Korespondenční Seminář z Programování

33. ročník **KSP** **Prosinec 2020**

Milí řešitelé, řešitelky a řešitelčata!

Dostává se k vám letošní vánoční vydání třetí série KSP-H 33. ročníku KSP.

Naleznete zde **4 normální úlohy**, z toho jednu kompetitivní nad reálnými daty Prahy, **bonusovou těžší úlohu** a pak také pokračování **seriálu o počítačové grafice**. Všechny díly seriálu můžete **odevzdávat v průběhu celého roku**, takže vůbec nevadí, že jste třeba první díl nestihli. Detaily naleznete u zadání seriálu. Připomínáme, že oproti loňskému ročníku 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. Za letošní rok půjde získat maximálně 300 bodů, takže hranice pro úspěšné řešitele je 150. Maturanti pozor, pokud chcete prominutí využít letos, musíte to stihnout do konce čtvrté série, pátá už bude moc pozdě. 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 21. února 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: **Třem nejúspěšnějším řešitelům kompetitivní úlohy pošleme sladkou odměnu.**

Třetí série třicátého třetího ročníku KSP

33-3-1 Bezpečný tunel — 9 bodů

Představme si svět zasažený vysoce nakažlivou nemocí. Aby lidé nebyli nakaženi, byla přijata přísná opatření proti šíření nemoci, mezi něž patří i omezení pohybu osob. Je zakázáno vzdalovat se více než 500 metrů od místa bydliště.

Kevin a Zuzka, jejichž domy jsou od sebe vzdáleny více než 1000 metrů, se ale nutně potřebují setkat. Naštěstí existuje způsob, jak se mohou i v tomto případě potkat, aniž by skončili ve vězení – mohou vykopat tajný tunel, který začíná a končí v oblasti, do které ještě mohou legálně dojít. V něm budou v bezpečí před policií i dalšími nezvanými hosty. Kopání tunelu je ale náročná věc, proto by chtěli, aby byl co nejkratší.

Máte zadáno město velikosti $M \times N$ jako bludiště ve čtvercové síti. Jednotkou je 1 metr, každá buňka je buď průchozí nebo je na ní zeď. V bludišti máte dva body A a B, což jsou pozice domů Kevina a Zuzky.

Od bodů A, B se smíte vzdálit maximálně na $K = 500$ metrů. Vzdálenosti se měří Manhattanskou metrikou, takže vzdálenost v dvou bodů $p = [x_1, y_1]$ a $q = [x_2, y_2]$ je definována jako $v = |x_1 - x_2| + |y_1 - y_2|$. Při průchodu městem se nesmíte v žádné chvíli vzdálit na víc než K metrů od A nebo B (není povoleno projít cestou, jejíž část prochází oblastí vzdálenou více než K, i kdyby se cesta poté zase vrátila do legální oblasti).

Chceme najít co nejkratší tunel, který má konce na pozicích T_A a T_B. K bodu T_A musí vést cesta z A, která se od A nikdy nevzdálí na víc než K metrů. To samé platí i pro T_B a bod B. Tunel je pod zemí, takže není nijak omezen zdmi v bludišti (podkope se pod nimi).

Od vás chceme vymyslet algoritmus na hledání tohoto nejkratšího tunelu. Dostane na vstupu zadanou mapu města a pozice domů A a B a najde nejkratší tunel.

33-3-2 Ospalý student a diktát — 10 bodů

Kevin včera do noci řešil KSP a na hodině matematiky se mu nedaří udržet pozornost. Učitel se mu opakovaně snaží nadiktovat posloupnost různých čísel (zadání domácího úkolu), ale Kevin vždy zachytí pouze útržky. Učitele to po několika pokusech přestane bavit a pokračuje ve výkladu.

Po návratu domů (a pořádném šlofíku) se Kevin dívá do sešitu, kde má několikrát napsané části domácího úkolu a zajímá ho, zda z nich je možné jednoznačně sestavit úkol celý. Délku posloupnosti si naštěstí zapamatoval.

Pro délku úkolu 4 a útržky $(2, 3, 5)$, $(2, 1)$ a $(1, 3)$ jde jednoznačně sestavit posloupnost $(2, 1, 3, 5)$. Pro $(2, 3, 5)$ a $(1, 3)$ již posloupnost jednoznačně sestavit nejde, protože pořadí 1 a 2 není jasně určeno. Pro $(2, 1, 3, 5)$ a $(3, 2)$ posloupnost sestavit také nejde, protože pořadí 2 a 3 neodpovídá (při diktování se musela stát chyba).

Znáte délku výsledné posloupnosti a útržky. Pomozte Kevinovi určit, zda jde posloupnost z útržků jednoznačně sestavit. Pokud nejde, určete proč.

33-3-3 Stříbrných stříkaček — 13 bodů

Přes výstupní kontrolu chodí různé modely stříbrných stříkaček. Víme, že každý model má jiný dostřik (vyjádřený celým číslem).

U každé stříkačky bychom rádi ihned věděli, kolik modelů s menším dostřikem již výstupní kontrolou prošlo.

Nyní si můžete rozmyslet, jak tuto úlohu vyřešit. To po vás ale tentokrát nebudeme chtít.

Vašek si na to napsal následující program s pomocí *treapů*, o kterém si myslel, že za jakýchkoliv okolností bude fungovat opravdu rychle. Doufal, že na každý dotaz odpoví v logaritmickém čase vzhledem k počtu předešlých dotazů.

Zde je zdrojový kód programu v jazyce C++. Na webu ještě naleznete ten samý zdrojový kód v Pythonu, do letáku se nám bohužel nevešel.

```
#include<bits/stdc++.h>
using namespace std;

int random_number()
{
    return rand();
}

struct Treap
{
    Treap *left=NULL, *right=NULL;
    int key, priority;
    int size=1;
};

void update(Treap *in)
{
    if (!in) return;
    in->size = 1;
    if (in->left) in->size += in->left->size;
    if (in->right) in->size += in->right->size;
}

Treap *merge(Treap *left, Treap *right)
{
    if (!left || !right) return left?left:right;
    if (left->priority < right->priority)
    {
        left->right = merge(left->right, right);
        update(left);
        return left;
    }
    else
    {
        right->left = merge(left, right->left);
        update(right);
        return right;
    }
}

pair<Treap*, Treap*> split(Treap *in, int key)
{
    Treap *left=in, *right=in;
    if (!in) return {NULL, NULL};
    if (in->key < key)
        tie(left->right, right) =
                split(in->right, key);
    else
        tie(left, right->left) =
                split(in->left, key);
    update(left);
    update(right);
    return {left, right};
}

Treap *treap;
int main()
{
    int n;
    scanf("%d",&n);
    for (int i=0;i<n;i++)
    {
        int input;
        scanf("%d",&input);
        Treap *a, *b;
        tie(a,treap) = split(treap,input);
        tie(treap,b) = split(treap,input+1);
        printf("%d\n",a?a->size:0);
        if (!treap)
        {
            treap = new Treap;
            treap->priority = random_number();
            treap->key = input;
        }
        treap = merge(merge(a,treap),b);
    }
    return 0;
}
```

Ukažte mu, že se může stát, že jeho algoritmus nebude tak rychlý, jak čekal. Vaším úkolem bude nalézt vstup, na kterém tento algoritmus poběží pomalu. Konkrétně Vaškovo řešení musí alespoň M krát spustit funkci `merge` nebo `split` (číslo M dostanete na vstupu).

Abyste mohli takový vstup vyrobit, tak budete potřebovat čísla z náhodného generátoru. Podařilo se vám zjistit, jaké hodnoty bude postupně po spuštění vracet funkce `random_number()`. Vaším úkolem bude vytvořit pořadí stříbrných stříkaček na vstupu.

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

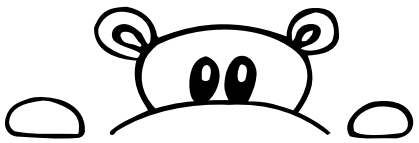
Formát vstupu: Na vstupu dostanete na prvním řádku čísla N a M. Číslo N udává počet náhodných čísel, které se vám povedlo zjistit, a zároveň limituje maximální počet stříbrných stříkaček. Číslo M je minimální počet operací, které musí odevzdaná posloupnost stříbrných stříkaček vygenerovat. Na dalších N řádcích jsou pak zjištěná náhodná čísla z náhodného generátoru v rozsahu mezi nulou a 10^6.

Pro jednotlivé vstupy platí:

Vstup	N	M
1.	3	20
2.	1 000	490 000
3.	1 000	490 000
4.	2 000	2 000 000

Formát výstupu: Na první řádek výstupu nejprve vypište $X \leq N$ – počet stříbrných stříkaček, které chcete předhodit programu na vstupu. Na následujících X řádcích pak vypište jednotlivá čísla. Každé z nich musí být celé číslo mezi 0 a 10^9

Ukázkový vstup:	*Ukázkový výstup:*
3 10 123 321 222	2 1 2

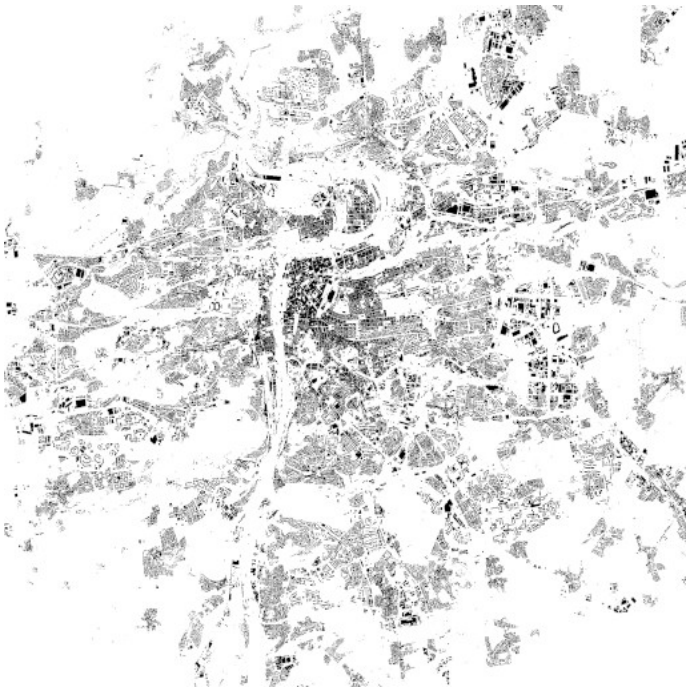


33-3-4 Obsazování území — 14 bodů

Během některých období řádění vysoce nakažlivé nemoci rozhodli vládci jednoho fiktivního království, říkejme mu třeba Fiktivní Praha, že se nikdo nesmí pohybovat dále, než 500 metrů od svého domu.

Ve Fiktivní Praze se vyskytoval i fiktivní Kevin a toho napadlo, že kdyby si koupil více domů, mohl by se procházet po větší části města. Začal svoji myšlenku rozvíjet dál a napadlo ho, že kdyby si koupil dostatečné množství domů, mohl by se procházet po celé Fiktivní Praze. Zajímalo by ho ale, kolik nejméně bude muset utratit, aby svého cíle dosáhl.

Fiktivní Praha se rozkládá na čtverečkové síti o rozměrech $2^{14} \times 2^{14}$ (neboli $16\,384 \times 16\,384$) políček. Některá políčka obsahují domy a dají se za nějakou cenu koupit (každé políčko se dá koupit samostatně bez ohledu na to, jestli patří do nějaké budovy rozkládací se přes více políček). Mapu Fiktivní Prahy můžete vidět na následujícím obrázku – ale nebojte se, data vám naservírujeme i v jednodušší podobě.



Na webu najdete i obrázky ve vyšším rozlišení ke stažení.

Pokud někdo vlastní budovu na políčku $[x, y]$, tak se může volně pohybovat na políčkách od $[x - 500, y - 500]$ až po $[x + 500, y + 500]$ včetně (mohli bychom říci, že může do vzdálenosti 500 od $[x, y]$ v maximové metrice).

Fiktivní Kevin by chtěl mít „zpřístupněná“ všechna zastavěná políčka, ale chtěl by za to utratit co nejméně. Vaším úkolem bude najít množinu políček, na nichž stojí domy a jejichž nákupem se zpřístupní všechna ostatní zastavěná políčka v mapě. Z takových množin políček se budete pokoušet najít tu, za kterou utratíte co možná nejméně (tedy budete minimalizovat součet cen vybraných políček).

Toto je speciální **kompetitivní úloha se statických vstupem**. Všichni řešitelé dostanou stejný vstup a přes Odevzdávátko pak odevzdají nejlepší řešení, které se jim povede najít. Obodování úlohy provedeme až po konci série a to tak, že nejlepší řešení dostane plný počet bodů a ostatní řešení dostanou body odstupňovaně podle toho, jak byla dobrá. Zároveň slibujeme, že každé korektní řešení (zpřístupňující všechna zastavěná políčka) dostane alespoň jeden bod.

V průběhu série se můžete s ostatními porovnávat pomocí průběžné online výsledkovky – upozorňujeme, že se v ní mohou vyskytnout i řešení od organizátorů.

Formát vstupu: Vstup je statický a stačí si ho stáhnout jen jednou. Je tvořen mapou cen v binárním souboru, který se sestává ze $16\,384 \times 16\,384$ dvoubajtových čísel bez znaménka (Cčkový datový typ `uint16_t`) zapsaných v little-endian formátu (méně významný bajt první a více významný bajt druhý). Data jsou uvedeny po řádcích a začínají v levém vrchním rohu mapy. S načtením vám mohou pomoci ukázky kódu níže.

Načtená čísla udávají cenu každého políčka (v korunách za metr čtvereční stavební parcely). V případě, že je číslo nula, tak se nejedná o zastavěné políčko (a není ho nutné zpřístupňovat, jedná se například o řeku nebo pole).

Načtení v jazyce C

```
#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>
#define S 16384

// Funkce na převod little-endian uint16_t
// do endianity procesoru (ať je jakákoliv)
uint16_t le16_to_cpu(const uint8_t *buf) {
  return (buf[0]) | (buf[1] << 8);
}

int main() {
  uint8_t *buffer = malloc(S*S*2);
  if (buffer == NULL) die("Nelze alokovat");
  fread(buffer, sizeof(uint8_t), S*S*2, stdin);

  uint16_t **mapa = malloc(sizeof(uint16_t*)*S);
  for (int r = 0; r < S; r++) {
    // Použijeme trik, kdy mapa bude bydlet
    // v prostoru alokovaném pro buffer, jenom
    // každé číslo převedeme funkcí, která
    // zajistí správnou endianitu
    mapa[r] = (uint16_t*)&buffer[2*r*S];

    for (int s = 0; s < S; s++) mapa[r][s] =
          le16_to_cpu(&buffer[2*r*S + 2*s]);
  }
}
```

Načtení v Pythonu 3

```
import sys
import numpy
from array import array
S = 16384

mapa = []
for y in range(S):
    # Musíme dát pozor na endianitu (data jsou
    # little-endian) a tak načtení svěříme numpy
    numpy_radek = numpy.fromfile(
        sys.stdin.buffer,
        dtype="<u2",  # unsigned 2bajtová čísla
        count=S       # v little-endian(<)
    )
    # Pro rychlejší přístup k prvkům převedeme
    # na interní Pythoní pole
    radek = array('H')  # minimálně 2b číslo
    radek.fromlist(numpy_radek.tolist())
    mapa.append(radek)
```

Formát výstupu: Výstup bude mít textový formát. Na první řádek výstupu uveďte počet domů, které kupujete, a na další řádky uveďte souřadnice zakoupených domu, na každý řádek jeden dům. Souřadnice uveďte jako číslo řádku a pak číslo sloupce oddělené mezerou (oboje indexujeme od nuly a bod $[0, 0]$ je levé vrchní políčko).

Maximální počet odevzdaných domů je 10 000 (do toho by se všechna rozumná řešení měla vejít).

Odevzdávátko spočítá cenu za nakoupené domy a přidá odevzdané řešení do průběžné výsledkovky.[1] Upozorňujeme, že od každého řešitele bereme v potaz vždy jeho poslední odevzdané řešení, i kdyby si tím měl zhoršit skóre. Proto vám doporučujeme si své řešení ukládat, abyste je případně mohli odevzdat znovu.

Zdroje: Mapu s cenami k této úloze jsme získali zpracováním dat OpenStreetMap (pozice domů na mapě) a také otevřených datových sad poskytovaných Hlavním městem Prahou (ceny za dům v jednotlivých oblastech).

33-3-X1 Z(a)tracené kouzlo — 10 bodů

Toto je bonusová úloha pro zkušenější řešitele, těžší než ostatní úlohy v této sérii. Nezískáte za ni klasické body, nýbrž dobrý pocit, že jste zdolali něco výjimečného. Kromě toho za správné řešení dostanete speciální odměnu a body se vám započítají do samostatných výsledků KSP-X.

Kouzelník Magiáš už několik hodin soustředěně hryže špičku kouzelnické hůlky. Rád by nabubřelému starostovi sousední vsi přičaroval oslí uši – to vždycky byla taková zábava! – jenže se mu nedaří vzpomenout si na správné zaklínadlo. Takhle přikouzlí oslí uši nanejvýš svým grimoárům.

Ale co už, kde nepomůže paměť, pomůže technika. Magiáš si je jistý, že kdyby zaklínadlo viděl, okamžitě ho pozná. Vytahuje tedy z truhlice počítač, pořádně zatočil klikou a vygeneroval náhodnou posloupnost znaků. Jestlipak v ní najde, co hledal?

Vymyslete algoritmus, který dostane nějakou *abecedu* (konečnou množinu obsahující A znaků), *zaklínadlo* (řetězec délky Z tvořený znaky abecedy) a číslo N. Výstupem algoritmu bude počet řetězců o N znacích abecedy, které obsahují zaklínadlo jako souvislý podřetězec. Jelikož výsledek může být obrovské číslo, počítejte modulo nějaké velké prvočíslo P.

Můžete předpokládat, že N je mnohem větší než Z a A.

Příklad: Uvažujme abecedu $\{\texttt{A}, \texttt{B}, \texttt{C}\}$. Všech řetězců délky $N = 5$ existuje $3^5 = 243$. Zaklínadlo `ABC` se vyskytuje ve 27 z nich, zaklínadlo `AAA` jen ve 21.

33-3-S Světlo a stín — 14 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 jej nejméně přečíst. A pokud si chcete úlohy z minulých dílů také naprogramovat, stále za ně můžete získat polovinu bodů.

Minule jsme si pořídili několik šumových funkcí. Nyní se podíváme, jak se z nich dá poskládat konkrétní povrch. Barva povrchu samotná ale není tolik zajímavá, proto zároveň s ní vygenerujeme i výškovou mapu povrchu, který definuje jeho tvar. Bude se nám hodit, až se naučíme simulovat chování světla a nějak zajímavě povrch osvětlíme. Nejdříve ale musíme mít co osvětlovat.

V tomto díle vás čekají jen dva úkoly. Prvním je udělat nějaký proceduální obraz nebo povrch, druhým je implementovat Phongův osvětlovací model. Jako návod pro první úkol je zde popis, jak udělat proceduální cihlovou zeď, po něm následuje teorie o světle a nakonec zadání druhého úkolu.

Proceduálně generovaný obraz

Úkol 1 [6b]:

Vytvořte v Shadertoy[2] nějaký libovolný, zajímavý, proceduálně generovaný obraz. Může být i animovaný. Mlhovina, vodní hladina, nějaký zajímavý vzor…

Pravděpodobně se vám budou hodit šumové funkce z minulého dílu seriálu. Mnoho inspirace a návodů naleznete v The Book of Shaders. Fantazii se meze nekladou. Nemusí se jednat o nic složitého, ale mělo by to být něco více než zkopírovaný kód ze seriálu nebo jiných zdrojů.

Též můžete vytvořit něco napodobující realistický povrch. Níže najdete právě příklad toho, jak ze šumů poskládat cihlovou zeď. Nemusí to být tak komplikované jako tento příklad (určitě nedělejte nic složitějšího) a můžete využívat kousky kódu ze seriálu. Pokud se vydáte touto cestou, soustřeďte se i na výšku povrchu, později ji využijeme.

Také vězte, že za vytvořením čehokoliv ze šumů se neskrývá žádný pevný postup, spíše se věci hackují dokud nevypadají tak jak mají. Příklad cihlové zdi níže je zde jako návod a inspirace.

A pokud se vám editor seká, zapausujte si překreslování obrazu tlačítkem v levém dolním rohu. Obraz se stále obnoví, pokud někam kliknete myší nebo překompilujete shader (na což se hodí klávesová zkratka `alt+enter`).

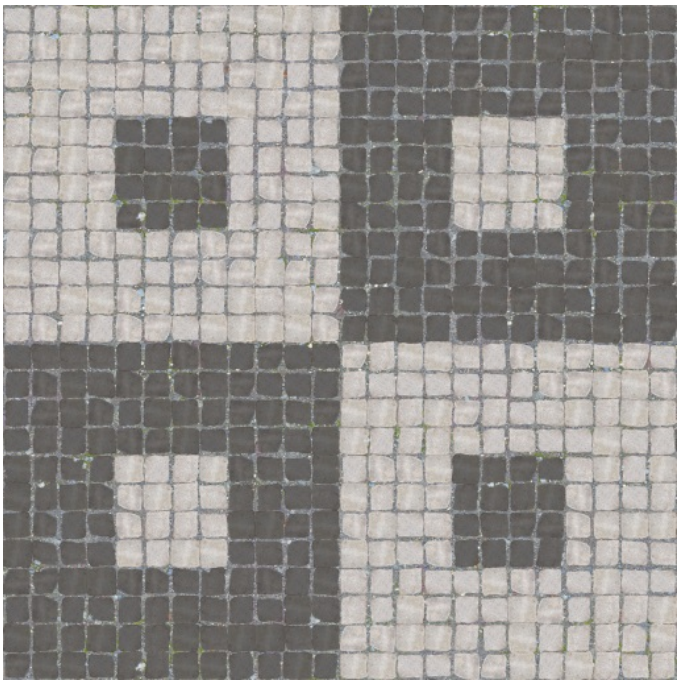
Bodování (celkem 6 bodů):

- základ 4 body za netriviální funkční obraz
- pokud se rozhodnete dělat povrch, +2 body za smysluplnou výškovou mapu
- pokud se rozhodnete dělat něco jiného, +2 body za smysluplnou animaci

Na další stránce naleznete pár nápadů na povrchy.

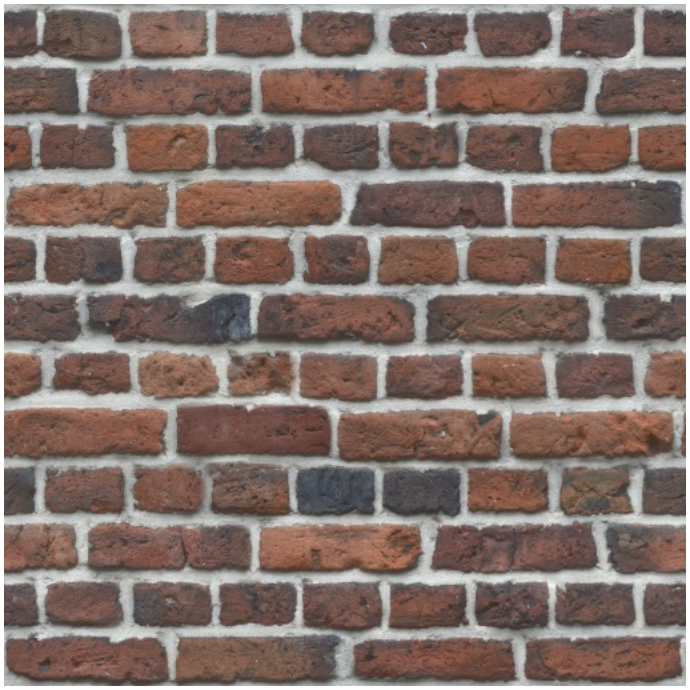
[1] `https://ksp.mff.cuni.cz/h/ulohy/33/33-3-4/vysledky`

[2] `https://www.shadertoy.com/new`



Od šumů k povrchu

Pokusíme se vytvořit funkci napodobující cihlovou zeď:



Začneme nejnápadnější strukturou v tomto povrchu: obdélníkovými cihlami. Obdélník není nic jiného než protáhlý čtverec, a jak si možná pamatujete z minulého dílu, se čtvercovými buňkami jsme se už setkali (u Perlinova šumu):

```
vec4 brick(vec2 uv)
{
    // Souřadnice v rámci buňky
    vec2 f = fract(uv);

    float height = max(f.x, f.y);

    return vec4(0.0, 0.0, 0.0, height);
}

void mainImage(out vec4 fragColor,
               in vec2 fragCoord)
{
    vec2 uv = fragCoord/iResolution.xy;

    // Převedeme uv do rozsahu -1..1
    uv = uv * 2.0 - 1.0;
    // "Natáhneme" X tak, aby byl zachován
    // poměr stran
    uv.x *= iResolution.x / iResolution.y;
    // Nyní je tedy čtverec v souřadnicích
    // i čtvercem na monitoru

    vec4 surface = brick(uv);

    fragColor = vec4(surface.www, 1.0);
}
```

Funkce `brick` zatím tvoří jen jakési čtvercové artefakty. Všimněte si, že tato funkce nevrací jen barvu povrchu v daném bodě, nýbrž i jeho výšku (jako komponentu w výsledného vektoru). Zatím na monitor zobrazujeme jen tuto výšku. Tmavé (malé) hodnoty výšky jsou dál od pozorovatele (hlouběji v povrchu), světlé jsou blíže. Výška se nám bude hodit, až budeme povrch osvětlovat.

Na povrchu chceme nějak zvýraznit okraje „cihlových“ buněk, kde bude vidět malta.

```
vec4 brick(vec2 uv)
{
    // Souřadnice v rámci buňky
    vec2 f = fract(uv);

    // Převedeme je do rozsahu -1..1
    f = f * 2.0 - 1.0;

    f = abs(f);

    float height = max(f.x, f.y);

    return vec4(0.0, 0.0, 0.0, height);
}
```

Tím, že jsme převedli `f` (tedy souřadnice uvnitř buňky) do rozsahu -1 až 1, tak platí, že u kraje buňky je aspoň jedna ze souřadnic v absolutní hodnotě blízko jedničce. Výšku získáme tím, že z absolutních hodnot x a y složek `f` vezmeme maximum.

Hodnotu `f` nyní nějak upravíme tak, aby výsledek vypadal trochu blíž tvaru cihel v předloze. Výšku invertujeme, aby platilo, že malta je hlouběji v povrchu.

```
vec4 brick(vec2 uv)
{
    // Souřadnice v rámci buňky
    vec2 f = fract(uv);

    // Převedeme je do rozsahu -1..1
    f = f * 2.0 - 1.0;

    f = abs(f);

    float height = max(f.x, f.y);
    height = max((height - 0.75) * 4.0, 0.0);
    height = pow(height, 3.0) * 3.0;
    height = 1.0 - clamp(height, 0.0, 1.0);

    return vec4(0.0, 0.0, 0.0, height);
}
```

Umocnění na třetí má za následek ostřejší přechod mezi černou a bílou. Násobení 4.0 a odečtení 0.75 zařídí, aby hodnoty z `max(f.x, f.y)` byly po tři čtvrtě vnitřku cihly rovny nule a až v poslední čtvrtině narostly do jedničky. Jinak by se celý vnitřek cihly ostře svažoval k okrajům (schválně to zkuste), takto je většina svahu „useklá“ tím, že jsme ji přesunuli do záporných hodnot. Poslední násobení třemi ovládá šířku okraje cihel.

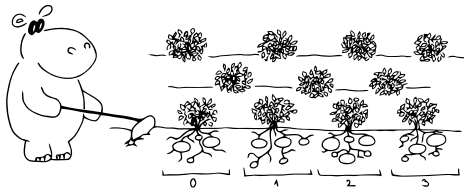
Zatím vidíme jen dvě řady cihel, chceme jich ale mít v rozsahu -1 až 1 více. Proto vstupní `uv` něčím vynásobíme. Všimněte si, že x-ovou komponentu `uv` násobíme menší hodnotou než y-ovou, tím cihly protáhneme podle x.

Též chceme každou druhou řadu cihel posunout, k čemuž použijeme už spočítané celočíselné souřadnice buňky v proměnné `cell`. Výsledný kód vypadá takto:

Také jsme si pro transformované souřadnice cihly zavedli proměnnou `brickuv`, jejíž celá část identifikuje cihlu a desetinná souřadnice uvnitř cihly. Původní netransformované `uv` se nám totiž bude později hodit.

```
vec4 brick(vec2 uv)
{
    // Chceme vidět v rozsahu -1..1
    // více řad cihel, navíc cheme,
    // aby byly cihly obdélníkové
    vec2 brickuv = uv * vec2(2.0, 4.0);

    int row = int(floor(brickuv.y));

    // Každou druhou řadu cihel posuneme
    if((row & 1) == 0)
        brickuv.x += 0.5;

    // Souřadnice v rámci buňky
    vec2 f = fract(brickuv);

    // Převedeme je do rozsahu -1..1
    f = f * 2.0 - 1.0;

    f = abs(f);

    float height = max(f.x, f.y);
    height = max((height - 0.75) * 4.0, 0.0);
    height = pow(height, 3.0) * 3.0;
    height = 1.0 - clamp(height, 0.0, 1.0);

    return vec4(0.0, 0.0, 0.0, height);
}
```

Výraz `(row & 1)` je bitový AND, který vrací jedničku právě když má `row` nastavený nejnižší bit. V podmínce tedy posune každý sudý řádek.



Nyní do tvaru cihel přidáme trochu chaosu. Použijeme k tomu *fractal brownian motion* z minulého dílu:

```
vec2 random2(vec2 st)
{
    vec2 s = vec2(
        dot(st, vec2(12.3456, 34.1415)),
        dot(st, vec2(42.2154, 15.2854))
    );
    return fract(sin(s) * 45678.9) * 2.0 - 1.0;
}

float gradientNoise(vec2 st)
{
    vec2 cell = floor(st);
    vec2 f = fract(st);

    vec2 s00 = random2(cell);
    vec2 s01 = random2(cell + vec2(0, 1));
    vec2 s10 = random2(cell + vec2(1, 0));
    vec2 s11 = random2(cell + vec2(1, 1));

    float d00 = dot(s00, f);
    float d01 = dot(s01, f - vec2(0, 1));
    float d10 = dot(s10, f - vec2(1, 0));
    float d11 = dot(s11, f - vec2(1, 1));

    vec2 u = smoothstep(0.0, 1.0, f);

    float noise = mix(
        mix(d00, d10, u.x),
        mix(d01, d11, u.x),
        u.y
    );
    return noise * 0.5 + 0.5;
}

float fbm(vec2 st)
{
    float val = 0.0;

    float p = 0.5;
```

```
    const float angle = 1.0;

    mat2 rotation = mat2(
        cos(angle), sin(angle),
       -sin(angle), cos(angle)
    );

    for (int i = 0; i < 5; i++)
    {
        val += gradientNoise(st) * p;
        p *= 0.5;
        st *= 2.0;

        st += vec2(1.181, 0.57);
        st = rotation * st;
    }

    return val;
}

vec4 brick(vec2 uv)
{
    // Chceme vidět v rozsahu -1..1
    // více řad cihel, navíc cheme,
    // aby byly cihly obdélníkové
    vec2 brickuv = uv * vec2(2.0, 4.0);

    int row = int(floor(brickuv.y));

    // Každou druhou řadu cihel posuneme
    if((row & 1) == 0)
        brickuv.x += 0.5;

    vec2 f = fract(brickuv);

    // Převedeme je do rozsahu -1..1
    f = f * 2.0 - 1.0;

    // Souřadnice, ze kterých bereme šum
    // pro cihly, jsou různé pro každou buňku
    vec2 noiseuv = brickuv * 2.0
        + floor(brickuv) * 42.2;
    // Náhodný šum převedený do rozsahu -1..1
    vec2 noise = vec2(
        fbm(noiseuv),
        fbm(noiseuv + vec2(12.244, 321.25))
    ) * 2.0 - 1.0;

    f = abs(f + noise * 0.25);

    float height = max(f.x, f.y);
    height = max((height - 0.75) * 4.0, 0.0);
    height = pow(height, 3.0) * 3.0;
    height = 1.0 - clamp(height, 0.0, 1.0);

    // Šum pro maltu v rozsahu 0..1
    // Používáme původní uv, nikoliv brickuv
    float backgroundNoise = fbm(uv * 64.0);

    height = clamp(
        height + backgroundNoise * 0.25,
        0.0, 1.0);

    // Šum pro cihly v rozsahu 0..1
    float foregroundNoise = fbm(uv * 16.0);

    // Šum umocníme na druhou
    foregroundNoise *= foregroundNoise;

    height = clamp(height
        - height * (foregroundNoise)* 0.5,
        0.0, 1.0);

    return vec4(0.0, 0.0, 0.0, height);
}
```

Šum spočítáme dvakrát, jednou pro x a jednou pro y. Aby byly tyto hodnoty různé, u y ke vstupu 'fbm' přidáme nějaké posunutí. Tento dvojrozměrný výsledek prostě přičteme k souřadnicím, ze kterých počítáme tvar cihly, čímž ho zkreslíme. Též si všimněte, že k souřadnicím, ze kterých šum počítáme, přičítáme `floor(uv) * 42.2`. Tato hodnota je vždy v rámci jedné cihly stejná, pro sousední cihly se ale bude velmi lišit. Tím zajistíme, že sousední cihly budou zkresleny různě (zkuste se podívat, jak by povrch vypadal, kdybychom to nedělali).

Též zaneseme nějaký šum do nízkých míst (malta) a nějaký šum o jiné frekvenci přidáme i do cihel.

Jako poslední úpravu přidáme povrchu barvu. Malta na pozadí bude jednolitě šedá, cihly budou mít různé odstíny červené, opět získané náhodnou funkcí, která bude různá pro každou cihlu.

```
vec4 brick(vec2 uv)
{
    // ...

    // Výsledek obarvíme
    vec3 color = vec3(0.4);

    vec3 brickColor = vec3(0.8, 0.55, 0.5)
        + vec3(random2(floor(brickuv)), 0.0)
        * vec3(0.15, 0.075, 0.0);

    color = mix(color, brickColor, height);

    return vec4(color, height);
}

void mainImage(out vec4 fragColor,
               in vec2 fragCoord)
{
    // ...

    // Nyní už barevný výstup
    fragColor = vec4(surface.rgb, 1.0);
}
```

A nyní, když už máme realistický povrch, tak bychom ho mohli i realisticky osvětlit!

Světlo

Simulace světla je asi největší část počítačové grafiky a budeme se jí věnovat téměř celý zbytek seriálu.

Světlo je, aspoň pro naše účely, proud nekonečně malých částic - fotonů. Každý foton má svou vlnovou délku, která určuje jeho barvu. Například vlnová délka 440nm odpovídá modré a 650nm červené. Intenzita světla je určena množstvím fotonů. V realtime vykreslování (což je vykreslování dost rychlé na to, aby bylo interaktivní, například pro hry) vlnovou délku fotonů víceméně ignorujeme a místo toho počítáme intenzitu světla „natřikrát", jednou pro červenou, zelenou a modrou barvu. Ostatní barvy vytvoříme kombinací těchto tří, stejně jako to dělají monitory.

Tím přijdeme o některé jevy, například žlutý povrch, který odráží červené a zelené světlo se někdy chová viditelně jinak než žlutý povrch, který odráží žluté světlo. Běžně ale moc nenarazíme na situace, kde to má vliv.

Globální iluminace

Když vidíme nějaký objekt, tak to co vnímáme je světlo, které z jeho povrchu doputovalo do našich očí. Většina věcí sama viditelné světlo nevyzařuje a co vidíme je tedy světlo, které k danému povrchu doputovalo odjinud a odrazilo

se směrem k nám. Tedy tím, že na nějaký povrch dopadá světlo, se tento povrch sám stává jakýmsi zdrojem světla a toto světlo se může dále odrážet od jiných objektů.

Představte si, že jste v noci v pokoji, kde je jediným zdrojem světla žárovka. I v místech, která jsou ve stínu, je něco vidět, přestože sem nedopadá světlo z žárovky přímo. Dopadá sem ovšem světlo odražené od těch částí místnosti, které ve stínu nejsou (a proto je tak častá bílá omítka na zdech – lépe odráží světlo a v místnosti je díky tomu lépe vidět).

Tomuto světlu, které k povrchu nedoputovalo přímo, ale nějak se po cestě odráželo od zbytku *scény* (v kontextu vykreslování se prostředí, které vykreslujeme, typicky říká scéna), se říká *nepřímé světlo* (anglicky *indirect light*). Naopak tomu světlu, které ze zdroje (žárovka, slunce) doputovalo na povrch přímo se říká *přímé světlo* (*direct light*).

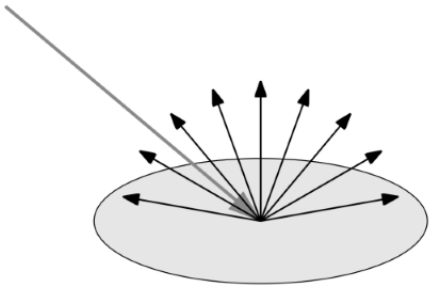
Někdy se tomuto jevu, kdy se každá část scény stáva zdrojem světla, také říká *globální iluminace*. Pokud ale nějaká hra nebo engine tvrdí, že umí globální iluminaci, většinou mají na mysli simulaci nepřímého světla vzniklého aspoň jedním difúzním odrazem (viz níže).

Nepřímé světlo je pro vzhled vykreslované scény nesmírně důležité. Nicméně je také nesmírně náročné jej simulovat. První pokusy o realtime globální iluminaci ve hrách se začaly objevovat teprve během posledních pár let a jejich plného nasazení se nejspíše dočkáme až s právě přicházející generací silnějšího hardwaru. Doteď se používala buď globální iluminace předpočítaná, nebo triky, které také později použijeme.

Odrazy

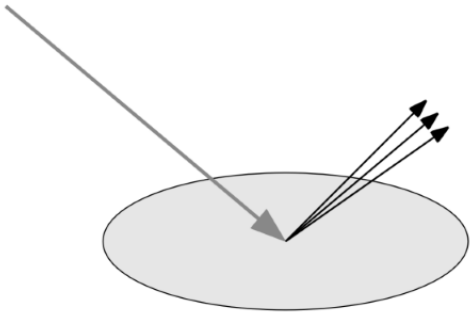
Co se stane, když světlo dopadne na nějaký povrch?

Většina světla se zlomí dovnitř povrchu, kde se bude chaoticky odrážet a rozptylovat od atomů či molekul materiálu. Část z tohoto světla materiál absorbuje, zbytek světla jej opět opustí, ovšem díky své chaotické cestě skrz materiál bude toto světlo vyzářeno do všech směrů rovnoměrně. Tomuto světlu se anglicky říká *diffuse light*, můžete se potkat i s pojmenováním jevu difúzní odrazy nebo difúzní rozptyl světla. Typicky jsou různé vlnové délky světla absorbovány různě, čímž se difúzní světlo viditelně zabarví.



Může se stát, že nezanedbatelná část světla materiál opustí viditelně jinde, než kde do něj vstoupilo (pokud je materiál tenký, tak třeba i na jeho opačné straně). Tomuto jevu se říká *subsurface scattering* a je patrný například na slunci svítícím skrz listí nebo na lidské kůži. Pro většinu materiálů jej ale lze zanedbat.

Část světla se do povrchu nezlomí a místo toho se od něj „zrcadlově“ odrazí. Tomuto světlu se říká *specular light*, odrazům se říká spekulární odrazy. Toto světlo nebývá zabarvené, protože s materiálem skoro neinteragovalo. Ovšem pro kovy může být takto odražena téměř všechna příchozí energie, navíc kovy, na rozdíl od většiny nevodivých materiálů, mají tendenci odražené světlo silně zabarvovat (a tím se liší banán od zlata, přestože oba materiály jsou „žluté“).



Na vzhled materiálu má vliv i jeho mikroskopický tvar. Především spekulární odrazy jsou jím velmi ovlivněny – hladké povrchy se lesknou jako zrcadlo, hrubé povrchy jsou matné.

Chování světla na povrchu je ovlivněno ještě mnoha dalšími parametry. Například na jinak hrubém povrchu může být tenká vrstva hladkého laku, která mění, jak budou vypadat odrazy. Také tvar mikroskopických nerovností má velký vliv na vzhled, zvlášť pokud jsou pravidelné, jako třeba na gramofonové desce nebo na DVD.

Parametry, kterými budeme popisovat materiál, se dají shrnout takto:

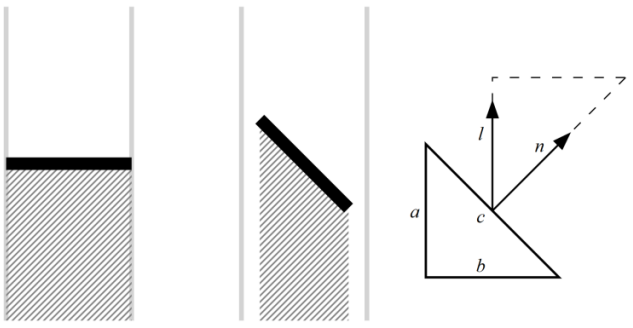
- difúzní barva – též intenzita difúzního světla, pro kovy slabá
- spekulární barva – pro nekovy bílá a slabá, pro kovy zabarvená a silná
- hrubost – ovlivňuje ostrost či matnost odrazů

Barvou v tomto případě myslíme jak odstín, tak světlost (tedy sílu odrazů). Větší hodnoty budou odrážet více světla.

Například Unreal Engine používá (mimo mnohé jiné) parametry *base color* a *metallic*, čili nějakou základní barvu a jednorozměrnou „kovovost“, ze kterých difúzní a spekulární barvu sám odvodí.

Lambertův zákon

Dřív, než začneme počítat, kolik světla se odrazilo z povrchu do našich očí (či spíše do naší virtuální kamery), je třeba zjistit, kolik světla na povrch dopadá. Sice počítáme osvětlení v nekonečně malém bodě, ale nyní si představme, že tento bod je ve skutečnosti malá ploška nějaké pevně dané velikosti, a víme, kolik světla by v tomto místě dopadlo na plochu stejného obsahu. Energie, která na tuto plošku skutečně dopadne, se ale bude výrazně lišit podle toho, jak je ploška orientovaná vzhledem ke zdroji světla.



Jak je vidět z obrázku, tak když je ploška orientovaná kolmo na příchozí paprsky světla, dopadá na ni všechna energie, ale pokud je nějak natočená, dopadá na ni jen část energie

a zbytek ji mine. Tato část energie je navíc přímo úměrná šířce „stínu“ plošky – čím širší stín, tím víc fotonů narazilo do plošky.

Poměr zachycené energie je tedy rovný poměru mezi velikostí plošky a velikostí stínu, v trojúhelníku toto odpovídá poměru stran b/c, tedy kosinu úhlu mezi těmito stranami. Tento úhel nalezneme též mezi zvýrazněnými vektory n a l (všimněte si, že dohromady tvoří trojúhelník podobný hlavnímu abc), kde n je vektor kolmý na plošku a l je vektor směřující ke světlu. Tomuto vztahu se říká Lambertův zákon, a toto je vše co potřebujeme pro spočítání (aspoň přibližného) difúzního osvětlení.

Spekulární odraz je složitější a existuje mnoho způsobů jak ho spočítat. My použijeme *Phongův osvětlovací model*. Je rychlý a jednoduchý, ale není nijak založený na fyzice světla, je navržen prostě tak, aby vypadal víceméně přirozeně.

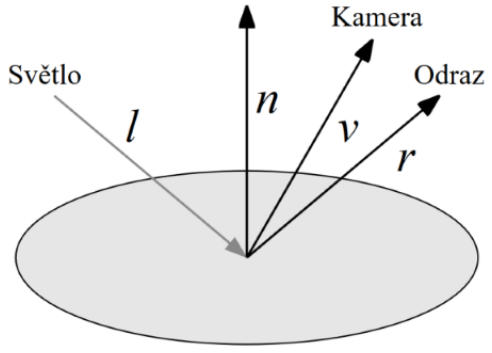
Pokud počítáme s tím, že náš zdroj světla je bod někde v prostoru, ještě potřebujeme brát v potaz vzdálenost našeho bodu od zdroje světla. Dejme tomu, že náš zdroj světla vysvítí do prostoru kolem sebe rovnoměrně nějakou energii. Co se stane, když se budeme od zdroje vzdalovat? Pokud si představíme kouli, jejíž střed je náš zdroj světla, vždy na ni dopadne všechna jeho energie. Když se bude poloměr koule (vzdálenost od zdroje) zvětšovat, celková plocha koule poroste s druhou mocninou jejího poloměru (povrch koule je roven $4pir^2$). Tedy „hustota“ energie bude s rostoucí vzdáleností od zdroje kvadraticky klesat.

Intenzita světla v nějakém bodě, jehož vzdálenost od zdroje je r, je tedy rovna $1/r^2$. Opět to není přesný vztah, protože předpokládá nekonečně malý zdroj světla (takové neexistují) a nebere vůbec v potaz konstanty, ale pro naše účely bohatě stačí.

Phongův osvětlovací model

Ke spočítání osvětlení v daném bodě budeme muset nějak popsat celkovou situaci, k čemuž se používá několik vektorů:

- n, takzvaný *normálový vektor* či *normal vector*, je kolmý na povrch a ukazuje směrem ven z objektu – říká nám tedy jak je povrch orientovaný
- l, *light vector*, ukazuje z aktuálního bodu směrem ke zdroji světla (tedy je rovnoběžný s příchozími paprsky)
- v, *view vector*, ukazuje z bodu směrem k pozorovateli (oku, kameře)
- r, vektor odrazu, ukazuje z bodu směrem kam by se světlo zrcadlově odrazilo (pokud by byl povrch absolutně hladký)



Úhel potřebný pro Lambertův zákon je vlastně úhel mezi vektory n a l. Připomeňme, že kosinus úhlu mezi dvěma vektory lze snadno spočítat pomocí skalárního součinu těchto dvou vektorů (v GLSL funkce `dot`), jen musí mít oba délku jedna, tedy musí být *normalizované* (v GLSL funkce `normalize`). Předpokládáme, že všechny vstupní vektory těchto rovnic normalizované jsou.

Pokud si nejste úplně jistí, co to je normalizace nebo skalární součin, odpověď najdete v našem krátkém úvodu do vektorů.[3]

Další hodnoty popisující situaci:

- I – příchozí světlo
- C_d – odchozí difúzní světlo
- C_s – odchozí spekulární světlo
- K_d – difúzní barva povrchu
- K_s – spekulární barva povrchu
- a – „lesklost“ povrchu (vyšší hodnoty jsou lesklejší)

Nyní můžeme formulovat rovnice pro oba typy světla. Začneme s difúzním:

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

Je vidět, že se nejedná o nic jiného než aplikaci Lambertova zákona na příchozí světlo a absorbci části světla materiálem. Tento model není *zcela* přesný, nicméně pro naše účely bohatě stačí. Zatím.

V praxi je potřeba si pohlídat, že pokud je výraz $n \cdot l$ záporný, tak místo něj použijeme nulu. To nastane, pokud se světlo nachází za rovinou povrchu (a tedy stejně jej nemůže nikdy osvětlit).

Spekulární světlo:

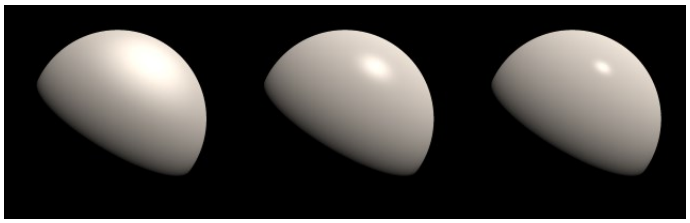
$$C_s = IK_s(n \cdot l)(v \cdot r)^a$$

Druhý skalární součin vyjde blízko k jedné jen tehdy, pokud je pozorovatel zhruba ve směru, kam se světlo odráží. Vyšší mocniny této hodnoty poté vedou k ostřejšímu, méně rozptýlenému odrazu. Je třeba ošetřit, aby skalární součiny nebyly záporné. To by nastalo, jakmile by úhel mezi vektory byl vyšší než 90°. V takovém případě místo hodnoty skalárního součinu použijeme nulu, čehož se v praxi docílí pomocí `max(dot(...), 0.0)`.

Odrazový vektor r počítáme pomocí zabudované funkce GLSL `reflect`.[4]

Parametr a je jakási lesklost povrchu. Stejně jako celá spekulární část Phongova modelu není nijak fyzikálně založený. Má vliv na velikost či ostrost spekulárních odrazů. Na obrázku níže jsou demonstrovány (zleva) hodnoty 4, 32 a 128.

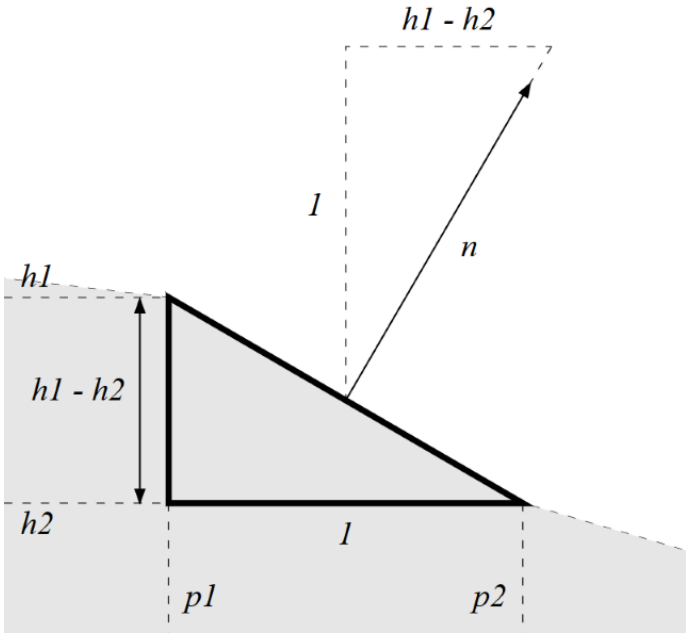
Jak difúzní světlo C_d a spekulární světlo C_s zkombinovat? Stačí je prostě sečíst. Pokud by scéna byla osvětlená více zdroji světla, tak bychom výsledné světlo od každého také posčítali. To stejné uděláme v příštím díle, až potkáme ambientní světlo.



[3] `http://ksp.mff.cuni.cz/encyklopedie/vektory.html`
[4] `https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/reflect.xhtml`

Normály

Nyní potřebujeme najít normálový vektor v libovolném bodě našeho povrchu. K tomu nám poslouží právě výška. Naklonění nějakého bodu můžeme zjistit z rozdílu výšky oproti jeho sousedům (posunutých o konstantu) v obou rozměrech:



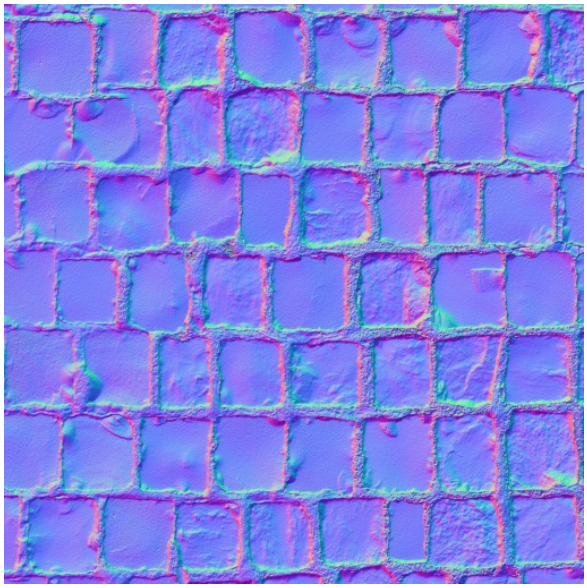
V bodě p_1 jsme spočítali výšku h_1, v bodě p_2 (vzdáleného od p_1 o jednu jednotku) výšku h_2, známe tedy rozměry tlustě nakresleného trojúhelníku, jehož horní strana reprezentuje povrch v tomto místě. Čárkovaný svislý trojúhelník nad ním, jehož přepona odpovídá hledanému normálovému vektoru, je mu podobný, normálový vektor tedy umíme spočítat. Z toho už vyplývá kód pro spočítání výšky:

```
float getHeight(vec2 uv) {
    return brick(uv).w;
}

vec3 computeNormal(vec2 uv)
{
    const float diff = 0.001;
    float here = getHeight(uv);

    vec3 n = vec3(
        here - getHeight(uv + vec2(diff, 0.0)),
        here - getHeight(uv + vec2(0.0, diff)),
        1.0
    );
    return normalize(n);
}
```

Normála ukazuje na našem povrchu směrem k pozorovateli, pro nás je to směrem do kladného z. Všimli jste si, že jsme se potichu přesunuli ze dvou rozměrů do tří? :-)

Zkuste si normály vizualizovat (nezapomeňte je přitom převést z rozsahu -1 až 1 do 0 až 1). Pokud se zdají být moc „ploché“ a modré, pronásobte něčím hodnoty, které vrací funkce `getHeight`.

Normálové mapy či textury se ve hrách často používají pro přidání jemných detailů do povrchů. I když je geometrie povrchu placatá, různé místa s různou normálou budou správně reagovat na světlo a vytvářet dojem výstupků a prohlubní. Níže je výřez normálové mapy, která patří k jedné z textur ze zadání prvního úkolu.



Počítáme světlo

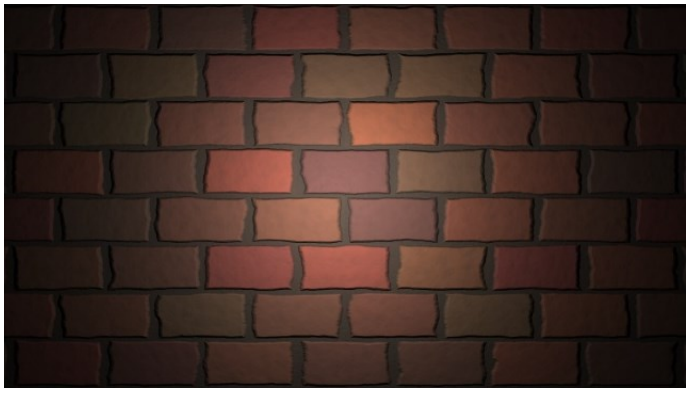
Úkol 2 [8b]:

Nasvětlete povrch (ať už cihly nebo váš vlastní) bodovým zdrojem světla, který lze ovládat pomocí kliknutí myši.

Pro získání pozice myši použijte zabudovaný vstup `iMouse.xy` (viz nápověda v Shadertoy),[5] ve kterém jsou uloženy souřadnice posledního pixelu, na který myš klikla. Nezapomeňte na ně aplikovat stejné korekce jako na souřadnice současného pixelu.

Světlu nastavte nějakou konstantní výšku nad povrchem (i když potom vizuální poloha světla nebude úplně odpovídat poloze kurzoru). Stejně tak si určete nějakou konstantní výšku kamery, aby bylo z čeho spočítat view vector. Dejte pozor na to, aby různé vektory skutečně ukazovaly, kam mají, a ne opačným směrem.

Výsledek by mohl vypadat nějak takto:



Bodování (celkem 8 bodů):

- 4 body za správnou implementaci funkce osvětlovacího modelu
- 2 body za ovládání světla myší
- 2 body za správné použití výškové mapy pro normály

Už víme, jak nějakým základním způsobem simulovat světlo. Teď by se hodilo nasvětlit něco zajímavějšího, než jen plochu. Právě v příštím díle začneme pracovat s prostorovými útvary a podíváme se na zoubek raytracingu.

Kuba Pelc

[5] `https://www.shadertoy.com/new`

Recepty z programátorské kuchařky: Treapy

Dnes si ukážeme datovou strukturu zvanou *treap*. Treapy jsou odrůdou binárních vyhledávacích stromů. Ovšem místo abychom je pracně udržovali vyvážené, použijeme jednoduché pravidlo založené na náhodných číslech. Díky němu budou treapy v průměru stejně rychlé jako vyvážené stromy, ale budou mnohem jednodušší. Také treapy naučíme mnoho dalších operací a nakonec vymyslíme, jak pomocí nich reprezentovat posloupnosti. Začneme ovšem trošku netradičně binárním vyhledáváním.

Binární vyhledávání

Při konstrukci vyhledávacích stromů vyjdeme z binárního vyhledávání. Proto si ho nejprve připomeneme:

Máme obrovské pole setříděných záznamů, třeba identifikačních čísel studentů nejmenované univerzity (záznamy však nemusí být čísla, stačí, když jsou navzájem porovnatelné). Naším úkolem je najít záznam z v poli s N záznamy $x_1 < x_2 < \ldots < x_N$.

Při použití binárního vyhledávání neboli půlení intervalu se podíváme na prostřední záznam x_m a porovnáme s ním naše z. Pokud $z < x_m$, víme, že se z nemůže vyskytovat „napravo“ od x_m, protože tam jsou všechny záznamy větší než x_m, a tím spíše než z. Analogicky pokud $z > x_m$, nemůže se z vyskytovat v první polovině pole. V obou případech nám zbude jedna polovina a v ní budeme pokračovat stejným způsobem. Tak budeme postupně půlit interval, ve kterém se z může nacházet, až buďto z najdeme, nebo vyloučíme všechny prvky, kde by prvek mohl být.

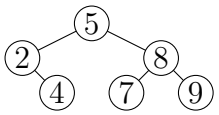
Tento algoritmus můžeme naprogramovat buďto rekurzivně, nebo pomocí cyklu, v němž budeme udržovat interval $\langle l, r\rangle$, ve kterém se hledaný prvek ještě může nacházet. My si ukážeme přístup s cyklem:

```
def bin_najdi(z):
    levy = 0
    pravy = N
    while levy <= pravy:
        median = (levy+pravy)//2
        # hledaná hodnota je vlevo
        if z < x[median]:
            pravy = median - 1
        # je vpravo
        elif z > x[median]:
            levy = median+1
        # našli jsme přímo hodnotu
        else:
            return median
        # hledaná hodnota nebyla nikde
        return -1
```

Binární vyhledávání je velmi rychlé, pokud máme možnost si data předem setřídit. Jakmile ale potřebujeme za běhu programu přidávat a odebírat záznamy, se zlou se potážeme. Buďto budeme mít záznamy uložené v poli a pak nezbývá než při zatřiďování nového prvku ostatní „rozhrnout“, což může trvat až N kroků, anebo si je budeme udržovat v nějakém seznamu. Do něho dokážeme přidávat v konstantním čase, jenže pak pro změnu nebudeme při vyhledávání schopni najít tolik potřebnou polovinu.

Vyhledávací stromy

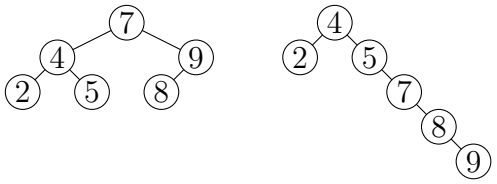
Představme si, jakými všemi možnými cestami se může v našem poli binární vyhledávání ubírat. Na začátku porovnáváme s prostředním prvkem a podle výsledku se vydáme jednou ze dvou možných variant (nebo rovnou zjistíme, že se jedná o hledaný prvek, ale to není moc zajímavý případ). V každé variantě nás zase čeká porovnání se středem příslušného intervalu a výsledek nás opět pošle jednou ze dvou dalších variant atd. To můžeme přehledně popsat pomocí stromu:



Jeden vrchol stromu prohlásíme za kořen a ten bude odpovídat celému poli (a jeho prostřednímu prvku). K němu budou připojené vrcholy obou polovin pole (opět obsahující příslušné prostřední prvky) a tak dále. Ovšem jakmile známe tento strom, můžeme náš půlící algoritmus provádět přímo podle stromu (ani k tomu nepotřebujeme vidět původní pole a umět v něm hledat poloviny): začneme v kořeni, porovnáme a podle výsledku se buďto přesuneme do levého, nebo pravého podstromu, a tak dále. Každý průběh algoritmu bude tedy odpovídat nějaké cestě z kořene stromu do hledaného vrcholu.

V každém vrcholu budeme mít uložený *klíč*, což bude hodnota onoho prostředního prvku. Podle klíčů se pak bude porovnávat.

Teď si ale všimněte, že aby hledání klíče podle stromu fungovalo, strom vůbec nemusel vzniknout půlením intervalu – stačilo, aby v každém vrcholu platilo, že všechny klíče v levém podstromu jsou menší než tento vrchol a naopak klíče v pravém podstromu větší. Hledání v témže poli by také popisovaly následující stromy (např.):



Hledací algoritmus podle jiných stromů samozřejmě už nemusí mít pěknou logaritmickou složitost (kdybychom hledali podle „degenerovaného“ stromu z pravého obrázku, trvalo by to dokonce lineárně). Důležité ale je, že takovéto stromy se dají poměrně snadno modifikovat a že je při troše šikovnosti můžeme udržet dostatečně podobné ideálnímu půlení intervalu. Pak bude hloubka stromu stále $\mathcal{O}(\log N)$, tím pádem i časová složitost hledání, a jak za chvilku uvidíme, i mnohých dalších operací.

Vyvažování stromů

Existují algoritmy, které umí udržovat stromy při přidávání a odebírání tak, aby jejich hloubka byla $\mathcal{O}(\log N)$.

Jedním z takovýchto algoritmů jsou AVL stromy.[6] Jejich implementace je ale poměrně zdlouhavá, a proto si dnes ukážeme jednodušší (v některých ohledech ale trošku horší) řešení.

[6] http://ksp.mff.cuni.cz/viz/kucharky/vyhledavaci-stromy

Treap

Při vyvažování si pomůžeme náhodou. V každý moment bude strom uspořádán nějak náhodně. Díky tomu bude opravdu malá šance, že by se jeho tvar podobal cestě.

Každému vrcholu při vytváření přiřadíme náhodně vybrané číslo z nějakého intervalu. Toto číslo označme jako *prioritu*.

Náš strom pak vždy bude splňovat tuto podmínku: *Každý vrchol má menší nebo stejnou prioritu než každý jeho syn.* Tedy strom bude na prioritách splňovat podmínky haldy. Z toho také název *treap*, který je spojením *tree* a *heap*. Pak platí následující tvrzení.

Věta: V případě, že jsou priority po dvou různé a klíče také, existuje pouze jeden možný tvar stromu.

Důkaz: Pojďme zkusit strom postavit. Do kořene musíme umístit vrchol s nejmenší prioritou. Ostatní vrcholy pak můžeme rozdělit podle toho, jestli mají větší nebo menší klíč než kořen. Z toho nám vzniknou dvě skupiny (mohou být i prázdné). Z každé musíme postavit jeden podstrom kořene. Na to rekurzivně použijeme stejný algoritmus.

Všechny kroky při vytváření ovšem jsou jednoznačné, a tedy existuje pouze jeden výsledek. *Q.E.D.*

V případě, že se vyskytne několik stejných priorit, tak už konstrukce nebude jednoznačná. Ovšem všimneme si, že rozdíly budou jen lokální. Tedy v případě, že shodných priorit nebude moc, možné tvary treapu se nebudou moc lišit. Priority tedy stačí vybírat jako (pseudo)náhodná čísla a nemusíme kolize nijak speciálně řešit.

Povšimněte si analogie s Quicksortem. V něm náhodně vybereme prvek a ostatní rozdělíme na dvě skupiny, které také zpracujeme rekurzivně. Jelikož jsou priority přidělovány náhodně, tak také vlastně vybíráme náhodný prvek. U obou algoritmů pak jen doufáme, že rekurze nebude moc hluboká, a tím pádem nám výpočty nezaberou moc času.

Přesněji řečeno platí, že pro libovolné klíče bude průměrná hloubka treapu $\mathcal{O}(\log N)$. Průměrujeme přitom pouze přes náhodný výběr priorit, na klíčích nezáleží.

Dokázat toto tvrzení je ale poměrně složité, proto ho zde uvádíme pouze bez důkazu. Zájemci si mohou přečíst článek[7] (v angličtině), případně si poslechnout přednášku.[8]

Dodejme ještě, že v případě různých klíčů existuje jen jedna možnost postavení treapu, tak nezáleží na tom, jakým postupem vznikl.

Základní operace: Merge a Split

V předchozí části jsme ukázali, co to treap je. Ovšem kdybychom s ním neuměli dělat nic jiného, než vyhledávání a postavení, tak by pro nás nebyl moc užitečný. Mohli bychom ho jednoduše nahradit binárním vyhledáváním v poli.

Nyní přidáme dvě poměrně nezajímavě vypadající operace a pomocí nich pak vybudujeme většinu ostatních operací.

Popišme vrchol treapu následující strukturou:

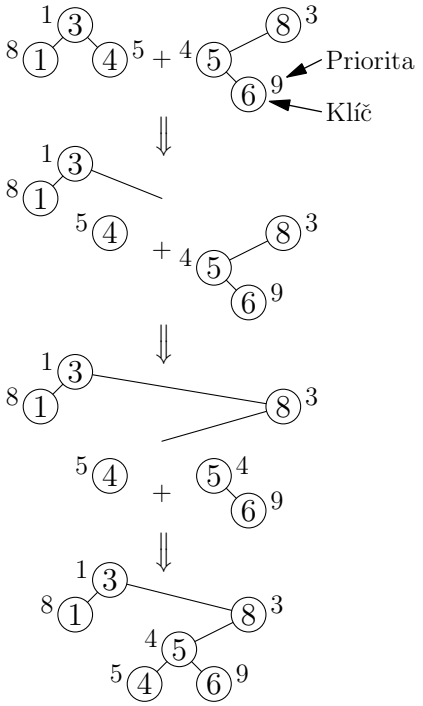
```
class Node:
    left = None
    right = None

    def __init__(self, priority, key):
        self.priority = priority
        self.key = key
```

Struktura tedy obsahuje dvě položky, kam lze umístit levý a pravý podstrom. Dále ještě obsahuje klíč a prioritu.

Merge. Tato operace vezme dva treapy a spojí je do jednoho. Ovšem musí platit, že všechny klíče v prvním z nich budou menší (nebo případně stejné) než klíče v druhém z nich.

Merge vstupní treapy rozebere a složí z nich svůj výsledek. Neprovádí žádné kopírování do nového stromu – to by bylo moc pomalé.

Pojďme nový treap postupně stavět. Sledujme obrázek:



Začneme od kořene. Tam dle definice musíme umístit vrchol s nejmenší prioritou. To ale musí být jeden z kořenů zdrojových treapů.

Dále uvažujeme, že se jedná o kořen prvního (levého) treapu. V opačném případě provedeme totéž, jen s prohozením stran. Všechny prvky v levém podstromu levého treapu musely být menší nebo stejné jako kořen, tedy mohou ve finálním treapu skončit také vlevo. Dále si povšimněme, že všechny ostatní prvky (pravý podstrom levého treapu a pravý treap) jsou větší nebo stejné jako kořen, a tedy mohou být umístěné do pravého podstromu.

Levý podstrom nového kořene tedy již máme vytvořený a stačí ho přiřadit. Pravý ještě vytvořený není. Je potřeba ho vytvořit ze dvou částí: z pravého podstromu levého treapu a z pravého treapu. Povšimneme si, že toto ale je zase Merge dvou treapů. Tedy nám stačí zavolat náš algoritmus rekurzivně.

V případě, že alespoň jeden vstup je prázdný treap, tak stačí ten druhý vrátit jako výsledek. Nahlédneme, že se naše rekurze vždy zastaví, protože se vždy zavolá na alespoň o jedna menší součet velikostí treapů. Nově vzniklý kořen už do rekurze určitě nezasahuje.

Pojďme ale tento odhad vylepšit. Všimneme si, že každé volání rekurze vrátí kořen, který se ve výsledném treapu pověsí pod kořen předchozí rekurze.

[7] http://faculty.washington.edu/aragon/treaps.html
[8] http://mj.ucw.cz/vyuka/2021/vkds/

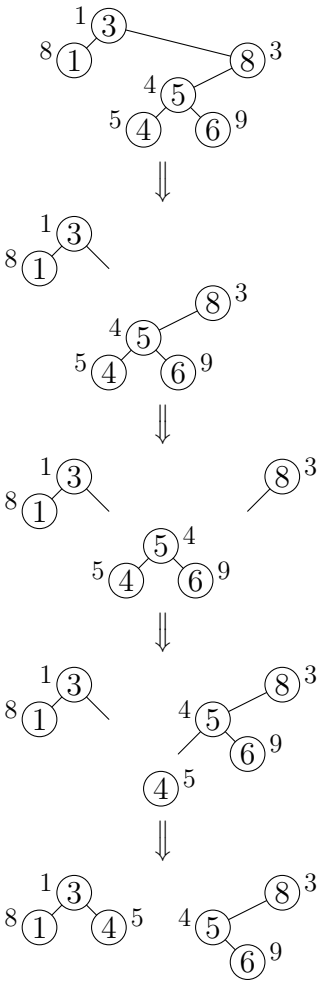
Tedy i-tý vrácený kořen bude v hloubce i. To ale znamená, že počet volání rekurze nebude větší než hloubka výsledného stromu. O ní ale doufáme, že bude malá.

```
def merge(left, right):
    if left is None or right is None:
        return left if right is None else right

    if left.priority < right.priority:
        left.right = merge(left.right, right)
        return left
    else:
        right.left = merge(left, right.left)
        return right
```

Split. Opačnou operací k Merge je Split. Ten nám treap rozřeže na dva. Na vstupu mu dáme jeden treap a klíč x. Zpět dostaneme dva treapy. První z nich obsahuje pouze položky, jejíž klič je menší než x. Druhý obsahuje zbytek, tedy položky s klíčem $\geq x$.

Uvědomíme si, že o každém vrcholu jednoznačně víme, jestli patří do levého, nebo do pravého podstromu. Stačí jeho klíč porovnat s x.

Pojďme tedy treap párat od kořene. Opět sledujme obrázek:



Dále uvažme, že kořen patří do levého výsledku. V opačném případě provedeme totéž, jen s prohozením stran. Kořen tedy bude kořenem levého výsledku. Všimneme si, že jeho levý podstrom už dále nebudeme muset řešit.

Je potřeba tedy jen rozdělit jeho pravý podstrom. Část má zůstat jeho pravý podstrom (a tedy v levém výsledku). Zbytek pak bude pravým výsledkem.

Tedy chceme znovu rozdělit menší treap na dva podle x. To ale zvládneme rekurzivním zavoláním operace Split.

V případě, že vstup je prázdný treap, tak oba výstupy jsou také prázdné treapy. Nahlédneme, že počet operací je nejvýše hloubka vstupního treapu, protože při každém zavolání rekurze se dostaneme o jedno patro níže.

```
def split(parent, key):
    if parent is None:
        return (None, None)

    if parent.key < key:
        left = parent
        left.right, right = split(
            parent.right, key)
    else:
        right = parent
        left, right.left = split(
            parent.left, key)

    return (left, right)
```

Treap jako vyhledávací strom

Insert. Pomocí spojování a slévání si postavíme již trošku zajímavější operace. Insert správně zatřídí jeden prvek do treapu.

Treap nejprve splitneme na dva podle klíče vkládaného prvku (označme po řadě A a B). Založíme nový prvek s náhodnou prioritou. Pak už jen stačí postupně spojovat. Nejprve spojíme A a nový prvek. Pak ho ještě spojíme s B.

Všimneme si, že při každém spojování skutečně platí nutná podmínka o pořadí klíčů.

Find. Hledat již umíme. V ruce máme vždy vyhledávací strom. Pojďme na to ale používat jen předchozí operace.

Označme hledaný klíč x. Stačí rozdělit treap před požadovaným klíčem a pak pravou část ještě po něm. Tedy provést Split na klič x. Na pravý výsledek pak ještě provedeme další Split. Tentokrát ale do levého výsledku pošleme všechny hodnoty $\geq x$. Evidentně není problém vytvořit Split s neostrou nerovností. Pro celá čísla nám ale jen stačí udělat Split s hodnotou $x + 1$.

Tím dostaneme tři treapy. Prostřední z nich obsahuje prvek s hledaným klíčem (v případě, že existuje). Ovšem nesmíme zapomenout na to, že pak zase tyto tři treapy musíme spojit, protože Split původní treap zničí.

Delete. Odstranění prvku s nějakým klíčem je jen mírnou modifikací předešlého algoritmu. Prostě poté, co daný prvek najdeme, tak je zahodíme a spojíme jen zbylé dvě části.

Lower bound. Dále ukážeme, jak najít prvek, který má nejmenší klíč větší než nějaké zadané číslo x.

Treap rozdělíme podle x. Pak už víme, že požadovaný vrchol musí být nejmenší z pravé části. Ten ale umíme snadno najít. Stačí jít od kořene pořád vlevo, dokud má vrchol levého potomka. Ani zde pak nesmíme zapomenout na finální slepení pracovních částí.

Případně můžeme s treapem pracovat pouze jako s vyhledávacím stromem. Lower bound lze provést průchodem od kořene jako při hledání. Poslední prvek, na který narazíme a má klíč $\geq x$, je totiž požadovaným výsledkem.

Rozšíření o počítání prvků

Hlavní výhoda treapu od již implementovaných algoritmů ve většině programovacích jazyků je, že treap můžeme snadno implementovat a dále upravovat k obrazu svému. Díky

tomu z něho můžeme vytvořit mnohem zajímavější datovou strukturu.

Pojďme u každého vrcholu počítat velikost podstromu pod ním. Uvědomíme si, u kterých vrcholů se tyto hodnoty mění při základních operacích Split a Merge. Ostatní vybudované operace vlastně jen využívají těchto, takže je nemusíme uvažovat zvlášť. Dále ještě vytváříme samostatné vrcholy, ale to také není žádná komplikace.

U Splitu stačí přepočítat počet vrcholů u vytvořených výsledků. Všimneme si, že modifikace v podstromech se dělají jen pomocí rekurze, a tedy se také počty správně opraví před tím, než budeme počítat aktuální velikost.

U operace Merge je situace podobná. Stačí jen přepočítat hodnotu u vraceného výsledku.

Přepočítání je jednoduché, stačí sečíst počet vrcholů v podstromech (jestli existují) a k tomu přičíst jedničku za aktuální vrchol.

Všimněte si, že takto vždy budeme mít v každém vrcholu spočtenou velikost podstromu, ale i tak složitost všech operací asymptoticky nevzroste.

K čemu je počítání velikostí podstromu ale dobré?

Když najdeme nějaký vrchol, můžeme se podívat, o kolikátý vrchol v původním treapu se jedná (když si je představíme seřazené podle klíčů). Stačí spočítat počet vrcholů v před aktuálním vrcholem (tedy s menším klíčem). Jedná se o velikost treapu, který vznikne před aktuálním prvkem v při destruktivním Splitu

Můžeme také najít n-tý nejmenší prvek v treapu. Ukážeme, jak provést Split, aby v levém výsledku skončilo n vrcholů. Budeme počítat, kolik ještě vrcholů chceme umístit do levého výsledného stromu. Jelikož se bude jednat o parametr rekurze, dále budeme značit také jako x. Mírně modifikujeme naši funkci Split. Nahradíme část, kde vybíráme, jestli aktuální vrchol půjde do levého či do pravého výsledku.

V případě, že počet vrcholů v levém podstromu a v kořeni je menší nebo stejný než n, tak kořen (a tedy nutně i celý levý podstrom) půjde doleva. Z pravého podstromu pak ještě rekurzivně vybereme zbývající počet potřebných vrcholů.

Když nalevo od aktuálního vrcholu je alespoň požadovaný počet vrcholů, tak aktuální vrchol umístíme do pravého výsledku. Do levého pak rekurzivně vybereme n vrcholů z levého podstromu.

Jelikož jsme vlastně nezměnili nic jiného než způsob výběru větve programu, časová složitost a jiné charakteristiky algoritmu se nezmění.

V případě, že chceme pouze najít n-tý prvek (a ne treap kolem něho rozdělit), můžeme použít i nedestruktivního postupu. Prostě jen půjdeme od kořene a stejným způsobem budeme počítat počet vrcholů, které ještě chceme přeskočit. Když bude počet vrcholů k přeskočení menší než velikost levého podstromu, tak půjdeme do levého podstromu. Když budou stejné, tak se jedná o aktuální vrchol. Ve zbývajících případech půjdeme do pravého podstromu. Od počtu vrcholů k přeskočení ovšem budeme muset odečíst velikost levého podstromu aktuálního vrcholu.

Pojďme se podívat na implementaci.

```
class Node:
    left = None
    right = None

    def __init__(self, priority, key):
        self.priority = priority
        self.key = key
        self.size = 1

def update(node):
    if node is None:
        return

    node.size = 1
    if node.left is not None:
        node.size += node.left.size
    if node.right is not None:
        node.size += node.right.size

def merge(left, right):
    if left is None or right is None:
        return left if right is None else right

    if left.priority < right.priority:
        left.right = merge(left.right, right)
        update(left)
        return left
    else:
        right.left = merge(left, right.left)
        update(right)
        return right

def split(parent, key):
    if parent is None:
        return (None, None)

    if parent.key < key:
        left = parent
        left.right, right = split(
            parent.right, key)
    else:
        right = parent
        left, right.left = split(
            parent.left, key)

    update(left)
    update(right)
    return (left, right)

def split_by_count(parent, n):
    if parent is None:
        return (None, None)

    left_count = 1 + (0 if parent.left is None
                      else parent.left.size)

    if left_count <= n:
        left = parent
        left.right, right = split_by_count(
            parent.right, n - left_count)
    else:
        right = parent
        left, right.left = split_by_count(
            parent.left, n)

    update(left)
    update(right)
    return (left, right)
```

Reprezentace posloupností

Pojďme udělat pokus. Odstraníme z vrcholů klíče, podle kterých byl treap setříděn. Sice teď v treapu nemůžeme hledat, ale pořád s ním jdou dělat zajímavé věci.

Všimneme si, že Merge je vlastně vůbec nevyužíval. Jen prostě dostane dva treapy a „nalepí“ je za sebe. Split je

sice využíval, ale místo něho jsme před chvílí vytvořili Split za n-tým prvkem, který klíče také nepotřebuje.

Máme tedy „bezejmenné“ vrcholy, ovšem stále s dobře definovaným pořadím. Nyní do každého vrcholu připíšeme nějakou *hodnotu.* Podle té ovšem není treap setříděn – nemusí to být číslo, dokonce ani něco, co se dá porovnávat.

Všimněte si, že jsme dostali reprezentaci posloupnosti. Můžeme se totiž ptát na n-tý prvek a případně ho změnit. Navíc umíme za nebo před libovolný prvek přidat další a také posloupnosti rozdělovat a spojovat.

Každá z těchto operací nám trvá průměrně $\mathcal{O}(\log N)$ času, kde N je počet prvků posloupnosti.

Mimochodem, původní případ, kdy jsme měli hodnoty setříděně, byl obdobou binárního vyhledávání na této posloupnosti.

Intervalový strom

V předchozí části jsme vybudovali posloupnost. Pojďme na ní postavit nějaké zajímavé operace.

Pokusíme se rychle odpovídat na dotazy, jaký je součet na určitém úseku posloupnosti. Dokonce se nemusí jednat jen o součet. Stačí nám jakákoliv asociativní operace. To je taková, kde nezáleží na uzávorkování. Například může jít o maximum, součin modulo prvočíslo a podobně, nebo dokonce v případě, že hodnota je maticí, tak se může jednat o součin matic. Pro jednodušší představu ale dále budeme mluvit o součtu.

V klasickém poli by se tento problém řešil pomocí intervalového stromu.[9]

Využijeme obdobného postupu jako u počítání velikostí podstromu. V každém vrcholu si budeme pamatovat součet.

Budeme součty přepočítávat vždy ve stejné momenty jako velikosti podstromu. Tedy ze stejného důvodu budou vždy aktuální a jednotlivé operace se asymptoticky nezpomalí.

Na součet se pak zeptáme opravdu jednoduše. Treap rozřízneme na tři části: před požadovaným intervalem, požadovaný interval a část za ním. Pak už se jen jednoduše zeptáme na součet prostřední části.

Líný intervalový strom. Dále se naučíme jednotlivé úseky rychle měnit. Problematiku si ukážeme pouze na konkrétním příkladu. Ke každému prvku úseku se pokusíme přičíst zadané číslo.

V každém vrcholu si budeme pamatovat, kolik se má do každého vrcholu v jeho podstromu ještě přičíst (označme *aktualizace*). Vždy předtím, než s daným vrcholem budeme něco dělat, tuto hodnotu propagujeme do podstromu. To znamená, že ji přičteme k hodnotě aktuálního vrcholu a k hodnotám aktualizace synů.

Jelikož všechny operace provádíme od kořene, celá cesta mezi kořenem a aktuálním vrcholem bude propagovaná. Hodnota aktuálního vrcholu tedy bude správná.

Přičtení k úseku je pak jednoduché. Treap rozdělíme zase na tři části a k aktualizaci prostředního treapu přičteme danou hodnotu. Nakonec vše zase poslepujeme k sobě.

Hodnoty se sice hned nezmění ve všech vrcholech, ale máme zaručeno, že až s nimi budeme pracovat, tak už bude aktuální.

Problém ovšem nastává, když chceme toto rozšíření aplikovat spolu s dotazy na součet (nebo jinou operaci) intervalů. Při propagování aktualizací totiž budeme muset upravovat i tyto součty. Na toto bohužel neexistuje žádná obecná poučka. Vždy je potřeba se zamyslet, jak spolu interagují obě dvě funkce. Když třeba přičítáme x ke každému vrcholu, je nutné zvýšit součet podstromu o nx, kde n je počet vrcholů v podstromu. Maximum bychom ale zvýšili pouze o x.

Příklad použití treapů

Na závěr si ukážeme jednu úlohu, na kterou se treapy hodí (je to úloha P-II-1 z 69. ročníku MO kategorie P).

Zadání: Na polici je vyznačeno n pozic a na každé z nich leží jedno jablíčko. Pozice jsou očíslovány zleva doprava čísly od 0 do $n-1$. Jablíčko na pozici i má velikost a_i. Všechna jablíčka mají navzájem různé velikosti. Chceme všechna jablíčka uspořádat podle velikosti od nejmenšího po největší. Kvůli přísným hygienickým předpisům je však nesmíme přemisťovat sami, musíme na to použít techniku. V místnosti s jablíčky máme mechanické rameno. Pomocí něho můžeme sebrat jablíčka z libovolného souvislého úseku police a vrátit je zpátky na polici na jejich původní místa, ale v opačném pořadí. Tuto operaci nazveme *reverze*. Každá reverze trvá stejně dlouho bez ohledu na to, jak dlouhý úsek obracíme.

Řešení pomocí treapu: Pokusíme se postupně výsledek stavět od začátku. Začneme umísťovat jablíčka od nejmenších velikostí. Tím začne vznikat prefix výsledku, do kterého pak již nebudeme zasahovat.

Je tedy vždy potřeba najít nejmenší zatím neumístěné jablko. Pak provedeme reverzi úseku od pozice, kam ho chceme dostat, po pozici, kde je. Tím se evidentně dostane na příslušnou pozici a nenaruší se ani prefix již dobře umístěných jablíček.

Jak ale udržovat aktuální pozice jablíček a hledat nejmenší na nějakém úseku? Využijeme treap.

Přidáme línou operaci proveď reverzi na podstromu. Tu budeme vyhodnocovat jednoduše. Stačí prohodit levého a pravého potomka a u obou z nich pak také provézt reverzi. To se ale zase udělá líným vyhodnocením. Ještě si všimneme, že udělat reverzi dvakrát je jako ji neudělat vůbec.

Také budeme počítat minimum v každém podstromu. Všimneme si, že reverze ho nijak neovlivní, a tedy vzájemnou interakci reverzí a minim nemusíme nijak řešit.

Najít nejmenší prvek v úseku je pak jednoduché. Nejprve úsek ořízneme do jednoho treapu. Pak už víme, jakou má mít nejmenší prvek hodnotu. U každého vrcholu zvládneme určit, jestli se jedná o aktuální prvek, nebo hledané minimum leží v levém či pravém podstromu. Podstrom, ve kterém leží hledaný prvek, totiž musí mít příslušné minimum. Můžeme ho tedy vytvořit například funkci Split za nejmenším prvkem.

Další podobné struktury

Na závěr dodejme, že princip budování operací ze Split a Merge a ostatní triky z této kuchařky nejsou nijak specifické pro treapy. Dají se provozovat i s jinými stromovitými strukturami. Například se Splay stromy – ty jsou sice pracnější, ale zaručují složitost $\mathcal{O}(logn)$ amortizovaně. A pokud této složitosti chceme dosáhnout i v nejhorším případě, jde použít (a,b)-strom nebo červeno-černý strom, ale tam už jsou algoritmy pro Split a Merge značně komplikované.

Jiří Kalvoda & Martin Mareš

[9] `http://ksp.mff.cuni.cz/viz/kucharky/intervalove-stromy`