**Wumpus World**

1. The Problem

* Create a program which makes a ‘cave’ filled with traps. Somewhere inside is a pile of gold, which the player wants to collect and bring back to the entrance without dying. The player dies by finding a Wumpus or falling into a Pit.
* The only hints the Player receives is in console prints. When the players gets within one tile from a trap or the gold, they receive a line of text indicating as such. If the player thinks they have found the Wumpus, the can also optionally face the tile and shoot an arrow.

1. Input

* The player can either use arrow keys or text commands to move their character
  1. The left and right arrow keys change the direction the player is facing. Pressing the up arrow will move the player towards the direction they are facing. The player can also optionally mark tiles that they think may be trapped; i.e. if they think the Wumpus is in the tile they are facing, they can mark it as being a possible Wumpus tile. This is done with W - Wumpus, P - Pit, G - Gold, and X - to clear a mark. Finally, they can also shoot their arrow with the F key
  2. Using text inputs has all the same functionality. Capitalization in all of these messages does not matter. Type in a cardinal direction (N - North, S - South, E - East, W - West) to face that direction. Type in U - Up, M - Move, Forward, or Walk to move in the direction the player is facing. Type Wumpus to mark a tile as a possible Wumpus, Pit for a Pit, Gold for Gold, and Clear to remove a marked tile.

1. Output

* The player is given signals throughout the game if he is near a trap. “The Robot smells and awful stench…” means you are near a Wumpus, “The Robot feels a breeze” means you are near a pit, “The Robot sees a glitter…” means you are near the gold.
* The only other output is instructions on how to move throughout the grid

1. User Interface

* The console window in Windows is the Interface in this program

////////////////////////////////////////////////////////////

// File: Main.cpp

// Author: Ben Odom

// Date Created: 09/04/2015

////////////////////////////////////////////////////////////

#include "LoopHandle.h"

void main()

{

LoopHandle::Init();

LoopHandle::Quit();

}

////////////////////////////////////////////////////////////

// File: LoopHandle.h

// Author: Ben Odom

// Date Created: 09/04/2015

// Brief: This file contains the namespace 'LoopHandle'

// which will act as a handle for the games main loop

// but not be part of it. It could technically handle

// multiple game loops within it or completely

// separate games entirely

////////////////////////////////////////////////////////////

#ifndef \_LOOPHANDLE\_H\_

#define \_LOOPHANDLE\_H\_

#include "MainLoop.h"

#include "EventHandle.h"

#include "Dungeon.h"

namespace LoopHandle

{

void Init();

void Quit();

}

#endif // \_LOOPHANDLE\_H\_

////////////////////////////////////////////////////////////

// File: LoopHandle.cpp

// Author: Ben Odom

// Date Created: 09/04/2015

////////////////////////////////////////////////////////////

#include "LoopHandle.h"

namespace LoopHandle

{

void Init()

{

System::Init();

MainLoop oMainLoop; // Creates an object of the 'MainLoop' class

oMainLoop.Run();

}

void Quit()

{

System::Quit();

}

}

////////////////////////////////////////////////////////////

// File: Loop.h

// Author: Ben Odom

// Date Created: 09/04/2015

// Brief: This file contains the class 'Loop'

// which will act as a template for all other game

// loops

////////////////////////////////////////////////////////////

#ifndef \_LOOP\_H\_

#define \_LOOP\_H\_

#include "EventHandle.h"

namespace LoopHandle

{

class Loop : public EventHandle

{

public:

void Run(); // Holds a prototype for any loops that inherit 'Loop'

virtual void Handle(); // A virtual function that will get called once per loop

virtual void Draw(); // A virtual function that will get called once per loop

Loop();

virtual ~Loop();

bool m\_bIsRunning; // Whether or not to keep looping

};

}

#endif // \_LOOP\_H\_

////////////////////////////////////////////////////////////

// File: Loop.cpp

// Author: Ben Odom

// Date Created: 09/04/2015

////////////////////////////////////////////////////////////

#include "Loop.h"

namespace LoopHandle

{

void Loop::Run()

{

while (m\_bIsRunning)

{

PollInput(); // From EventHandle

Handle();

}

}

void Loop::Handle()

{

// Virtual, Do Nothing...

}

void Loop::Draw()

{

// Virtual, Do Nothing...

}

Loop::Loop()

{

}

Loop::~Loop()

{

// Virtual, Do Nothing...

}

}

////////////////////////////////////////////////////////////

// File: MainLoop.h

// Author: Ben Odom

// Date Created: 09/04/2015

// Brief: This file contains the class 'MainLoop' which

// inherits from the 'Loop' class. This class will

// handle the main loop of the Wumpus World and call

// all necessary functions related to that part of

// the program

////////////////////////////////////////////////////////////

#ifndef \_MAINLOOP\_H\_

#define \_MAINLOOP\_H\_

#include "Loop.h"

#include "Dungeon.h"

#include "Robot.h"

#include <vector>

namespace LoopHandle

{

class MainLoop : public Loop

{

public:

void Handle(); // Inherited from 'Loop'. Will call 'Robot's 'Handle' function as well

void OnKeyPress(int a\_iKey); // Inherited from 'EventHandle'. Will pass this information on to 'Robot's 'OnKeyPress' as well

void OnKeyRepeat(int a\_iKey); // Inherited from 'EventHandle'.

void OnKeyRelease(int a\_iKey); // Inherited from 'EventHandle'.

MainLoop(); // Will ask the user which kind of input they want to use

~MainLoop();

private:

int m\_iControls; // Will be set to 0 for text input, and 1 for regular input

};

}

#endif // \_MAINLOOP\_H\_

////////////////////////////////////////////////////////////

// File: MainLoop.cpp

// Author: Ben Odom

// Date Created: 09/04/2015

////////////////////////////////////////////////////////////

#include "MainLoop.h"

const unsigned int PRINT\_WIDTH = 50;

const unsigned int PRINT\_POSX = 30;

namespace LoopHandle

{

void MainLoop::Handle() // Inherited from 'Loop'. Will call 'Robot's 'Handle' function as well

{

if (m\_iControls == 0)

m\_bIsRunning = Dungeon::Handle();

}

void MainLoop::OnKeyPress(int a\_iKey) // Inherited from 'EventHandle'. Will pass this information on to 'Robot's 'OnKeyPress' as well

{

switch (a\_iKey)

{

case VK\_ESCAPE: m\_bIsRunning = false; break;

default:

{

if (m\_iControls == 1)

m\_bIsRunning = Dungeon::OnKeyPress(a\_iKey);

break;

}

}

}

void MainLoop::OnKeyRepeat(int a\_iKey) // Inherited from 'EventHandle'.

{

switch (a\_iKey)

{

default:break;

}

}

void MainLoop::OnKeyRelease(int a\_iKey) // Inherited from 'EventHandle'.

{

switch (a\_iKey)

{

default: break;

}

}

MainLoop::MainLoop() // Will ask the user which kind of input they want to use

{

m\_bIsRunning = true;

System::Print("Do you want to use text input or the arrow keys to control your character?", PRINT\_WIDTH, PRINT\_POSX, 0);

System::Print(">> ", PRINT\_WIDTH, PRINT\_POSX, 3);

char cBuffer[256];

System::ShowCursor();

std::cin.getline(cBuffer, 256);

System::ToLower(cBuffer, 256);

if (!strcmp(cBuffer, "text") || !strcmp(cBuffer, "text input")) // Remeber that strcmp returns 0 if it finds the requested char\*

m\_iControls = 0; // Text input

else

m\_iControls = 1; // Regular input

Dungeon::Init();

}

MainLoop::~MainLoop()

{

Dungeon::Quit();

}

}

////////////////////////////////////////////////////////////

// File: EventHandle.h

// Author: Ben Odom

// Date Created: 09/04/2015

// Brief: This file contains the 'EventHandle' class which

// contains all the virtual functions needed for

// input. All of the 'OnKey' functions get called

// automatically in the loop, and simply need to be

// redefined by the inheriting class to be used.

////////////////////////////////////////////////////////////

#ifndef \_EVENTHANDLE\_H\_

#define \_EVENTHANDLE\_H\_

#include "System.h"

#include <vector>

#include <Windows.h>

#include <ctime>

class EventHandle

{

public:

void ParseInput(); // Decides which keys have been pressed/released and which need to be repeated

void OrderInput(); // If input has moved around in the vector, re-orders it to match

void PollInput(); // Grabs input from 'GetAsyncKeyState()'

// There is no built-in functionality for key presses and releases, so calling these functions when inputs change will simulate it

virtual void OnKeyPress(int a\_iKey); // Gets called when a key is pressed down initially

virtual void OnKeyRepeat(int a\_iKey); // Gets called when the delay time has been reached, and everytime the repeat time has been reached

virtual void OnKeyRelease(int a\_iKey); // Gets called whne the key is no longer recognized as being pressed down

EventHandle();

virtual ~EventHandle();

private:

// This vector hold all the keys pressed on the current frame and the last one respectively

std::vector<int> m\_vKeys, m\_vPrevKeys;

// These vectors hold the time in milliseconds when the key can be considered held long enough to request a single repeat, and then continuous repetition respectively

std::vector<int> m\_vUntilRepeat, m\_vUntilPressed;

};

#endif // \_EVENTHANDLE\_H\_

////////////////////////////////////////////////////////////

// File: EventHandle.cpp

// Author: Ben Odom

// Date Created: 09/04/2015

////////////////////////////////////////////////////////////

#include "EventHandle.h"

const unsigned int KEY\_REPEAT\_PRESS\_DELAY = 60;

const unsigned int KEY\_PRESS\_DELAY = 200;

const unsigned int MAX\_KEYS = 8;

void EventHandle::ParseInput() // Decides which keys have been pressed/released and which need to be repeated

{

if (m\_vKeys.size() < m\_vPrevKeys.size())

{

for (int i = m\_vKeys.size(); i < m\_vPrevKeys.size(); ++i)

OnKeyRelease(m\_vPrevKeys[i]);

}

for (int i = 0; i < m\_vKeys.size(); ++i)

{

if ((i >= m\_vPrevKeys.size()) || (m\_vKeys[i] != m\_vPrevKeys[i]))

{

if (i >= m\_vPrevKeys.size())

{

m\_vUntilPressed.push\_back(clock() + KEY\_PRESS\_DELAY);

m\_vUntilRepeat.push\_back(clock() + KEY\_PRESS\_DELAY + KEY\_REPEAT\_PRESS\_DELAY);

}

OnKeyPress(m\_vKeys[i]);

}

else if ((m\_vKeys[i] == m\_vPrevKeys[i]) && (clock() > m\_vUntilRepeat[i]) && (clock() > m\_vUntilPressed[i]))

{

m\_vUntilRepeat[i] = clock() + KEY\_REPEAT\_PRESS\_DELAY;

OnKeyRepeat(m\_vKeys[i]);

}

}

m\_vPrevKeys = m\_vKeys;

m\_vKeys.clear();

}

void EventHandle::OrderInput() // If input has moved around in the vector, re-orders it to match

{

for (int i = 0; i < m\_vKeys.size(); ++i)

{

for (int j = 0; j < m\_vPrevKeys.size(); ++j)

{

if ((i < m\_vPrevKeys.size()) && (m\_vKeys[i] == m\_vPrevKeys[j]))

{

iter\_swap(m\_vPrevKeys.begin() + i, m\_vPrevKeys.begin() + j);

iter\_swap(m\_vUntilRepeat.begin() + i, m\_vUntilRepeat.begin() + j);

iter\_swap(m\_vUntilPressed.begin() + i, m\_vUntilPressed.begin() + j);

}

else if ((j < m\_vKeys.size()) && (m\_vKeys[i] == m\_vPrevKeys[j]))

iter\_swap(m\_vKeys.begin() + i, m\_vKeys.begin() + j);

}

}

}

void EventHandle::PollInput() // Grabs input from 'GetAsyncKeyState()'

{

int i = 0;

FlushConsoleInputBuffer(GetStdHandle(STD\_INPUT\_HANDLE));

if (GetAsyncKeyState(VK\_LEFT) & 0x8000) { m\_vKeys.push\_back(VK\_LEFT); ++i; }

if (GetAsyncKeyState(VK\_RIGHT) & 0x8000) { m\_vKeys.push\_back(VK\_RIGHT); ++i; }

if (GetAsyncKeyState(VK\_UP) & 0x8000) { m\_vKeys.push\_back(VK\_UP); ++i; }

if (GetAsyncKeyState(VK\_DOWN) & 0x8000) { m\_vKeys.push\_back(VK\_DOWN); ++i; }

if (GetAsyncKeyState('W') & 0x8000) { m\_vKeys.push\_back(0x57); ++i; } // W Key

if (GetAsyncKeyState('A') & 0x8000) { m\_vKeys.push\_back(0x41); ++i; } // A Key

if (GetAsyncKeyState('S') & 0x8000) { m\_vKeys.push\_back(0x53); ++i; } // S Key

if (GetAsyncKeyState('D') & 0x8000) { m\_vKeys.push\_back(0x44); ++i; } // D Key

if (GetAsyncKeyState('P') & 0x8000) { m\_vKeys.push\_back(0x50); ++i; } // P Key

if (GetAsyncKeyState('G') & 0x8000) { m\_vKeys.push\_back(0x47); ++i; } // G Key

if (GetAsyncKeyState('X') & 0x8000) { m\_vKeys.push\_back(0x58); ++i; } // X Key

if (GetAsyncKeyState('F') & 0x8000) { m\_vKeys.push\_back(0x46); ++i; } // F Key

if (GetAsyncKeyState(VK\_RETURN) & 0x8000) { m\_vKeys.push\_back(VK\_RETURN); ++i; }

if (GetAsyncKeyState(VK\_PRIOR) & 0x8000) { m\_vKeys.push\_back(VK\_PRIOR); ++i; }

if (GetAsyncKeyState(VK\_NEXT) & 0x8000) { m\_vKeys.push\_back(VK\_NEXT); ++i; }

if (GetAsyncKeyState(VK\_ESCAPE) & 0x8000) { m\_vKeys.push\_back(VK\_ESCAPE); ++i; }

OrderInput();

ParseInput();

if (!i)

{

m\_vPrevKeys.clear();

m\_vUntilPressed.clear();

m\_vUntilRepeat.clear();

}

}

void EventHandle::OnKeyPress(int a\_iKey) // Gets called when a key is pressed down initially

{

// Virtual, Do Nothing...

}

void EventHandle::OnKeyRepeat(int a\_iKey) // Gets called when the delay time has been reached, and everytime the repeat time has been reached

{

// Virtual, Do Nothing...

}

void EventHandle::OnKeyRelease(int a\_iKey) // Gets called whne the key is no longer recognized as being pressed down

{

// Virtual, Do Nothing...

}

EventHandle::EventHandle()

{

// Empty Constructor

}

EventHandle::~EventHandle()

{

// Virtual, Do Nothing...

}

////////////////////////////////////////////////////////////

// File: Dungeon.h

// Author: Ben Odom

// Date Created: 09/04/2015

// Brief: This file contains the 'Dungeon' namespace which

// will act as the Wumpus World handler so to speak.

// It has an member object of the Player 'Robot' and

// will make sure to send output to the user as to

// what's happening in the Wumpus World.

////////////////////////////////////////////////////////////

#ifndef \_DUNGEON\_H\_

#define \_DUNGEON\_H\_

#include "Robot.h"

namespace Dungeon

{

bool Handle(); // This is called by the 'MainLoop' when text input is selected

bool OnKeyPress(int a\_iKey); // THis is called by the 'MainLoop' when standard input is selected

void Init();

void Quit();

}

#endif // \_DUNGEON \_H\_

////////////////////////////////////////////////////////////

// File: Dungeon.cpp

// Author: Ben Odom

// Date Created: 09/04/2015

////////////////////////////////////////////////////////////

#include "Dungeon.h"

namespace Dungeon

{

Robot oRobot;

bool Handle() // This is called by the 'MainLoop' when text input is selected

{

if (oRobot.IsAlive())

oRobot.Handle();

return oRobot.IsAlive(); // Returns if the program should continue or not

}

bool OnKeyPress(int a\_iKey) // THis is called by the 'MainLoop' when standard input is selected

{

if (oRobot.IsAlive())

oRobot.OnKeyPress(a\_iKey);

return oRobot.IsAlive(); // Returns if the program should continue or not

}

void Init()

{

oRobot = Robot(1, 1, DIRECTION::DOWN); // Create a player character at (1, 1) facing downwards

}

void Quit()

{

}

}

////////////////////////////////////////////////////////////

// File: Robot.h

// Author: Ben Odom

// Date Created: 09/04/2015

// Brief: This file contains the functions required to move

// the player character either by text commands or

// by regular input. 'OnKeyPress' will be the regular

// input and 'Handle' will be text input

////////////////////////////////////////////////////////////

#ifndef \_ROBOT\_H\_

#define \_ROBOT\_H\_

#include "Grid.h"

namespace Dungeon

{

class Robot

{

public:

void Handle(); // Handles text input and parses it to move the player

void RecieveSignal(); // Handles both text and regular input signals that get sent to the console to alert the player

bool Collision(); // Checks to make sure the player hasn't died, found gold, returned gold, or gone into a wall

void Draw(); // Draws the player and grid onto the screen

void FireArrow(); // Attempts to fire an arrow at the Wumpus. Can fail and waste the arrow

void KillWumpus(int a\_iPosX, int a\_iPosY); // Removes the Wumpus from the grid

void FailShot(); // When the user fails to hit the Wumpus

void GetPos(int &a\_iPosX, int &a\_iPosY, int &a\_iFacing); // Returns the current position of the player and where they are facing. Currently unused

bool IsAlive(); // Returns whether or not the player is alive. When you get the gold or die the loop will exit

void OnKeyPress(int a\_iKey); // Handles regular input and moves the player accordingly. Check 'EventHandle.h' for the technicalities of how input is gathered

void Clear(); // Clears the screen and redraws it

Robot(); // Empty constructor

Robot(int a\_iPosX, int a\_iPosY, int a\_iFacing); // Overloaded constructor with parameters to set it's values

~Robot(); // Empty deconstructor

private:

int m\_iPosX, m\_iPosY; // X and Y position in the grid

int m\_iFacing; // Direction the player is facing. Check 'enum DIRECTION' for possible values

int m\_iArrowCount; // The amount of arrows the player is able to use

int m\_iPrintLine; // What line to print text onto since the x value is set to 40 and will otherwise wrap around poorly

Grid m\_oGrid, m\_oPercievedGrid; // The actual game grid, and the one that the player can customize respectivly

bool m\_bIsAlive, m\_bHasGold; // If the player is alive, and if he has collected the gold

};

enum DIRECTION : int // Hold the possible directions the player can face

{

UP,

LEFT,

DOWN,

RIGHT,

D\_COUNT // How many possible directions

};

}

#endif // \_ROBOT\_H\_

////////////////////////////////////////////////////////////

// File: Robot.cpp

// Author: Ben Odom

// Date Created: 09/04/2015

////////////////////////////////////////////////////////////

#include "Robot.h"

// Used to avoid magic numbers

const unsigned int MAX\_ROWS = 6; // The size of the grid rows

const unsigned int MAX\_COLLUMNS = 4; // The size of the grid collumns

const unsigned int PRINT\_WIDTH = 50;

const unsigned int PRINT\_POSX = 30;

const unsigned int LAST\_SPACE\_OF\_CONSOLE = 80;

const unsigned int LAST\_LINE\_OF\_CONSOLE = 24;

namespace Dungeon

{

void Robot::Handle()

{

Clear();

System::ShowCursor(); // Shows the console cursor

System::Print("Capitalization does \*not\* matter", PRINT\_WIDTH, PRINT\_POSX, m\_iPrintLine); m\_iPrintLine += 2;

System::Print("Face: N / North | S / South | E / East | W / West", PRINT\_WIDTH, PRINT\_POSX, m\_iPrintLine++);

System::Print("Move: U / Up | M / Move | Forward | Walk", PRINT\_WIDTH, PRINT\_POSX, m\_iPrintLine++);

System::Print("Mark: Wumpus | Pit | Gold | Clear", PRINT\_WIDTH, PRINT\_POSX, m\_iPrintLine++);

System::Print("Arrow: F / Fire | A / Arrow | Shoot | Kill", PRINT\_WIDTH, PRINT\_POSX, m\_iPrintLine++);

System::Print("--------------------------------------------------", PRINT\_WIDTH, PRINT\_POSX, m\_iPrintLine); m\_iPrintLine += 2;

System::Print("Position:", PRINT\_WIDTH, PRINT\_POSX, m\_iPrintLine);

System::SetCursor(PRINT\_POSX + 10, m\_iPrintLine, 0);

printf\_s("%d : %d", m\_iPosX, m\_iPosY); m\_iPrintLine += 2;

System::Print("What will you do?", PRINT\_WIDTH, PRINT\_POSX, m\_iPrintLine); m\_iPrintLine += 2;

System::Print(">> ", PRINT\_WIDTH, PRINT\_POSX, m\_iPrintLine); m\_iPrintLine += 2;

char cBuffer[256]; // Will hold user input to be parsed later

System::ShowCursor();

std::cin.getline(cBuffer, 256);

System::ToLower(cBuffer, 256); // Will convert the input string to all lower case letters

if (!strcmp(cBuffer, "n") || !strcmp(cBuffer, "north"))

m\_iFacing = DIRECTION::UP;

if (!strcmp(cBuffer, "w") || !strcmp(cBuffer, "west"))

m\_iFacing = DIRECTION::LEFT;

if (!strcmp(cBuffer, "s") || !strcmp(cBuffer, "south"))

m\_iFacing = DIRECTION::DOWN;

if (!strcmp(cBuffer, "e") || !strcmp(cBuffer, "east"))

m\_iFacing = DIRECTION::RIGHT;

bool bMoved = false; // If the player moves, this value will be set to true and the screen will be redrawn

if (!strcmp(cBuffer, "u") || !strcmp(cBuffer, "up") ||

!strcmp(cBuffer, "m") || !strcmp(cBuffer, "move") ||

!strcmp(cBuffer, "forward") ||

!strcmp(cBuffer, "walk"))

{

System::SetCursor(m\_iPosX, m\_iPosY, 0); std::cout << m\_oGrid.m\_vcGrid[m\_iPosX][m\_iPosY];

switch (m\_iFacing)

{

case DIRECTION::UP: --m\_iPosY; break;

case DIRECTION::LEFT: --m\_iPosX; break;

case DIRECTION::DOWN: ++m\_iPosY; break;

case DIRECTION::RIGHT: ++m\_iPosX; break;

}

bMoved = true;

Collision();

}

char cTemp = NULL; // Will hold the tile value the user wants to mark as questionable

if (!strcmp(cBuffer, "wumpus"))

cTemp = TILE::WUMPUS\_Q;

if (!strcmp(cBuffer, "pit"))

cTemp = TILE::PIT\_Q;

if (!strcmp(cBuffer, "gold"))

cTemp = TILE::GOLD\_Q;

if (!strcmp(cBuffer, "clear"))

cTemp = TILE::EMPTY;

if (!strcmp(cBuffer, "f") || !strcmp(cBuffer, "fire") ||

!strcmp(cBuffer, "a") || !strcmp(cBuffer, "arrow") ||

!strcmp(cBuffer, "shoot") ||

!strcmp(cBuffer, "kill"))

FireArrow(); // Attempts to shoot the Wumpus

if (cTemp > 0) // If the user wants to mark a tile as a possible trap...

{

switch (m\_iFacing) // Check the tile they want to mark

{

case DIRECTION::UP: m\_oPercievedGrid.m\_vcGrid[m\_iPosX][m\_iPosY - 1] = cTemp; break;

case DIRECTION::LEFT: m\_oPercievedGrid.m\_vcGrid[m\_iPosX - 1][m\_iPosY] = cTemp; break;

case DIRECTION::DOWN: m\_oPercievedGrid.m\_vcGrid[m\_iPosX][m\_iPosY + 1] = cTemp; break;

case DIRECTION::RIGHT: m\_oPercievedGrid.m\_vcGrid[m\_iPosX + 1][m\_iPosY] = cTemp; break;

}

}

m\_oPercievedGrid.Draw();

Draw();

if (bMoved && !m\_bHasGold)

RecieveSignal();

}

void Robot::RecieveSignal()

{

bool bMustPause = false; // This will be set to true if the user is next to a trap

// Check for Wumpus on all 4 directions

if ((m\_oGrid.m\_vcGrid[m\_iPosX + 1][m\_iPosY] == TILE::WUMPUS) ||

(m\_oGrid.m\_vcGrid[m\_iPosX - 1][m\_iPosY] == TILE::WUMPUS) ||

(m\_oGrid.m\_vcGrid[m\_iPosX][m\_iPosY + 1] == TILE::WUMPUS) ||

(m\_oGrid.m\_vcGrid[m\_iPosX][m\_iPosY - 1] == TILE::WUMPUS))

{

bMustPause = true;

System::Print("The Robot smells an awful stench...", PRINT\_WIDTH, PRINT\_POSX, m\_iPrintLine++);

}

// Check for Gold on all 4 directions

if ((m\_oGrid.m\_vcGrid[m\_iPosX + 1][m\_iPosY] == TILE::GOLD) ||

(m\_oGrid.m\_vcGrid[m\_iPosX - 1][m\_iPosY] == TILE::GOLD) ||

(m\_oGrid.m\_vcGrid[m\_iPosX][m\_iPosY + 1] == TILE::GOLD) ||

(m\_oGrid.m\_vcGrid[m\_iPosX][m\_iPosY - 1] == TILE::GOLD))

{

bMustPause = true;

System::Print("The Robot sees a glitter...", PRINT\_WIDTH, PRINT\_POSX, m\_iPrintLine++);

}

// Check for Pit on all 4 directions

if ((m\_oGrid.m\_vcGrid[m\_iPosX + 1][m\_iPosY] == TILE::PIT) ||

(m\_oGrid.m\_vcGrid[m\_iPosX - 1][m\_iPosY] == TILE::PIT) ||

(m\_oGrid.m\_vcGrid[m\_iPosX][m\_iPosY + 1] == TILE::PIT) ||

(m\_oGrid.m\_vcGrid[m\_iPosX][m\_iPosY - 1] == TILE::PIT))

{

bMustPause = true;

System::Print("The Robot feels a breeze...", PRINT\_WIDTH, PRINT\_POSX, m\_iPrintLine++);

}

if (bMustPause)

{

System::Pause();

Clear();

}

}

bool Robot::Collision() // Makes sure the players 'm\_iPosX' and 'm\_iPosY' are valid, and check for traps

{

// Makes sure the player doesn't go off the grid on the left side

if (m\_iPosX < 1)

m\_iPosX = 1;

if (m\_iPosY < 1)

m\_iPosY = 1;

// Makes sure the player doesn't go off the grid on the right side

if (m\_iPosX > MAX\_ROWS)

m\_iPosX = MAX\_ROWS;

if (m\_iPosY > MAX\_COLLUMNS)

m\_iPosY = MAX\_COLLUMNS;

if (m\_iFacing < 0)

m\_iFacing = 3;

if (m\_iFacing > 3)

m\_iFacing = 0;

// If the player ran into the Wumpus

if (m\_oGrid.m\_vcGrid[m\_iPosX][m\_iPosY] == TILE::WUMPUS)

{

Clear();

m\_bIsAlive = false;

System::Print("You have been gobbled up by the Wumpus!", PRINT\_WIDTH, PRINT\_POSX, 0);

m\_oGrid.Draw();

System::Pause();

return true;

}

// If the player fell into a Pit

else if (m\_oGrid.m\_vcGrid[m\_iPosX][m\_iPosY] == TILE::PIT)

{

Clear();

m\_bIsAlive = false;

System::Print("You have fallen to your death!", PRINT\_WIDTH, PRINT\_POSX, 0);

m\_oGrid.Draw();

System::Pause();

return true;

}

// If the player has found the gold

else if (m\_oGrid.m\_vcGrid[m\_iPosX][m\_iPosY] == TILE::GOLD)

{

Clear();

m\_bHasGold = true;

m\_oGrid.m\_vcGrid[m\_iPosX][m\_iPosY] = TILE::EMPTY;

System::Print("You got the gold, now get out!", PRINT\_WIDTH, PRINT\_POSX, 0);

m\_oPercievedGrid.Draw();

Draw();

System::Pause();

system("cls");

return true;

}

// If the player has returned to the starting position with the gold

if ((m\_iPosX == 1 && m\_iPosY == 1) && (m\_bHasGold))

{

Clear();

m\_bIsAlive = false;

System::Print("You escaped with the gold!", PRINT\_WIDTH, PRINT\_POSX, 0);

m\_oGrid.Draw();

Draw();

System::Pause();

return true;

}

return false;

}

void Robot::Draw() // Draws the player in the grid

{

System::SetCursor(m\_iPosX, m\_iPosY, 0);

switch (m\_iFacing)

{

case DIRECTION::UP: std::cout << "^"; break;

case DIRECTION::LEFT: std::cout << "<"; break;

case DIRECTION::DOWN: std::cout << "v"; break;

case DIRECTION::RIGHT: std::cout << ">"; break;

}

}

void Robot::FireArrow()

{

if (m\_iArrowCount) // If the player still has an arrow

{

switch (m\_iFacing) // Check each direction

{

case DIRECTION::UP:

{

if (m\_oGrid.m\_vcGrid[m\_iPosX][m\_iPosY - 1] == TILE::WUMPUS)

KillWumpus(m\_iPosX, m\_iPosY - 1);

else

FailShot();

break;

}

case DIRECTION::LEFT:

{

if (m\_oGrid.m\_vcGrid[m\_iPosX - 1][m\_iPosY] == TILE::WUMPUS)

KillWumpus(m\_iPosX - 1, m\_iPosY);

else

FailShot();

break;

}

case DIRECTION::DOWN:

{

if (m\_oGrid.m\_vcGrid[m\_iPosX][m\_iPosY + 1] == TILE::WUMPUS)

KillWumpus(m\_iPosX, m\_iPosY + 1);

else

FailShot();

break;

}

case DIRECTION::RIGHT:

{

if (m\_oGrid.m\_vcGrid[m\_iPosX + 1][m\_iPosY] == TILE::WUMPUS)

KillWumpus(m\_iPosX + 1, m\_iPosY);

else

FailShot();

break;

}

}

}

else // When the player has no arrow left

{

System::Print("Your Robot has no arrows to fire...", PRINT\_WIDTH, PRINT\_POSX, m\_iPrintLine++);

System::Pause();

Clear();

}

}

void Robot::KillWumpus(int a\_iPosX, int a\_iPosY)

{

m\_oGrid.m\_vcGrid[a\_iPosX][a\_iPosY] = TILE::EMPTY;

--m\_iArrowCount;

System::Print("The Wumpus was slain!", PRINT\_WIDTH, PRINT\_POSX, m\_iPrintLine++);

System::Pause();

Clear();

}

void Robot::FailShot()

{

System::Print("Nothing happened.", 0, 40, m\_iPrintLine++);

System::Print("You are out of arrows...", PRINT\_WIDTH, PRINT\_POSX, m\_iPrintLine++);

System::Pause();

Clear();

}

void Robot::GetPos(int &a\_iPosX, int &a\_iPosY, int &a\_iFacing) // Unused currently

{

a\_iPosX = m\_iPosX;

a\_iPosY = m\_iPosY;

a\_iFacing = m\_iFacing;

}

bool Robot::IsAlive() // Returns whether the player is currently alive or not

{

return m\_bIsAlive;

}

void Robot::OnKeyPress(int a\_iKey)

{

bool bMoved = false; // Will determine if the player moves this step or not

char cTemp = NULL; // Will hold the tile value the user wants to mark as questionable

Clear();

System::Print("Turn : <- or ->", PRINT\_WIDTH, PRINT\_POSX, m\_iPrintLine++);

System::Print("Move Forward: Up Arrow", PRINT\_WIDTH, PRINT\_POSX, m\_iPrintLine++);

System::Print("Mark Tile : W - Wumpus | P - Pit | G - Gold", PRINT\_WIDTH, PRINT\_POSX, m\_iPrintLine++);

System::Print("Clear Mark : X - Clear", PRINT\_WIDTH, PRINT\_POSX, m\_iPrintLine++);

System::Print("Shoot Arrow : F - Fire", PRINT\_WIDTH, PRINT\_POSX, m\_iPrintLine); m\_iPrintLine += 2;

System::Print("Shoot Arrow : F - Fire", PRINT\_WIDTH, PRINT\_POSX, m\_iPrintLine); m\_iPrintLine += 2;

switch (a\_iKey)

{

case VK\_LEFT: ++m\_iFacing; break;

case VK\_RIGHT: --m\_iFacing; break;

case VK\_UP:

{

if (m\_iFacing == DIRECTION::UP)

--m\_iPosY;

if (m\_iFacing == DIRECTION::LEFT)

--m\_iPosX;

if (m\_iFacing == DIRECTION::DOWN)

++m\_iPosY;

if (m\_iFacing == DIRECTION::RIGHT)

++m\_iPosX;

bMoved = true;

break;

}

case 0x57: cTemp = TILE::WUMPUS\_Q; break; // W Key

case 0x50: cTemp = TILE::PIT\_Q; break; // P Key

case 0x47: cTemp = TILE::GOLD\_Q; break; // G Key

case 0x58: cTemp = TILE::EMPTY; break; // X Key

case 0x46: FireArrow(); break; // F Key

default: break;

}

if (cTemp > 0)

{

switch (m\_iFacing)

{

case DIRECTION::UP: m\_oPercievedGrid.m\_vcGrid[m\_iPosX][m\_iPosY - 1] = cTemp; break;

case DIRECTION::LEFT: m\_oPercievedGrid.m\_vcGrid[m\_iPosX - 1][m\_iPosY] = cTemp; break;

case DIRECTION::DOWN: m\_oPercievedGrid.m\_vcGrid[m\_iPosX][m\_iPosY + 1] = cTemp; break;

case DIRECTION::RIGHT: m\_oPercievedGrid.m\_vcGrid[m\_iPosX + 1][m\_iPosY] = cTemp; break;

}

}

if (!Collision())

{

m\_oPercievedGrid.Draw();

Draw();

if (bMoved && !m\_bHasGold)

RecieveSignal();

}

}

void Robot::Clear()

{

System::Clear(PRINT\_POSX, 0, LAST\_LINE\_OF\_CONSOLE);

m\_iPrintLine = 0; // Reset where the which line the console prints to

}

Robot::Robot()

{

// Empty Constructor

}

Robot::Robot(const int a\_icPosX, const int a\_icPosY, const int a\_icFacing)

{

m\_iPosX = a\_icPosX;

m\_iPosY = a\_icPosY;

m\_iFacing = a\_icFacing;

m\_iArrowCount = 1;

m\_bIsAlive = true;

m\_bHasGold = false;

Clear();

m\_oGrid.Fill(MAX\_ROWS, MAX\_COLLUMNS);

m\_oPercievedGrid.Fill(MAX\_ROWS, MAX\_COLLUMNS);

m\_oGrid.Trap(MAX\_ROWS, MAX\_COLLUMNS);

m\_oPercievedGrid.Draw();

Draw();

}

Robot::~Robot()

{

// Nothing Yet

}

}

////////////////////////////////////////////////////////////

// File: Grid.h

// Author: Ben Odom

// Date Created: 09/04/2015

// Brief: This file contains the 'Grid' struct which

// is part of the namespace 'Dungeon'. I decided to

// make this a struct instead of a class because

// even though it has one function, all members of

// it should always be public to it's parent without

// any worry of corruption or anything since only

// the parent has access to it, and it doesn't modify

// it's own values after initialization. This may be

// bad programming, but that's my reasoning...

////////////////////////////////////////////////////////////

#ifndef \_GRID\_H\_

#define \_GRID\_H\_

#include "System.h"

namespace Dungeon

{

struct Grid

{

std::vector<std::vector<char>> m\_vcGrid; // Heard you like grids...Just hold the actual grid values. See 'enum TILE' for possible tile values

void Draw(); // Draw the grid on the top left of the screen

void Fill(int a\_iRow, int a\_iCollumn); // Fills the grid with empty tiles and surrounding walls

void Trap(int a\_iRow, int a\_iCollumn); // This will add traps to the cave

};

enum TILE // Holds all the possible tile combinations so they can be 'switch()'ed to easily

{

WALL,

EMPTY,

WUMPUS,

WUMPUS\_Q,

PIT,

PIT\_Q,

GOLD,

GOLD\_Q,

T\_COUNT // How many possible tiles

};

}

#endif // \_GRID\_H\_

////////////////////////////////////////////////////////////

// File: Grid.h

// Author: Ben Odom

// Date Created: 09/04/2015

////////////////////////////////////////////////////////////

#include "Grid.h"

// To avoid using magic numbers, all constants are represented as such

// These are just the ASCII values of each tile set. A '\_Q' is the user marked tiles; As in the user can set a tile to these values

// in order to remember that they think something might be there. So a 'WUMPUS\_Q' means the user thinks a Wumpus is in that tile

const unsigned char WALL\_ASCII = 219;

const unsigned char EMPTY\_ASCII = 176;

const unsigned char WUMPUS\_ASCII = 'W';

const unsigned char WUMPUS\_Q\_ASCII = 'w';

const unsigned char PIT\_ASCII = ' ';

const unsigned char PIT\_Q\_ASCII = ' ';

const unsigned char GOLD\_ASCII = 233;

const unsigned char GOLD\_Q\_ASCII = 233;

// The 'PIT\_RATE' represents how many pits will appear. It's calculated as:

// (NumOfRowsInGrid \* NumOfCollumnsInGrid) / PIT\_RATE

const unsigned int PIT\_RATE = 9;

// This is the direct number of Wumpus in the cave.

const unsigned int WUMPUS\_RATE = 1;

namespace Dungeon

{

void Grid::Draw() // Draws the grid in ASCII at the top left of the screen

{

for (int i = 0; i < m\_vcGrid.size(); ++i)

{

for (int j = 0; j < m\_vcGrid[i].size(); ++j)

{

System::SetCursor(i, j, 0); // Set the console cursor at (i, j) and set it's visibility to 0

switch (m\_vcGrid[i][j]) // Just switches to evaluate 'm\_vcGrid[i][j]' using the enum 'TILE'

{

case TILE::EMPTY: std::cout << EMPTY\_ASCII; break;

case TILE::WALL: std::cout << WALL\_ASCII; break;

case TILE::WUMPUS: std::cout << WUMPUS\_ASCII; break;

case TILE::WUMPUS\_Q: std::cout << WUMPUS\_Q\_ASCII; break;

case TILE::PIT: std::cout << PIT\_ASCII; break;

case TILE::PIT\_Q: std::cout << PIT\_Q\_ASCII; break;

case TILE::GOLD: std::cout << GOLD\_ASCII; break;

case TILE::GOLD\_Q: std::cout << GOLD\_Q\_ASCII; break;

}

}

}

}

void Grid::Fill(int a\_iRow, int a\_iCollumn) // Fills the grid with empty tiles and surrounding walls

{

// In order to '.push\_back' the 'm\_vcGrid' with an entire row, I use this variable to hold 'a\_iCollumn' collumns and then push it to the row

std::vector<char> vcTemp;

for (int i = 0; i <= a\_iRow + 1; ++i)

{

for (int j = 0; j <= a\_iCollumn + 1; ++j)

{

// If it's a surrounding tile, make it a 'TILE::WALL', if it's inside make it 'TILE::EMPTY'

if (((i == 0) || (i == a\_iRow + 1)) ||

((j == 0) || (j == a\_iCollumn + 1)))

vcTemp.push\_back(TILE::WALL);

else

vcTemp.push\_back(TILE::EMPTY);

}

m\_vcGrid.push\_back(vcTemp); // Push back the full row of 'a\_iCollumn' collumns

vcTemp.clear(); // Allow 'vcTemp' to get more values by erasing it's contents

}

}

void Grid::Trap(int a\_iRow, int a\_iCollumn) // This will add traps to the cave

{

struct CanTrap

{

int iRow, iCollumn;

};

std::vector<CanTrap> voCanTrap; // This struct will hold the tile which can currently randomly be used as a trap

for (int i = 1; i <= a\_iRow; ++i)

{

for (int j = 1; j <= a\_iCollumn; ++j)

{

if (((i != 1) || (j != 1)) &&

((i != 1) || (j != 2)) &&

((i != 2) || (j != 1)))

voCanTrap.push\_back({ i, j }); // Fill the vector with acceptable values in an attempt to make a solvable cave layout

}

}

int iTemp; // Hold the randomly generated value

for (int i = 0; i < (a\_iRow\*a\_iCollumn) / PIT\_RATE; ++i) // See 'PIT\_RATE' comment

{

iTemp = rand() % voCanTrap.size();

m\_vcGrid[voCanTrap[iTemp].iRow][voCanTrap[iTemp].iCollumn] = TILE::PIT;

voCanTrap.erase(voCanTrap.begin() + iTemp); // Removed the randomly generated tile position from the 'voCanTrap' vector since it now has a pit in it

}

for (int i = 0; i < WUMPUS\_RATE; ++i)

{

iTemp = rand() % voCanTrap.size();

m\_vcGrid[voCanTrap[iTemp].iRow][voCanTrap[iTemp].iCollumn] = TILE::WUMPUS;

voCanTrap.erase(voCanTrap.begin() + iTemp); // Remove the Wumpus from the randomly generated tile position from the 'voCanTrap' vector

}

iTemp = rand() % voCanTrap.size();

m\_vcGrid[voCanTrap[iTemp].iRow][voCanTrap[iTemp].iCollumn] = TILE::GOLD; // Throw some gold in there

}

}

////////////////////////////////////////////////////////////

// File: Dungeon.h

// Author: Ben Odom

// Date Created: 09/04/2015

// Brief: This file contains the 'System' namespace. This

// namespace will have functions to make printing

// to the console simple and with high functionality

// rather than a simple per line print like

// std::cout provides alone. It will also include

// random number and input functionality.

////////////////////////////////////////////////////////////

#ifndef \_SYSTEM\_H\_

#define \_SYSTEM\_H\_

#include <vector>

#include <iostream>

#include <iomanip>

#include <string>

#include <windows.h>

#include <stdlib.h>

#include <time.h>

namespace System

{

enum COLOR : int // Holds the color values that windows uses for the console colors for font and background

{

BLACK,

BLUE,

GREEN,

CYAN,

RED,

MAGENTA,

BROWN,

LIGHT\_GRAY,

DARK\_GRAY,

LIGHT\_BLUE,

LIGHT\_GREEN,

LIGHT\_CYAN,

LIGHT\_RED,

LIGHT\_MAGENTA,

YELLOW,

WHITE,

C\_COUNT

};

void Print(const int ac\_iPosX, const int ac\_iPosY, const char \*ac\_pcFormat, ...); // Broken function that attempted to set the console print line and then pass the remaining arguments to 'printf\_s'

void Print(const std::string ac\_sTextToPrint, const int ac\_iLength, const int ac\_iPosX = -1, const int ac\_iPosY = -1);

void Clear(const int ac\_iPosX, const int ac\_iPosY, const int ac\_iLineLength);

void ToLower(char \* a\_pcUppercaseWord, const int ac\_iSize);

void SetCursor(short a\_iPosX, short a\_iPosY, bool a\_bShowCursor);

void ShowCursor();

void HideCursor();

void Pause();

// Uses the math function 'floor' after adding 0.5f to ensure numbers are rounded upwards

int Round(float a\_fNum);

// Will return a random number as either a float or an int based on which type is passed

// Expects a starting number and an ending number such as 1, 10 which gets a number between 1 and 10

// Make sure to put an 'f' at the end of number constants to define it as float such as '1.0f'

int Random(int a\_iMin, int a\_iMax);

float Random(float a\_fMin, float a\_fMax);

void Init(const int a\_ciColor = LIGHT\_GRAY);

void Quit();

}

#endif // \_SYSTEM\_H\_

////////////////////////////////////////////////////////////

// File: Dungeon.cpp

// Author: Ben Odom

// Date Created: 09/04/2015

////////////////////////////////////////////////////////////

#include "System.h"

const unsigned int LAST\_SPACE\_OF\_CONSOLE = 80;

const unsigned int LAST\_LINE\_OF\_CONSOLE = 24;

namespace System

{

HANDLE hStdout;

CONSOLE\_CURSOR\_INFO hCursorInfo;

CONSOLE\_SCREEN\_BUFFER\_INFO hBufferInfo;

void Print(const int ac\_iPosX, const int ac\_iPosY, const char \*ac\_pcFormat, ...) // Broken function that attempted to set the console print line and then pass the remaining arguments to 'printf\_s'

{

va\_list arguments;

va\_start(arguments, ac\_pcFormat);

SetCursor(ac\_iPosX, ac\_iPosY, hCursorInfo.bVisible);

printf\_s(ac\_pcFormat, arguments);

}

void Print(std::string ac\_sTextToPrint, const int ac\_iLength, const int ac\_iPosX, const int ac\_iPosY)

{

SetCursor(ac\_iPosX, ac\_iPosY, hCursorInfo.bVisible);

if (ac\_sTextToPrint.length() > ac\_iLength)

{

int i = 0; // Will keep track of where in the 'substr' we've gone thus far

std::vector<std::string> vsTextToPrint;

bool bLoop = true;

while (bLoop)

{

if (i + ac\_iLength >= ac\_sTextToPrint.length())

bLoop = false;

std::string sTemp = ac\_sTextToPrint.substr(i, ac\_iLength); // Holds the 'substr' of 'ac\_sTextToPrint' that we're cutting into pieces.

vsTextToPrint.push\_back(ac\_sTextToPrint.substr(i, i + sTemp.rfind(" "))); i += sTemp.rfind(" ") + 1;

}

for (i = 0; i < vsTextToPrint.size(); ++i)

{

SetCursor(ac\_iPosX, ac\_iPosY + i, hCursorInfo.bVisible);

std::cout << vsTextToPrint[i];

}

}

else

std::cout << ac\_sTextToPrint;

}

void Clear(const int ac\_iPosX, const int ac\_iPosY, const int ac\_iLineLength)

{

std::string sFill = "";

for (int i = 0; i < LAST\_SPACE\_OF\_CONSOLE - ac\_iPosX; ++i)

sFill += " ";

for (int i = 0; i < LAST\_LINE\_OF\_CONSOLE - ac\_iPosY; ++i)

{

SetCursor(ac\_iPosX, ac\_iPosY + i, 0);

std::cout << sFill;

}

}

void ToLower(char \* a\_pcUppercaseWord, const int ac\_iSize)

{

for (int i = 0; i < ac\_iSize; ++i)

{

if (a\_pcUppercaseWord[i] <= 90 && a\_pcUppercaseWord[i] >= 65)

a\_pcUppercaseWord[i] += 32;

}

}

void SetCursor(short a\_iPosX, short a\_iPosY, bool a\_bShowCursor)

{

hCursorInfo.bVisible = a\_bShowCursor;

SetConsoleCursorPosition(hStdout, { a\_iPosX, a\_iPosY });

SetConsoleCursorInfo(hStdout, &hCursorInfo);

}

void ShowCursor()

{

hCursorInfo.bVisible = true;

SetConsoleCursorInfo(hStdout, &hCursorInfo);

}

void HideCursor()

{

hCursorInfo.bVisible = false;

SetConsoleCursorInfo(hStdout, &hCursorInfo);

}

void Pause()

{

SetCursor(0, LAST\_LINE\_OF\_CONSOLE, 0);

printf\_s("<Press Enter To Continue>");

Sleep(300);

while (!(GetAsyncKeyState(VK\_RETURN) & 0x8000)) {}

}

int Round(float a\_fNum)

{

return floor(a\_fNum + 0.5f);

}

int Random(int a\_iMin, int a\_iMax)

{

return a\_iMin + rand() % ((a\_iMax + 1) - a\_iMin);

}

float Random(float a\_fMin, float a\_fMax)

{

return a\_fMin + (float)rand() / ((float)RAND\_MAX / (a\_fMax - a\_fMin));

}

void Init(int a\_ciColor)

{

srand(time(NULL));

hStdout = GetStdHandle(STD\_OUTPUT\_HANDLE);

SetConsoleOutputCP(CP\_UTF8);

SetConsoleTextAttribute(hStdout, a\_ciColor);

hCursorInfo.dwSize = 10;

hCursorInfo.bVisible = 1;

SetConsoleCursorInfo(hStdout, &hCursorInfo);

}

void Quit()

{

}

}