cs380su21-meta.sg

Dashboard / My courses / cs380su21-meta.sg / 17 May - 23 May / Assignment 2 (Flood-Fill)

Description Submission </> Edit Submission view

Grade

Reviewed on Wednesday, 26 May 2021, 7:00 PM by Automatic grade
grade: 100.00 / 100.00

Assessment report [-]

[+]**Summary of tests**

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

functions.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
/*!*****************************************************************************
\file functions.h
\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
*******************************************************************************/

#ifndef FUNCTIONS_H
#define FUNCTIONS_H

#include <stack>
#include <algorithm>
#include <queue>

#include "data.h"

#define UNUSED(expr) (void)expr;

namespace AI
{
    // Domain specific functor that returns adjacent nodes
    class GetMapAdjacents : public GetAdjacents
    {
        int* map; // the map with integers where 0 means an empty cell
        int size; // width and hight of the map in elements

    public:

        GetMapAdjacents(int* map=nullptr, int size=0)
            : GetAdjacents(), map{ map }, size{ size }{}

        virtual ~GetMapAdjacents(){}

        /*!*********************************************************************
        \brief
        Set grid positions and grid color

        \param key
        Position of cell in grid

        \param color
        Color of cell

        \return
        None.
        ***********************************************************************/
        void setValue(Key key, int color)
        {
            int j = key.j;
            int i = key.i;

            this->map[j * this->size + i] = color;
        }

        /*!*********************************************************************
        \brief
        An Operator Overloading function that finds all empty adjacent cells and
        insert into an array vector of nodes and return it.

        \param key
        Position of cell in grid

        \return
        Returns an array vector of nodes
        ***********************************************************************/
        std::vector<AI::Node*> operator()(Key key)
        {
            int j = key.j;
            int i = key.i;

            std::vector<AI::Node*> list = {};

            // Find and return all empty adjacent cells
            if (j >= 0 && j < this->size && i >= 0 && i < this->size)
            {
                if (i > 0 && this->map[j * this->size + i - 1] == 0)
                {
                    Node* newNode = new Node;
                    newNode->key.i = i - 1;
                    newNode->key.j = j;

                    list.push_back(newNode);
                }

                if (i < this->size-1 && this->map[j * this->size + i + 1] == 0)
                {
                    Node* newNode = new Node;
                    newNode->key.i = i + 1;
                    newNode->key.j = j;

                     list.push_back(newNode);
                }

                if (j > 0 && this->map[(j - 1) * this->size + i] == 0)
                {
                    Node* newNode = new Node;
                    newNode->key.j = j - 1;
                    newNode->key.i = i;

                    list.push_back(newNode);
                }

                if (j < this->size-1 && this->map[(j + 1)* this->size + i] == 0)
                {
                    Node* newNode = new Node;
                    newNode->key.j = j + 1;
                    newNode->key.i = i;

                    list.push_back(newNode);
                }
            }

            return list;
        }
    };

    // Domain specific functor that returns shuffled adjacent nodes
    class GetMapStochasticAdjacents : public GetMapAdjacents
    {
    public:

        GetMapStochasticAdjacents(int* map, int size)
            : GetMapAdjacents{ map, size }{}

       /*!***********************************************************************
       \brief
       An Operator Overloading function that finds all empty adjacent cells and
       insert into an array vector of Nodes, then shuffles the result and
       return it.

       \param key
       Position of cell in grid

       \return
       Returns an array vector of nodes
       *************************************************************************/
        std::vector<AI::Node*> operator()(Key key)
        {
            std::vector<AI::Node*> list = {};

            // Find and return all empty adjacent cells
            // Use the base class operator() and then shuffle the result

            list = GetMapAdjacents::operator()(key);
            std::random_shuffle(list.begin(), list.end());

            return list;
        }
    };

    // Wrappers that provide same interface for queue and stack

    struct Interface
    {
        virtual void clear() = 0;
        virtual void push(Node* pNode) = 0;
        virtual Node* pop() = 0;
    };

    struct Queue : Interface
    {
        std::vector<Node*> Q;
        int count = 0;

       /*!***********************************************************************
       \brief
       Clears all nodes in the vector array

       \param
       None.

       \return
       None.
       *************************************************************************/
        void clear()
        {
            Q.clear();
            count = 0;
        }

        /*!**********************************************************************
        \brief
        Add nodes by pushing back to the vector array

        \param
        None.

        \return
        None.
        ************************************************************************/
        void push(Node* pNode)
        {
           Q.push_back(pNode);
           ++count;
        }

       /*!***********************************************************************
       \brief
       Remove nodes by popping it from the back of the vector array

       \param
       None.

       \return
       None.
       *************************************************************************/
       Node* pop()
       {
           Node* pNode = nullptr;

           pNode = Q.back();
```

```
           Q.pop_back();
           --count;

           return pNode;
       }

       /*!**********************************************************************
       \brief
       Check if the vector array is empty

       \param
       None.

       \return
       Returns true if vector is empty, else returns false
       ************************************************************************/
       bool empty()
       {
           return (count == 0) ? true : false;
       }
    };

    struct Stack : Interface
    {
        std::vector<Node*> Stack;
        int count = 0;

        /*!*********************************************************************
        \brief
        Clears all nodes in the vector array

        \param
        None.

        \return
        None.
        ***********************************************************************/
        void clear()
        {
            Stack.clear();
        }

        /*!*********************************************************************
        \brief
        Add nodes by pushing back to the vector array

        \param
        None.

        \return
        None.
        ***********************************************************************/
        void push(Node* pNode)
        {
            Stack.push_back(pNode);
            ++count;
;        }

        /*!*********************************************************************
        \brief
        Remove nodes by popping it from the back of the vector array

        \param
        None.

        \return
        None.
        ***********************************************************************/
        Node* pop()
        {
            Node* pNode = nullptr;

            pNode = Stack.back();
            Stack.pop_back();
            --count;

            return pNode;
        }

        /*!*********************************************************************
        \brief
        Check if the vector array is empty

        \param
        None.

        \return
        Returns true if vector is empty, else returns false
        ***********************************************************************/
        bool empty()
        {
            return (count == 0) ? true : false;
        }
    };

    // Recursive Flood Fill
    class Flood_Fill_Recursive
    {
        GetAdjacents* pGetAdjacents;

    public:
        Flood_Fill_Recursive(GetAdjacents* pGetAdjacents)
            : pGetAdjacents{ pGetAdjacents }{}

        /*!*********************************************************************
        \brief
        Implement Recursive Flood Fill Algorithm
```

```cpp
        \param key
        Position of cell in grid

        \param color
        Color of cell

        \return
        None
        **********************************************************************/
        void run(Key key, int color)
        {
            // Implement the flood fill
            std::vector<AI::Node*> adjacentList =
                this->pGetAdjacents->operator()(key);

            for (auto adj : adjacentList)
            {
                GetMapAdjacents* mapAdj =
                    dynamic_cast<GetMapAdjacents*>(this->pGetAdjacents);

                mapAdj->setValue(adj->key, color);
                this->run(adj->key, color);
            }
        }
    };

    // Iterative Flood Fill
    // Type T defines is it depth- or breadth-first
    template<typename T>
    class Flood_Fill_Iterative
    {
        GetAdjacents* pGetAdjacents;
        T openlist;

    public:
        Flood_Fill_Iterative(GetAdjacents* pGetAdjacents)
            : pGetAdjacents{ pGetAdjacents }, openlist{}{}

        /*!*********************************************************************
        \brief
        Implement Iterative Flood Fill Algorithm, depth or breadth-first

        \param key
        Position of cell in grid

        \param color
        Color of cell

        \return
        None
        **********************************************************************/
        void run(Key key, int color)
        {
            // Implement the flood fill
            openlist.clear();
            openlist.push(new Node(key));

            while (!openlist.empty())
            {
                Node* current = openlist.pop();
                std::vector<Node*> adjacentList =
                    this->pGetAdjacents->operator()(current->key);

                for (auto adj : adjacentList)
                {
                    GetMapAdjacents* mapAdj =
                        dynamic_cast<GetMapAdjacents*>(this->pGetAdjacents);

                    mapAdj->setValue(adj->key, color);
                    this->openlist.push(adj);
                }
            }
        }
    };

} // end namespace

#endif
```

VPL

◄ Showcase: The Lines Game

Jump to...

Slides ►