

Influence Map Assignment

by Mathew Wade Alex Taylor

FILE	2885_MATHEW_WADE_ALEX_TAYLOR_INFLUENCE_MAP_ASSIGNMENT_177048_845881511.DOCX (377.33K)		
TIME SUBMITTED	22-MAR-2019 05:58PM (UTC+0000)	WORD COUNT	6083
SUBMISSION ID	103038285	CHARACTER COUNT	32942

Influence Maps

Student Name: Mathew Taylor

P Number: 149885

Module Title: Advanced AI and Machine Learning

Assignment Title: Influence Maps

Module Lecturer's Name: Gordon Dickers

Table of Contents

Table of Figures.....	2
Introduction	3
Task 1	4
Simple Influence.....	4
Convolution and Gaussian	4
Task 2	7
The Chosen Algorithm.....	7
Testing.....	7
Task 3	8
Game world and AI	8
Testing.....	9
References	10
Appendix	11
Task 2	11
Influence Map Equation Code Example	11
Task 3	11
Influence Map Implementation Code.....	11

Table of Figures

Figure 1 - Influence Map[1].....	3
Figure 2 - Rapid initial drop off with a larger range of influence[2]	4
Figure 3 - Plateaus first before rapidly trailing off[2]	4
Figure 4 - Gaussian Convolution Update Matrix[2]	4
Figure 5 - Update Matrix for a corner of a map[2]	5
Figure 6 - Update Matrix for the bottom edge of a map[2]	5
Figure 7 - Equation to calculate the influence on a single point in a map[2]	5
Figure 8 - Red influence source.....	7
Figure 9 - Green influence source.....	7
Figure 10 - Two opposite influences next to each other	7
Figure 11 - User selects the map they wish to load	8
Figure 12 - AI in between two different coloured sources	8
Figure 13 - AI moving where it should, towards the green	8
Figure 14 - Implementation of the equation	11
Figure 15 - Green Source is removed when the AI touches it	11

Introduction

Influence maps are grids placed over game worlds that contain values to each square based entirely on some functions which represent spatial features. [1][2] Influence maps are generally used to analyse a map or world by the AI in the game. A common game genre that uses influence maps are RTS (Real-Time Strategy) games, in this example the game might have one influence map for the enemies, one for the allies, and one for resources. The allied AI could find areas on the map with high resource density, low ally density, and low enemy density making a safe, valuable source of resources that nobody is already using. [3][4]

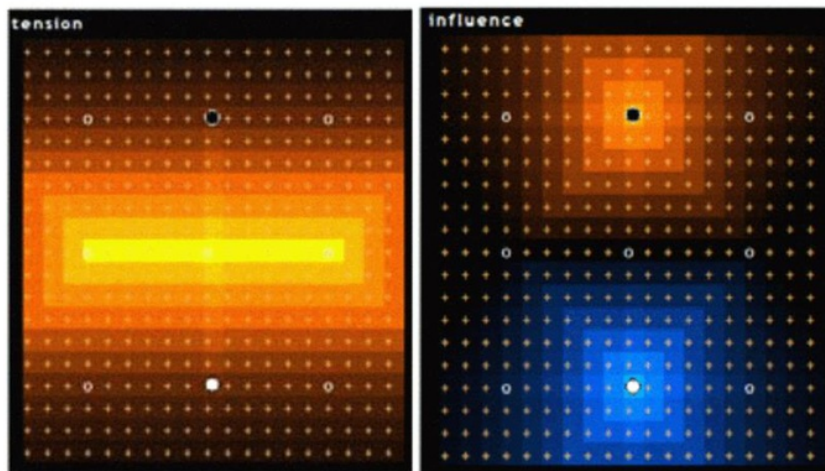


Figure 1 - Influence Map[1]

Task 1

Simple Influence

The equation shown in figure 2 has I_d as the influence to be calculated by using the given values of distance, d , and I_0 which is the influence at a distance of 0.[2]

Another option is to use a method in which there is a more rapid initial drop off, but has longer range of influence, the equation below in figure 2 will achieve this.

$$I_d = \frac{I_0}{\sqrt{1+d}},$$

Figure 2 - Rapid initial drop off with a larger range of influence[2]

Another option is to use the equation in figure 3 that maintains influence only close to the influence origin, but then rapidly trails off at a distance from the origin.[2]

$$I_d = \frac{I_0}{(1+d)^2}$$

Figure 3 - Plateaus first before rapidly trailing off[2]

Convolution and Gaussian



A popular way of updating analyses that involve spreading values out from their source is to use an image blurring algorithm. Convolution filters require a defined update matrix to tell the user how the value of a location on the map will be updated based on the values of its neighbours and its own value.[2]

Example:

Value of a location on the map and its neighbours.

20.0, 20.0, 18.0
22.0, 24.0, 26.0
24.0, 26.0, 30.0

Gaussian filter template (update matrix).

$$M = \frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}.$$

Figure 4 - Gaussian Convolution Update Matrix[2]

Multiply the Location values with the update matrix.

20.0, 20.0, 18.0 1/16, 2/16, 1/16 20/16, 40/16, 18/16
22.0, 24.0, 26.0 X 2/16, 4/16, 2/16 = 44/16, 96/16, 52/16
24.0, 26.0, 30.0 1/16, 2/16, 1/16 24/16, 52/16, 30/16

Add the nine output values together, then divide by the number of values to get the average.

$$20/16 + 40/16 + 18/16 + 44/16 + 96/16 + 52/16 + 24/16 + 52/16 + 30/16 = 376/16 = 23.5$$

$$23.5 / 9 = 2.6111...$$

If the location on the map being worked on happens to be either in the corner of the map or on the edge, then the update matrix will have to be altered. Instead of using the update matrix used in the previous example one could use the update matrixes shown below.

For a corner.

$$\frac{1}{9} \begin{bmatrix} 4 & 2 \\ 2 & 1 \end{bmatrix}$$

Figure 5 - Update Matrix for a corner of a map[2]

For an edge

$$\frac{1}{12} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \end{bmatrix}$$

Figure 6 - Update Matrix for the bottom edge of a map[2]

Using a Gaussian convolution template as shown in figure 4 with the value shown above with a location of those values gives an intensity on that point of 2.6111. This process must be completed for each location. This can cause issues, if you run the calculations and change the values on the influence map as you go along then the previous values calculated will affect the new values being calculated. To fix this problem two copies of the map will be needed, one contains the source data that will be read from and as calculations are done the new values are saved to the second map.[2]

The Gaussian filter method is used to blur images and remove noise, so in this scenario it makes the influence area around the source much smoother than the original.

While this method does reduce noise, it is considered to take time to execute and isn't as detailed as other methods.

The time and memory complexity is determined by using the big 'O' notation. The convolution method has a complexity of $O(mn)$, m is the number of points on the map, while n is the number influence sources. In this implementation 'm' is 2500 and 'n' is 17, giving a complexity of 42500.[2][5]

Another method for determining the influence is the equation shown in figure 7, which calculates the influence, I_d based upon the distance, d from an influence source, I_o that has an influence range, r .

$$I_d = \frac{I_o}{(e^{(d-r)} + 1)}$$

Figure 7 - Equation to calculate the influence on a single point in a map[2]

This method allows for the influence to be flat around a given distance from the centre of influence, but then tail off very rapidly. When it comes to using this method and algorithm two nested loops that loop through for the number of points in the map by the x coordinates and y coordinates will be used. Every time it goes through the second loop it will use the current position on the map being looked at and will grab the details of the sources, perform the calculation above for each source and then add the calculated values together, saving it to the point on the map.

Example using two sources:

Source 1
Position = (1,1)
Io1 = 3.5

Source 2
Position = (1,6)
Io2 = 4.3

Cell
Position = (4,5)
Radius = 9

The first step is to calculate the distance between the source and the cell being looked at. There are two methods to figure out the distance between the two points, there's the Pythagoras method and the trigonometry method. The trigonometry method requires the angles within the triangle that's made by the difference in the x and y so this method won't be usable as the angles are not known and would take extra computation to calculate.[6]

The method decided on to calculate the distance is Pythagoras due to familiarity with the method and its ease of use. When using Pythagoras, it might be necessary to work in square space when implementing it to reduce the computational cost, this will not be necessary for the example shown below though.

$$d = \text{square root of } (4 - 1)^2 + (5 - 1)^2$$

$$d = 5$$

Then all the values are put into the actual equation to calculate the influence from that source on the point.

$$Id = 3.5 / (e^{(5 - 9)} + 1)$$

$$Id = 3.44$$

Then the same calculations are done for the other source which gave an influence of 4.29

So the total influence on the point from the two source is the two calculated influences added together, giving a total influence of 7.73

Equation 1 isn't very likely to be used in influence maps due to its high computational cost as the distance check uses a square root and this calculation is done on every point in the map. A relatively simple way to alter this equation to improve its efficiency is to work in square space, so there's no need for the square root, this will drastically decrease the computational cost.

The complexity of the equation method is given by the square root and the division, which give it a complexity of $O(mn)$. [2][7]

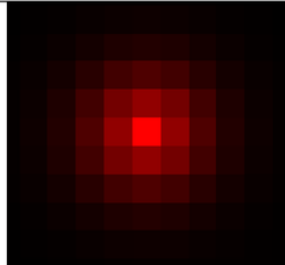
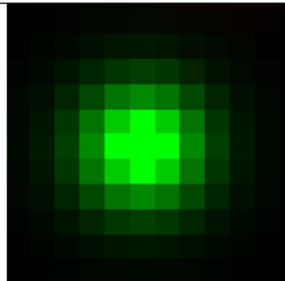
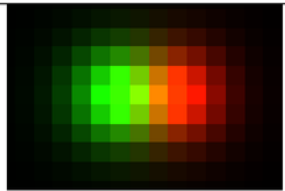
Task 2

The Chosen Algorithm

The chosen algorithm that was implemented is the equation method shown in figure 7, this method was chosen because it provides more detailed influence mappings than the convolution method as the points are not affected by the neighbouring point's influences. The only down side to using this method is that the distance check used in it is very computationally expensive, but this can easily be rectified by working in square space, so no square roots are used.

To implement this method there will be a two-dimensional array containing the information for the map. A nested for loop will be used to read through the two-dimensional array, and then another for loop will be used to check every source in the scene. Within the third for loop the actual calculations for the chosen method will take place, and the RGB values will be saved to the array, the RGB values saved represent the influence on that point, please see figure 14 in the appendix for the code detailing this implementation.

Testing

Test	Description	Expected result	Result	Pass?
1	Placing a red source on the map and displaying its influence.	Will see a red box with more red boxes spreading out from it gradually getting darker.	 <i>Figure 8 - Red influence source</i>	Yes
2	Placing a green source on the map and displaying its influence	Will see a green box with more green boxes spreading out from it gradually getting darker.	 <i>Figure 9 - Green influence source</i>	Yes
3	Placing a red and green influence source next to each other	The area in between the red and green source should become a yellowish/orange as it will have an influence from both sources.	 <i>Figure 10 - Two opposite influences next to each other</i>	Yes

Task 3

Game world and AI

When running the developed program, the user will be asked to pick a map to use as shown in figure 8. The map that was chosen map will be loaded uses opengl, and displayed on the screen. The data read from the map file is the location of the player AI, the enemy positions, the goal positions, and the wall positions.

```
Enter the influence map you want to load in by entering the corresponding number
1: Map1.txt
2: Map2.txt
```

Figure 11 - User selects the map they wish to load

The AI in the game world looks at the point above it, to the right of it, below it, and to the left of it.

The AI looks at the RGB values in the four points it's looking at, it multiplies the red value by two, add the blue value to it, then subtract the green value, this is because the AI needs to move towards the green influence and it's set up so it moves towards the smaller value and away from the higher value which will be whichever one has more red in it.

To stop the AI from returning to its last position repeatedly, a blue trail has been placed behind the AI so there's influence in it. If the AI has to choose between going into blue and going into red then it will go into blue, unless the red is much smaller, this is the reason for multiplying the red value by two.

The last check the AI does before moving is if the position it's looking at is a wall, if it is then set that value to the max out of the four grabbed values so it's never chosen.

Figure 9 shows the player AI in between a red point it should move away from and a green point it should move towards. While figure 10 shows the actual result, where the AI moves towards the green and away from the red, while leaving a very slight blue trail behind it.



Figure 12 - AI in between two different coloured sources



Figure 13 - AI moving where it should, towards the green

Testing

Test	Description	Expected Result	Result	Pass
1	Check if AI moves towards green.	AI will move towards green.	AI moves towards green sources. See Figure 13.	Yes
2	Check if the AI moves away from red.	AI will move away from red.	AI moves away from red sources. See figure 13.	Yes
3	Check if the AI will avoid going on walls.	AI will not move onto walls.	AI doesn't move onto walls.	Yes
4	Check if the green source disappears when the AI bot makes contact with it.	Green source will disappear when AI touches it.	Green source disappears when AI touches it. See Figure 15 in the appendix.	Yes
5	Check if the user can input anything other than the given choices for maps.	The user won't be able to select anything other than the choices.	The user is only able to input the shown choices.	Yes
6	Check if the correct map is loaded in when the user selects one.	The correct map will be loaded.	The correct map is loaded.	Yes

References

- [1] S. H. Jang and S. B. Cho, "Evolving neural NPCs with layered influence map in the real-time simulation game 'conqueror,'" *2008 IEEE Symp. Comput. Intell. Games, CIG 2008*, pp. 385–388, 2008.
- [2] I. Millington and J. Funge, *Artificial Intelligence for Games, Second Edition*. 2009.
- [3] J. Evans, "Influence Maps," pp. 1–3, 2013.
- [4] N. Hansson, "Game School Gems: Influence Maps I." 2010.
- [5] "Analysis of Algorithms: Lecture 7." [Online]. Available: <https://www.geeksforgeeks.org/analysis-algorithms-big-o-analysis/>. [Accessed: 22-Mar-2019].
- [6] "Measurement Session 5: Measurement and Trigonometry." [Online]. Available: https://www.learner.org/courses/learningmath/measurement/session5/part_a/measuring.html. [Accessed: 22-Mar-2019].
- [7] R. Bell, "A beginner 's guide to Big O notation," *A beginner's Guid. to Big O Not.*, pp. 1–5, 2015.

Appendix

Task 2

Influence Map Equation Code Example

```
//Loop through for the GRIDSIZE
for (int i = 0; i < GRIDSIZE; i++)
{
    //Loop through for the GRIDSIZE
    for (int j = 0; j < GRIDSIZE; j++)
    {
        //Loop through for the number of enemies
        for (int k = 0; k < enemyCounter; k++)
        {
            //Check if the current point being looked at is the position of the enemy
            if (i == enemy[k].x && j == enemy[k].y)
                tempGrid[i][j].r = 1.0; //Set the current point being looked at to red
            else
            {
                //Calculate the distance from the source point to the current point being looked at on the map
                distance = sqrt(((enemy[k].x - i) * (enemy[k].x - i)) + ((enemy[k].y - j) * (enemy[k].y - j)));

                float tempRValue = enemy[k].influence / (exp(distance - enemy[k].range) + 1);
                //Calculate the influence the source has on the point in the map
                tempGrid[i][j].r += tempRValue;

                if (tempRValue > highestInfluenceEnemy)
                    highestInfluenceEnemy = tempRValue;
            }
        }
    }
}
```

Figure 14 - Implementation of the equation

Task 3



Figure 15 - Green Source is removed when the AI touches it

Influence Map Implementation Code

//Includes

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <iostream>

#include <fstream>

#include <string>

#include <stack>

#include "glut.h"

using namespace std;

```
#define GRIDSIZE 50 //Grid size in the x and in the y
//#define ENEMYCOUNT 9 //Number of Enemies
#define MAZEWIDTH 16 //Maze width
#define MAZEHEIGHT 16 //Maze Height
//#define GOALCOUNT 8 //Number of goals
```

```
const char WALL = '#';
```

```
int enemyCounter, goalCounter, wallCounter;
```

```
float cameraX = GRIDSIZE / 2; //Camera x position
```

```
float cameraY = GRIDSIZE / 2; //Camera y position
```

```
float cameraZ = GRIDSIZE / 4; //Camera z position
```

```
float totalAngleX = 0.0;
```

```
float totalAngleY = 0.0;
```

```
typedef struct rgbtag {
```

```
    float r; //Red value
```

```
    float g; //Green value
```

```
    float b; //Blue value
```

```
}rgb;
```

```
typedef struct entities
```

```
{
```

```
    float influence; //Influence of the point
```

```
    float range; //Range of influence of the point
```

```
    int x; //x coordinate of the point
```

```
    int y; //y coordinate of the point
```

```
    int speedX; //speed of the point if required
```

```
    int speedY; //speed of the point if required
```

```
}entity;
```

```
char loadGrid[GRIDSIZE*GRIDSIZE]; //Holds the 1 dimensional values of the influence map being  
loaded in as chars
```

```
char loadGrid2d[GRIDSIZE][GRIDSIZE]; //Holds the 2 dimensional values of the influence map being  
loaded in as chars
```

```
rgb grid[GRIDSIZE][GRIDSIZE]; //Hold the rgb values for each point on the map
```

```
rgb tempGrid[GRIDSIZE][GRIDSIZE]; //Temporarily holds the new rgb values to be saved to the grid
```

```
entity *walls; //Holds the data for the walls
```

```
entity *goal; //Holds the data for the goals
```

```
entity *enemy; //Holds the data for the enemies
```

```
entity player; //Holds the data for the player
```

```
//function prototypes
```

```
void init(void);
```

```
void display(void);
```

```
void drawBox(float w, float h, float l);
```

```
void reshape(int w, int h);
```

```
void setInfluence(void);
```

```
void playerAI(void);
```

```
void updateGrid(void);
```

```
void readMapIn(void);
```

```
void init()
```

```
// Initialize
```

```
//*****
```

```
// desc: initializes OpenGL
```

```
{
```

```
    readMapIn();
```

```
    player.influence = 0.1; //Set the player's influence (The value that alters the last position the  
player was at
```

```

        glClearColor(0.0f, 0.0f, 0.0f, 0.0f);           // clears colour buffer
        glEnable(GL_DEPTH_TEST);                       // hidden surface removal
        glEnable(GL_CULL_FACE);                         // do not calculate
inside of poly's
        glFrontFace(GL_CCW);                           // counter clock-wise
polygons are out

```

```

int i, j, k;

```

```

//Loop through for the GRIDSIZE

```

```

for (i = 0; i < GRIDSIZE; i++)

```

```

{

```

```

    //Loop through for the GRIDSIZE

```

```

    for (j = 0; j < GRIDSIZE; j++)

```

```

    {

```

```

        //Set the whole grid to black

```

```

        grid[i][j].r = 0;

```

```

        grid[i][j].g = 0;

```

```

        grid[i][j].b = 0;

```

```

        //Loop through for ever enemy

```

```

        for (k = 0; k < enemyCounter; k++)

```

```

        {

```

```

            //Check if i and j match the coordinates of the enemy

```

```

            if (i == enemy[k].x && j == enemy[k].y)

```

```

            {

```

```

                grid[i][j].r = 1.0; //Set the enemy's position to red

```

```

            }

```

```

        }

```

```

        //Loop through for every goal

```

```

        for (k = 0; k < goalCounter; k++)
        {
            //Check if i and j match the coordinates of the goal
            if (i == goal[k].x && j == goal[k].y)
            {
                grid[i][j].g = 1.0; //Set the goal's position to green
            }
        }

        //Check if i and j match the coordinates of the player AI
        if (i == player.x && j == player.y)
        {
            grid[i][j].g = 1.0; //Set the player AI's position to green
        }
    }
}

setInfluence(); //Call the function that sets all the initial influences
}

void setInfluence()
{
    float distance; //Hold the calculated distance from one source on the map to another point
    on the map
    float highestInfluenceEnemy = 0.0;
    float highestInfluenceGoal = 0.0;

    //Loop through for the GRIDSIZE
    for (int i = 0; i < GRIDSIZE; i++)
    {
        //Loop through for the GRIDSIZE

```

```

for (int j = 0; j < GRIDSIZE; j++)
{
    //Loop through for the number of enemies
    for (int k = 0; k < enemyCounter; k++)
    {
        //Check if the current point being looked at is the position of the
enemy
        if (i == enemy[k].x && j == enemy[k].y)
            tempGrid[i][j].r = 1.0; //Set the current point being looked at
to red
        else
        {
            //Calculate the distance from the source point to the
current point being looked at on the map
            distance = sqrt(((enemy[k].x - i) * (enemy[k].x - i)) +
((enemy[k].y - j) * (enemy[k].y - j)));

            float tempRValue = enemy[k].influence / (exp(distance -
enemy[k].range) + 1);

            //Calculate the influence the source has on the point in the
map
            tempGrid[i][j].r += tempRValue;

            if (tempRValue > highestInfluenceEnemy)
                highestInfluenceEnemy = tempRValue;
        }
    }
    //Loop through for the number of goals
    for (int k = 0; k < goalCounter; k++)
    {
        //Check if the current point being looked at is the position of the
goal
        if (i == goal[k].x && j == goal[k].y)

```

```

                                tempGrid[i][j].g = 1.0; //Set the current point being looked
at to green
                                else
                                {
                                    //Calculate the distance from the source point to the
current point being looked at on the map
                                    distance = sqrt((goal[k].x - i) * (goal[k].x - i) + (goal[k].y - j) *
(goal[k].y - j));

                                    float tempGValue = goal[k].influence / (exp(distance -
goal[k].range) + 1);

                                    //Calculate the influence the source has on the point in the
map
                                    tempGrid[i][j].g += tempGValue;

                                    if (tempGValue > highestInfluenceGoal)
                                        highestInfluenceGoal = tempGValue;
                                }
                            }
                    }
                }
            }
            //player.y++;
            //Loop through for the GRIDSIZE
            for (int i = 0; i < GRIDSIZE; i++)
            {
                //Loop through for the GRIDSIZE
                for (int j = 0; j < GRIDSIZE; j++)
                {
                    tempGrid[i][j].r /= highestInfluenceEnemy; //Normalise the red values
                    tempGrid[i][j].g /= highestInfluenceGoal; //Normalise the green value

                    //Loop through for the GRIDSIZE

```

```

        for (int k = 0; k < wallCounter; k++)
        {
            //Create the white wall surrounding the scene so the player AI
            doesn't leave the area

            //Create white walls on the left
            if (i == walls[k].x && j == walls[k].y)
            {
                tempGrid[i][j].r = 1.0;
                //tempGrid[i][j].g = 1.0;
                //tempGrid[i][j].b = 1.0;
            }
        }

        //Check if i and j match the coordinates of the player AI
        if (i == player.x && j == player.y)
        {
            tempGrid[i][j].g = 1.0; //Set the position of the player in the map to
            green
        }
    }

    //copy calculated array into original
    for (int i = 0; i < GRIDSIZE; i++)
    {
        for (int j = 0; j < GRIDSIZE; j++)
        {
            grid[i][j] = tempGrid[i][j];
        }
    }
}

```

```

void drawBox(float w, float h, float l)

//drawBox

//*****

//This function first translates a further x,y,z units then draws a box of

//width w, height h, length l

{

    glPushMatrix();    //save modelview

    w = w / 2.0; h = h / 2.0; l = l / 2.0; //adjust values so centre is in middle of box

    box                                                         //draw faces of

    glBegin(GL_POLYGON);

    glVertex3f(w, l, h);

    glVertex3f(w, l, -h);

    glVertex3f(-w, l, -h);

    glVertex3f(-w, l, h);

    glEnd();

    glBegin(GL_POLYGON);

    glVertex3f(w, l, h);

    glVertex3f(-w, l, h);

    glVertex3f(-w, -l, h);

    glVertex3f(w, -l, h);

    glEnd();

    glBegin(GL_POLYGON);

    glVertex3f(w, l, h);

    glVertex3f(w, -l, h);

    glVertex3f(w, -l, -h);

    glVertex3f(w, l, -h);

    glEnd();

    glBegin(GL_POLYGON);

    glVertex3f(-w, l, h);

    glVertex3f(-w, l, -h);

```

```

        glVertex3f(-w, -l, -h);
        glVertex3f(-w, -l, h);
        glEnd();
        glBegin(GL_POLYGON);
        glVertex3f(-w, -l, h);
        glVertex3f(-w, -l, -h);
        glVertex3f(w, -l, -h);
        glVertex3f(w, -l, h);
        glEnd();
        glBegin(GL_POLYGON);
        glVertex3f(w, -l, -h);
        glVertex3f(-w, -l, -h);
        glVertex3f(-w, l, -h);
        glVertex3f(w, l, -h);
        glEnd();
        glPopMatrix(); //restore previous modelview matrix so leaving things as you found them
    }

// display
//*****
// desc: handles drawing of scene
void display()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);           // clear screen and
    depth buffer
    glLoadIdentity();
    // reset modelview matrix

    glRotatef(totalAngleX, 0, 1, 0); //Rotate around the y
    glRotatef(totalAngleY, 1, 0, 0); //Rotate around the x

```

```

        gluLookAt(cameraX, cameraY, cameraZ, cameraX, cameraY, cameraZ - 1, 0.0, 1.0, 0.0);
//reset camera position

        glColor3f(0, 0, 0); //set colour

        glTranslatef(0.5, 0.5, 0.5); //move by x,y and z units


//Loop through for the GRIDSIZE
for (int i = 0; i < GRIDSIZE; i++)
{
    //Loop through for the GRIDSIZE
    for (int j = 0; j < GRIDSIZE; j++)
    {
        if (grid[i][j].r > grid[i][j].b)
        {
            glColor3f(grid[i][j].r, 0, 0); //set the colour to red
        }
        else
        {
            glColor3f(0, 0, grid[i][j].b); //set colour to blue
        }

        if (grid[i][j].r > grid[i][j].g)
        {
            glColor3f(grid[i][j].r, 0, 0); //set colour to red
        }
        else
        {
            glColor3f(0, grid[i][j].g, 0); //set colour to green
        }
        if (grid[i][j].g != 0)
        {
            glColor3f(grid[i][j].r, grid[i][j].g, grid[i][j].b);
        }
    }
}

```

```

        }

        glPushMatrix(); //save current position/orientation, etc.
        glTranslatef(i, j, 0);
        drawBox(1, 1, 1); //draw
        glPopMatrix(); //restore previous "current" position/orientation etc.
    }
}

glFlush(); //force drawing

glutSwapBuffers();           // bring backbuffer to foreground
                             //any errors then display error
codes
    GLenum errCode;
    const GLubyte *errString;
    if ((errCode = glGetError()) != GL_NO_ERROR)
    {
        errString = gluErrorString(errCode);
        fprintf(stderr, "OpenGL error : %s\n", errString);
    }

}

//Handles the user inputs
void keyboardread(unsigned char key, int x, int y)
{
    //Values used to alter the camera
    float vx = 0, vy = 0, vz = 0, xAngle = 0, yAngle = 0;

```

```
switch (key)
{
case 'w':
{
    vy += 1; //Move Camera up
    break;
}
case 'd':
{
    vx += 1; //Move Camera right
    break;
}
case 's':
{
    vy -= 1; //Move Camera down
    break;
}
case 'a':
{
    vx -= 1; //Move Camera left
    break;
}
case 'j':
{
    xAngle += 10; //Turn image to the left
    break;
}
case 'l':
{
    xAngle -= 10; //Turn image to the right
    break;
}
```

```

    }
    case 'i':
    {
        vz -= 1; //Move camera forwards
        break;
    }
    case 'k':
    {
        vz += 1; //Move camera backwards
        break;
    }
    case 'u':
        break;
    case '':

        //Reset the tempGrid to all black except for the blue value
        for (int i = 0; i < GRIDSIZE; i++)
        {
            for (int j = 0; j < GRIDSIZE; j++)
            {
                tempGrid[i][j].r = 0.0;
                tempGrid[i][j].g = 0.0;
            }
        }

        playerAI(); //Run the playerAI function

        int oldPlayerX = player.x; //Holds player's last x position
        int oldPlayerY = player.y; //Holds player's last y position

        int oldEnemyX[GRIDSIZE * GRIDSIZE]; //Holds enemy's last x position

```

```

int oldEnemyY[GRIDSIZE * GRIDSIZE]; //Holds enemy's last y position

//Loop through for as many enemies as there are
for (int k = 0; k < enemyCounter; k++)
{
    oldEnemyX[k] = enemy[k].x; //Save the enemy's last x position
    oldEnemyY[k] = enemy[k].y; //Save the enemy's last y position

    //Check if the enemy has hit a wall and if so reverse their speed in the axis
    for (int l = 0; l < wallCounter; l++)
    {
        if (enemy[k].x == walls[l].x && enemy[k].y + 1 == walls[l].y ||
            enemy[k].x == walls[l].x && enemy[k].y - 1 == walls[l].y)
            enemy[k].speedY *= -1;

        if (enemy[k].x + 1 == walls[l].x && enemy[k].y == walls[l].y ||
            enemy[k].x - 1 == walls[l].x && enemy[k].y == walls[l].y)
            enemy[k].speedX *= -1;
    }

    enemy[k].x += enemy[k].speedX;
    enemy[k].y += enemy[k].speedY;
}

//Loop through for the number of goals or green targets on the map
for (int i = 0; i < goalCounter; i++)
{
    //Check if the player has landed on a goal
    if (player.x == goal[i].x && player.y == goal[i].y)
    {
        goal[i].x = -10;
        goal[i].y = -10;
    }
}

```

```

        //tempGrid[goal[i].x][goal[i].y].g = 0.0;
        //goal[i].influence = 0.0; //Set the goal's influence to 0.0
    }

}

//Loop through for the GRIDSIZE
for (int i = 0; i < GRIDSIZE; i++)
{
    //Loop through for the GRIDSIZE
    for (int j = 0; j < GRIDSIZE; j++)
    {
        //Loop through for the number of enemies
        //for (int k = 0; k < enemyCounter; k++)
        //{
            ///Check if i and j match the coordinate points of the
current enemy position being looked at
            //if (i == enemy[k].x && j == enemy[k].y)
            //    tempGrid[i][j].r = 1.0; //Set the r value of the
current enemy position to 1.0

            ///Check if i and j match the coordinate points of the last
position of the enemy being looked at
            //if (i == oldEnemyX[k] && j == oldEnemyY[k])
            //    tempGrid[i][j].r = 0.0; //Set the r value of the last
position the enemy was at to 0.0
            //}
            //Check if i and j match the coordinate points of the last player AI
position
            if (i == oldPlayerX && j == oldPlayerY)
            {
                tempGrid[i][j].b += player.influence; //Add the player
influence to b value in the last location of the player AI
            }
        }
    }
}

```

```

        tempGrid[i][j].g = 0.0; //Set the g value in the last location of
the player AI to 0.0
    }
    //Check if i and j match the coordinate points of the current player
AI position
    if (i == player.x && j == player.y)
    {
        tempGrid[i][j].g = 1.0; //Set the g value in the current player
position to 1.0
    }
}
}
updateGrid(); //Call the upgradeGrid function
break;
}

cameraX += vx; //Change the cameraX value, making the camera move in the x
cameraY += vy; //Change the cameraY value, making the camera move in the y
cameraZ += vz; //Change the cameraZ value, making the camera move in the z

totalAngleX += xAngle; //Change the totalAngleX value, making the camera rotate around
the y
totalAngleY += yAngle; //Change the totalAngleY value, making the camera rotate around
the z

glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glutPostRedisplay();
}

//Updates the grid
void updateGrid()
{

```

```

float distance; //Hold the calculated distance from one source on the map to another point
on the map

float highestInfluenceEnemy = 0.0;

float highestInfluenceGoal = 0.0;

//Loop through for the GRIDSIZE
for (int i = 0; i < GRIDSIZE; i++)
{
    //Loop through for the GRIDSIZE
    for (int j = 0; j < GRIDSIZE; j++)
    {
        //Loop through for the number of enemies
        for (int k = 0; k < enemyCounter; k++)
        {
            //Check if the current point being looked at is the position of the
enemy
            if (i == enemy[k].x && j == enemy[k].y)
            {
                tempGrid[i][j].r = 1.0; //Set the current point being looked at
to red
            }
            else
            {
                //Calculate the distance from the source point to the
current point being looked at on the map
                distance = sqrt(((enemy[k].x - i) * (enemy[k].x - i)) +
((enemy[k].y - j) * (enemy[k].y - j)));

                float tempRValue = enemy[k].influence / (exp(distance -
enemy[k].range) + 1);

                //Calculate the influence the source has on the point in the
map
                tempGrid[i][j].r += tempRValue;

                if (tempRValue > highestInfluenceEnemy)

```

```

highestInfluenceEnemy = tempRValue;

    }

}

//Loop through for the number of goals
for (int k = 0; k < goalCounter; k++)
{
    //Check if the current point being looked at is the position of the
goal
    if (i == goal[k].x && j == goal[k].y)
        tempGrid[i][j].g = 1.0; //Set the current point being looked
at to green
    else
    {
        //Calculate the distance from the source point to the
current point being looked at on the map
        distance = sqrt((goal[k].x - i) * (goal[k].x - i) + (goal[k].y - j) *
(goal[k].y - j));

        float tempGValue = goal[k].influence / (exp(distance -
goal[k].range) + 1);

        //Calculate the influence the source has on the point in the
map
        tempGrid[i][j].g += tempGValue;

        if (tempGValue > highestInfluenceGoal)
            highestInfluenceGoal = tempGValue;
    }
}

}

}

//Loop through for the GRIDSIZE

```

```

for (int i = 0; i < GRIDSIZE; i++)
{
    //Loop through for the GRIDSIZE
    for (int j = 0; j < GRIDSIZE; j++)
    {
        tempGrid[i][j].r /= highestInfluenceEnemy;
        tempGrid[i][j].g /= highestInfluenceGoal;

        //Loop through for the GRIDSIZE
        for (int k = 0; k < wallCounter; k++)
        {
            //Create the white wall surrounding the scene so the player AI
doesn't leave the area

            //Create white walls on the left
            if (i == walls[k].x && j == walls[k].y)
            {
                tempGrid[i][j].r = 1.0;
                //tempGrid[i][j].g = 1.0;
                //tempGrid[i][j].b = 1.0;
            }
        }
    }
}

//copy calculated array into original
for (int i = 0; i < GRIDSIZE; i++)
{
    for (int j = 0; j < GRIDSIZE; j++)
    {
        grid[i][j] = tempGrid[i][j];
    }
}

```

```

    }
}

void reshape(int w, int h)
//reshape
//*****
//OpenGL reshape callback is called when the window is resized
{
    glViewport(0, 0, (GLsizei)w, (GLsizei)h); //respecify size of view port to match new w and h
    glMatrixMode(GL_PROJECTION); //go into projection mode
    glLoadIdentity(); //reset projection matrix
    //glOrtho(0.0, GRIDSIZE, 0.0, GRIDSIZE, 0.0, 100.0); //defines orthoganal projection (parrallel

    // viewing volume

    // window limits of display

    // left =0.0, right =2.0,

    // bottom = 0.0, top = 2.0

    // near =0.0, far = 100.0
    gluPerspective(500.0,(GLfloat)w/(GLfloat)h, 0.1, 100.0); //redefine projection matrix to
match changes in w and h
    glMatrixMode(GL_MODELVIEW); //switch back to modelview mode
    glLoadIdentity(); //reset modelview matrix
    gluLookAt(GRIDSIZE / 2, GRIDSIZE / 2, 7.0f, GRIDSIZE / 2, GRIDSIZE / 2, 6.0f, 0.0, 1.0, 0.0);
//reset camera position
}

void main(int argc, char** argv)
{
    glutInit(&argc, argv);

```

```

    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);

    glutInitWindowSize(500, 500);

    glutInitWindowPosition(100, 100);

    glutCreateWindow(argv[0]);

    init();

    glutDisplayFunc(display);

    glutReshapeFunc(reshape);

    glutKeyboardFunc(keyboardread);

    glutMainLoop();
}

void playerAI()
{
    int movementCheck = 0; //0 = player moves up, 1 = player moves right, 2 = player moves
down, 3 = player moves left

    float upR, rightR, downR, leftR; //Holds the altered rgb(influence) values

    //Grab the red value and multiply by 2 because you don't want the player to move towards
red

    //then add the blue value because you don't want the player to move towards blue,

    //but don't multiply it as you want the player to move to blue instead of red

    //then subtract the green value from it because you want the player to move towards green

    //Do this for above the player, to the right of the player, below the player, and to the left of
the player

    upR = (grid[player.x][player.y + 1].r) * 2 + (grid[player.x][player.y + 1].b) -
(grid[player.x][player.y + 1].g);

    rightR = (grid[player.x + 1][player.y].r) * 2 + (grid[player.x + 1][player.y].b) - (grid[player.x +
1][player.y].g);

    downR = (grid[player.x][player.y - 1].r) * 2 + (grid[player.x][player.y - 1].b) -
(grid[player.x][player.y - 1].g);

    leftR = (grid[player.x - 1][player.y].r) * 2 + (grid[player.x - 1][player.y].b) - (grid[player.x -
1][player.y].g);

```

```
if (player.y + 1 >= GRIDSIZE - 1)
    upR += rightR + downR + leftR + upR;
if (player.y - 1 <= 0)
    downR += upR + rightR + leftR + downR;
if (player.x + 1 >= GRIDSIZE - 1)
    rightR += upR + downR + leftR + rightR;
if (player.x - 1 <= 0)
    leftR += upR + downR + rightR + leftR;

//Check if right has the smallest influence
if (rightR < upR && rightR < downR && rightR < leftR)
{
    movementCheck = 1;
}

//Check if down has the smallest influence
if (downR < upR && downR < rightR && downR < leftR)
{
    movementCheck = 2;
}

//Check if left has the smallest influence
if (leftR < upR && leftR < downR && leftR < rightR)
{
    movementCheck = 3;
}

//Check if up has the smallest influence
if (upR < leftR && upR < downR && upR < rightR)
{
    movementCheck = 0;
}
```

```
//Move the player differently depending on the value of movementCheck
switch (movementCheck)
{
case 0:
{
    player.y++; //Move the player up
    break;
}
case 1:
{
    player.x++; //Move the player right
    break;
}
case 2:
{
    player.y--; //Move the player down
    break;
}
case 3:
{
    player.x--; //Move the player left
    break;
}
default:
    break;
}

}

void readMapIn()
{
    string line;
```

```

string infMapFile;

ifstream infMapList;

int count;
char choice[32];

do
{
    count = 0;
    infMapList.open("Influence Map List.txt"); //Opens the file containing the list of
influence maps

    cout << "Enter the influence map you want to load in by entering the corresponding
number" << endl;

    //Read the entire file and print out the options for the user
    while (getline(infMapList, line))
    {
        count++;
        cout << count << ": " << line << endl;
    }

    cin >> choice; //User inputs the influence map file they wish to load

    infMapList.close(); //close the file

    infMapList.open("Influence Map List.txt"); //Opens the file containing the list of
influence maps

    count = 0;

```

```

        //Read the entire file and check if the choice the user input was valid
        while (getline(infMapList, line))
        {
            count++;

            //Check if the input was valid
            if (count == atoi(choice))
            {
                infMapFile = line; //Grab the name of the file that the user chose
            }
        }

        infMapList.close(); //Close file

    } while (infMapFile.empty()); //Loop until the input is valid and so infMapFile isn't empty

    ifstream infMap(infMapFile); //Opens the file containing the maze

    count = 0;

    //Loop through until the entire file has been read
    while (getline(infMap, line))
    {
        line.copy(&loadGrid[GRIDSIZE*count], GRIDSIZE, 0); //Read in the maze and add it to
the maze variable
        count++;
    }

```

```
infMap.close(); //Close the file

int enemyCount = 0, goalCount = 0, wallCount = 0;

//Count the total number of enemies, goals, and walls
for (int i = 0; i < GRIDSIZE; i++)
{
    for (int j = 0; j < GRIDSIZE; j++)
    {
        if (loadGrid[GRIDSIZE*i + j] == 'E' || loadGrid[GRIDSIZE*i + j] == 'B')
            enemyCount++;

        if (loadGrid[GRIDSIZE*i + j] == 'G')
            goalCount++;

        if (loadGrid[GRIDSIZE*i + j] == '#')
            wallCount++;
    }
}

//Save the read in map into a 2 dimensional array
for (int i = 0; i < GRIDSIZE; i++)
{
    for (int j = 0; j < GRIDSIZE; j++)
    {
        loadGrid2d[i][j] = loadGrid[GRIDSIZE*i + j];
    }
}

enemy = new entity[enemyCount]; //Set the size of the enemy array of entities
goal = new entity[goalCount]; //Set the size of the goal array of entities
```

```
walls = new entity[wallCount]; //Set the size of the walls array of entities
```

```
//Save the counts to global variables for future use
```

```
enemyCounter = enemyCount;
```

```
goalCounter = goalCount;
```

```
wallCounter = wallCount;
```

```
enemyCount = 0, goalCount = 0, wallCount = 0; //Reset the counts
```

```
//Save the read in amp into a 2 dimensional array
```

```
for (int i = 0; i < GRIDSIZE; i++)
```

```
{
```

```
    for (int j = 0; j < GRIDSIZE; j++)
```

```
    {
```

```
        //Load in the all the positions and other variables for the enemies
```

```
        if (loadGrid2d[i][j] == 'E')
```

```
        {
```

```
            enemy[enemyCount].x = i;
```

```
            enemy[enemyCount].y = j;
```

```
            enemy[enemyCount].influence = 0.5;
```

```
            enemy[enemyCount].range = 1.0;
```

```
            //Randomise the direction that the enemies are moving in
```

```
            int speedx = rand() % 2;
```

```
            int speedy = rand() % 2;
```

```
            int xDirection = rand() % 2;
```

```
            int yDirection = rand() % 2;
```

```
            if (speedx == 1)
```

```
            {
```

```

        if (xDirection == 1)
            enemy[enemyCount].speedX = 1;
        else
            enemy[enemyCount].speedX = -1;
    }
    else
    {
        enemy[enemyCount].speedX = 0;
    }
    if (speedy == 1)
    {
        if (yDirection == 1)
            enemy[enemyCount].speedY = 1;
        else
            enemy[enemyCount].speedY = -1;
    }
    else
    {
        enemy[enemyCount].speedY = 0;
    }
    enemyCount++;
}

//Load in all the details for the big enemy
if (loadGrid2d[i][j] == 'B')
{
    enemy[enemyCount].x = i;
    enemy[enemyCount].y = j;
    enemy[enemyCount].influence = 0.5;
    enemy[enemyCount].range = 3.0;
    enemy[enemyCount].speedX = 0.0;

```

```

        enemy[enemyCount].speedY = 0.0;
        enemyCount++;
    }

    //Load in all the details for the goals
    if (loadGrid2d[i][j] == 'G')
    {
        goal[goalCount].x = i;
        goal[goalCount].y = j;
        goal[goalCount].influence = 1.0;
        goal[goalCount].range = 1.0;
        goalCount++;
    }

    //Load in all the details for the walls
    if (loadGrid2d[i][j] == '#')
    {
        walls[wallCount].x = i;
        walls[wallCount].y = j;
        wallCount++;
    }

    if (loadGrid2d[i][j] == 'S') //look for the S in the maze and set the player's
start position to it
    {
        player.x = i;
        player.y = j;
    }
}
}
}

```

Influence Map Assignment

GRADEMARK REPORT

FINAL GRADE

GENERAL COMMENTS

Instructor

83/100

PAGE 1

PAGE 2

PAGE 3

PAGE 4



Comment 1

apostrophes denote possession - none here so lose the apostrophe.

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

PAGE 13

PAGE 14

PAGE 15

PAGE 16

PAGE 17

PAGE 18

PAGE 19

PAGE 20

PAGE 21

PAGE 22

PAGE 23

PAGE 24

PAGE 25

PAGE 26

PAGE 27

PAGE 28

PAGE 29

PAGE 30

PAGE 31

PAGE 32

PAGE 33

PAGE 34

PAGE 35

PAGE 36

PAGE 37

PAGE 38

PAGE 39

PAGE 40

MATHEW WADE ALEX TAYLOR

83

RESEARCH-INFORMED LITERATURE



7.5/10 Good review of literature relevant to the problem specification. References appropriate. A good range but avoid too many electronic refs where possible.

7.5

KNOWLEDGE AND UNDERSTANDING



20/25 V Good understanding demonstrated both in code and concepts.

20

ANALYSIS



16/20 Convolution calc correct but no need to divide by 9.. Distance calcs also correct. Well done. Analysis of complexity in evidence.

16

PRACTICAL APPLICATION AND DEP.



36/40 Loads influence map file. Good AI bot behaviours. Both positive and negative influences demonstrated and AI bot contributes to influence map-good. Algorithms used tested against the port's test calculations. Evidence of good SE practices. Excellent.

36

SKILLS FOR PROFESSIONAL PRAC.



3.5/5 Well structured report. Very minor typo/grammar errors. Refs appropriate.

3.5