

WonderMaize

5/11/16

Team D

Matthew Persing

Alec Tiefenthal

Jack Porter

Contents

Introduction.....	3
WonderMaize User Guide	4
Objective	4
Basic Controls	4
Changing Maze Size.....	4
Gameplay	5
System Requirements.....	6
Operating System Compatibility.....	6
Hardware Compatibility	6
Installation Instructions	6
Maintenance Guide	8
Tools.....	8
Troubleshooting	8
Software Requirements Specification (SRS)	9
Software and Architecture Design Specification (SADS)	10
Test Plan/Strategy	12

Introduction

WonderMaize is a maze game for maze enthusiasts. Great for all ages so long as the user thinks they can navigate a maze. WonderMaize is restricted to operating on 64-bit Windows platforms. This document contains an in depth user guide that explains how to start and play WonderMaize as well as an installation and maintenance guide with more detailed information on how to install and troubleshoot the system. This document also contains requirements and design documents alongside a detailed test plan.

WonderMaize User Guide

Objective

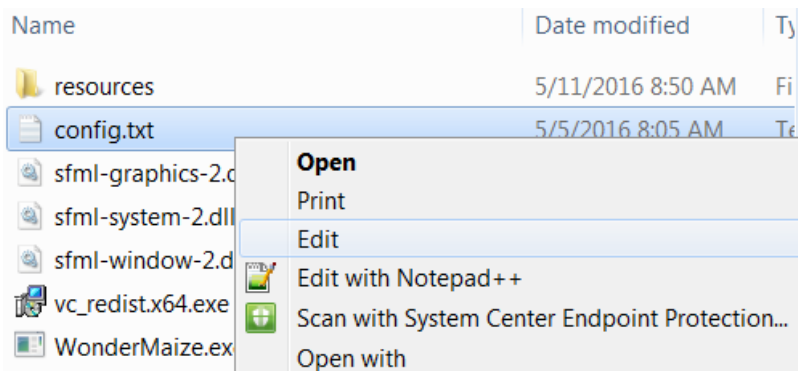
The objective of the WonderMaize game is to navigate through the randomly generated maze to find 5 different mysterious objects that are hidden throughout.

Basic Controls

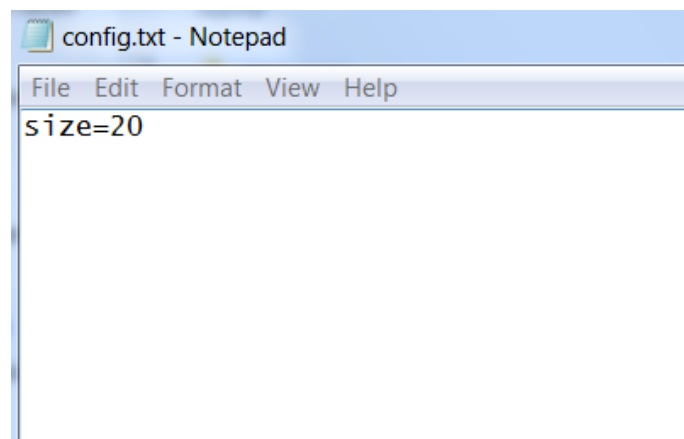
Key	Action
W	Move forward
A	Turn left
S	Move backwards
D	Turn right
Esc	Quit the game

Changing Maze Size

To change the maze size, open the config.txt file in folder with the game executable in a text editor. Notepad will be used for this example.



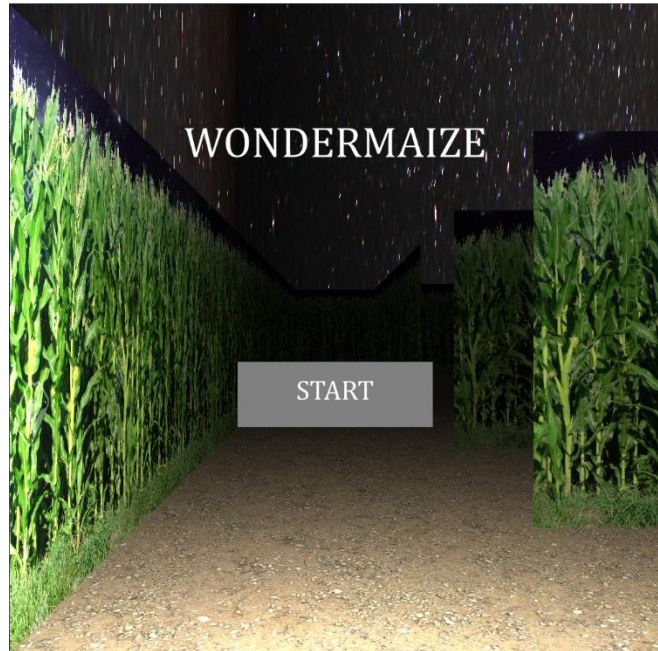
Once open in the text editor, simply change the number following the equal sign and save the file. Mazes with a size lower than 7 will not work properly and mazes of very large sizes are much more computer intensive. For a good combination of challenge and performance, sizes of 20 and 30 are recommended. For the change to be reflected in the game, the game should be restarted.



Gameplay

Upon opening the game, a box will pop up to start the game. Click in the box to begin playing the game.

Once in the maze, the goal is to find all 5 hidden objects. The objects that are hidden in the maze are a Cow, Gourd, Sphere, Teapot, and Teddy Bear. Initially all objects are colored red. To actually find an object, run close to it and a swirly effect will appear on the screen and the object will change color to yellow.



After finding all of the models, a screen showing that the player has won will appear. To restart the game, press the “Restart” button in the middle of the screen. After pressing the “Restart” button, a new maze will be randomly generated and the objects to find will be randomly scattered throughout the maze.



System Requirements

Operating System Compatibility

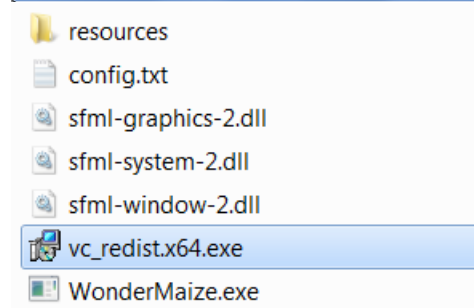
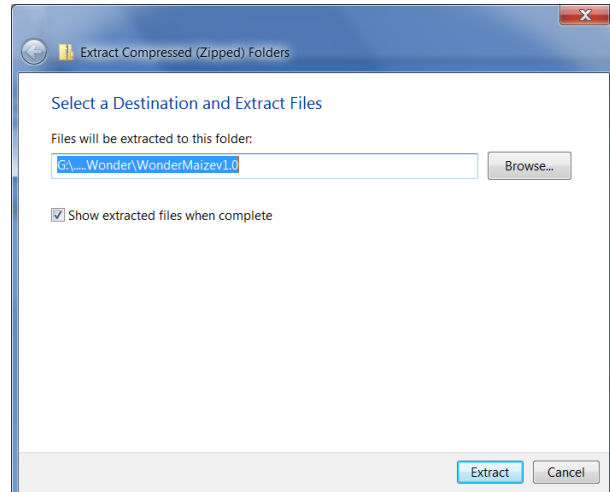
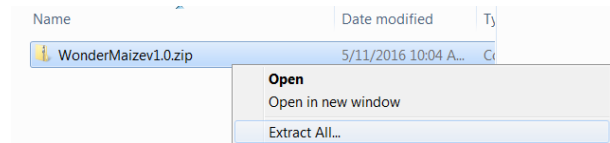
Windows 7 (32-bit)	Not Supported
Windows 7 (64-bit)	Supported
Windows 8 (32-bit)	Not Supported
Windows 8 (64-bit)	Not Supported
Windows 10 (32-bit)	Not supported
Windows 10 (64-bit)	Supported
Mac OSX	Not supported
Linux	Not supported

Hardware Compatibility

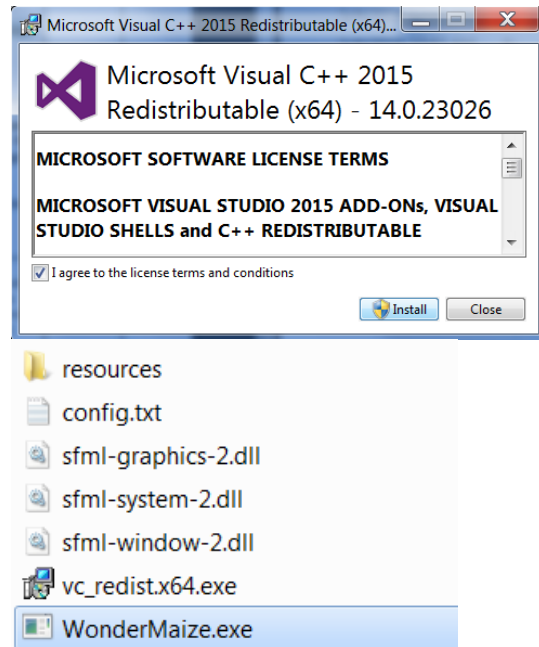
A graphics card with OpenGL 3.3 or newer is required for proper operation.

Installation Instructions

1. Right click the .zip file to open the context menu and left click Extract All. If you have another tool that extracts .zip files such as WinRAR, it is also acceptable to use that tool and you can skip to step 3.
2. Choose a destination folder for the game, then click extract. It is okay to leave the destination as the default.
3. Run the vc_redist.x64.exe installer to install the Visual Studio 2015 Runtime. This is required to run the WonderMaize.exe



4. Read the terms and conditions to install the Visual Studio 2015 Runtime, click Install, then follow the onscreen instructions. If an error message pops up indicating the Visual Studio 2015 Runtime has already been installed, you should be able to run the game. Simply continue to the next step.
5. Double click on WonderMaize.exe to open the game. The previous steps do not need to be repeated in the future.



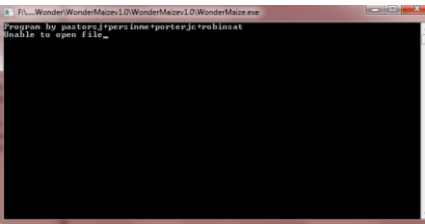
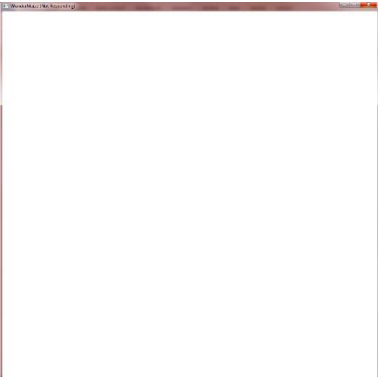
Maintenance Guide

Tools

- Blender
- Glew
- GitLab
- Microsoft Visual Studio (2013 or newer)
- OpenGL
- SFML

The tools used created an environment in which 3D graphics software could be written and ran. GitLab was used to create a safe development environment for testing and allowing for continuous integration. The continuous integration was set up on Rose-Hulman's ADA GitLab server, with the .gitlab-ci.yml file in the root directory of the project controlling the automated build and test process. The .zip file ready for distribution to users is found in the repository at /artifacts/WonderMaizev1.0.zip. Blender was used to work on models and get the models' surface normals to be calculated in an easy to access way.

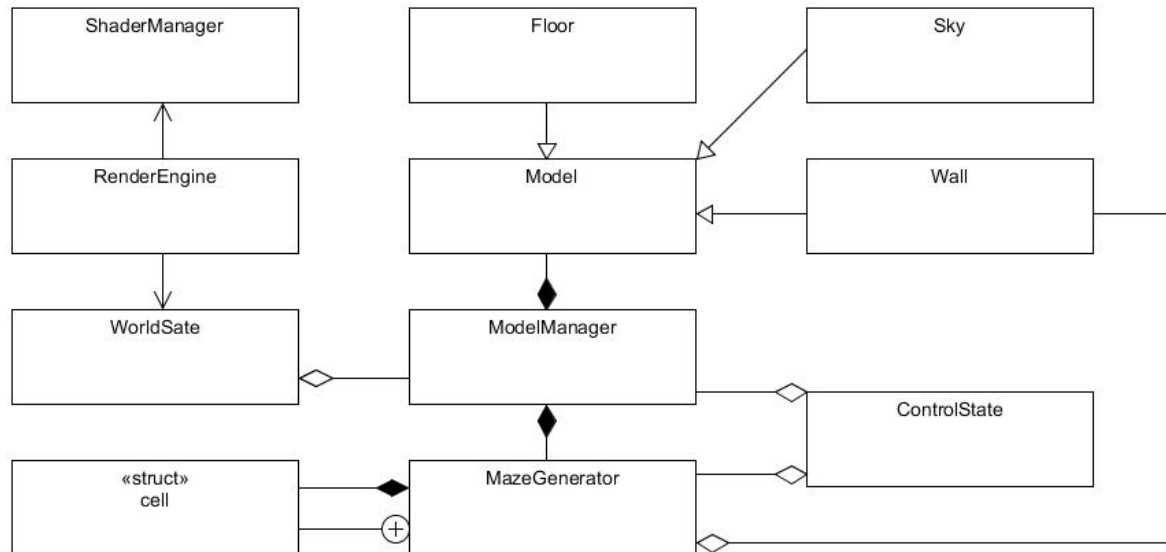
Troubleshooting

Program not start when launched		If this fails try reinstalling and making sure all files are arranged appropriately. If this does not help contact the developers at persinme@rose-hulman.edu
Program does not run		Check software compatibility to ensure the software will run on your system. Ensure files are as they were after the initial installation.
Receive unable to open file error.		Ensure the config.txt file is located in the directory that the game is located in. For the release build look in the x64/Release folder. If attempting to run from Visual Studio look in the WunderMaze folder.
White screen appears and program is not responding.		Check that the maze size in the config.txt file is set to at least 7. If the size is set to less than 7, change it to 7 or greater.

Software Requirements Specification (SRS)

The application should present the user with a randomized 3D maze. The controls for the game should be intuitive, requiring little or no instruction to users who have played any type of 3D game on a computer before. The maze should have some kind of end condition, like finding various objects hidden within it. The maze should be easily repeatable and customizable, with the customization made possible with at least some kind of configuration file. The program should not encounter any errors caused on its own accord. In the event that there is an error, fatal or otherwise, due to an issue outside the scope of the application, the error should *never* affect the running of a new instance of the application (if the application is closed and reopened, it should work regardless of what happened in the previous session).

Software and Architecture Design Specification (SADS)



The above figure is a UML diagram of our system. The following is an explanation of the different parts of the diagram. Main is not shown, but it loops until the game is won or closed, using SFML to display the window and get user input. NOTE: Many of these classes utilize external libraries such as glem and OpenGL which are not shown here.

- **ShaderManager** – This is the class that manages the various shaders used in our program. This class is responsible for the actual logic behind compiling and loading the shaders.
- **RenderEngine** – This is the class that handles the various aspects of actually rendering a scene for the user. This class contains the majority of the OpenGL calls in our system.
- **WorldState** – This class is responsible for holding the ModelManager of the world, as well as the ControlState for the world. In addition to storing those objects, WorldState holds several important pieces of information which required to know what the world “looks” like, such as where the camera is, and which direction is up. The logic to perform a single timestep is also contained within this class.
- **ModelManager** – This class is in charge of initializing the Models that are needed for our program, storing those Models, and getting various pieces of information about those models that are relevant to OpenGL. This class is also where the MazeGenerator is initially called to create a new maze.
- **MazeGenerator** – Holds all of the logic for actually generating a maze. Also declares the cell struct.
 - **cell** – A simple data struct that assists in representing the layout of the maze.
- **ControlState** – Holds information concerning movement in the scene. Contains logic to move in a single step.
- **Model** – The base class for a Model. This class contains the logic for utilizing an objLoader (external library) to load in information concerning an object model file (.obj). This includes the location of the vertices in the model, the normal vectors of the model, as well as texture information for the model. Some matrices are also stored in this class which are used for giving the model a location, and a rotation. In addition to finding and

storing Model information, this class contains the logic to draw a model in the scene using OpenGL.

- The following are special types of Models. Although the standard model could be used for this task and not require hard coding of these models, doing it this way allows for significantly lower runtime costs when using OpenGL.
- Floor –There is no .obj file associated with Floor, the attributes are coded in such that there appears to be a floor in every cell of the maze, but otherwise uses the logic from the Model class.
- Sky –There is no .obj file associated with Sky, the attributes are coded in such that there appears to be four sky objects in the scene, but otherwise uses the logic from the Model class.
- Wall –There is no .obj file associated with Wall, the attributes are coded in such that there appears to be a wall in every spot in the maze where there should be a wall which is why it has knowledge of the MazeGenerator being used, but otherwise uses the logic from the Model class.

Test Plan/Strategy

The WonderMaize game's initial manifestation was for demonstrating Computer Graphics, and therefore a large percentage of the code is for front end graphics calculations and execution. To reduce potential errors in the WonderMaize, a test harness has been created in MSTest to test the backend code for the project. The front end code consists of calls to SFML and OpenGL and therefore is not practical to test. The backend classes that are tested in the test harness are MazeGenerator, ControlState, and Model. These are the primary classes that don't have interaction with OpenGL and are therefore able to be tested decently well.

The MazeGenerator tests are largely characterization tests that test the properties of the MazeGenerator. The tests included for MazeGenerator are:

- TestInitialization: Basic test to make sure the MazeGenerator loads.
- TestMazeSizeGetters: Make sure that the getXSize() and getYSize() methods reflect the arguments that makeMaze was called with.
- TestMazeBorders: Make sure that the maze has all outer walls created in a simple square maze.
- TestNonsquareMazeBorders: Make sure that the maze has all outer walls created in a rectangular maze.
- TestMinMazeSize: Boundary value. Make sure that the maze properly generates at the smallest allowed size.
- TestMazeXSizeTooSmall: Boundary value. Make sure that the maze is not created given too small of an X dimension.
- TestMazeYSizeTooSmall: Boundary value. Make sure that the maze is not created given too small of a Y dimension.

The tests for ControlState cover the user input side of the game's state. The tests included for ControlState are:

- TestInitialization: Basic test to make sure ControlState loads.
- TestStep: Very simple test to check that basic movement within a cell updates state correctly.
- TestStepWithTurning: A more exhaustive test that checks that steps update information correctly after repeated sequences of steps and turns.
- TestCellLocation: An exhaustive test that checks to see that the state handles exploring into cells in 3D space correctly and accurately.

The tests for Model cover the loading in of a Model, assuring that all of the values loaded are correct. The tests included for Model are:

- TestModelInitialization: Basic test to make sure the Model constructor doesn't error out.
- TestModelLoadNotFail: Tests that a Model can load from a .obj without erroring out, but does not confirm if the data loaded is accurate.
- TestModelCorrectVertexOrder: Tests to see if the vertices for faces on a test object are loaded into the Model as expected.
- TestModelCorrectNormalOrder: Tests to see if the normal for faces on a test object are loaded into the Model as expected.