cs380su21-meta.sg

≡ Description   ▦ Submission view

Grade

Reviewed on Friday, 28 May 2021, 4:38 PM by Yi Lin YEE
grade: 85.00 / 100.00

Assessment report ⬚ **[-]**

[-]**Summary of tests**

```
+----------------------------+
| 10 tests run/10 tests passed |
+----------------------------+
```

Memory leak -15%

Submitted on Wednesday, 26 May 2021, 7:00 PM (Download)

functions.cpp

```cpp
/*!**************************************************************************
\file functions.cpp
\author Vadim Surov, Goh Wei Zhe
\par DP email: vsurov\@digipen.edu, weizhe.goh\@digipen.edu
\par Course: CS380
\par Section: A
\par Programming Assignment 2
\date 05-26-2021
\brief
This file has declarations and definitions that are required for submission
***************************************************************************/
#include "functions.h"

namespace AI
{


}
```

functions.h

```cpp
1   /*!************************************************************************
2   \file functions.h
3   \author Vadim Surov, Goh Wei Zhe
4   \par DP email: vsurov\@digipen.edu, weizhe.goh\@digipen.edu
5   \par Course: CS380
6   \par Section: A
7   \par Programming Assignment 2
8   \date 05-26-2021
9   \brief
10  This file has declarations and definitions that are required for submission
11  ************************************************************************/
12
13  #ifndef FUNCTIONS_H
14  #define FUNCTIONS_H
15
16  #include <stack>
17  #include <algorithm>
18  #include <queue>
19
20  #include "data.h"
21
22  #define UNUSED(expr) (void)expr;
23
24  namespace AI
25  {
26      // Domain specific functor that returns adjacent nodes
27      class GetMapAdjacents : public GetAdjacents
28      {
29          int* map; // the map with integers where 0 means an empty cell
30          int size; // width and hight of the map in elements
31
32      public:
33
34          GetMapAdjacents(int* map=nullptr, int size=0)
35              : GetAdjacents(), map{ map }, size{ size }{}
36
37          virtual ~GetMapAdjacents(){}
38
39          /*!************************************************************************
40          \brief
41          Set grid positions and grid color
42
43          \param key
44          Position of cell in grid
45
46          \param color
47          Color of cell
48
49          \return
50          None.
51          ************************************************************************/
52          void setValue(Key key, int color)
53          {
54              int j = key.j;
55              int i = key.i;
56
57              this->map[j * this->size + i] = color;
58          }
59
60          /*!************************************************************************
61          \brief
62          An Operator Overloading function that finds all empty adjacent cells and
63          insert into an array vector of nodes and return it.
64
65          \param key
66          Position of cell in grid
67
68          \return
69          Returns an array vector of nodes
70          ************************************************************************/
71          std::vector<AI::Node*> operator()(Key key)
72          {
73              int j = key.j;
74              int i = key.i;
75
76              std::vector<AI::Node*> list = {};
77
78              // Find and return all empty adjacent cells
79              if (j >= 0 && j < this->size && i >= 0 && i < this->size)
80              {
81                  if (i > 0 && this->map[j * this->size + i - 1] == 0)
82                  {
83                      Node* newNode = new Node;
84                      newNode->key.i = i - 1;
85                      newNode->key.j = j;
86
87                      list.push_back(newNode);
88                  }
89
90                  if (i < this->size-1 && this->map[j * this->size + i + 1] == 0)
91                  {
92                      Node* newNode = new Node;
93                      newNode->key.i = i + 1;
94                      newNode->key.j = j;
95
96                      list.push_back(newNode);
97                  }
98
99                  if (j > 0 && this->map[(j - 1) * this->size + i] == 0)
100                 {
101                     Node* newNode = new Node;
102                     newNode->key.j = j - 1;
103                     newNode->key.i = i;
104
105                     list.push_back(newNode);
106                 }
107
108                 if (j < this->size-1 && this->map[(j + 1)* this->size + i] == 0)
```

```cpp
109             {
110                 Node* newNode = new Node;
111                 newNode->key.j = j + 1;
112                 newNode->key.i = i;
113
114                 list.push_back(newNode);
115             }
116         }
117
118         return list;
119     }
120 };
121
122 // Domain specific functor that returns shuffled adjacent nodes
123 class GetMapStochasticAdjacents : public GetMapAdjacents
124 {
125 public:
126
127     GetMapStochasticAdjacents(int* map, int size)
128         : GetMapAdjacents{ map, size }{}
129
130     /*!************************************************************************
131     \brief
132     An Operator Overloading function that finds all empty adjacent cells and
133     insert into an array vector of Nodes, then shuffles the result and
134     return it.
135
136     \param key
137     Position of cell in grid
138
139     \return
140     Returns an array vector of nodes
141     *************************************************************************/
142     std::vector<AI::Node*> operator()(Key key)
143     {
144         std::vector<AI::Node*> list = {};
145
146         // Find and return all empty adjacent cells
147         // Use the base class operator() and then shuffle the result
148
149         list = GetMapAdjacents::operator()(key);
150         std::random_shuffle(list.begin(), list.end());
151
152         return list;
153     }
154 };
155
156 // Wrappers that provide same interface for queue and stack
157
158 struct Interface
159 {
160     virtual void clear() = 0;
161     virtual void push(Node* pNode) = 0;
162     virtual Node* pop() = 0;
163 };
164
165 struct Queue : Interface
166 {
167     std::vector<Node*> Q;
168     int count = 0;
169
170     /*!************************************************************************
171     \brief
172     Clears all nodes in the vector array
173
174     \param
175     None.
176
177     \return
178     None.
179     *************************************************************************/
180     void clear()
181     {
182         Q.clear();
183         count = 0;
184     }
185
186     /*!************************************************************************
187     \brief
188     Add nodes by pushing back to the vector array
189
190     \param
191     None.
192
193     \return
194     None.
195     *************************************************************************/
196     void push(Node* pNode)
197     {
198         Q.push_back(pNode);
199         ++count;
200     }
201
202     /*!************************************************************************
203     \brief
204     Remove nodes by popping it from the back of the vector array
205
206     \param
207     None.
208
209     \return
210     None.
211     *************************************************************************/
212     Node* pop()
213     {
214         Node* pNode = nullptr;
215
216         pNode = Q.back();
```

```
217              Q.pop_back();
218              --count;
219
220              return pNode;
221          }
222
223          /*!******************************************************************
224          \brief
225          Check if the vector array is empty
226
227          \param
228          None.
229
230          \return
231          Returns true if vector is empty, else returns false
232          ******************************************************************/
233          bool empty()
234          {
235              return (count == 0) ? true : false;
236          }
237      };
238
239      struct Stack : Interface
240      {
241          std::vector<Node*> Stack;
242          int count = 0;
243
244          /*!******************************************************************
245          \brief
246          Clears all nodes in the vector array
247
248          \param
249          None.
250
251          \return
252          None.
253          ******************************************************************/
254          void clear()
255          {
256              Stack.clear();
257          }
258
259          /*!******************************************************************
260          \brief
261          Add nodes by pushing back to the vector array
262
263          \param
264          None.
265
266          \return
267          None.
268          ******************************************************************/
269          void push(Node* pNode)
270          {
271              Stack.push_back(pNode);
272              ++count;
273  ;          }
274
275          /*!******************************************************************
276          \brief
277          Remove nodes by popping it from the back of the vector array
278
279          \param
280          None.
281
282          \return
283          None.
284          ******************************************************************/
285          Node* pop()
286          {
287              Node* pNode = nullptr;
288
289              pNode = Stack.back();
290              Stack.pop_back();
291              --count;
292
293              return pNode;
294          }
295
296          /*!******************************************************************
297          \brief
298          Check if the vector array is empty
299
300          \param
301          None.
302
303          \return
304          Returns true if vector is empty, else returns false
305          ******************************************************************/
306          bool empty()
307          {
308              return (count == 0) ? true : false;
309          }
310      };
311
312      // Recursive Flood Fill
313      class Flood_Fill_Recursive
314      {
315          GetAdjacents* pGetAdjacents;
316
317      public:
318          Flood_Fill_Recursive(GetAdjacents* pGetAdjacents)
319              : pGetAdjacents{ pGetAdjacents }{}
320
321          /*!******************************************************************
322          \brief
323          Implement Recursive Flood Fill Algorithm
324
```

```
325          \param key
326          Position of cell in grid
327
328          \param color
329          Color of cell
330
331          \return
332          None
333          **********************************************************************/
334          void run(Key key, int color)
335          {
336              // Implement the flood fill
337              std::vector<AI::Node*> adjacentList =
338                  this->pGetAdjacents->operator()(key);
339
340              for (auto adj : adjacentList)
341              {
342                  GetMapAdjacents* mapAdj =
343                      dynamic_cast<GetMapAdjacents*>(this->pGetAdjacents);
344
345                  mapAdj->setValue(adj->key, color);
346                  this->run(adj->key, color);
347              }
348          }
349      };
350
351      // Iterative Flood Fill
352      // Type T defines is it depth- or breadth-first
353      template<typename T>
354      class Flood_Fill_Iterative
355      {
356          GetAdjacents* pGetAdjacents;
357          T openlist;
358
359      public:
360          Flood_Fill_Iterative(GetAdjacents* pGetAdjacents)
361              : pGetAdjacents{ pGetAdjacents }, openlist{}{}
362
363          /*!**********************************************************************
364          \brief
365          Implement Iterative Flood Fill Algorithm, depth or breadth-first
366
367          \param key
368          Position of cell in grid
369
370          \param color
371          Color of cell
372
373          \return
374          None
375          **********************************************************************/
376          void run(Key key, int color)
377          {
378              // Implement the flood fill
379              openlist.clear();
380              openlist.push(new Node(key));
381
382              while (!openlist.empty())
383              {
384                  Node* current = openlist.pop();
385                  std::vector<Node*> adjacentList =
386                      this->pGetAdjacents->operator()(current->key);
387
388                  for (auto adj : adjacentList)
389                  {
390                      GetMapAdjacents* mapAdj =
391                          dynamic_cast<GetMapAdjacents*>(this->pGetAdjacents);
392
393                      mapAdj->setValue(adj->key, color);
394                      this->openlist.push(adj);
395                  }
396              }
397          }
398      };
399
400  } // end namespace
401
402  #endif
```

[VPL](#)

◄ Showcase: The Lines Game   Jump to...   Attendance cs380su21-meta.sg Friday
                                                        28/5/2021 9:00am-12:20pm ►