Gymnázium, Praha 7, Nad Štolou 1

Maturitní práce

Jednoduchá počítačová hra

Autor práce: Ivan Dostál
Vedoucí práce: Bc. Martin Sourada
Ročník: 6S
Školní rok: 2020/2021

Abstrakt

Práce popisuje tvorbu jednoduché hry, ve které uživatel hraje jako had, který se sbíráním jídla zvětšuje a snaží se přimět protivníka narazit. Práce se zaměřuje zejména na naprogramování protivníka ovládaného počítačem a vysvětluje fungování algoritmů k tomu použitých.

Prohlášení

Prohlašuji, že jsem maturitní práci vypracoval samostatně, použil jsem pouze podklady uvedené v přiloženém seznamu a postup při zpracování a dalším nakládání s prací je v souladu se zákonem č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů.

V Praze dne 23. března 2021

Poděkování

Tímto bych chtěl poděkovat vedoucímu mé maturitní práce, panu Bc. Martinovi Souradovi, za jeho čas, vstřícnost, ochotu a rady, které byly pro dokončení této práce nezbytné.

Obsah

Úvod

Hra, která bude vytvářena, je založená na principu klasické arkádové hry *Snake*. Hráč v ní pohybuje hadem po čtvercové či obdélníkové hrací ploše a jeho cílem je posbírat co nejvíce jídla, které se na hrací ploše objevuje. Hráč začne jako malý had, ale po každém sebrání jídla se o trochu zvětší. Cílem hry je být co nejdelší, aniž by had naboural sám do sebe nebo do stěn, které hrací pole ohraničují. Pokud had narazí, hra končí.

V některých verzích hry *Snake* není hrací pole ohraničené stěnami a pokud z něj had vyjede na jednom kraji, tak se objeví se na opačném. Tento způsob zde ale pro jednoduchost použit nebude a hrací pole bude ohraničené stěnami, do kterých had narazit nesmí.

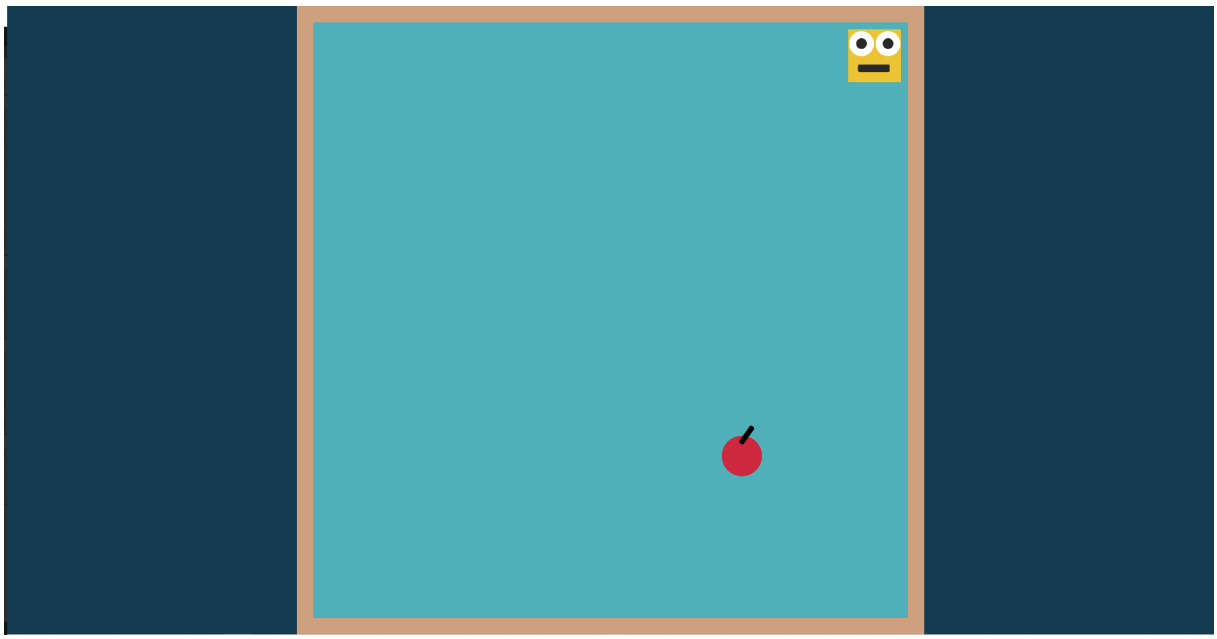
Rozdíl oproti klasické hře *Snake* bude ten, že na hracím poli bude kromě hráče ještě druhý had ovládaný počítačem. Při vzájemné kolizi hadů prohraje ten, který narazí hlavou. Cílem hry bude vydržet naživu déle než druhý had.

V první kapitole čtenářům stručně přiblížím, jak bylo vytvořeno herní prostředí. V následující kapitole vysvětlím, jakým způsobem funguje algoritmus pro hledání nejkratší cesty a popíši, jak byl protivník dále zdokonalen. Poté shrnu vytvoření hlavní nabídky a v poslední kapitole rozeberu, jak se hra ovládá a co všechno uživatel může dělat.

Celá hra bude vytvářena v programu *Unity* [1] a vývojovém prostředí Microsoft Visual Studio [2]. Tyto programy byly vybrány na základě předchozích zkušeností a jejich relativní jednoduchosti. Hra bude naprogramována jazykem *C#* a je dostupná ke stažení z veřejného repozitáře na *githubu* na adrese https://github.com/idostik/Snake2.git. V práci není zmíněn veškerý zdrojový kód, ale pouze jeho složitější části.

1 Vytvoření herního prostředí

Prvním krokem při tvorbě hry bylo vytvoření hracího pole, hráčem ovládaného hada a potom jídla, které bude had sbírat.



Obrázek 1: Hrací pole

Velký světle modrý čtverec představuje hrací pole. Jeho velikost si bude uživatel moci zvolit. Had ovládaný hráčem začíná v pravém horním rohu jako malý žlutý čtverec (tento čtverec je pouze jeden článek, sbíráním jídla budou přibývat další) a jídlo se na hracím poli objevuje jako v podobě jablka.

1.1 Pohyb hada

Had se pohybuje pravidelně v předem zvoleném časovém intervalu. Délka hada je kontrolována pomocí proměnné `snakeLength`, která se zvětší při každém sebrání jídla. Pohyb je uskutečněn vytvářením nových článků hada a odstraňováním starých. Vedle hlavy hada (prvního článku) je ve směru zvoleným hráčem vytvořen článek nový a jeho pozice je přidána do kolekce `snakePosList` (řádek 2). Pokud je počet pozic v `snakePosList` větší

než aktuální délka hada `snakeLength`, je poslední pozice v kolekci odstraněna a na základě toho zničen i samotný článek na této pozici [3].

Aby byla hra přehlednější, jsou mezery mezi jednotlivými články hada vyplňovány (aby byl had spojitý) podobným způsobem, jakým se vytvářejí nové články.

```
1  currentPos = directionVector;
2  Instantiate(snakePart, currentPos, Quaternion.identity);
3  snakePosList.Insert(0, currentPos);
4
5  if (snakeLength >= 2)
6  {
7    Vector2 gapPos = (snakePosList[0] + snakePosList[1]) / 2f;
8    Instantiate(gapFiller, gapPos, Quaternion.identity);
9    gapPosList.Add(gapPos);
10 }
```

Kód 1: Vytvoření nového článku hada a vyplnění mezer

1.2 Jídlo a kolize

Jídlo se objevuje na náhodně zvolené pozici, na které se zároveň nevyskytuje had. Na hracím poli může být v jednu chvíli pouze jedno jídlo a teprve po jeho sebrání se objeví nové.

Kolize hada sama se sebou jsou kontrolovány pomocí vestavěné funkce `OnTriggerEnter2D` a kolize hadů mezi sebou budou zjišťovány pomocí listu pozic jednotlivých článků `snakePosList`.

2 Had ovládaný počítačem

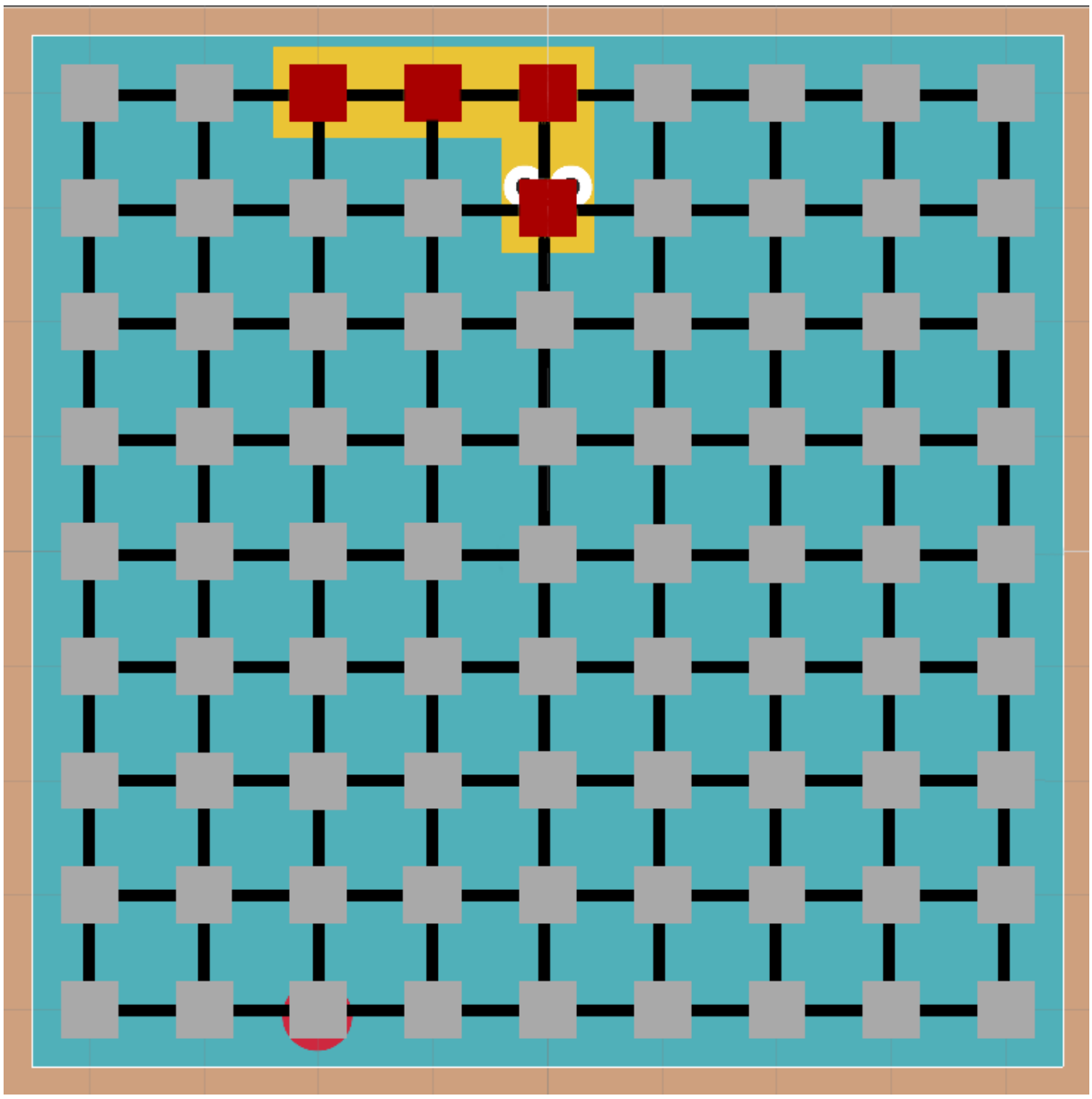
Protivník ovládaný počítačem bude začínat v opačném rohu hracího pole než hráč a jeho cílem bude sbírat jídlo a vydržet naživu co nejdéle. Had potřebuje umět rozpoznat, kam se může pohnout a kam ne, aby se vyhnul kolizi. Musí být schopen nalézt co nejkratší cestu k jídlu, a pokud přímá cesta k jídlu neexistuje, měl by umět přežít bez nárazu, dokud se cesta znovu neumožní. Ideálně by se had ovládaný počítačem měl také aktivně pokoušet zapříčinit, aby hráč narazil a prohrál.

2.1 Algoritmus *A**

Prvním krokem při vytváření protivníka bylo naprogramování algoritmu na hledání cesty v grafu. Takových algoritmů existuje hned několik, například *Breadth first search*, *Depth first search*, *Dijkstra*, *D** nebo *A**. Použit byl algoritmus *A**, protože je v hledání nejkratší cesty nejefektivnější. Na rozdíl od většiny ostatních algoritmů totiž hledá cestu nejdříve ve směru, kterým nejkratší cesta vede nejpravděpodobněji, a je proto časově nejrychlejší [4].

2.1.1 Vytvoření grafu

Jak bylo již výše zmíněno, algoritmus *A** slouží k hledání nejkratší cesty v grafu [5]. Ve hře *Snake* je tento graf tvořen všemi pozicemi, na kterých se mohou vyskytovat jednotlivé články hada. Protože se had může pohybovat pouze ve čtyřech směrech a vzdálenost, o kterou se každým pohybem posune, je pořád stejná, tak tento graf tvoří čtvercová síť uzlů, jejíž velikost záleží na velikosti hracího pole.



Obrázek 2: Znázornění grafu, ve kterém je hledána cesta

Na obrázku výše je graf, ve kterém je hledána cesta, znázorněn. Čtverce představují jednotlivé uzly grafu a černé spojnice ukazují, jak se had může mezi jednotlivými uzly pohybovat.

Každému uzlu je při tvorbě grafu přiřazeno několik proměnných. Konkrétně schůdnost, světová pozice a x-ová a y-ová souřadnice uzlu v grafu (kód 3, řádek 18). Schůdnost určuje, jestli se na pozici daného uzlu vyskytuje článek hada (nezáleží na tom, jestli článek patří hráči nebo protivníkovi). Na předchozím obrázku jsou neschůdné uzly zvýrazněny

červenou barvou. Světová pozice udává umístění uzlu vůči počátku světa, který je v tomto případě ve středu hracího pole. Poslední dva atributy označují umístění uzlu v grafu. Například uzel v levém dolním rohu bude mít x-ovou i y-ovou souřadnici 0 a uzel, na jehož pozici se na obrázku nachází hlava hada, bude mít x-ovou souřadnici 4 a y-ovou souřadnici 7.

```
1  public class Node
2  {
3    public bool walkable;
4    public Vector2 worldPosition;
5    public int gridX;
6    public int gridY;
7    public int gCost;
8    public int hCost;
9    public Node parent;
10
11   public Node(bool _walkable, Vector2 _worldPos, int _gridX, int _gridY)
12   {
13     walkable = _walkable;
14     worldPosition = _worldPos;
15     gridX = _gridX;
16     gridY = _gridY;
17   }
18
19   public int FCost
20   {
21     get
22     {
23       return gCost + hCost;
24     }
25   }
26 }
```

Kód 2: Uzel

Samotný kód pro vytvoření sítě uzlů je vidět v rámečku níže. Síť je tvořena dvourozměrným polem uzlů `grid` o velikosti hrací plochy `gridSize`. Pole se postupně prochází a každému uzlu jsou přiřazeny konkrétní hodnoty proměnných zmíněných výše [6].

```
1  public void CreateGrid()
2    {
3      grid = new Node[gridSize, gridSize];
4      for (int x = 0; x < gridSize; x++)
```

```
5       {
6         for (int y = 0; y < gridSize; y++)
7         {
8           Vector2 worlPoint = startPosAI + Vector2.right * x
9           + Vector2.up * y;
10          bool walkable = true;
11          if (
12          snakePosListAI.Contains(worlPoint)
13          || playerScript.snakePosList.Contains(worlPoint)
14          )
15          {
16            walkable = false;
17          }
18          grid[x, y] = new Node(walkable, worlPoint, x, y);
19        }
20      }
21    }
```

Kód 3: Vytvoření sítě uzlů

2.1.2 Implementace algoritmu

Po sestavení sítě uzlů přichází na řadu samotný *A** algoritmus, který hledá nejkratší cestu mezi dvěma body. Začne převedením zadaných pozic na odpovídající uzly (řádek 3). Poté se vytvoří seznam `openSet`, do kterého se přidá počáteční uzel, a prázdný seznam `closedSet` (řádek 6). Seznam `openSet` slouží k ukládání uzlů, které ještě nebyly vyhodnocené, a do seznamu `closedSet` se ukládají již vyhodnocené uzly [7].

```
1   public List<Node> FindPath(Vector2 startPos, Vector2 targetPos)
2     {
3       Node startNode = aIScript.NodeFromWorldPoint(startPos);
4       Node targetNode = aIScript.NodeFromWorldPoint(targetPos);
5
6       List<Node> openSet = new List<Node>();
7       HashSet<Node> closedSet = new HashSet<Node>();
8       openSet.Add(startNode);
```

Kód 4: A algoritmus: připravení potřebných proměnných*

Po vytvoření seznamů začne hlavní cyklus, který se opakuje, dokud existují nějaké nevyhodnocené uzly nebo dokud se nenarazí na cílový uzel. Algoritmus nejprve projde se-

znam `openSet` a vybere uzel, který má nejnižší hodnotu `FCost` (řádek 12). Když je tato hodnota shodná u více uzlů, rozhodne hodnota `gCost` (co tyto hodnoty znamenají je vysvětleno dále). Vybraný uzel je uložen do proměnné `currentNode` a je přemístěn ze seznamu `openSet` do seznamu `closedSet` (řádek 23).

Pokud je v seznamu `openSet` pouze jeden uzel, například při prvním cyklu algoritmu, tak se výše zmíněný výběr přeskočí a tento uzel je automaticky označen jako `currentNode` a následně přemístěn do seznamu `closedSet`.

```
9     while (openSet.Count > 0)
10    {
11      Node currentNode = openSet[0];
12      for (int i = 1; i < openSet.Count; i++)
13      {
14        if (
15        openSet[i].FCost < currentNode.FCost
16        || openSet[i].FCost == currentNode.FCost
17        && openSet[i].hCost < currentNode.hCost
18        )
19        {
20          currentNode = openSet[i];
21        }
22      }
23      openSet.Remove(currentNode);
24      closedSet.Add(currentNode);
```

Kód 5: A algoritmus: výběr uzlu s nejnižší hodnotou `FCost`*

Dále se kontroluje, jestli je právě vybraný uzel `currentNode` cílový. Pokud ano, algoritmus vrátí funkci `RetracePath`, která sestaví finální cestu (kód 8).

```
25      if (currentNode == targetNode)
26      {
27        return RetracePath(startNode, targetNode);
28      }
```

Kód 6: A algoritmus: dosažení cílového uzlu*

Následuje další cyklus, který prochází všechny uzly sousedící s uzlem `currentNode`. Pokud je sousední uzel neschůdný nebo pokud byl již vyhodnocen (už je v seznamu `closedSet`),

tak se přeskočí (řádek 31). Pro všechny ostatní uzly se spočítá nová hodnota `gCost`. Tato hodnota udává vzdálenost od počátečního uzlu po již objevené cestě (nebo-li přes kolik uzlů se k danému uzlu došlo). Pokud je nová hodnota `gCost` menší než aktuální `gCost`, nebo pokud daný uzel ještě není v seznamu `openSet` a tudíž žádnou hodnotu přiřazenou nemá, tak je mu přiřazena hodnota `gCost` spočítaná výše a další dvě proměnné: `hCost` a `parent` (řádek 43). Hodnota `hCost` je vzdálenost daného uzlu od cílového uzlu. Tato vzdálenost se spočítá jako délka cesty po grafu (nepočítá se úhlopříčně, ale pravoúhle). Proměnná `parent` určuje, z kterého sousedního uzlu se na daný uzel přišlo, a proto se nastaví na uzel `currentNode`. Nakonec se přidají do seznamu `openSet` uzly, které tam ještě nejsou a kterým byly nově přiřazeny proměnné (řádek 49).

Hodnota `FCost` zmíněná na začátku algoritmu, podle které se vybírá uzel `currentNode` (řádek 15), se vypočítá jako součet hodnot `gCost` a `hCost`. Tento výpočet se děje v kódu samotného uzlu (kód 2, řádek 19).

```
29        foreach (Node neighbour in aIScript.GetNeighbours(currentNode))
30        {
31          if (!neighbour.walkable || closedSet.Contains(neighbour))
32          {
33            continue;
34          }
35
36          int newMovementCostToNeighbour = currentNode.gCost + 1;
37
38          if (
39          newMovementCostToNeighbour < neighbour.gCost
40          || !openSet.Contains(neighbour)
41          )
42          {
43            neighbour.gCost = newMovementCostToNeighbour;
44            neighbour.hCost = GetDistance(neighbour, targetNode);
45            neighbour.parent = currentNode;
46
47            if (!openSet.Contains(neighbour))
48            {
49              openSet.Add(neighbour);
50            }
51          }
52        }
53      }
54      return null;
55    }
```

Kód 7: A algoritmus: přiřazení hodnot novým uzlům*

Pokud algoritmus nenarazí na cílový uzel, tak se celý cyklus opakuje. Znovu se projde seznam `openSet` a jako `currentNode` se vybere uzel s nejnižší hodnotou `FCost`. Ten se potom přemístí do seznamu `closedSet` a následně se projdou všechny jeho sousední uzly, kterým se přiřadí proměnné podle podmínek popsaných výše. Dále se přidají nové uzly do seznamu `openSet` a cyklus se znovu opakuje, dokud neprojde všechny dostupné uzly (cesta potom neexistuje) nebo dokud nedojde k cílovému uzlu.

Jestliže algoritmus narazí na cílový uzel, tak se zavolá funkce `RetracePath`, která pomocí předchůdců (`parent`) přiřazených jednotlivým uzlům vysleduje cestu zpátky na začátek a vytvoří z nich nový seznam `path`, který po otočení představuje hotovou nejkratší cestu.

```
1  List<Node> RetracePath(Node startNode, Node endNode)
2    {
3      List<Node> path = new List<Node>();
4      Node currentNode = endNode;
5
6      while (currentNode != startNode)
7      {
8        path.Add(currentNode);
9        currentNode = currentNode.parent;
10     }
11     path.Reverse();
12     return path;
13   }
```

Kód 8: Sestavení finální cesty

2.1.3 Úspěšnost hada po implementaci algoritmu

Algoritmus je při hře volán před každým pohybem hada ovládaného počítačem, a ten se tak dokáže vyhnout kolizi s hráčem ve většině případů. Problém nastává v pozdější fázi hry, kdy je had už delší a vznikají situace, při kterých cesta k jídlu neexistuje. V takových případech had pokračuje rovně, a pokud se cesta k jídlu znovu neobjeví, tak narazí.

Po krátkém testování bylo přidáno pravidlo, že rozdíl velikostí hadů nesmí být větší než 10, aby se zabránilo tomu, že hráč nesbírá jídlo a jenom čeká, až se had ovládaný počítačem zvětší a sám naboura. Pokud je jeden z hadů o 10 článků kratší než druhý, prohraje.

I po přidání tohoto pravidla ale zkušenější hráči nemají problém vyhrát. Hráč nemusí nijak útočit, stačí aby hrál pasivně, sbíral jídlo a vyhýbal se. Had ovládaný počítačem časem narazí sám.

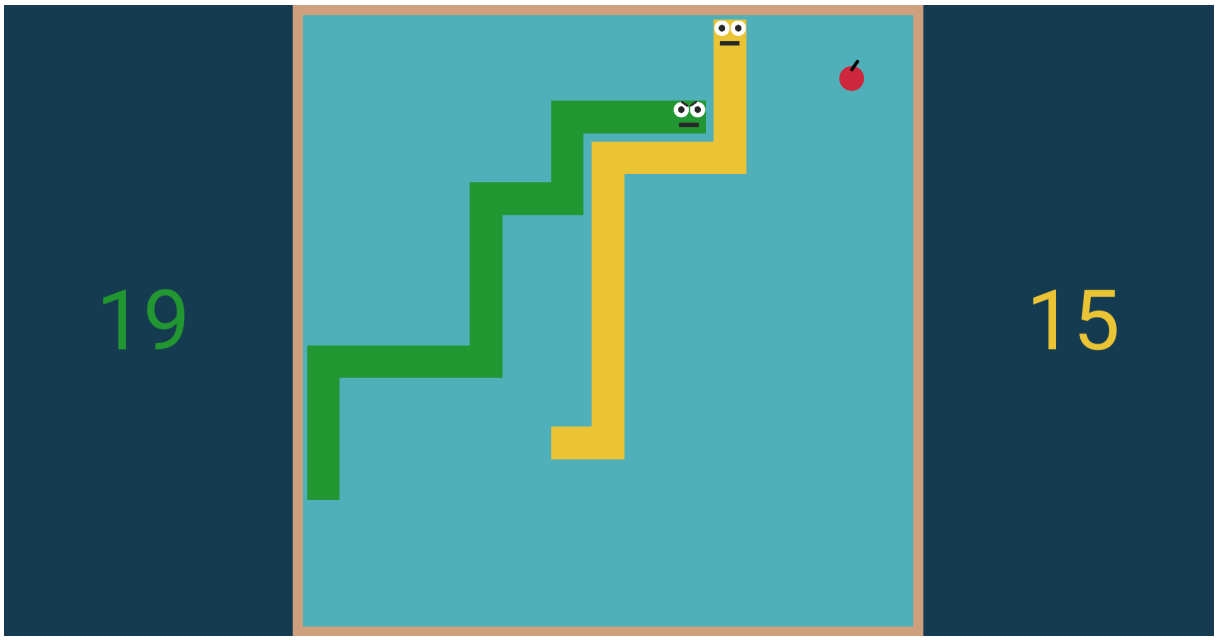
Algoritmus totiž počítá pouze s tím, jak se dostat co nejkratší cestou k jídlu, ale už nedbá na to, jestli bude mít had po sebrání jídla možnost pokračovat. a pokud cesta k jídlu neexistuje vůbec, algoritmus $A*$ je nepoužitelný.

2.2 Další vylepšení hada

Had ovládaný počítačem sice už dokáže vyhledat nejkratší cestu k jídlu, ale pořád je docela jednoduše porazitelný. Proto bylo naprogramováno několik dalších vylepšení, které z něj udělají mnohem těžšího protivníka.

2.2.1 Hledání nejdelší cesty a únikového uzlu

Prvním problémem je, že had ovládaný počítačem nemá žádné instrukce, které by mu říkaly, co má dělat, když cesta k jídlu zrovna neexistuje. Na následujícím obrázku je příklad takové situace. Had ovládaný počítačem (zelený had) měl naplánovanou cestu k jídlu podél horního okraje hracího pole, ale hráč (žlutý had) mu tuto cestu zablokoval. Samotný algoritmus $A*$ si s touto situací poradit nedokáže a had ovládaný počítačem by proto příštím pohybem narazil a prohrál.



Obrázek 3: Situace, kdy cesta k jídlu neexistuje

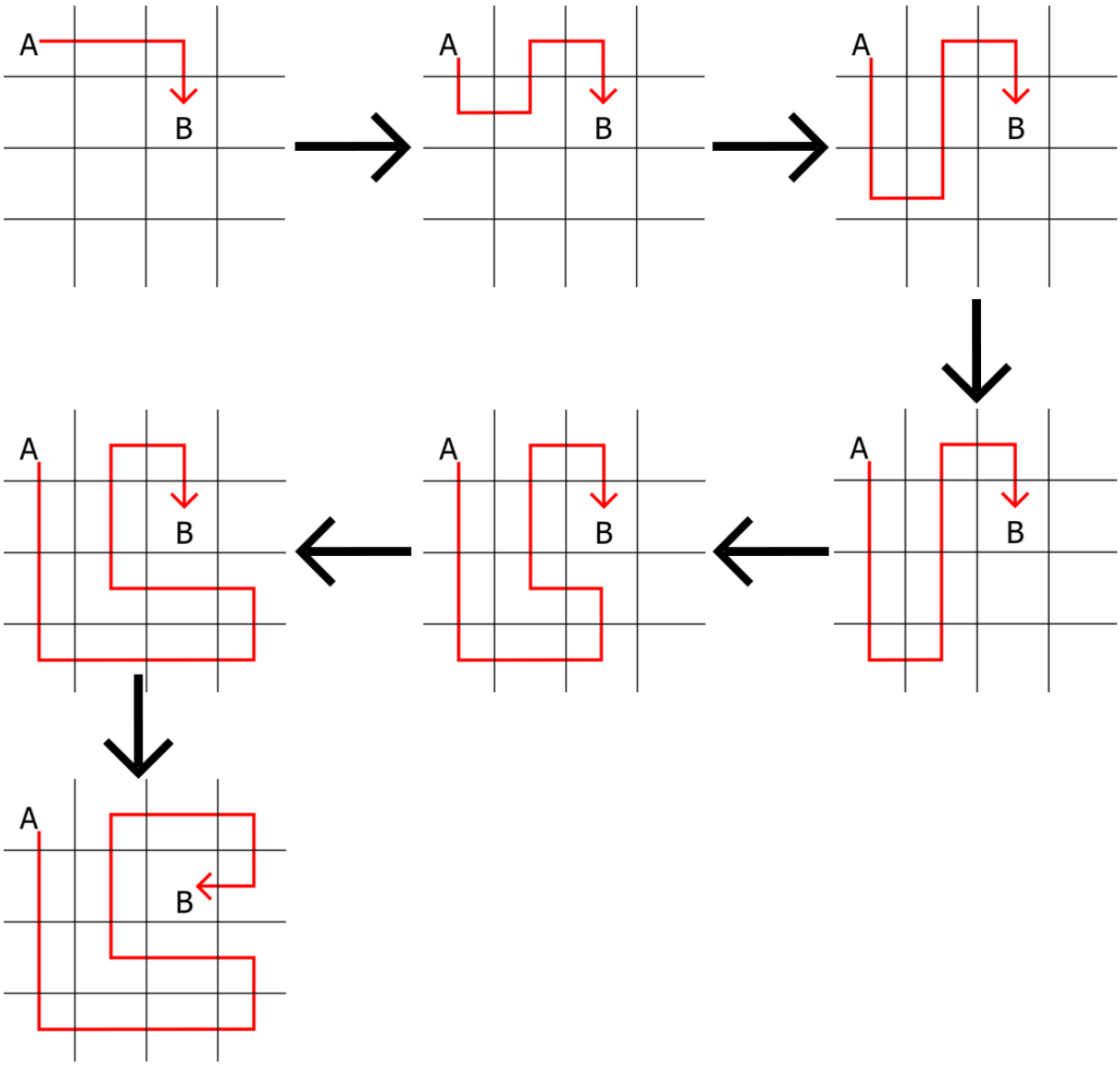
Řešením této situace je otočit se a jet co nejdelší cestou k místu, kde se cesta ven z uzavřeného prostoru otevře jako první, v tomto případě ke středu levé hranice hracího pole.

Uzel, u kterého se jako první objeví cesta ven, je hledán pomocí funkce `GetExitNode`. Ta projde všechny schůdné uzly, na které se had ovládaný počítačem v tuto chvíli může dostat, a z nich vybere uzel sousedící s článkem hada, který zmizí nejdříve (článek, který má v seznamu `snakePosListAI` nejvyšší index).

```
1  public Node GetExitNode()
2  {
3    List<Node> accessibleNodes = GetAccessibleNodes(
4    NodeFromWorldPoint(currentPosAI)
5    );
6    int maxIndex = -2;
7    Node exitNode = NodeFromWorldPoint(currentPosAI);
8    foreach(Node n in accessibleNodes)
9    {
10     List<Node> neighbours = GetNeighbours(n);
11     for (int i = 0; i < neighbours.Count; i++)
12     {
13       int neighbourIndex = snakePosListAI.IndexOf(
14       neighbours[i].worldPosition
15       );
16       if (neighbourIndex > maxIndex)
17       {
18         maxIndex = neighbourIndex;
19         exitNode = n;
20       }
21     }
22   }
23   return exitNode;
24 }
```

Kód 9: Hledání únikového uzlu

K sestavení nejdelší cesty je použita nejkratší cesta k únikovému uzlu. Uzly této cesty se postupně po dvojicích procházejí, a pokud to je možné, tak se mezi dvojici uzlů vloží další dva sousední uzly a cesta se tak prodlouží. Celá cesta se takto projde dvakrát, aby se žádné prodloužení nevynechalo a výsledná trasa byla co nejdelší.



Obrázek 4: Tvorba nejdelší cesty

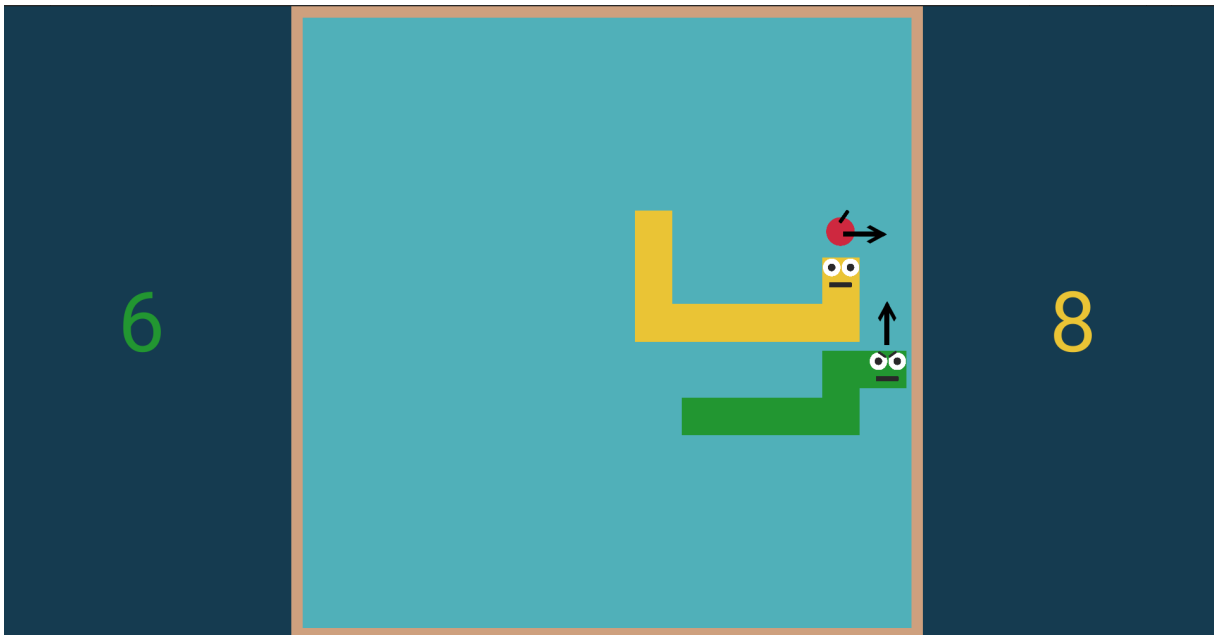
2.2.2 Bezpečné sebrání jídla

Po vylepšení zmíněném v předchozí kapitole se had ovládaný počítačem umí dostat ze situací, kdy neexistuje přímá cesta k jídlu. To ale zvládne jenom v případě, že má k dispozici dostatek místa na kličkování, než se cesta k jídlu znovu objeví. Proto byla přidána funkce `IsEatingSafe`, která před každým pohybem kontroluje, jestli bude mít had po sebrání jídla dostatek místa na bezpečné pokračování.

Pokud existuje cesta k jídlu, funkce změní schůdnost uzlů tak, jak by to vypadalo kdyby had ovládaný počítačem jídlo sebral. Situace je následně vyhodnocena jako bezpečná, pokud je z budoucí pozice hada dostupný dostatek schůdných uzlů, nebo pokud je vedle některého z dostupných uzlů ocas hada. Když had skončí po sebrání jídla vedle ocasu, může se pohybovat pořád za ním (ocas bude stejným tempem mizet), dokud se neobjeví lepší cesta. Po vyhodnocení situace se schůdnost uzlů změní zpět na původní stav. Pokud se cesta jeví jako riziková, had bude zdržovat pomocí nejdelší cesty a únikového uzlu z předešlé kapitoly, dokud se cesta k jídlu nevyhodnotí jako bezpečná.

2.2.3 Obrana proti zatlačení ke stěně

I přes všechna předchozí vylepšení je možné hada ovládaného počítačem docela jednoduše porazit zatlačením ke stěně. Pokud nejkratší cesta k jídlu vede mezi hráčem a stěnou, had ovládaný počítačem se po ní bude pohybovat a hráčovi potom stačí jenom jeden pohyb do strany aby vyhrál. Příklad této situace je vidět na následujícím obrázku.



Obrázek 5: Prohra zatlačením ke stěně

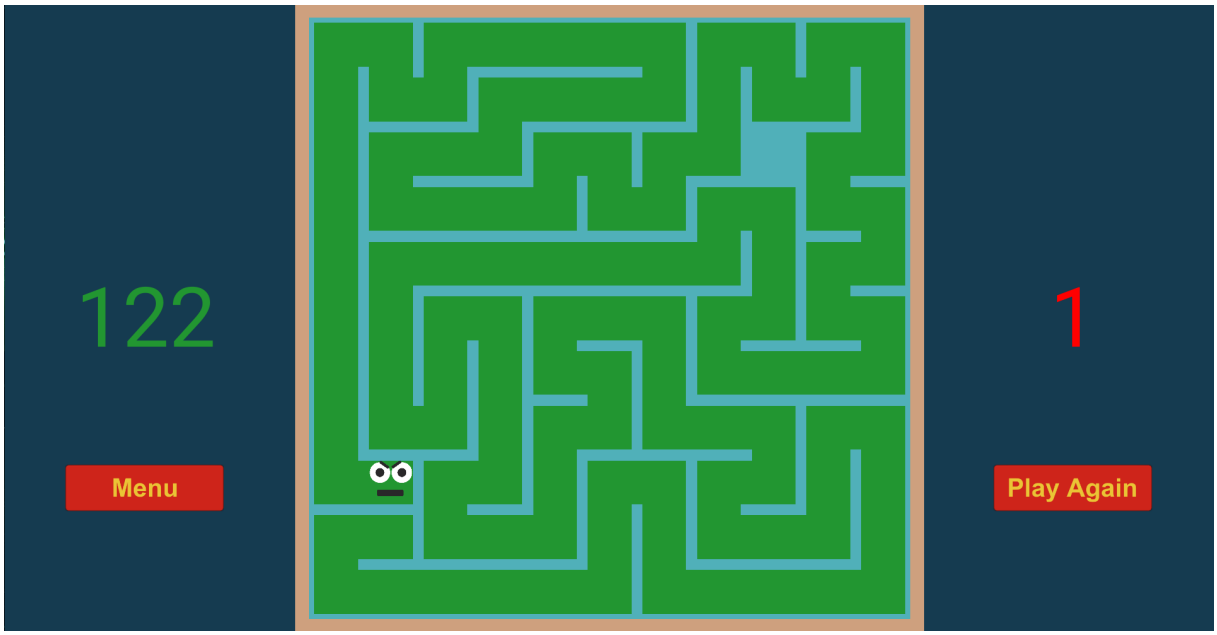
Tento problém byl vyřešen jednoduchou funkcí, která kontroluje, jestli se první uzel cesty vyskytuje mezi stěnou a hráčem. Pokud ano, had ovládaný počítačem po této cestě pokračovat nebude.

2.2.4 Útok

Posledním přidaným vylepšením bylo útočení. Pokud je had ovládaný počítačem v blízkosti hráče, tak se změní cíl jeho cesty na uzel před hlavou hráčova hada (podmínkou je, že tato cesta existuje). Útočení tímto způsobem sice občas zapříčiní, že had ovládaný počítačem nesebere jídlo, ale útoky jsou zato velmi nepředvídatelné a hráč se jim musí obtížně vyhýbat.

2.2.5 Úspěšnost hada po přidání vylepšení

Po naprogramování všech výše zmíněných vylepšení je hra proti protivníkovi ovládaným počítačem výrazně obtížnější, než se samotným algoritmem na hledání nejkratší cesty. Protivník se zvládá vyhýbat většině pokusů hráče o zablokování jeho cesty a dokáže si poradit i v případech, kdy cesta k jídlu neexistuje.



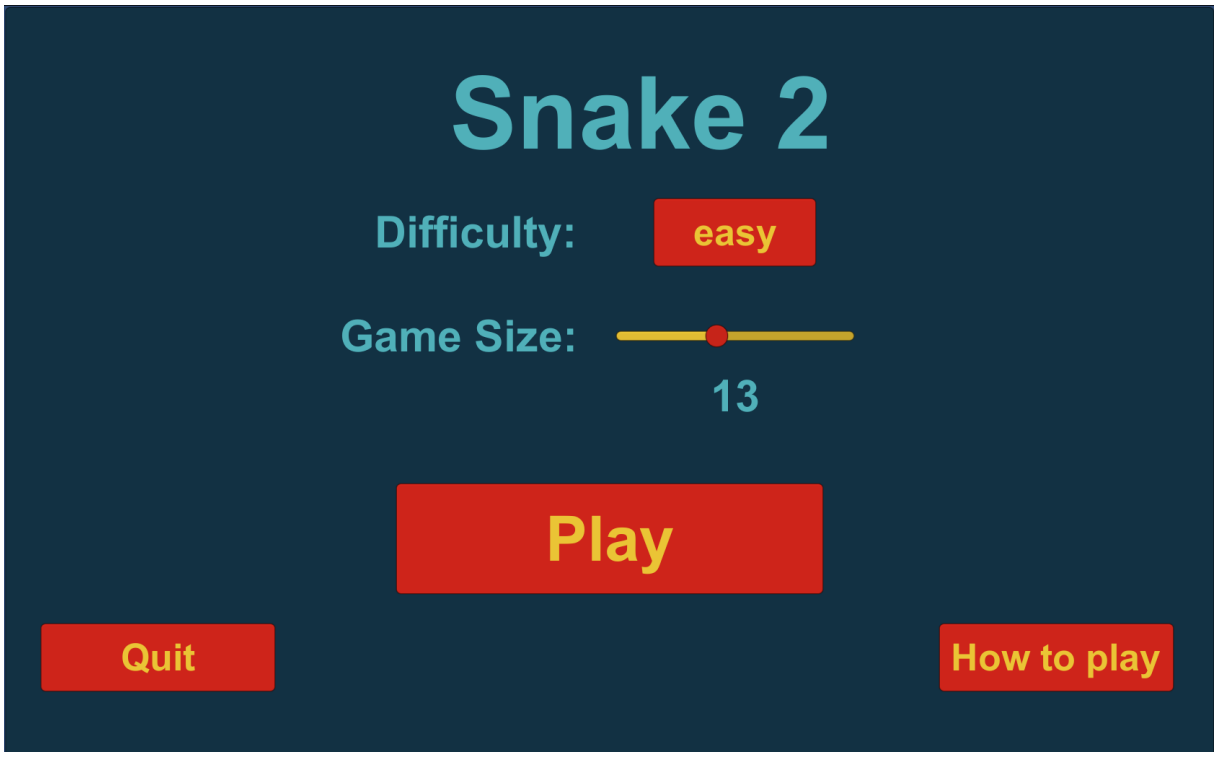
Obrázek 6: Had ovládaný počítačem zaplnil téměř celé hrací pole

Pokud se had ovládaný počítačem nechá hrát sám, dokáže dokonce zaplnit skoro celé hrací pole. To se ve hře *Snake* nepodaří téměř žádnému hráči.

Protivník ale není bez nedostatků. Může být například poražen, pokud ho cesta k jídlu vede podél stěny. Hráči pak k němu stačí ze strany přijet a zablokovat mu cestu pryč. Protivník je potom nucen pokračovat rovně, dokud v rohu hracího pole nenarazí. Způsob, jakým jsou některé části kódu naprogramované, zároveň není úplně nejefektivnější a rychlost hry je proto do určité míry omezena výkonem počítače, na kterém je hra spuštěna.

3 Hlavní nabídka

Po dokončení tvorby samotné hry přišla na řadu hlavní nabídka. V ní má hráč možnost si zvolit velikost hracího pole a obtížnost protivníka. Dále se zde může dozvědět základní pravidla a jak se hra ovládá.



Obrázek 7: Hlavní nabídka

Hráč si může vybrat jednu ze tří obtížností. Pokud je hra nastavena na lehkou obtížnost, had ovládaný počítačem používá pouze algoritmus $A*$. Na střední obtížnost je hra o něco rychlejší a protivník kromě nejkratší cesty umí najít i cestu nejdelší a je schopen předvídat, kdy je sebrání jídla nebezpečné. V případě, že hráč nastaví obtížnost na těžkou, jsou použita všechny vylepšení zmíněná v předchozích kapitolách a pohyb hadů je ještě zrychlen.

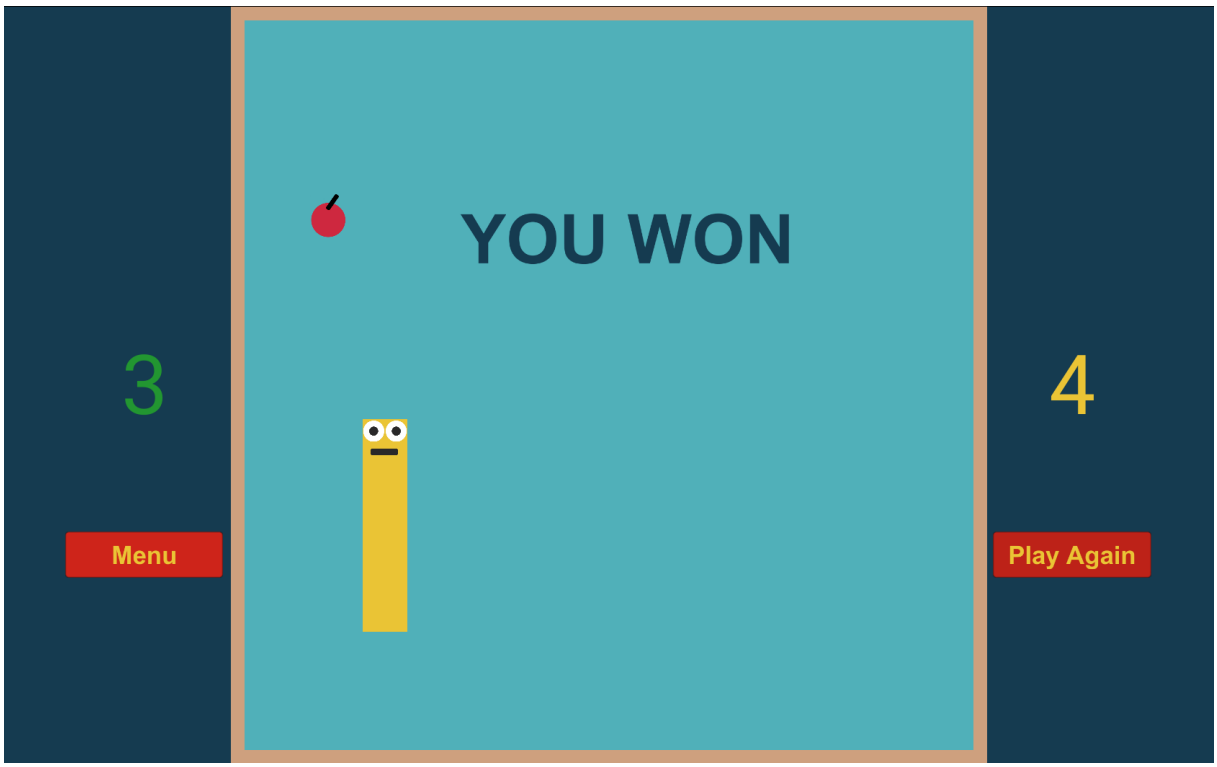
4 Uživatelská dokumentace

Vytvořená hra je variantou klasické hry *Snake*. Hráč se pohybuje po hracím poli jako had a hraje proti druhému hadovi ovládanému počítačem. Cílem hry je vydržet naživu déle než protivník.

Po spuštění hry se uživatel ocitne v hlavní nabídce (obrázek 7). Zde má možnost nastavit specifikace hry. Obtížnostmi může procházet kliknutím na červené tlačítko vedle nápisu *Difficulty* a velikost hracího pole je možno změnit přetažením posuvníku vedle nápisu *Game Size*. Dále si hráč může přečíst stručné instrukce ke hře po kliknutí na tlačítko *How to play* v pravém dolním rohu. Odejít ze hry lze tlačítkem *Quit*. Pokud je uživatel spokojen s nastavením, pomocí velkého tlačítka *Play* vstoupí do hry.

Hráč hraje za žlutého hada, který začíná jako jeden žlutý čtverec v pravém horním rohu hracího pole (obrázek 1). Protivník ovládaný počítačem hraje za modrého hada a začne v rohu opačném. Světle modrá hrací plocha je ohraničená béžovou zdí. Hra začne po dvou vteřinách od stlačení tlačítka *Play*. Oba hadi se pohybují v pravidelném časovém intervalu, jehož rychlost závisí na nastavené obtížnosti. Hráč ovládá směr pohybu žlutého hada pomocí šipek na klávesnici. Hadi mohou sbírat jídlo, které se na hracím poli objevuje jako červená jablka. Po každém sebrání jídla se had prodlouží o jeden článek. Pokud had narazí buď do druhého hada, sám do sebe nebo do zdi ohraničující hrací pole, prohraje. Prohrát lze také, pokud je hráč o deset článků kratší než protivník. Délka hadů je znázorněna číslem na stranách vedle hracího pole. Hráčovým úkolem je zapříčinit, aby had ovládaný počítačem narazil a prohrál.

Pokud jeden z hadů skončí, objeví se nápis, který říká, jestli hráč vyhrál nebo prohrál, a dvě tlačítka (obrázek 8). Jedním tlačítkem může hráč spustit hru znovu se stejným nastavením a druhým se může vrátit do hlavní nabídky. Pokud hráč vyhraje, může pokračovat ve hře sám a zkusit dosáhnout co největší délky. Při výhře hada ovládaného počítačem bude protivník pokračovat, dokud nenarazí.



Obrázek 8: Výhra hráče

Závěr

Cílem této práce bylo naprogramovat jednoduchou hru, kterou bude hrát uživatel proti počítači. Povedlo se mi naprogramovat variantu hry *Snake* s chytrým protivníkem ovládaným počítačem, s kterým může hráč soupeřit. V práci jsem vysvětlil, jak jsem při programování postupoval a jak jednotlivé použité algoritmy fungují. Cíl práce považuji za splněný a s prací jsem celkově spokojen.

Do budoucna bych rád celý kód zefektivnil, aby hledání cesty mohlo probíhat co nejrychleji. Dále bych přidal více map, mezi kterými by si hráč mohl vybrat. Případně by hru bylo možné udělat zajímavější přidáním více typů jídel, které by hadům dávaly různé bonusy.

Seznam obrázků

Seznam kódů

Seznam použité literatury

1. UNITY TECHNOLOGIES. *Unity* [online]. Verze 2019.4.10f1 Personal [cit. 2021-03-18]. Dostupné z: https://unity.com//.
2. MICROSOFT CORPORATION. *Microsoft Visual Studio Community 2019* [online]. Verze 16.7.7 [cit. 2021-03-28]. Dostupné z: https://visualstudio.microsoft.com/cs/vs/community/.
3. CODEMONKEY. Making Snake in Unity: Snake Grow (Unity Tutorial for Beginners). In: *Youtube* [online]. 2019 [cit. 2021-03-21]. Dostupné z: https://www.youtube.com/watch?v=KifUCu1LLgs.
4. LAGUE, Sebastian. A* Pathfinding (E01: algorithm explanation). In: *Youtube* [online]. 2014 [cit. 2021-03-21]. Dostupné z: https://www.youtube.com/watch?v=-L-WgKMFuhE.
5. WIKIPEDIA CONTRIBUTORS. A* search algorithm. In: *Wikipedia: The Free Encyclopedia* [online]. [B.r.]. Datum poslední revize 2021-03-16 [cit. 2021-03-18]. Dostupné z: https://en.wikipedia.org/w/index.php?title=A*_search_algorithm&oldid=1012527716.
6. LAGUE, Sebastian. A* Pathfinding (E02: node grid). In: *Youtube* [online]. 2014 [cit. 2021-03-18]. Dostupné z: https://www.youtube.com/watch?v=nhiFx28e7JY.
7. LAGUE, Sebastian. A* Pathfinding (E03: algorithm implementation). In: *Youtube* [online]. 2014 [cit. 2021-03-18]. Dostupné z: https://www.youtube.com/watch?v=mZfyt03LDH4.