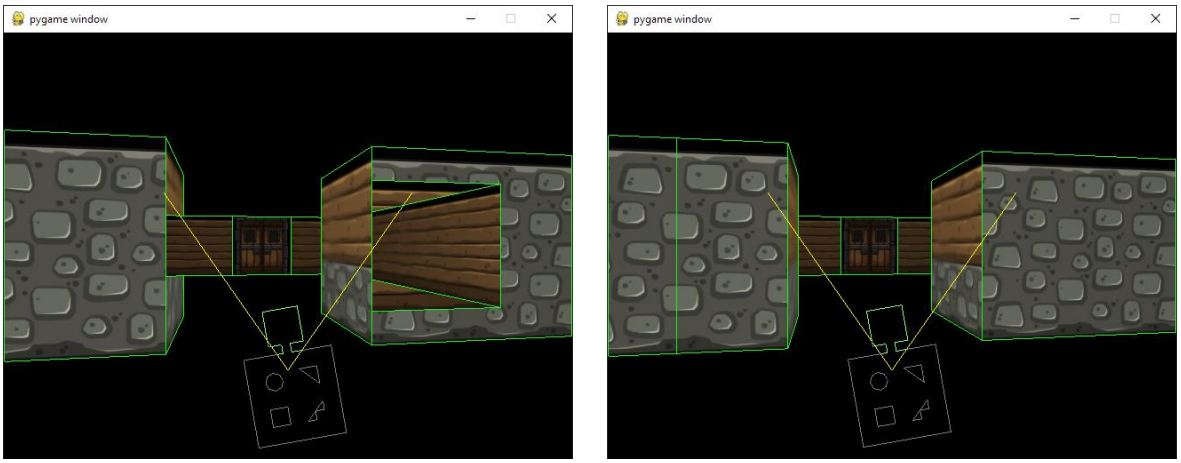
Cílem sedmé úlohy je využít modifikace binárního vyhledávacího stromu (ang. Binary Search Tree, zkráceně BST) pro řešení problému viditelnosti stěn v jednoduchém herním enginu. S obdobnou úlohou jsme se již setkali ve druhém zadání, kde jsme implementovali tzv. ray marching algoritmus pro nalezení nejbližší stěny ve scéně při pohledu z perspektivy hráče pohybujícího se po mapě tvořené pravidelnou mřížkou znaků reprezentujících překážky. Nyní se pokusíme vyřešit stejný problém pomocí stromu pro binární rozdělování prostoru (ang. Binary Space Partitioning Tree, zkráceně BSP tree) [1]. Tato technika byla využita ve známé hře Doom (1993) pro zajištění vykreslování pouze viditelných polygonů [2]. Výhoda nového přístupu spočívala v efektivnějším vykreslování viditelných polygonů, které navíc mohou být ve scéně rozmístěny téměř libovolně. Předchozí hry jako Wolfenstein 3D (1992) postavené na ray marchingu se omezovaly pouze na vykreslování zdí zarovnaných se souřadnými osami a navíc jejich rozmístění bylo svázáno s pevně daným krokem pravidelné mřížky herní mapy.

Korektního vykreslení scény je pomocí této techniky docíleno tak, že libovolně rozmístěné stěny (a případně i další herní překážky) jsou nejprve vloženy do BSP stromu. Vkládání jednotlivých překážek reprezentovaných např. úsečkami probíhá obdobným způsobem, jako je tomu při vkládání klíče (hodnoty) do binárního vyhledávacího stromu (tj. menší číslo než je hodnota daného uzlu putuje doleva, větší pak doprava). Při následném vykreslování každého nového snímku scény jsou tyto překážky během průchodu BSP stromu v určitém smyslu setřízeny podle své polohy vůči hráči. Následné vykreslení jednotlivých překážek provedené od těch nejvzdálenějších k nejbližším zajistí korektní zobrazení scény (jedná se o tzv. malířův algoritmus, ang. Painter's algorithm) [3]. Na obrázcích níže můžete vidět výsledek při vykreslení jednotlivých stěn bludiště v náhodném pořadí (vlevo) a po jejich setřízení vůči pozorovateli (vpravo).



V předpřipravené šabloně, která je součástí této úlohy, máte již hotovu většinu funkcí potřebných pro realizaci jednoduchého herního enginu postaveného na řešení problému viditelnosti pomocí BSP stromu. Stačí příslušným způsobem upravit implementaci binárního vyhledávacího stromu z předchozího zadání na BSP strom a využít ho pro správné zobrazení předdefinované scény. Následující body by vás měly postupně provést detaily implementace jednotlivých úkolů.

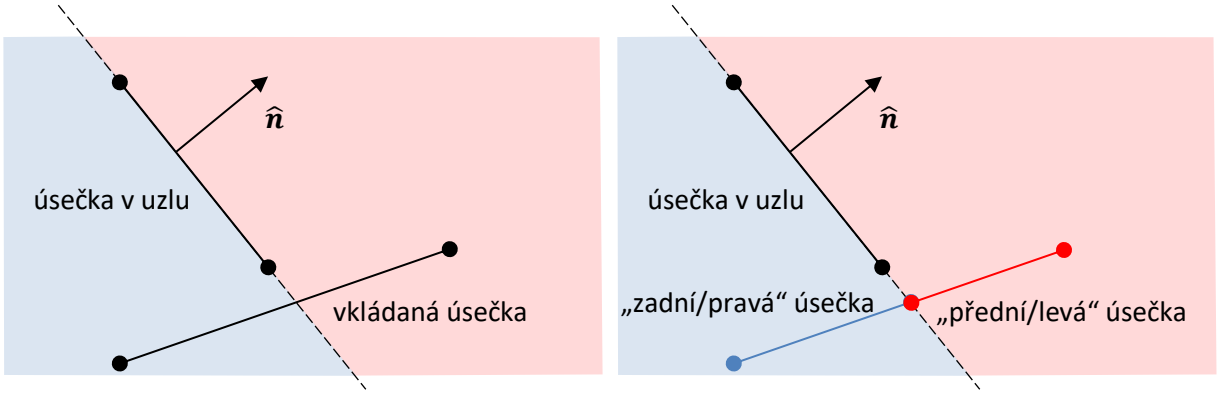
Do virtuálního prostředí si před započetím práce nainstalujte pygame a PyOpenGL (pozn., OpenGL je zde využito pouze pro efektivní vykreslení otexturovaných 2D lichoběžníků):

```
pip install pygame
pip install PyOpenGL PyOpenGL_accelerate
```

Úkol 1: (1 bod)

Vložení uzlu do BSP stromu.

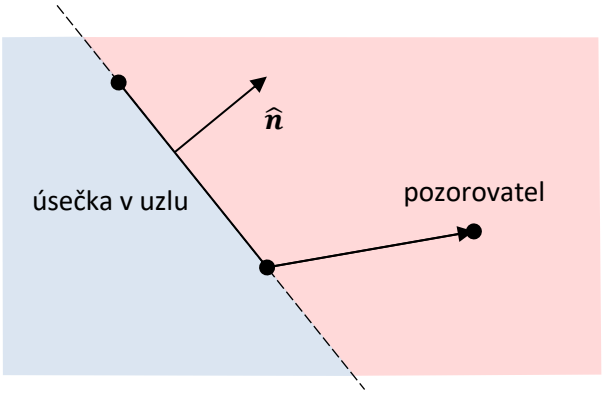
Ve třídě BSPNode, která vznikla na základě naší třídy Node z předchozí úlohy, doplňte metodu `insert` následujícím způsobem. Předem ještě poznamenejme, že vkládanou hodnotou je u námi uvažovaného BSP stromu úsečka, nikoliv číslo. Vkládanou úsečku rozdělte metodou `split_by` na potenciálně dvě nové úsečky (`front_line`, `back_line`), přičemž dělící úsečkou je ta z aktuálního uzlu. Nově vytvořenou úsečku, která leží v kladné polorovině, vložte do levé části stromu a úsečku z opačné poloroviny vložte do pravé části stromu. Z přiloženého obrázku je zřejmé, že jedna z výsledných úseček nemusí existovat.



Úkol 2: (1 bod)

Průchod stromem.

Upravte rekurzivní metodu `traverse_inorder` tak, aby přijímala bod (`Point`) reprezentující polohu pozorovatele a provedla následující operace. Nejprve zjistěte, zda se pozorovatel nachází v kladné nebo záporné polorovině. Připomeňme úmluvu z předchozího úkolu, že kladná polorovina je určena kladným směrem normály úsečky v daném uzlu. Pak platí, že pozorovatel se nachází v kladné polorovině, když výsledek skalární součinu vektoru vedoucího z jednoho ze dvou krajních bodů úsečky do bodu pozorovatele a normálového vektoru úsečky je kladný. Je-li výsledek záporný, pak se pozorovatel nachází v záporné polorovině. K další úvaze ponechme situaci, kdy je pozorovatel přesně na rozmezí polorovin a skalární součin vyjde nulový (pozn., řešení této třetí alternativy nemá zásadní vliv na výsledek).



Víme-li nyní, ve které ze dvou polorovin se pozorovatel nachází, můžeme projít strom do hloubky následujícím způsobem, který je principiálně shodný s průchodem BST: Je-li pozorovatel v kladné polorovině, navštívíme nejprve opačnou (zápornou) polorovinu (tj. pravého potomka), pokračujeme vrácením uzlové úsečky a následně navštívíme (kladnou) polorovinu pozorovatele (tj. levého potomka). V případě, že se pozorovatel vyskytuje v záporné polorovině, proces jen obrátíme.

Metodu opět realizujte jako generátor (místo `return self.__value` použijte `yield self.__value`). Zopakujme, jelikož se ale jedná o rekurzivní metodu, musíte při vracení návratové hodnoty z jejího rekurzivního volání využít místo jednoduchého `yield` variantu `yield from` z PEP 380 [4].

Úkol 3: (1 bod)

Vykreslení překážek v „back-to-front“ pořadí.

Ve třídě `BSPTree` doplňte metodu `draw`, která přijímá bod s polohou pozorovatele a vrátí „back-to-front“ setřízený seznam úseček. Opět připomeňme, že tato funkcionalita je prakticky totožná s metodou `sort` z BST.

Úkol 4: (1 bod)

Integrace `BSPNode` do `FPSEngine2`.

V metodě `InitScene` ve třídě `FPSEngine2` vytvořte privátní atribut `__scene_root`, který bude obsahovat kořen BSP stromu sestaveného z listu úseček `__scene`.

```
self.__scene_root = None # TODO put the root of the BSP tree here
```

Rovněž upravte v metodě `MainLoop` třídy `FPSEngine2` cyklus starající se o vykreslování jednotlivých stěn tak, aby tyto byly vykresleny v požadovaném pořadí odzadu. Poloha pozorovatele (hráče) je uložena v lokální proměnné `viewer`.

```
for line in self.__scene: # TODO replace this line with BSP tree in-order traverse
```

Reference

[1] S. Chen and D. Gordon. Front-to-Back Display of BSP Trees. IEEE Computer Graphics & Algorithms, pp 79–85. September 1991.
[2] https://en.wikipedia.org/wiki/Doom_engine
[3] https://en.wikipedia.org/wiki/Painter%27s_algorithm
[4] https://www.python.org/dev/peps/pep-0380/