na vstupu zadaný polynom pátého stupně a posloupnost $N$ čísel, kterými jsou nyní obrazy označené (což jsou všechno funkční hodnoty zadaného polynomu pro celá $x$ od 1 do $N$), a vydá *permutaci*, kterou kurátor vrátí obrazy do původního pořadí.

Pod *permutací* si v tomto případě představujeme výstup typu „1. obraz přesuň na 5. pozici, 5. obraz na 3. pozici, 3. obraz na 1. pozici, 2. obraz na…“. Slibujeme, že si šotek vybral hezkou funkci a tak jsou všechna pozměněná čísla obrazů celá a unikátní.

*Upozornění:* Plné body získáte pouze za řešení, které poběží v lineárním čase vzhledem k $N$ i v nejhorším případě (což třeba řešení s použitím hešování splňovat asi nebude).

### 33-5-2 Zakázaná písmena — 10 bodů

V jedné nejmenované, ale velmi pokrokové zemi se rozhodli zakázat některá písmena (pravděpodobně protože se protivila typografickému cítění některých skupin obyvatel, ale přesný důvod už si vlastně nikdo nepamatuje).

Pro každé zakázané písmeno byla vyhlášena oficiální posloupnost jiných písmen a číslic, kterou se mají všechna použití zakázaného písmene nahradit (např. všechna `A` nahradit za `cd007xD`). Rozlišujeme velikost, tedy např. `D` a `d` považujeme za různá písmena. Avšak stále se nedařilo uspokojit každého obyvatele a zákazů přibývalo a přibývalo… až byla nakonec zakázána všechna písmena, která se v této zemi používala (a bylo dovoleno používat jen číslice).

Váš program dostane na vstupu seznam zákazů (dvojic zakázané písmeno + náhrada) a starý text obsahující spoustu zakázaných písmen. Vaším úkolem bude všechna nahradit za jejich oficiálně uznávané náhrady. Má to však háček: protože jak zákazy přibývaly, nikomu se nechtělo aktualizovat staré zákazy, spousta náhrad obsahuje dnes už zakázaná písmena. Nahrazování zakázaných písmen je tedy třeba provádět opakovaně, dokud v textu nějaká zbývají. Na konci vám zůstane text, který obsahuje pouze číslice.

Alespoň vám můžeme slíbit, že v nahrazovacích pravidlech neexistuje cyklus – tedy se nemůže stát, že např. začnete s písmenem `A`, nějakou dobu nahrazujete a v textu se znovu objeví nějaké `A` – to bychom pak mohli pokračovat donekonečna. Nahrazovací pravidla jsou také jednoznačná – pro každé zakázané písmeno existuje jen jedna náhrada.

Protože výsledný text může být hodně dlouhý, budeme od vás chtít vypsat pouze 10 znaků od nějaké konkrétní pozice.

Toto je praktická open-data úloha. V odevzdávacím systému si necháte vygenerovat vstupy a odevzdáte příslušné výstupy. Záleží jen na vás, jak výstupy vyrobíte.

*Formát vstupu:* Na prvním řádku dostanete výchozí text – posloupnost písmen a číslic (neobsahuje mezery ani žádné jiné speciální znaky) délky nejvýše 100 znaků.

Na druhém řádku dostanete dvě čísla oddělená mezerou: $Z$ udávající počet zakázaných písmen a $P$ udávající pozici, od které chceme vypisovat výstup. Na každém z dalších $Z$ řádků je popis jednoho zákazu. Ten se skládá ze zakázaného písmene (velké nebo malé písmeno), mezery a předepsané náhrady (posloupnost písmen a číslic bez mezer).

*Formát výstupu:*

Na výstup vypište jeden řádek obsahující 10 číslic, které se nachází ve výsledném textu (po provedení všech náhrad) na pozicích $P$ až $P+9$ (znaky číslujeme od nuly, tedy pro $P = 0$ bychom chtěli vypsat prvních 10 číslic ze začátku textu). Slibujeme, že těchto 10 číslic vždy bude existovat – nedostanete např. $P$, které by bylo tři znaky před koncem textu. Délka výsledného textu bude menší než $2^{63}$ (tedy $P < 2^{63} - 10$).

| *Ukázkový vstup:* | *Ukázkový výstup:* |
|---|---|
| `Aa44`<br>`2 1`<br>`A 33aa`<br>`a 101` | `3101101101` |

Nahrazování by mohlo probíhat například takto: `Aa44` → `33aaa44` → `3310110110144`.

## 33-5-3 Těžký vozík — 10 bodů

Jste ve skladu plném různých beden. Vaším úkolem je co nejrychleji převézt vozík na určité místo. Vozíkem se sice dá jezdit poměrně rychle, ale je opravdu těžký a kvůli tomu je obtížné ho tlačit (a nebo brzdit). Zvládne se tedy pohybovat pouze s omezeným zrychlením.

Skladiště si představte jako čtverečkovou mřížkou $N \times M$. Každé políčko je buď volné, nebo blokované bednou. Dále dostanete zadanou původní pozici vozíku a cíl, kam se musí vozík dostat. Na začátku i na konci jeho cesty musí mít nulovou rychlost.

Vozík se pak pohybuje v tazích. Na začátku tahu má nějakou vertikální a horizontální rychlost $u$ a $v$ z předešlého tahu. Poté lze vozík zrychlit nebo zbrzdit – obě dvě hodnoty $u$ i $v$ můžete zvýšit nebo snížit maximálně o 1 (klidně i současně). Poté se vozík posune o $u$ políček ve vertikálním směru a o $v$ v horizontálním.

Políčka po cestě ovšem musí být volná. Pro každý tah musí platit, že obdélník (rovnoběžný s mřížkou) s protilehlými vrcholy na políčkách, kde vozík začíná a končí, musí být prázdný (tedy nesmí obsahovat políčko s bednou).

Vaším úkolem je vymyslet algoritmus, který najde nejrychlejší cestu (co do počtu tahů) ze startovního do cílového políčka.

## 33-5-4 Nevěřící Bob — 14 bodů

Alice vymyslela úžasný algoritmus, který dokáže během okamžiku zjistit, zda zadané obrovské číslo je prvočíslem. Hned se jde pochlubit Bobovi. Ten vymýšlí jedno číslo za druhým a pokaždé dostane odpověď, že to není prvočíslo. To už mu přijde značně podezřelé, tak se Alici směje, že napsat program, který pokaždé odpoví NE, by uměl i snáz. Alice na něj ale snadno vyzrála: upravila program tak, aby pro každé složené číslo $x$ své NE doprovodil i nějakým dělitelem čísla $x$ (netriviálním, tedy různým od 1 a od $x$). Bob sice neumí rychle testovat prvočíselnost, ale dělení mu docela jde, takže si pokaždé snadno ověří, že výsledek programu je netriviálním dělitelem zadaného čísla.

Ponecháme Alici s Bobem jejich osudu ... a také úvahám o tom, jak by Alice mohla Boba přesvědčit i o odpovědi ANO. Místo toho se na vzniklou situaci podíváme obecněji.

Alicin algoritmus je *dokazovatel* – pro zadaný *vstup* vydá jednak *výstup* a jednak nějakou dodatečnou informaci, které se říká *certifikát* nebo také *důkaz.*

Bob neumí ze vstupu spočítat výstup, ale má jiný algoritmus – *ověřovatel.* Ten dostane vstup, výstup a důkaz a odpoví, jestli důkaz schvaluje.

Chceme přitom, aby pro každý vstup spočítal Alicin dokazovatel jak správný výstup, tak příslušný důkaz, a Bobův ověřovatel tuto dvojici výstup + důkaz schválil. Naopak pokud je výstup špatně, všechny domnělé důkazy musí Bobův ověřovatel spolehlivě zamítat. Kombinaci dokazovatele a ověřovatele říkáme *důkazový systém* pro danou úlohu.

Všimněte si podobnosti s dokazováním matematických vět: Pokud je věta pravdivá, má k ní existovat důkaz, který nás o tom přesvědčí. A naopak k nepravdivé větě neexistuje nic, co bychom jako důkaz uznali.

Na rozdíl od matematiků nás ale zajímá i efektivita dokazování: chceme, aby důkazy byly co nejkratší a aby je bylo možné ověřit co nejrychleji. Zejména rychleji, než by si Bob správný výstup našel sám.

Vymyslete důkazové systémy pro následující tři úlohy:

- *Různost prvků.* Je dána posloupnost čísel $x_1, \ldots, x_n$. Navrhněte důkaz délky $\mathcal{O}(n)$, pomocí nějž půjde v čase $\mathcal{O}(n)$ ověřit, zda jsou všechna $x_i$ navzájem různá.
- *Nejkratší cesta.* Je dán souvislý neorientovaný graf s $n$ vrcholy a $m$ hranami. Jeden z vrcholů je určen jako start, jiný jako cíl. Hrany jsou ohodnocené přirozenými čísly. Chceme najít nejkratší cestu ze startu do cíle a důkaz, pomocí nějž půjde v čase $\mathcal{O}(n+m)$ ověřit, že cesta je nejkratší.
- *Editační vzdálenost.* Jsou dány dva řetězce $\alpha$ a $\beta$. Chceme najít nejkratší možnou posloupnost editačních operací, která z $\alpha$ vyrobí $\beta$. Editační operace jsou přidání jednoho znaku, smazání jednoho znaku, případně změna jednoho znaku na jiný. Předpokládejte, že délka optimální posloupnosti operací je mnohem menší než délka obou řetězců.

Pro výpočet editační vzdálenosti se může hodit nahlédnout do kuchařky o dynamickém programování (kapitola o nejdelší společné podposloupnosti).

## 33-5-X1 — 10 bodů

Toto je praktická open-data úloha. V odevzdávacím systému si necháte vygenerovat vstupy a odevzdáte příslušné výstupy. Záleží jen na vás, jak výstupy vyrobíte.

*Výše uvedené zadání úlohy je kompletní a přesně takové, jaké má být. Žádná část mu nechybí ani nepřebývá.*

## 33-5-S Globální iluminace a path tracing 15 bodů

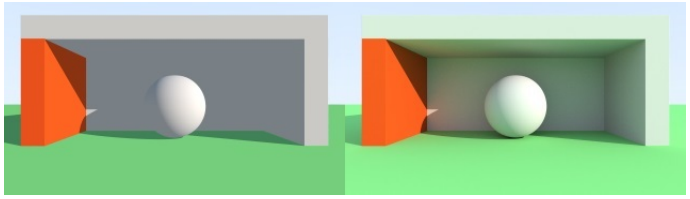
*Toto je seriálová úloha, která navazuje na podobné úlohy v minulých sériích. Pokud jste předchozí díly seriálu neřešili, pro pochopení tohoto dílu je dobré si je nejméně přečíst. A pokud si chcete úlohy z minulých dílů také naprogramovat, stále za ně můžete získat polovinu bodů.*

V minulých dílech jsme zmínili globální iluminaci – efekt, kdy se z libovolného objektu, na který dopadá světlo, také stává pomyslný zdroj světla jen tím, že dopadající světlo dále odráží. Toto odražené světlo má velký vliv na vzhled scény a je klíčovou součástí fotorealistického vykreslování. V tomto díle si ukážeme způsob, jak alespoň přibližně (a pomalu) spočítat i nepřímé difúzní osvětlení.

Pokud máte notebook, který má dvě GPU (integrované a dedikované), nyní je ten správný čas se ujistit, že váš prohlížeč používá to rychlejší z nich.

Také budeme potřebovat jedno matematické kladivo: integrály. Možná je už znáte, ale pokud ne, vše potřebné pro tento díl se dočtete v článku o integrálech v naší encyklopedii.[1]

Níže je motivační obrázek. Vlevo je ambientní světlo dělané tak, jak jsme si ukázali v minulých dílech. Je skoro konstantní, protože se používá barva téměř jednobarevné oblohy čtená ve směru normály povrchu. Vpravo je výsledek realistické simulace difúzních odrazů, jak si ji ukážeme nyní.



### Zobrazovací rovnice

Doteď jsme počítali jen s jediným zdrojem světla, a to se sluncem, takže stačilo poslat směrem ke slunci „stínový“ paprsek a aplikovat Lambertův zákon spolu s Phongovým osvětlovacím modelem. V reálné scéně by ale na toto místo dopadalo i světlo z oblohy nebo světlo odražené ze zbytku scény. My toto světlo velmi tupě aproximovali konstantní hodnotou „ambientního“ světla.

Náš postup není zrovna obecný. Pojďme tedy odvodit nějaký obecný vztah pro počítání osvětlení, který zahrnuje jak slunce, tak oblohu a odražené ambientní světlo.

Co vlastně chceme spočítat? Pro nějaké místo scény nás zajímá, kolik z něj odejde světla konkrétním směrem, třeba do naší kamery. Toto odchozí světlo budeme značit $L_o$ ($L$ jako *light*, $o$ jako *outgoing*). Směr odchozího světla bude $\omega_o$.

Také potřebujeme vědět, kolik světla do tohoto místa přichází. Pro směr $\omega_i$ budeme značit příchozí světlo funkcí $L_i(\omega_i)$ ($i$ jako *incoming*). Toto světlo bude samozřejmě modulováno Lambertovým zákonem.

Poté potřebujeme nějak modelovat odrazové vlastnosti materiálu v daném místě scény. Tedy chceme funkci, která nám řekne, kolik příchozího světla ze směru $\omega_i$ se odrazí směrem $\omega_o$ a jak se zabarví. Této funkci se anglicky říká *bidirectional reflectance distribution function*, zkráceně BRDF.

Poslední složkou bude světlo, které materiál sám vyzáří směrem $\omega_o$. To budeme značit $L_e(\omega_o)$ ($e$ jako *emission*).

Nyní vše poskládáme dohromady. Odchozí světlo se bude rovnat vyzářenému světlu, plus odraženému světlu od zbytku scény. Odražené světlo spočítáme tak, že vyhodnotíme BRDF pro každý možný směr příchozího světla, tedy zintegrujeme BRDF přes polokouli viditelnou z daného bodu scény. Tuto polokouli budeme značit $\Omega$. Dostaneme tedy tohle:

$$L_o(\omega_o) = L_e(\omega_o) + \int_\Omega L_i(\omega_i)\mathrm{brdf}(\omega_i, \omega_o)(n \cdot \omega_i)\ \mathrm{d}\omega_i$$

Tomu, co jsme právě odvodili, se říká *zobrazovací rovnice*, anglicky *rendering equation*. A my se pokusíme najít její aspoň přibližné řešení.

Nejdříve chci ale upozornit na jednu hezkou vlastnost zobrazovací rovnice. Představme si, že pomocí ní počítáme světlo v nějakém místě stěny, která je osvětlena žárovkou – emissivní koulí. Pokud je žárovka blízko, zabírá velkou část viditelné polokoule a počítané místo bude osvětleno silně. Pokud bude daleko, bude zabírat mnohem menší místo na polokouli a místo bude osvětleno mnohem slaběji. Klesání intenzity světla se vzdáleností od zdroje, které jsme ve třetím díle explicitně počítali (byť jen přibližně), s vykreslovací rovnicí funguje samo od sebe a dokonce zcela přesně!

Ještě než se pustíme do počítání, celou situaci si značně zjednodušíme. Nebudeme vůbec uvažovat spekulární odrazy, jen **difúzní**.

### Správnější rovnice pro difúzní odrazy

Než se pohneme dál, je třeba opravit jednu lež z minulých dílů ohledně počítání difúzního osvětlení. Pro připomenutí, ukázali jsme si jej takto:

$$C_d = IK_d(n \cdot l)$$

Kde $C_d$ je světlo odražené k pozorovateli, $I$ je intenzita příchozího světla, $K_d$ je barva (difúzní odrazivost) povrchu, $n$ je normálový vektor povrchu a $l$ je vektor směrem ke zdroji světla. Skalární součin $n$ a $l$ je aplikací Lambertova zákona. V této formulaci má tento výpočet difúzního světla jeden zásadní problém: nezachovává energii.

Pokud by byla barva povrchu rovna jedné (tedy neabsorboval by žádné světlo, všechno by bylo difúzně odraženo) a ze všech směrů by na povrch dopadalo světlo hodnoty jedna, tak celkové odražené světlo povrchu by bylo větší než celkové příchozí světlo. Konkrétně by bylo $\pi$-krát větší. Proto se difúzní světlo správně počítá takto:

$$C_d = I\frac{K_d}{\pi}(n \cdot l)$$

Pro správnou simulaci světla je potřeba, abychom energii zachovávali, proto budeme pro difúzní osvětlení používat tento vztah. Podrobné odvození, proč se toto děje, naleznete zde: http://www.joshbarczak.com/blog/?p=272

Naše BRDF tedy vypadá následovně:

$$\frac{K_d}{\pi}$$

Skalární součin z Lamberova zákona v BRDF není, protože už je započtený do zobrazovací rovnice.

[1] `http://ksp.mff.cuni.cz/encyklopedie/integraly.html`

**Monte Carlo integrace**

Co s integrálem v zobrazovací rovnici? Určitě jej neupočítáme nějakým upravováním rovnic, protože příchozí světlo a obecně scéna může vypadat jakkoliv složitě. Ale můžeme použít starou dobrou hrubou sílu.

Integrál dvourozměrné funkce je dotaz na obsah plochy pod touto funkcí. V případě integrálu přes polokouli se jedna o dotaz na objem prostoru pod trojrozměrnou funkcí. Můžeme si to představit jako výškovou mapu nad povrchem polokoule – pro každý bod na polokouli určený směrem $\omega_i$ nám vnitřek integrálu určuje výšku funkce nad tímto bodem (pro jednoduchost zatím počítejme jen intenzitu světla bez barvy).

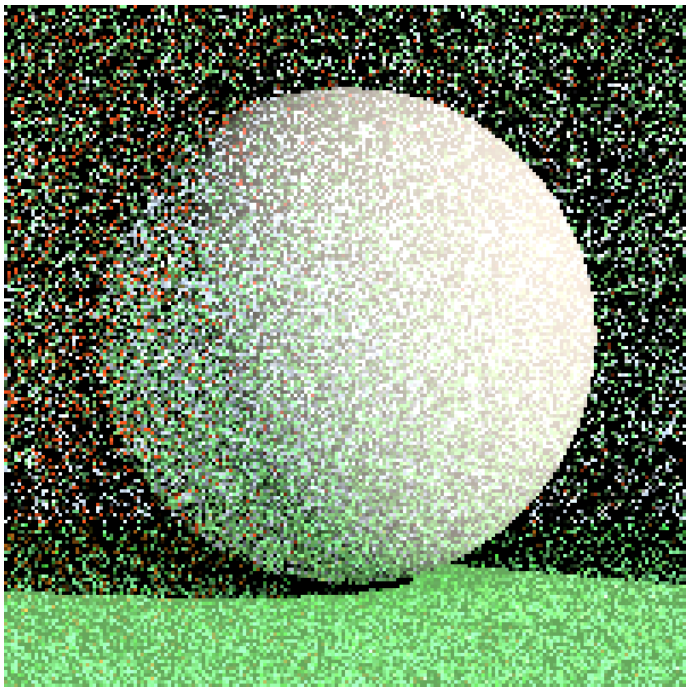
Přesnou hodnotu integrálu neupočítáme, ale můžeme se k ní aspoň přiblížit. V principu si vybereme několik míst na této rovině, pro ty funkci uvnitř integrálu vyhodnotíme, a z výsledků nějak odhadneme hodnotu celého integrálu.

Ale můžeme si to ještě zjednodušit. Vybereme si *jediné* místo na rovině (jediný směr příchozího světla), spočítáme funkci uvnitř integrálu, a budeme se tvářit, že tuto hodnotu má tato funkce úplně všude. Tedy předpokládáme, že integrál se ptá na objem nějakého kvádru. A ten umíme spočítat. Kromě výšky, kterou spočítáme vyhodnocením funkce uvnitř integrálu, potřebujeme znát ještě obsah plochy, nad kterou integrál počítáme. Vlastně je to povrch polokoule, a ten je roven $2\pi$ (počítáme s polokoulí o poloměru 1).

Samozřejmě tento výsledek bude velmi nepřesný, a výsledek se bude velmi lišit podle toho, v jakém místě funkci spočítáme.

Můžeme si ale těchto nepřesných výsledků pořídit více, každý z nějakého náhodně vygenerovaného bodu, přičemž každý bod má stejnou pravděpodobnost, že se na něj zeptáme, a pak výsledky zprůměrovat. Na čím více místech spočítáme vnitřní funkci, tím lepší bychom měli mít obrázek o jejím skutečném tvaru.

Tohle sice zní jako strašný hack, ale skutečně to funguje. Dokonce se dá dokázat, že čím více těchto náhodných *vzorků* (tak se říká jednomu výsledku) spočítáme, tím blíž bude jejich průměr skutečné hodnotě integrálu.



V praxi každý snímek spočítáme jeden vzorek na pixel a výsledek zprůměrujeme se všemi dosud spočítanými snímky. A protože zpočátku bude barva každého pixelu záviset jen na několika málo náhodných vzorcích, které budou pro každý pixel různé, tak bude obraz zašuměný. S více vzorky se šum bude postupně zlepšovat.

**Path tracing**

Už víme, že pokud chceme aproximovat vykreslovací rovnici pro nějaký pixel, tedy na konkrétním místě pro konkrétní odchozí směr, stačí nám k tomu vybrat si jeden náhodný příchozí směr a započítat světlo, které z něj přichází. Jak ale spočítat to příchozí světlo?

Pokud paprsek vystřelený pro příchozí směr $d$ dopadl do oblohy, příchozí světlo je prostě to, co obloha v tomto směru vyzáří, tedy její barva. Pokud ale dopadl na nějaký povrch ve scéně, příchozí světlo zjistíme (rekurzivním) vyhodnocením vykreslovací rovnice pro toto místo, jen jako $L_o$ použijeme $-d$.

Pro vyhodnocení rovnice můžeme opět použít trik s aproximací integrálu jediným vzorkem. A ani nemusíme nějak speciálně řešit průměrování, stačí, když jej děláme na finálních barvách pixelu.

Tento proces vyhodnocování vykreslovací rovnice jedním vzorkem můžeme rekurzivně opakovat. Abychom to nedělali donekonečna, můžeme si říct, že po nějakém počtu odrazů skončíme. Pro poslední vyhodnocení vykreslovací rovnice prostě prohlásíme, že příchozí světlo je nulové (už jej nechceme počítat), takže vrátíme vlastně jen světlo, které materiál sám vyzáří.

Všimněte si, že paprsky použité pro vyhodnocení vykreslovacích rovnic tvoří jakousi cestu, kterou se nějaký paprsek světla může ve scéně odrážet. Proto se tomuto postupu někdy říká *path tracing*, neboli sledování cest.

**Slunce**

Jak do zobrazovací rovnice a path tracingu zapadá přímé osvětlení od slunce z minulého dílu?

Pokud bychom simulovali scénu s realistickou oblohou, tak by slunce bylo, stejně jako ve skutečnosti, malé, ale velmi světlé místo na obloze. A některé z našich náhodných paprsku by prostě měly to štěstí a trefili by se právě do něj.

Náš model slunce z předchozího dílu ale počítá s tím, že se jedná o nekonečně malý bod nekonečně daleko. A náhodný směr, který by se trefil zrovna do tohoto bodu, nevygenerujeme nikdy. Místo toho můžeme při každém vyhodnocení vykreslovací rovnice slunce započítat explicitně. Prostě vystřelíme jeden stínový paprsek, a pokud odletí do nekonečna, přičteme k odchozímu světlu iluminaci ze slunce (samozřejmě modulovanou Lambertovým zákonem a difúzní barvou aktuálního povrchu).

Slunce modelované nekonečně malým bodem je docela hack, ale je to jednoduché a funguje to.

**Progresivní vykreslování**

Obraz, který se budeme snažit spočítat, bude vyžadovat na každý pixel vystřelit velké množství paprsků, řádově stovky až tisíce. My ale nechceme na každý snímek čekat několik minut. Proto začneme s nekvalitním, zašuměným obrazem, a ten budeme postupně vylepšovat.

Budeme zobrazovat průměr všech už vykreslených snímků. Budeme si pamatovat rozpracovaný obraz. Každý snímek vystřelíme jen jeden paprsek na pixel, poté načteme hod-

notu z rozpracovaného obrazu, vynásobíme ji počtem už vykreslených snímků, k výsledku přičteme právě spočítaný nový snímek a toto celé opět vydělíme novým celkovým počtem snímků a uložíme.

Budeme potřebovat nějaké úložiště pro rozpracovaný obraz. K tomu použijeme *texturu*. Textura je něco jako pole hodnot (byť toho umí mnohem více než obyčejné pole). Typicky se setkáváme s dvojrozměrnými texturami, kde je v každém prvku uložena barva jako trojice či čtveřice čísel. Dvojrozměrné textury jsou tedy něco jako obrázek.

Shadertoy umí výsledek shaderu uložit právě do textury. Při počítání dalšího snímku můžeme tento předchozí výsledek nějak využít.

**Kostra pro řešení úkolů**

Protože na řešení úkolů v tomto díle je potřeba jednak dost specificky nastavit Shadertoy, druhak mít přístup ke spoustě pomocných funkcí, můžete k řešení použít následující kostru, ve které je vše připraveno:
https://www.shadertoy.com/view/WtdBWr

Jediné, co v kostře potřebujete měnit, je funkce `computeImage` (v záložce „Buffer A“), případně můžete přidat své vlastní funkce. Dále je třeba doplnit implementaci funkce `raymarch` z minulého dílu.

Také se v kostře nacházejí nějaké pomocné funkce. Ta nejdůležitější je funkce `randomPointOnUnitHemisphere`, která pro různé seedy generuje rovnoměrně rozmístěné body na povrchu jednotkové polokoule. Dá se tedy použít k tomu, abyste pro nějaké místo na povrchu vygenerovali náhodný směr odraženého paprsku.

Při psaní shaderu doporučuji zapausovat překreslování a resetovat čas na nulu (pomocí tlačítek na spodním kraji zobrazovacího okna). Poté sice budete vidět jen první velmi zašuměný snímek, ale při překompilování shaderu se ihned překreslí a uvidíte nový výsledek. Jinak by se při překompilování jen do zprůměrovaných snímků starého shaderu začaly pomalu průměrovat snímky nového.

Kostra též obsahuje ukázkovou definici scény: proměnné `sunlightDirection` a `sunlightColor` a funkce `sceneSdf` a `getSky` pro scénu, pro kameru jsou to proměnné `cameraPosition`, `cameraLookAt` a `cameraFov`. Pokud je použijete, měli byste dostávat stejné výsledky jako jsou na obrázcích níže.

**Poznámka k nastavení Shadertoy**

Pokud byste si chtěli Shadertoy nastavit pro progresivní vykreslování sami (třeba proto, že na něm máte účet a ukládáte si rozpracované shadery přímo v Shadertoy – i v takovém případě prosím stále odevzdávejte kód shaderu a ne jen odkaz na shadertoy), dělá se to následujícím postupem.

Shadertoy nám umožňuje použít několik zabudovaných textur, pod editorem naleznete čtyři černé obdélníky. Vyberte ten nadepsaný „iChannel0“ a z výběru (v „Misc“) vyberte „Buffer A“. Tím říkáme, že pod názvem „iChannel0“ chceme v shaderu přistupovat k této textuře. Poté v levém horním rohu editoru klikněte na tlačítko „plus“ a vyberte „Buffer A“. V editoru vznikne nová záložka nadepsaná „Buffer A“ (vedle původní záložky „Image“). Shader v této záložce nebude kreslit do pixelů obrazovky, nýbrž přímo do pixelů textury „Buffer A“. I pro shader v „Buffer A“ nabindujte „iChannel0“ na texturu „Buffer A“.

**Shrnutí**

Abychom vám implementaci usnadnili, zde je seznam všech důležitých faktů z textu výše:

- Neváhejte použít své řešení (nebo kousky kódu ze zadání) minulého dílu
- Odchozí světlo spočítáme aproximací vykreslovací rovnice pomocí jediného vzorku.
- Protože integrál se ptá na objem pod nějakou dvojrozměrnou funkcí, potřebujeme znát „plochu“ této funkce, která je v tomto případě rovna ploše polokoule, tedy $2\pi$.
- Jako BRDF použijeme $K_d/\pi$ (tedy pouze difúzní, spekulární neuvažujeme).
- Nezapomeňte aplikovat i Lambertův zákon.
- Při každém vyhodnocování vykreslovací rovnice (pokud ji budete vyhodnocovat vícekrát) nezapomeňte počítat se správnými vektory: směr příchozího a odchozího světla, normála povrchu…
- Pokud budete psát i dva odrazy, nezapomeňte pro druhý obraz vygenerovat náhodný směr z jiného seedu než jste použili pro první odraz.
- Pokud chcete své výsledky vizuálně porovnávat s obrázky zde, použijte celou definici scény a kamery z kostry.

**Úkol 1– Hlavní** [8b]:

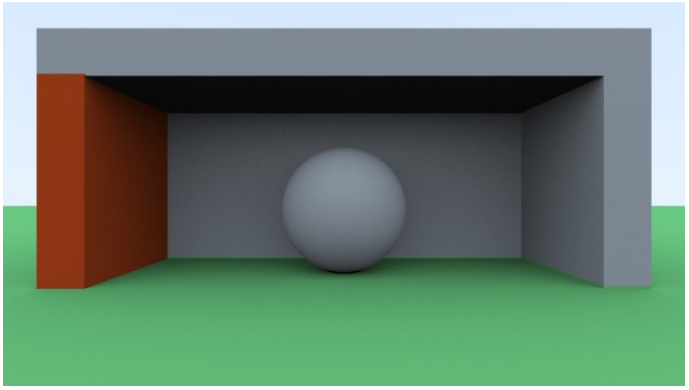
Naprogramujte osvětlení od oblohy pomocí výše popsaných postupů, které umí jediný difúzní odraz. Tím je myšleno, že od místa dopadu paprsku z kamery vystřelíme ještě jeden paprsek náhodným směrem. Pokud vyletí do nekonečna, přijde po něm světlo z daného místa oblohy, pokud se trefí do nějaké překážky, tak z něj žádné světlo nepřijde.

Jako vždy odevzdávejte zdrojový kód vašeho shaderu (stačí „Buffer A“).

Při implementaci použijte pro každý snímek jiné náhodné hodnoty, třeba pomocí zabudovaného vstupu `iFrame`, jako to dělá ukázkový obsah funkce `computeImage` v kostře. Při generování náhodného směru paprsku pro nějaké místo scény je dobré do funkce `randomPointOnUnitHemisphere` jako `seed` dosadit třeba pozici ve scéně (samozřejmě třeba posunutou o `iFrame`, aby byla výsledná náhodná hodnota i pro stejné pozice každý snímek jiná).

Nezapomeňte použít difúzní BRDF s dělením $\pi$. Pokud je váš obraz málo světlý nebo naopak příliš světlý, zkontrolujte, že správně počítáte integrál ze zobrazovací rovnice, viz sekce „Monte Carlo integrace“ výše.

Výsledek úkolu by měl vypadat nějak takto:

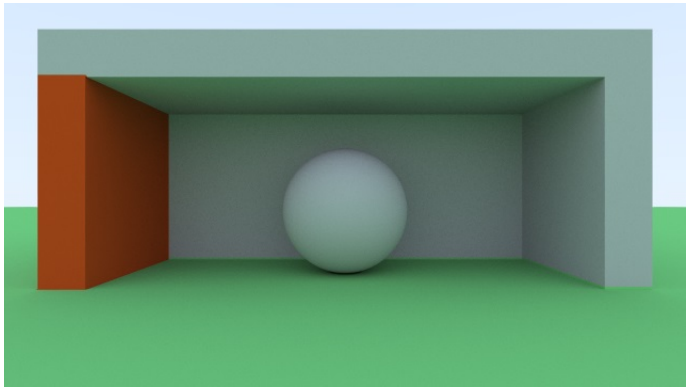


**Úkol 2– Dva odrazy** [3b]:

Rozšiřte vaše řešení předchozího úkolu o druhý odraz. Tedy pokud původní nepřímý paprsek (vystřelený náhodným směrem z místa dopadu paprsku z kamery) dopadl na nějakou překážku, opakujeme předchozí postup, tedy vystřelíme ještě jeden paprsek nějaký **jiným** náhodným směrem.

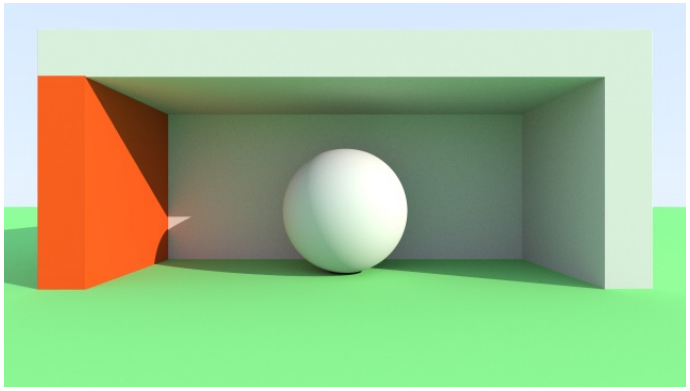
Tady vás to možná navádí k napsání něčeho, co umí konfigurovatelný počet odrazů. Doporučuji se do toho nepouštět a prostě to zadrátovat na dva odrazy, protože rekurzi použít nemůžete (GPU ji neumí) a při psaní něčeho takového smyčkami se dá velmi snadno ztratit.

Výsledek by měl vypadat nějak jako na obrázku níže. Všimněte si, že strop místnosti je zabarven lehce do zelena světlem odraženým od země.



**Úkol 3– Slunce** [2b]:

Přidejte k řešení minulého úkolu osvětlení od slunce výše popsaným způsobem. Níže je obrázek výsledku pro dva nepřímé odrazy. Všimněte si, že místnost je na levé straně viditelně zabarvená světlem odraženým od červené zdi. Tento efekt by měl být viditelný i ve vašem řešení.



**Antialiasing**

Doteď jsme na každý pixel vystřelili jediný paprsek, který vždy procházel jeho středem. Představme si ale, že ve scéně je nějaká vertikální tyč, která je natolik tenká, že se celá vejde mezi středy dvou sousedních pixelů. Potom by ve výsledném obrázku nebyla vůbec!

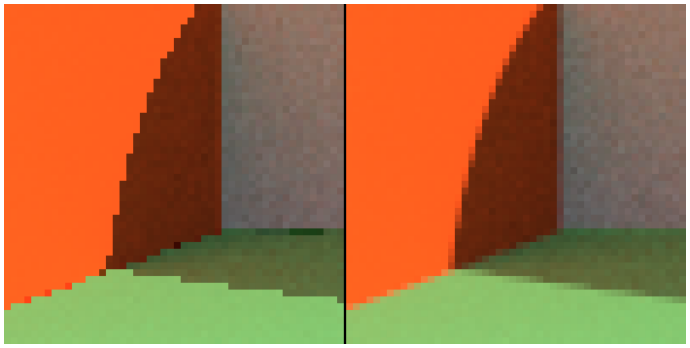
Problém se dá opravit tak, že obrázek vykreslíme v mnohem větším rozlišení a pak ho zmenšíme na původní rozlišení. Třeba jeden pixel výsledného obrázku by byl ve skutečnosti zprůměrováním bloku 8x8 pixelů obrazu ve vysokém rozlišení. Tomuto postupu se říká *supersampling*.

Nicméně supersampling není všemocný. Vždycky si totiž můžeme pořídit ještě tenčí tyč, která se vejde i mezi středy hustší mřížky pixelů. Problém je v tom, že pixel není nějaký nekonečně malý bod, ale je to ve skutečnosti malý čtvereček s nenulovou plochou. Sice jej rozsvítíme celý jen jednou barevnou hodnotou, ale stále by tato hodnota měla reprezentovat celou část scény, která se za plochou pixelu skrývá.

Problému, který se snažíme vyřešit, se obecně říká *aliasing*. Vzniká proto, že nějakou složitou funkci (barvu scény) vzorkujeme nepříliš hustě a v pravidelných rozestupech, tudíž nám může velká část informace unikat. (Když už jsme u nedostatečně hustého vzorkování složité funkce, všimli jste si, jakým způsobem generujeme náhodná čísla? ;) )

Ideální by bylo pro každý pixel nějak zprůměrovat barvu scény za ním, a to zobrazit. A to s mašinerií, kterou už máme kvůli průměrování pixelů v čase, můžeme implementovat velmi snadno. Stačí nestřílet paprsek z kamery středem pixelu, ale náhodně vygenerovaným bodem na ploše pixelu. Takže čím víc snímku akumulujeme, tím lépe bude hodnota každého pixelu odpovídat reálně barvě scény za ním.

Na obrázku níže je vlevo obraz s aliasingem, vpravo stejný obraz, ale spočítaný s antialiasingem podle výše popsaného způsobu.



**Úkol 4** [1b]:

Implementujte výše popsaným způsobem antialiasing. Obraz by měl vypadat stejně jako bez antialiasingu, jen hrany objektů a ostré stíny od slunce by měly být viditelně jemnější (poté, co se naakumuluje několik snímků).

Náhodný posun v pixelu dokonce může být i pro všechny pixely stejný, jen by se měl každý snímek změnit.

**Světýlko**

Zatím jsme nepoužili jednu zbývající komponentu vykreslovací rovnice: emissivní světlo. Pokud jste udělali předchozí úkoly, jeho implementace by měla být velmi snadná: stačí k odchozímu světlu z vykreslovací rovnice emissivní světlo přičíst. Jediné, co zbývá, je definovat si, který objekt ve scéně bude světlo vyzařovat. My rozzáříme kouli uprostřed, a to třeba takto:

```
vec3 emissiveLight = vec3(0.0);
if (sdfSphere(vec3(0.0, 0.8, 2.0),
    0.8, currentPosition) < 0.02)
{
    emissiveLight = vec3(1.0, 0.8, 0.01) * 4.0;
}
```

Kde v `emissiveLight` je emissivní světlo a v `currentPosition` je pozice ve scéně, kde vykreslovací rovnici vyhodnocujeme.

Aby byla scéna zajímavější, přidáme vedle svítící koule ještě jednu menší kuličku, která bude vrhat nějaký stín. K tomu stačí modifikovat funkci `sceneSdf` tím, že na její konec (ale ještě před `return` samozřejmě) přidáme následující kód:

```
// Stínová koule
d = min(d, sdfSphere(vec3(1.8, 1.3, 2.0),
    0.4, p));
```

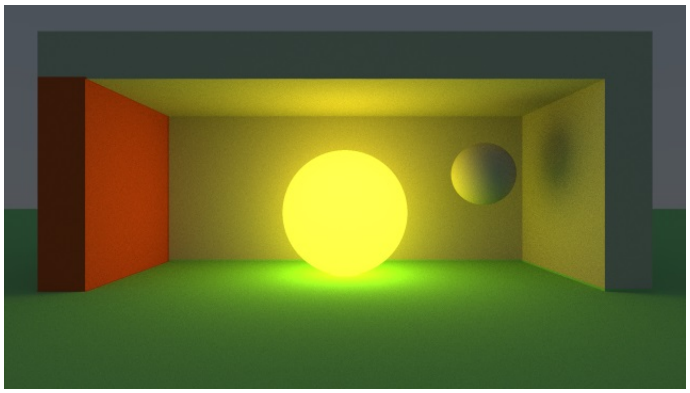
A nakonec ještě „vypneme“ sluneční světlo a zeslabíme světlost oblohy, aby svítivost koule hezky vynikla:

```
vec3 getSky(vec3 direction)
{
    return mix(vec3(0.8, 0.9, 1.0),
               vec3(0.5, 0.7, 1.0),
               max(direction.y, 0.0)) * 0.1;
}

// ...

vec3 sunlightColor = vec3(0.0);
```

**Úkol 5** [1b]:

Implementujte emissivní kouli postupem popsaným výše. Pro dva odrazy by výsledek měl vypadat jako na obrázku níže.

Všimněte si stínu, který vzniká za menší kuličkou, a že je správně jemný, tedy každé místo je osvětleno právě podle toho, jak velkou část svítící koule vidí. Nic z toho jsme nemuseli explicitně implementovat, všechno prostě funguje samo, stačí správně řešit vykreslovací rovnici!



**Jak je to doopravdy**

Pokud jste splnili úkoly tohoto seriálu, napsali jste malý renderer, který dokáže simulovat světlo o poznání sofistikovaněji než spousta komerčních her, a co víc, běží na integrovaných grafikách, zatímco raytracing ve hrách vyžaduje (aspoň zatím) velké, drahé a energeticky žravé grafické karty.

Tady něco nesedí, že?

Dále nenásledují žádné úkoly ani nic k nim relevantního, ale vysvětlení, v čem je přístup z posledních dvou dílů tohoto seriálu omezený. Jaký je rozdíl mezi naším SDF pathtracerem a renderery, které se používají na filmy nebo architektonickou vizualizaci, a také hrami a enginy, které raytracing začínají čím dál více využívat.

**Rasterizace**

Pokud chceme vykreslit trojúhelníkovou scénu s perspektivní projekcí (tu jsme používali v seriálu), můžeme využít toho, že trojúhelník zůstane trojúhelníkem i po této projekci, a je tedy snadné spočítat, které pixely na obrazovce zabírá. Víceméně místo našeho postupu „pro každý pixel najdi průsečík se scénou“ použijeme „pro každý trojúhelník obarvi pixely, které zabírá“. Také si pro každý pixel pamatujeme jeho „hloubku“ ve scéně, aby se nestalo, že vzdálenější pixely zakryjí ty bližší.

Tomuto postupu se říká rasterizace a lze implementovat velice efektivně jak v softwaru, tak v hardwaru. Toto je ta technika, co se používá ve hrách a jiných realtime grafických aplikacích. Nevýhodou je, že není použitelná pro velké množství nesouvisejících dotazů typu „kde paprsek poprvé protne scénu“, které jsou při počítání sofistikovanějšího osvětlení velmi užitečné.

Jak si pořídit něco, co na „paprskové“ dotazy odpovídat umí?

**Hardwarový raytracing**

Reálné scény ve hrách se skládají z velkého množství trojúhelníků (dnes až desítky milionů), takže pokud chceme najít průsečík paprsku se scénou, nestačí spočítat kolizi paprsku pro každý trojúhelník, musíme nějak rychle umět vyřazovat ty trojúhelníky, které paprsek neovlivní. K tomu se používají různé stromové struktury, typicky *Bounding Volume Hiearchy* neboli *BVH*, což je strom, kde jsou v listech uloženy trojúhelníky scény a každý nelistový vrchol si pamatuje kvádr dost velký na to, aby se do něj vešel celý podstrom pod tímto vrcholem. Tudíž pokud paprsek tento kvádr neprotne, nemá smysl podstrom uvažovat.

Procházet nějakým stromem není výkonnostně ideální, protože děláme spoustu malých a hlavně náhodných (tedy neefektivních) čtení z paměti, navíc každé následující záleží na výsledku předchozího. Procesor se s takovou úlohu nějak popere, nicméně ani ty nejrychlejší procesory nejsou dostatečně výkonné na to, aby tohle bylo použitelné na nějaké grafické úlohy pro hry.

A grafické karty? Nic nám nebrání implementovat průchod stromem v shaderu.[2] GPU sice bude průchod stromem nejspíš vadit ještě víc než procesoru, přesto bude výsledek řádově rychlejší. Nicméně stále ne dost rychlý pro hry. Až s hardwarovou implementací některých částí počítání průsečíku paprsku se scénou, zkombinovanou s nejnovějším a nejrychlejším hardwarem, se dostáváme na něco použitelného. (Hardware konkrétně řeší minimálně počítání průsečíku paprsku s trojúhelníkem nebo kvádrem, což jsou jinak poměrně náročné úlohy, vzhledem k obřímu množství průsečíků, které je třeba spočítat.)

**Reprezentace scény**

Proč nám tedy v seriálu raytracing prochází? Trik je právě v tom, že náš popis scény není žádný obří strom, ale jednoduchá funkce. Nepotřebujeme z paměti přečíst *vůbec nic*, a spočítat toho musíme jen málo. Naše scéna je vlastně uložena uvnitř zkompilovaného shaderu, který je i s ní dost malý na to, aby se vešel do všelijakých keší na hardwaru. Sice SDF vyhodnocujeme vícekrát, ale při průchodu stromem bychom průsečík s vrcholem stromu také počítali opakovaně. Hlavně mezitím nemusíme čekat, než nám dorazí pár bytů z náhodného místa paměti (takové čekání může klidně trvat stejný čas jako provedení stovek nebo tisíců instrukcí typu „sečti dvě čísla“).

Reprezentace scény funkcí má samozřejmě to omezení, že scéna musí být velmi jednoduchá. Naše funkce také v principu počítá „pro každý pixel zkontroluj každý objekt“, a s rostoucím počtem objektů by poměrně brzo začal být výhodnější nějaký stromový přístup.

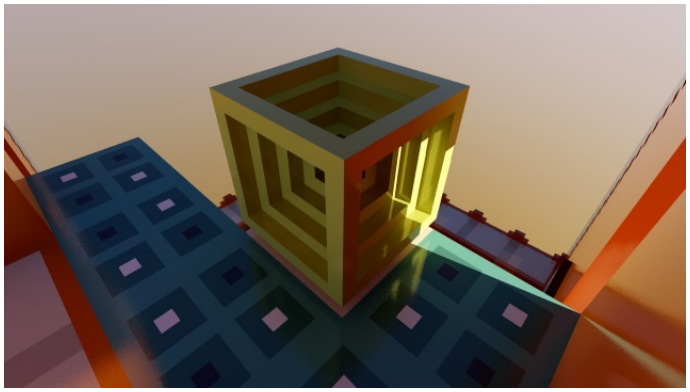
Mimochodem, k Signed Distance Functions existuje alternativa, které se říká Signed Distance *Fields*, což je SDF

[2] https://interplayoflight.wordpress.com/2018/07/04/hybrid-raytraced-shadows-and-reflections/

předpočítaná do prostorové mřížky. Toto umí používat pro počítání nepřímého osvětlení třeba Unreal Engine nebo Godot Engine.

Dalším trikem, jak si pořídit použitelný raytracing bez specializovaného hardwaru, je omezit scénu na čistě voxelovou – hranatou ve stylu Minecraftu. Poté je i softwarová implementace na GPU dostatečně rychlá.

Níže je screenshot z voxelového raytraceru, scéna je jednou z testovacích scén programu MagicaVoxel.[3]



**V jednoduchosti (není?) síla**

Řešení vykreslovací rovnice tak, jak je popsáno výše, je jednoduché a funguje. Dokonce jsme mohli nějakému objektu nastavit, aby vyzařoval světlo, a vše fungovalo!

Teda, skoro.

Principem je náš způsob vykreslování správný. Nicméně představme si klasickou žárovku, konkrétně tenký rozžhavený drát uvnitř. Je to velmi malý, ale velmi světlý objekt. Jaká je pravděpodobnost, že se při path tracingu nějaký náhodně vyslaný paprsek trefí právě do něj? Velmi, velmi malá. Abychom tedy dostali nezašuměný obraz, museli bychom čekat opravdu dlouho.

Na stejný problém bychom narazili, kdybychom místo difúzního BRDF použili nějaké, které má i spekulární odrazy, a zkusili s nimi simulovat zrcadlo. Zrcadlo je vlastně povrch, jehož BRDF je nulové pro všechny páry příchozího a odchozího směru kromě těch párů, kde by se příchozí světlo odrazilo skoro přesně směrem odchozího. Opět by byla velmi malá šance, že by náhodný paprsek letěl právě tím směrem, aby z něj zrcadlo odrazilo směrem k pozorovateli nenulové množství energie.

V praxi se tyto problémy řeší tím, že si pořídíme „cinknutou“ náhodu, která s větší pravděpodobností posílá paprsky tam, kde se s nimi stane něco zajímavého. Tuto větší pravděpodobnost poté kompenzuje tak, že tyto paprsky dostanou menší váhu. Těmto postupům se říká *Next Event Estimation* (pro zdroje světla) a *Importance Sampling* (pro složité BRDF).

**Kam dál**

Tento seriál byla jen malá ukázka z obřího pole počítačové grafiky. Pokud se do něj chcete ponořit hlouběji, nebo si jen přečíst nějaké zajímavosti třeba o vykreslování ve hrách, níže najdete odkazy na další materiály.

Pokud vás shadery nadchly, podívejte se na už mnohokrát zmiňovanou *The Book of Shaders*.[4] Je to velmi hezky zpracovaná webová učebnice shaderů, ve které najdete také různé šumové funkce, vzory a co se s nimi dá všechno dělat. Tento seriál je jí inspirovaný a je na ní částečně založený.

*Shadertoy*[5] je sám o sobě obří galerii nejrůznějších shaderů, od procedurálně generovaných scén a fraktálů přes cartoonové animace a dema až po roztodivné vizualizace různých nových grafických algoritmů, které vytvářejí inženýři pracující na špičkových herních enginech.

Jako zdroj různých zajímavých grafických experimentů či algoritmů využitelných v shaderech můžete použít i domovskou stránku *Iniga Quileze*.[6] Některé jeho články jsou velmi užitečné a bylo na ně odkázáno v seriálu.

Pokud vás zajímá grafika z pohledu fotorealistických offline rendererů, přečtěte si krátkou webovou knihu *Ray Tracing in One Weekend*[7] či její pokračování *Raytracing the next week* a *Raytracing the rest of your life*.

Jestli chcete jen rychlou ochutnávku různých postupů a triků, co hry využívají, doporučuji *GTA V Graphics Study*[8] či nějakou podobnou studii z tohoto seznamu.[9]

Ale jestli si chcete napsat vlastní malý vykreslovací engine, doporučuji *LearnOpenGL*,[10] což je rozsáhlý tutorial zaměřující se na grafické API OpenGL a na základní algoritmy používané v realtime renderingu a ve hrách. Vězte ale, že herní vykreslování v praxi je něco úplně jiného a mnohem méně elegantního než to, co si ukazujeme v tomto seriálu. Ale pokud se do něj přece jen pustíte, chcete vědět o nástroji *Renderdoc*,[11] což je tzv. frame debugger, který vám umožňuje dívat se, jak GPU postupně kreslí snímek, a zjistit, proč se vám třeba nějaký mesh nekreslí.

A pokud vás zajímá, co a proč se děje uvnitř grafické karty, velmi detailní popis je *A trip through the graphics pipeline*.[12]

*Kuba Pelc*

[3] https://github.com/ephtracy/voxel-model/tree/master/vox/monument
[4] https://thebookofshaders.com/
[5] https://www.shadertoy.com/
[6] https://iquilezles.org/
[7] https://raytracing.github.io/
[8] http://www.adriancourreges.com/blog/2015/11/02/gta-v-graphics-study/
[9] http://www.adriancourreges.com/blog/2020/12/29/graphics-studies-compilation/
[10] https://learnopengl.com/
[11] https://renderdoc.org/
[12] https://fgiesen.wordpress.com/2011/07/09/a-trip-through-the-graphics-pipeline-2011-index/

*Tento studijní text vznikl pro účely seriálu o počítačové grafice a shaderech, a popisuje jen to nejnutnější pro jeho pochopení.*

Představte si, že píšete nějakou hru, kde hráč může skákat. Nachází se v nějaké pozici (výšce) $h$ na ose Y, má svou aktuální vertikální rychlost $v$ a dolů ho přitahuje gravitace o zrychlení $g$. Ostatní osy pohybu zanedbáváme.

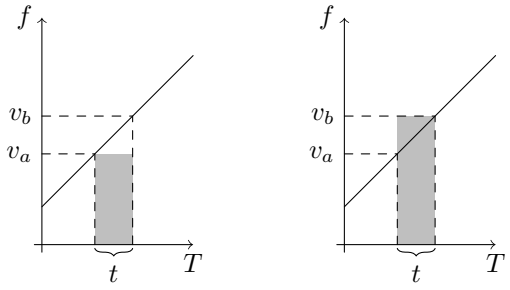
Logiku hry vyhodnocuje nějaká smyčka, která se spustí každých $t$ sekund. V každé její iteraci chceme spočítat novou pozici (vlastně jen výšku) a rychlost hráče na základě předchozí pozice a rychlosti, a také času, který od minulé iterace uplynul. Jak by něco takového mohlo vypadat v pseudokódu?

```
// Varianta A:

h = h + v * t;
v = v + g * t;

// Varianta B:

v = v + g * t;
h = h + v * t;
```

Která z variant je ale správně? Spočítáme nejdřív novou pozici pomocí staré rychlosti a pak rychlost updatujeme, nebo nejdřív spočítáme novou rychlost a tu použijeme pro spočítání pozice?

Správně není ani jedno. Kdybyste zkusili takovou hru spustit s reálným gravitačním zrychlením, postava by se chovala velice divně. Důvod je ten, že v reálném světě by se rychlost postavy neměnila v oddělených krocích, ale plynule během celého časového úseku $t$.

Vyjádříme si rychlost $v$ jako funkci $f$ závislou na $t$. Ve variantě A vlastně používáme pro celý časový úsek hodnotu rychlosti z bodu $v_a$, ve variantě B hodnotu $v_b$.




Pozici v obou variantách změníme o hodnotu $vt$, ať už jako $v$ použijeme $v_a$ nebo $v_b$. Všimněte si, že tato hodnota odpovídá obsahu obdélníku výšky $v$ a šířky $t$. Nyní je z obrázku patrné, že k pozici nechceme přičítat ani jeden z obdélníků, ale lichoběžník, jehož horní strana je tvořena funkcí $f$.

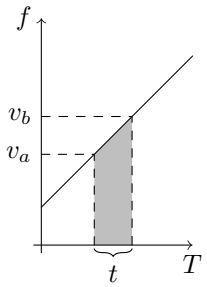
Jinak řečeno, zajímá nás obsah pod funkcí $f$. Toto vyjadřují *integrály*. S integrálem bychom pseudokód nahoře zapsali takto:

$$h = h + \int_0^t (v + gx)\ \mathrm{d}x$$
$$v = v + gt$$

Znak pro integrál je $\int$ a říká nám, že chceme spočítat obsah plochy pod danou funkcí, v našem případě $v + gx$. Zajímá nás jen plocha od bodu 0 do bodu $t$ (bodem 0 uvažujeme počátek časového úseku, který nás zajímá) na vodorovné ose. Značení $\mathrm{d}x$ na konci říká, že se jedná o funkci proměnné $x$, právě hodnota $x$ se bude měnit od 0 do $t$.

Pro výpočet tohoto integrálu můžeme použít obyčejnou geometrii, pomocí které spočítáme obsah lichoběžníku, který integrálu odpovídá. Jeho hodnota bude:

$$\int_0^t (v + gx)\ \mathrm{d}x = vt + \frac{1}{2}gt^2$$




Zde jsme jen vzali obsah obdélníku pod původní hodnotou $v$ a přičetli k němu obsah trojúhelníku od $v_a$ do $v_b$. A odtud se také bere vzorec pro zrychlení z fyziky.

V tomto případě šla funkce uvnitř integrálu *zintegrovat* velmi jednoduše. Ne vždy tomu tak je, v některých případech ani nelze řešení vyjádřit vzorečkem. Jedna možnost, jak alespoň číselně spočítat integrál nějaké složité funkce, je popsána v pátém díle seriálu o grafice.[13]

Toto je ovšem jen extrémně stručné a zjednodušené vysvětlení integrálů, proto ho prosím berte trochu s rezervou.

*Článek pro vás sepsal*
*Kuba Pelc*

[13] `http://ksp.mff.cuni.cz/viz/33-5-S`