View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by ScholarWorks@UARK

University of Arkansas, Fayetteville
ScholarWorks@UARK

Computer Science and Computer Engineering Undergraduate Honors Theses
Computer Science and Computer Engineering

5-2015

WIP

Armon Nayeraini
University of Arkansas, Fayetteville

Follow this and additional works at: http://scholarworks.uark.edu/csceuht

Part of the Programming Languages and Compilers Commons

Recommended Citation
Nayeraini, Armon, "WIP" (2015). *Computer Science and Computer Engineering Undergraduate Honors Theses*. 33.
http://scholarworks.uark.edu/csceuht/33

This Thesis is brought to you for free and open access by the Computer Science and Computer Engineering at ScholarWorks@UARK. It has been accepted for inclusion in Computer Science and Computer Engineering Undergraduate Honors Theses by an authorized administrator of ScholarWorks@UARK. For more information, please contact scholar@uark.edu.

WIP (Work In Progress)

An Undergraduate Honors College Thesis

in the

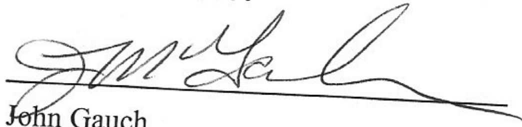
Depart of Computer Science
College of Engineering
University of Arkansas
Fayetteville, AR

by

Armon Nayeraini

This thesis is approved.

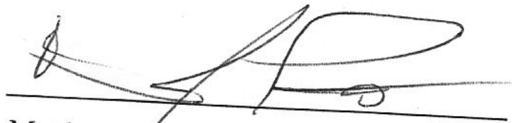
Thesis Advisor:



John Gauch

Thesis Committee:

Michael Gashler



Matthew Patitz

Abstract

Games have been shown to be capable tools in teaching. Additionally, programming can be a hard skill to learn. The objective of this research was to create a game that helps student learn coding concepts by playing a video game. The result of our work is a fully functional game that introduces the beginning concepts of programming: sequential, conditional, iterative operations.

1.0: Introduction

1.1: An Introduction To Video Games

What is a video game? One definition is “a game played by electronically manipulating images produced by a computer program on a television screen or other display screen.” But, what is a game then? An answer for that could be “a form of play or sport, especially a competitive one played according to rules and decided by skill, strength, or luck.” This is an acceptable definition. However, for the purposes of this paper, a game is simply “a set of rules that are followed.”

1.2: Video Games in Education

The video game industry has existed over three decades. Nearly as long, there has been interest by educators in the potential use of video games in education.

One early example coincides with the release of PacMan in the early eighties. The game's wild popularity amongst the youth spawned a question of whether “the magic of ‘Pac-Man- ‘cannot be bottled and unleashed in the classroom to enhance student involvement, enjoyment, and commitment” (Bowman 1982, p. 14). Indeed, this early work noted that “It (Pac-Man) is an action system where skills and challenges are progressively balanced, goals are clear, feedback is immediate and unambiguous, and relevant stimuli can be differentiated from irrelevant stimuli. Together, this combination contributes to the formation of a flow experience” (Bowman, 1982 p. 15).

Such interest continues to show itself in more recent times, as well. Kurt Squire, a well known researcher in game design for education, states that “Video games... make it possible to 'learn while doing'” (Squire, 2005). Specifically, the paper goes in depth about how various games allow players to experience and learn about specific topics they otherwise not have been able to. Full Spectrum Warrior is one such game. In this game, the player takes the role of a United States Army commander responsible for leading squads of soldiers in various conflicts. As commander, it is up to he or she to decide when, where, and how to engage. The soldiers, on the other hand, know how to fight using various tactics based on real life. The result is the player having to “build meanings on the spot as they navigate [through the game].” (Squire, 2005). Another game mentioned is Madison 2200, an urban planning simulation. “Players get a project directive from the mayor, addressed to them as city

planners, including a city budget plan and letters from concerned citizens about crime, revenue, jobs, waste, traffic, and affordable housing." (Squire, 2005). Like in Full Spectrum Warrior, the player is placed in a situation where they must derive connections and patterns to find success. "One player commented: 'I really noticed how [urban planners] have to . . . think about building things . . . like, urban planners also have to think about how the crime rate might go up, or the pollution or waste, depending on choices.'" (Squire, 2005).

Continuing, there has been a great deal of research in the potential of games as an academic tool. One experiment took place in Chile during the early two thousands. The study contained over one thousands first and second graders hailing from several elementary schools. The experiment was designed such that "[the] children's attention was focused on playing and not on learning. Learning contents were an intrinsic part of the game, so that their learning was incidental." (Rosas et all). A variety of games were developed and ran on a GameBoy like device. These games ranged from playing as Hermes, a Greek god, who had to save his friends in a temple to working as magical wizard searching in a city to strengthen her magical abilities. Behind these premised laid gameplay which was rooted in educational activities such as spelling and basic mathematics. At the end of the experiment, it was found that students whom had played the video games were more focused and interested in learning, than those who had not played.(Rosas et all).

1.3: Problem Introduction

One may argue that the field of programming is approximately seventy five years old, since this is when the first machines to run assembly began to be seen. However, the languages that one sees today give more of their heritage to the languages that arose during the late sixties and early seventies, such as C. In which case one may say the field is roughly four decades old. Regardless of which age one chooses, there has been time for several generations of programmers to arise and pass on what they learned. Still, there is still common claim learning how to program is difficult.

Indeed, one paper, (Jenkins, 2002), lists a number of potential reasons on why programming is difficult to learn. The first reason focuses on student aptitude. This “question of aptitude” asks if a student's ability to comprehend mathematics is a factor. Another possibility is that professors are unable to convey the information in a way that all students can understand. Finally, it could be that the first programming language that students are introduced is not intuitive to a beginner and acts as a major roadblock to learning.

Another paper, (Gomes and Mendes, 2007), also discusses the potential difficulties encountered while learning to program. One reason the authors mention is that teaching focuses too much on learning the language syntax and not on how programmers should break down and solve the problem. Another issue discussed is that coding requires relating experiences and knowledge that a new student does

not yet have. Hence there is a steep unavoidable learning curve students must overcome when learning to program.

Based on the information above, we decided to develop a video game that introduces basic coding concepts without focusing on syntax or programming ideas. Our hope is that this will be a better introduction to programming for young students.

1.4: Solution Introduction

The goal of this research is to develop a video game that will teach beginning programmers the fundamental tools and concepts of programming. Because we are targeting young students, our approach to will be engage the players in unique puzzles that implicitly introduce programming concepts. In particular, we will create puzzles that introduce the programming concepts of sequential, conditional, and iterative operations. The remainder of thesis describes our design, implementation, and evaluation of this educational game which will be referred to as WIP.

1.5: Prior Work in Game Development

This research will build upon our prior game development experience. During the last six years we have created several games, both personal and commercial. One game, "FBLA Game: The Game", was a PC arcade game that competed in a national game design competition. Another, "Dot Wars", was released for Xbox 360. It received praise as "one of the best on the Xbox Live Indie Game market"

(1UpOrPoison, 2013). Our last released game was "Steve: Snake Evolved" which was developed for the Android market, were it holds a four and a half star rating on the Android Play Store. It is due to this past work that the methodical game development process employed in this research.

2.0: Approach

2.1: Approach Introduction

Development of WIP was approached in three phases. The first was Pre-Production. This was where most game design takes place. Next came the Production phase. Here, code was written and non-code assets created. The final stage was Post-Production. In this phase, we debugged, polished, and refined the game. The sections below describe in detail the importance and purpose behind each phase of game development and what tasks we completed during that phase.

2.1: Pre-Production

Pre-Production in this paper is the term that defines all planning for a project that occurs before code is written and assets are created. This is because it is not enough to simply state what the objective of the game and begin Production. The game must be designed. Previously, it was mentioned that a game is nothing other than a set of rules that are followed. Game design is where these rules are created. The whole game revolves around these rules as they define exactly what the game is and what

the game does. If the game is not codified into a clear set of rules then the result is ambiguity. This leads to a Production phase that becomes mired in unclear objectives as to how to implement the game. The result is at best an unfocused game filled with untapped potential and at worst a sloppy game that frustrates the player. In WIP, Pre-Production was completed through the employment of a design document.

2.1.1: Design Document

A design document is a written material that acts a bible during a video game's development. This document performs two roles. The first is that a design document acts as a record for all design questions and answers that have occurred during the project. A design question is any question related to a designer's development of the game's rule set. These questions vary greatly. What genre of game is this? Should the game be fast-paced? Is this a single-player game or multiplayer? As mentioned, not every designer asks these specific questions, but they do give an idea of what goes on during game design. Regardless of what questions are asked, their answers are to be recorded in the design document. That way, if the same or similar question appears again, the answer is already there. And, if the answer is no longer acceptable in context of the project, it can be revised.

The second role of a design document is to act as a framework for the implementation. As the design document is filled out, a shape of what the project is

to become begins to form. This outline provides a reference. New ideas can be checked against the document to see if they mesh. Old ideas that are revisited can be changed or removed as the project direction becomes stronger. By following this process we ensure consistent design decisions and thus a consistent end product.

Just as there are many ways to approach Pre-Production, there are also many ways to implement a design document. For WIP, the design document was broken down into categories which aimed to define the five key components of a video game: Player Gameplay, Player Input, Level Design, Art Design, and Audio Design. In the following sections, we define these components in more detail.

2.1.2: Player Gameplay

Player Gameplay describes all actions the player may take in game. For example, in Tic-Tac-Toe the player may only take one action: place one of their pieces in an empty slot on the gameboard. Now, consider something more complex, such as Chess. There, the player may take two actions: move his or her piece to an empty location or take another opponents piece. While almost as simple as Tic-Tac-Toe on the surface, Chess has much more to consider. To start, there are multiple types of pieces. Each piece moves uniquely. Additionally, several pieces have unique abilities. Pawns are able to promote. A rook and king are able to perform a “castle”. This highlights an important lesson as a designer. Often, something simple such as “a player can move a piece” entails far more detail and outlining than first thought.

In WIP, players position “tools” on a board to solve various problems. There are five actions that can be taken during gameplay: select a tool, place a tool, remove a tool, set a tool, and execute a solution. Before these actions are detailed, the concept of a tool will first be explained.

A tool, game-wise, represents an abstraction of various programming concepts. In game, there are three tools: math, conditional, and loop. The math tool encompasses various mathematical functions one may accomplish in actual code such as addition, subtraction, multiplication, and division. Next is the conditional tool which embodies the conditional statements if and else. The last tool, loop, allows players to introduce iteration into their solutions.

With tools now defined, one may return to what the player gameplay is in WIP. To reiterate, the player may do the following: select a tool, place a tool, set a tool, remove a tool, and execute a solution. “Select a Tool” refers to the action of choosing which tool the player wishes to use from a menu. “Place a Tool” is the act of placing a tool onto the gameboard. “Set a Tool” is where the player sets the attributes of a tool they are placing. “Remove a Tool” is where the player may delete a tool they have placed from the gameboard. Finally, there is “Execute a Solution”. After the player has set up his or her various tools, they are able to test their solution to see if it solves the puzzle. This is akin to compiling and running code. During the execution phase,

random inputs are given and the player's solution is checked to see if it can return the desired outputs.

2.1.3: Player Input

Whereas Player Gameplay is defined as what the player can do in game, Player Input defines what the player must do in the real world to act in game. Returning to the Tic-Tac-Toe analogy, it was declared that the only action a player could take in game was to place a piece on the gameboard. How does the player place their piece? If the game is played on paper, they must draw their X or O on the game board. In Chess, he or she must grab their piece and move it to where they desire. Player Input is crucial to the success of Player Gameplay and the game as whole because it is almost solely responsible for how the player physically feels during play. If playing the game is uncomfortable or taxing, the user is less inclined to play and thus renders the rest of the game moot. A common example of a game with clunky Player Input is the claw machine, the arcade games where one uses a claw to grab a prize such as a stuffed animal.

In WIP, Player Input is handled by keyboard and mouse. For tool selection, the player may use hotkeys (a hotkey is single key or combination of keys that is linked to an action in game) or clicks to chose their tool. Placement is handled by simple left clicks on the grid. Removal of tools is handled by right clicks on the tool that is to be removed. Setting the tool's attributes after being placed, such as a newly created

conditional, is done via clicks in a dialogue box. Finally, execution is handled by pressing the spacebar.

The reasoning behind these controls is that they follow traditional PC game controls. Usually, the role of the keyboard is provide instant access to selecting a utility of some kind. In Real-Time-Strategy (RTS) games this is a building or unit. In First-Person-Shooters (FPS) this is usually a gun in particular slot. Meanwhile, the mouse acts as the actual action execution in game. In RTS games this means attacking with one's armies. In FPS's it means shooting one's gun. Thus, by following the expected roles of the keyboard and mouse, WIP is able to feel comfortable to any experienced PC game user.

2.1.4: Level Design

Level Design can be described as the thought behind the creation of the game-space by where the game is played. Its importance cannot be understated. Without proper Level Design all Player Input design and Player Gameplay design is rendered moot. Returning to the Tic-Tac-Toe analogy, the game-space can be said to be an empty 3 by 3 board where pieces are placed. Without the board, one could not actually play the game. If the board was partially or completely filled the dynamic of the game would change entirely. Chess is an even better example of Level Design. The size of the board, how the pieces are initially placed and the checkering of black and white squares are examples of Level Design. Like Tic-Tac-Toe, if any one of the above

details were changed, the whole nature of the game would also change. Imagine if White was forced to play with nothing but pawns or Black only started with a king and four rooks. This new game would not be the Chess that is known today.

In WIP, Level Design is designed to gamify foundational programming problems. Before diving into that, there are a few traits that all levels hold. The first is that all levels hold an N by N grid, where N is a variable number. This grid holds N squared number of slots for the player to place tools. Depending on the level, N may greater or lesser. Additionally, all levels hold an input and an output. These are always located in top middle of the grid and bottom middle of the grid. The input square is where the player receives various level inputs during solution execution. It is then up to the player's solution to take proper inputs deliver them to the output square and reject invalid inputs. Otherwise, the level grid starts out empty.

As mentioned above, levels are based off of programming problems. Examples of this are “return only even numbers” and “count to the number hundred (100)”. This way, as players solve problems they also learn to analyze and breakdown a problem like a programmer would. Making available the tools that a coder would have and throwing problems a coder would typically deal with, it is hoped that the user is able to connect the two and learn these fundamental coding concepts.

2.1.5: Art Design

Art Design is the visual component of a game. In the Tic-Tac-Toe example this could be the game being presented by ASCII characters. In Chess, it could be three dimensional models of the pieces seen in an isometric view. However, as simple as it might seem, Art Design has many factors to keep in mind. Is the game two or three dimensional? For a two dimensional game, will the player see things top down (also known as a birds-eye view) or from the side (often called sidescrolling)? Then there is the matter of the art style. Images hand drawn or, perhaps, pixelated in similar manner to the classic arcade games?

All these questions are important because they exemplify what the purpose of Art Design is. Like a book cover, Art Design is responsible for setting the expectations and tone of a game. In addition, there is another more subtle role Art Design holds. Say a game has two teams, one enemy and one friendly. It might be best if all allies were color coded blue and the enemy team coded red. This is because there is a connotation between blue as friendly and red as enemy that reinforces this game concept of friend and foe. This is gameplay clarity. Gameplay clarity is ensuring that the gameplay makes sense and can be followed by the player. In the case of our hypothetical game, by taking to colors that contrast each other the player is able to differentiate between the two teams easily.

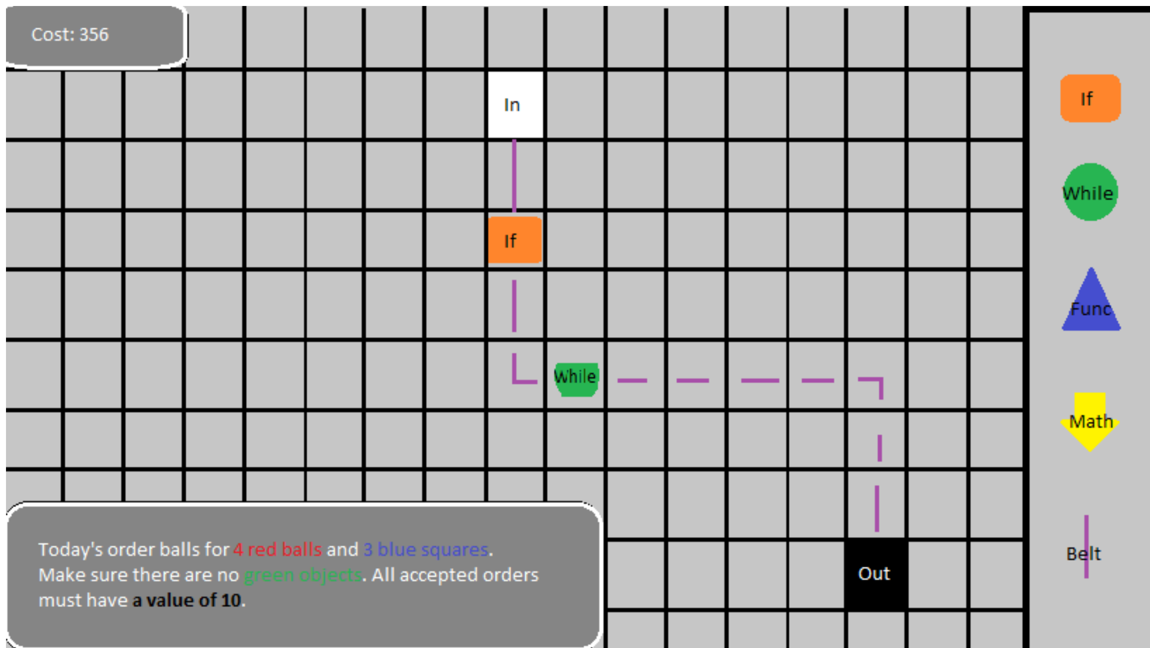


Illustration 1: A Design Mockup of WIP from the WIP Design Document. Here one may see how Gameplay, Level Design, and Art Design are shown.

In WIP, the art style is a minimalistic and two dimensional with a top down view. A mockup example of this can be seen in Illustration 1. The decision to choose two dimensions was due to gameplay. Since the gameplay in WIP revolves around creating a sort of flow (see Illustration 1 for an example), two dimensions proved to be a clearer way of representing this. The minimalistic (simple shapes and colors) art style was chosen emulate working in IDEs. In IDEs keywords are often color-coded while everything else is left simple black and white text. In a similar fashion, only programming concept tools in game receive in bold coloring.

2.1.6: Audio Design

Whereas Art Design dealt with what the player saw, Audio Design encompasses what the player hears. Perhaps there is “clink” noise each time a player places a piece in Tic-Tac-Toe. In Chess, the player may hear pleasant elevator music as they decide their next move. Like Art Design, Audio Design holds two roles: setting the tone of the game and gameplay clarity. Examples of the first would be playing powerful orchestral music during the climax of the game or playing a lone string instrument as the player character prepares a last stand against impossible odds. In the case of the second, the distinct pop of a sniper rifle being shot would be an example. Even if the player doesn’t know where the sniper is, the shot would inform them that they need to keep their head down.

Like Art Design, Audio Design in WIP was designed to be simple and non-intrusive. For the sake of clarity, simple sound effects were designed to give the player feedback when he or she has taken an action. An example of this would be a pop sound when a tool is removed. In addition, a quiet, but pleasant melody was designed such that the player does not need to play in silence.

2.2: Production

Production is where all aspects of game development occurs. Code is written, art is drawn, and audio is created. It is here that Pre-Production plans are implemented. During Production it is acceptable to see Pre-Production concepts fail and be scrapped. Depending on their importance, this may result in a return to

brainstorming to replace these concepts. However, this typically delays the progress of Production. Fortunately for WIP, the transition from Pre-Production to Production went very smoothly. In this section, we describe our own production process in more detail.

2.1.1: Development Tools

The selection of development tools is an important first step in Production. Due to the planning done in the Design Document, there were three fields of development: code, art, and audio.

For code development, we used Java and the LibGDX library. There were several reasons why Java was chosen over other languages. Besides the developer's personal proficiency in Java, the language is compatible with many game development libraries. One in particular is LibGDX, a powerful library that handles sprite rendering and asset management. By using these tools we were able to rapidly implement and debug the game.

Development of art work for WIP was handled by a program called Paint.net. This software, similar to Microsoft Paint, in that is it is an image creation and processing program. However, in addition to all the utilities MS Paint holds, it capable of additional features such as image layering. Because of its robust nature, as well as status as freeware, it proved to be an excellent choice of software for artwork

creation. Using Paint.net, we created the WIP board sprite, all of the various tool sprites, and other graphical elements.

All audio components for WIP was created using Music Creator 6 and Audacity. Music Creator 6 is a synthetic music creator that allows for the development of musical pieces without need of a live musician. This is done by first loading instrument tracks from Music Creator 6's library. When the desired instruments are loaded the user then writes notes for the instruments to play on a music staff. An example of this can be seen in Illustration 2.

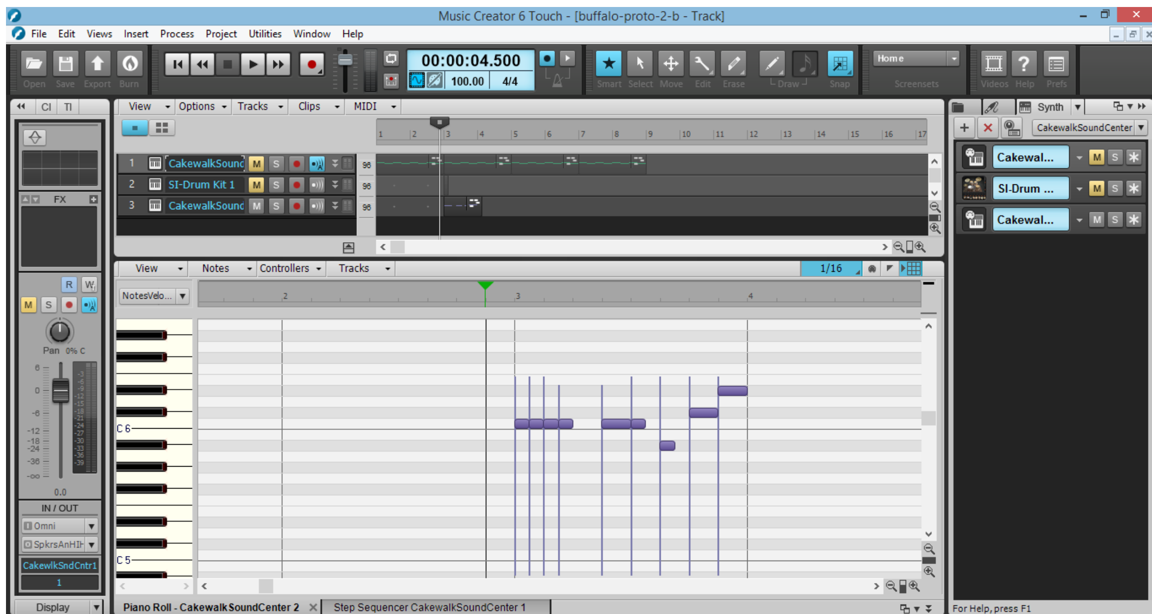


Illustration 2: A Screenshot of Writing Music in Music Creator 6

For sound effects, public domain audio clips were employed. Before they were used they were first put through Audacity, a freeware audio editing program. After an is

created or found audio piece it must first be edited. Editing can range from pitch changes to clips being combined to make a new sound. Illustration 3 is an example of such sound editing. There a sound clip has been loaded and is in the middle of being edited.

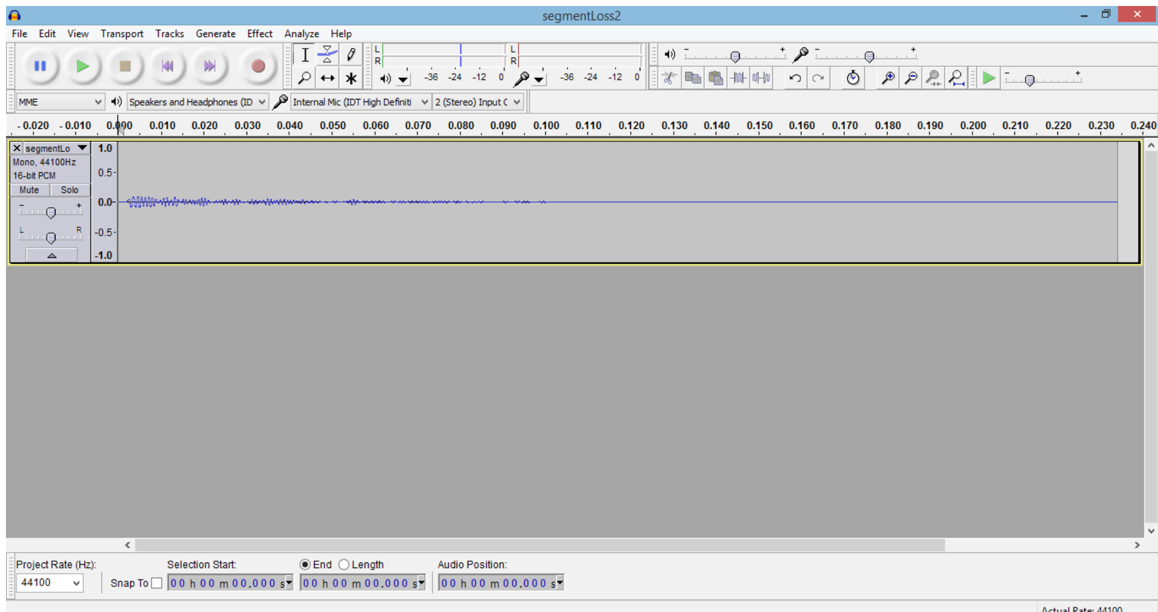


Illustration 3: A Screenshot of Sound Editing in Audacity

Although the creation of audio was a time consuming process, the final result was very successful. This result is all sound effects and music found in WIP.

2.2.2: Class Hierarchy Overview

Because this game was coded in Java, the overall structure of the code is object-oriented. Every element the user interacts with is an object of some form. It is because of this that a pyramid structure was chosen for the class hierarchy. Within a “source” folder lays packages. Each package holds classes that are related closely to

each other and the name of the package. Thus, the package “tools” holds everything tool related. While “levels” hold everything that has to do with levels. In this section, we will describe the implementation of all classes in WIP.

2.2.3: “Forms” Package Overview

It was previously mentioned that tools take input. In order to ensure that user places the proper input and is informed of it, the “Forms” package came to be. Inside “Forms” is one primary class, “Form.java”, that provides basic functionality. There are four classes that inherit from “Form.java” (see Illustration 4). Three are tool specific: “MathForm.java”, “IfForm.java” and “LoopForm.java”. These forms prompt the user for tool inputs. For example, “MathForm.java” asks the user for the MathTool's mathematical operator and an augment value. The forth form derived from “Form.java” is “LevelComplete.java”. As the name suggests, this is executed when the user has completed the level.

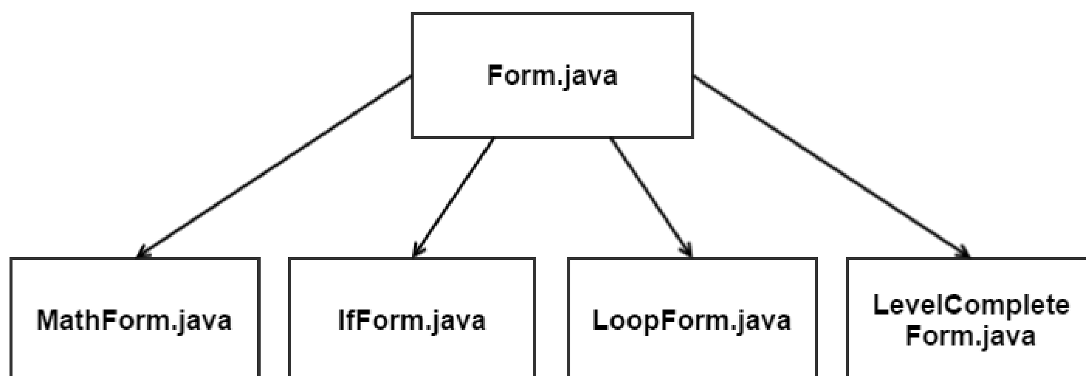


Illustration 4: The Class Hierarchy of the "Forms" Package

2.2.4: “Levels” Package Overview

The levels package consists of one primary class “Level.java” and three derived classes “LevelOne.java”, “LevelTwo.java”, and “LevelThree.java” that encapsulate the problem solving objectives in each game level. This is shown in Illustration 5 below.

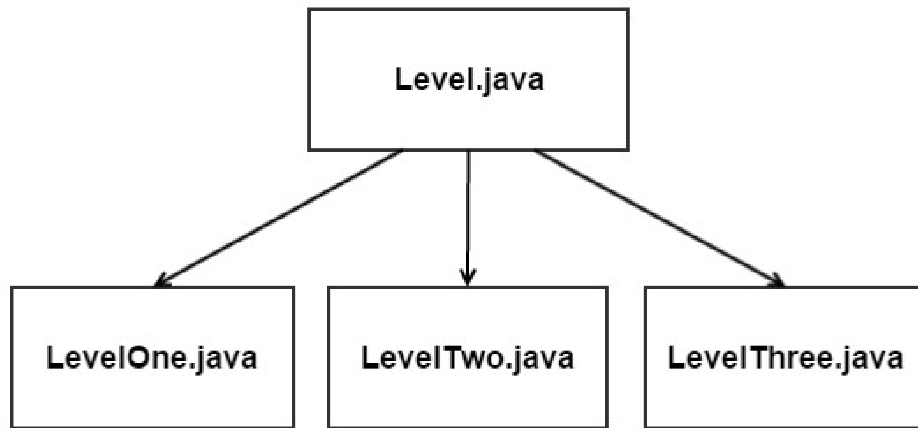


Illustration 5: The Class Hierarchy of the "Levels" Package

The “Level.java” class sets up all the fundamental workings of a level game flow. This entails a number of important tasks. The first is drawing the user interface which is broken into three parts.

- Dashboard – Where the player selects tools to place on the game board
- Game Grid – Where the player places and removes tools
- Heads Up Display (HUD) – Where messages to player are displayed

In addition to drawing the user interface, “Level.java” is also responsible for

handling user input. When the player left or right clicks a location, “Level.java” decides what action, if any, needs to be taken. This decision is context sensitive. For example, clicking on tools on the dashboard will have a different effect from clicking on tools that are already placed on the game grid.

Finally, it is within “Level.java” that the slot for a level's win condition is set and checked. Because of how “Level.java” is implemented, it is easy to create new levels in game by merely creating classes that extend “Level.java” and define what the new level's win condition is. The remaining classes within this package are just that, children of “Level.java” with unique win conditions. By specifying different win conditions, such as “accept only even numbers” or “increase the input value until you reach the value one hundred” we challenge to player to master basic programming skills such as conditions and loops.

2.2.5: “Tools” Package Overview

The “Tools” Package holds only one primary class, “Tool.java”, which is responsible for creating the variables and functions that are universal amongst tools but are unique in execution. These include the tool’s sprite, its position, and an update method that specifies what action the tool should take when the level input has entered the tool. The “Tools” package also contains six classes derived from “Tool.java” that are specialized to the individual requirements of the tools in game: “IfTool.java”, “InputTool.java”, “LoopTool.java”, “MathTool.java”, “OutputTool.java”, and

“PipeTool.java”. This can be seen in Illustration 6 below.

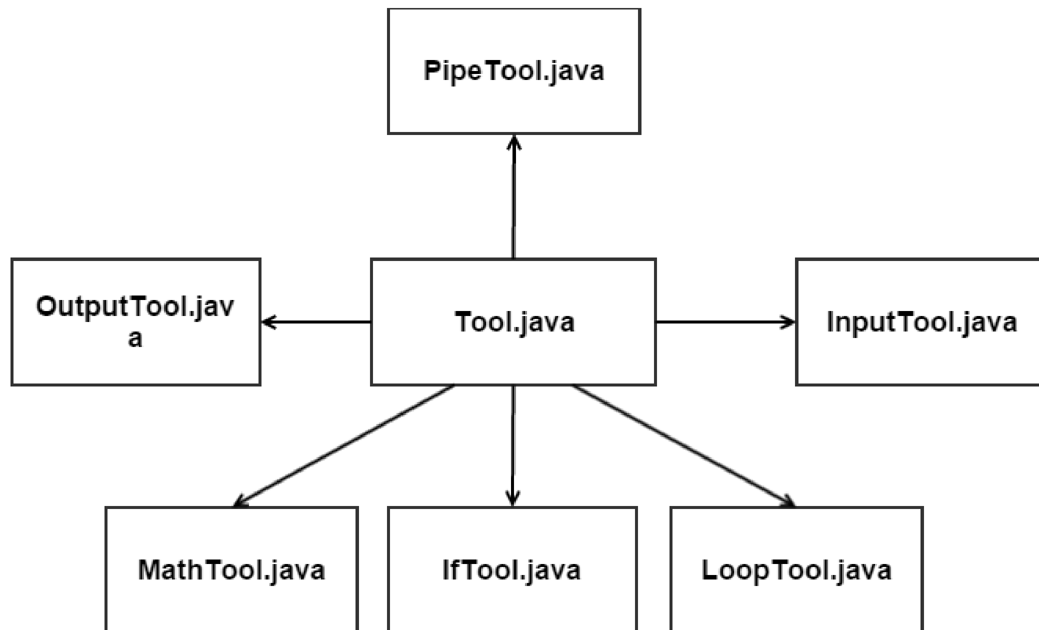


Illustration 6: The Class Hierarchy of the "Tools" Package

Our implementation classes in the “Tools” package mirrors the list of tools in our design document, so it is natural to have a “MathTool.java” to handle mathematical operations, “IfTool.java” to handle conditional operations, and “LoopTool.java” to handle iteration operations. We have also introduced three new tools: “InputTool.java”, “OutputTool.java” and “PipeTool.java”. These six classes are described in detail below:

- “InputTool.java” – Creates random input values for the game. These input values moves from square to square on the game grid and are transformed by other tools until the desired output value is created.

- “OutputTool.java” – Serves as an end point for the level. When values reach the output square, they are tested to see if they match the level requirements.

- “PipeTool.java” – This tool transports data values from one square on the game grid to the next square, and is used to connect other game tools. Special care was taken to display pipe connections as naturally as possible. For example, if a sequence of pipes are placed in consecutive squares going vertically or horizontally across the game grid, we display the pipes as vertical or horizontal line segments. More importantly, we automatically select an appropriate corner pipe to display when a player connects vertical pipes to horizontal pipes.

- “IfTool.java” – This tool handles conditional operations. When the tool is created, the user is prompted to give a Boolean comparison operator (such as “=” or “<” or “>”) and an integer value that is used as a comparison value. At run time, we compare the level input to the comparison value using the specified comparison operator. For example, if the player selects “<” and a value of “10”, we calculate the value of “level input < 10” and return either true or false. Using this information, the level input moves to either the next tool down on “true” side or “false” side. Although our IfTool handles only one comparison operation, it is possible to connect a sequence of IfTools together to perform more complex conditional operations.

- "LoopTool.java" – This tool handles iterative operations. The user interface for this tool is very similar "IfTool.java". When the "LoopTool.java" is created, the user is prompted for a Boolean comparison operator and an integer value to compare with. At run time, this tool mimics the behavior of a while loop. After performing the Boolean calculation, the level input will will go down either the "true" or "false" side. In addition, "LoopTool.java" contains an additional input slot that allows the level input to return to the tool and compare again. This enables the player to place a number of tools on the "true" side that transform the input in some way, and then go back and evaluate the Boolean expression. It is possible for players to create infinite loops using "LoopTool.java".

- "Math.java" – This tool performs mathematical operations to modify the input value. When this tool is created, it prompts the user for an integer argument. But, instead of requesting a comparison operator, this tool requests a mathematical operator (such as "+" for addition or "*" for multiplication). When the level input arrives to the math tool, it performs the specified mathematical operation using the integer argument. For example, if the user selects "+" and an argument of "1", the math tool will simply increment the input value by one. Similarly, if the user selects the modulo operation "%" and an argument of "2", the Math tool will calculate the remainder after integer division by 2. If the input value is 31 going into the

Math tool, it will be 1 after this operations is performed. By connecting a sequence of Math tools together, players can perform very complex mathematical operations.

2.2.6: “UI” Package Overview

While “Level.java” from the “Level” package handles the display of most game elements on screen, there several elements that are too complex to be handled by “Level.java” that warranted their own classes. We have placed these classes in the “UI” package. Within this package are two primary classes: “HUD.java” and “Dashboard.java” that are not connected to other classes in our class hierarchy. This can be seen in Illustration 7.

The “HUD.java” class is responsible for drawing any relevant and dynamic information to the player such as his or her score. The score is calculated based on how many tools were placed on the game grid to solve the level. This is compared to an ideal solution to the level to give the user points. This way, players have an incentive to find the smallest possible solution to a level. The “Dashboard.java” class renders the dashboard which consists of the toolbar and a selection sprite informing the player on what tool is currently selected.

Illustration 7: The Class Hierarchy of the "UI" Package

2.2.7: “WIP” Package Overview

The final package is unusual compared to the others. Here, the relationship between classes isn't what they hold in common, but rather that they share nothing. In other words, this is package that contains all the important classes needed for the game to run, but have no similarities. The first class is “WIP.java”. It is the driver classes for the game. When loaded, this is the class that is called and run. From it, everything else is created, from program window to tool image. The other is known as “Menu.java”. This is where the player is sent when the game begins. Here he or she may select the level they wish to play. When the level is completed, they are returned here to select another.

Illustration 8: The Class Hierarchy of the "WIP" Package

2.3: Post-Production

The final WIP package is unusual compared to the others. Here, the relationship between classes is not what they hold in common, but rather that they share nothing. In other words, WIP is a package that contains all the important classes needed for the game to run, but have no similarities. The first class in this package is "WIP.java". It is the driver classes for the game. When loaded, this is the class that is called and run. From it, everything else is created, from the program window to the tool image.

The other class in this package is "Menu.java". This is where the player is sent when the game begins. As the name suggests, this class displays a menu of options to the player, where they may select the level they wish to play. When the level is completed, they are returned to "Menu.java" to make another selection. At this time,

we have no restrictions on what order a player can visit different game levels. They could play level three first, and then go back to play level one if they wished.

2.3.1: Post-Production in WIP

Post-Production is the final phase in our game development approach. It is here that we refactor and refine what has been created during the Production phase. In terms of code, this means debugging and cleaning up what has been written. In particular, this includes removing old functions, rewriting inefficient code, and finding and correcting bugs. Our goal was to take our initial working program and bring it up to release quality. We were successful in this task, and our current game release is fully operational and bug free.

A similar cleaning up process takes place for visual and audio elements during Post-Production. This typically involves editing or recoloring sprites to make them more visually clear and editing the volume of music for better balance with sound effects. What Post-Production should not entail is adding new content, such as a new sprites or sound effects.

During production, we created a sprite asset page that worked, but had several small visual flaws. During Post-Production, we refactored the sprite asset page to have a cleaner and leaner aesthetic. The difference can be seen in Illustration 9 and Illustration 10 below. In Illustration 9 there are several flaws: the pipe line is off-

center, the grid lines are too thick, and the tool box does not have a title. All of these problems were corrected in our final game which is illustrated in Illustration 10.

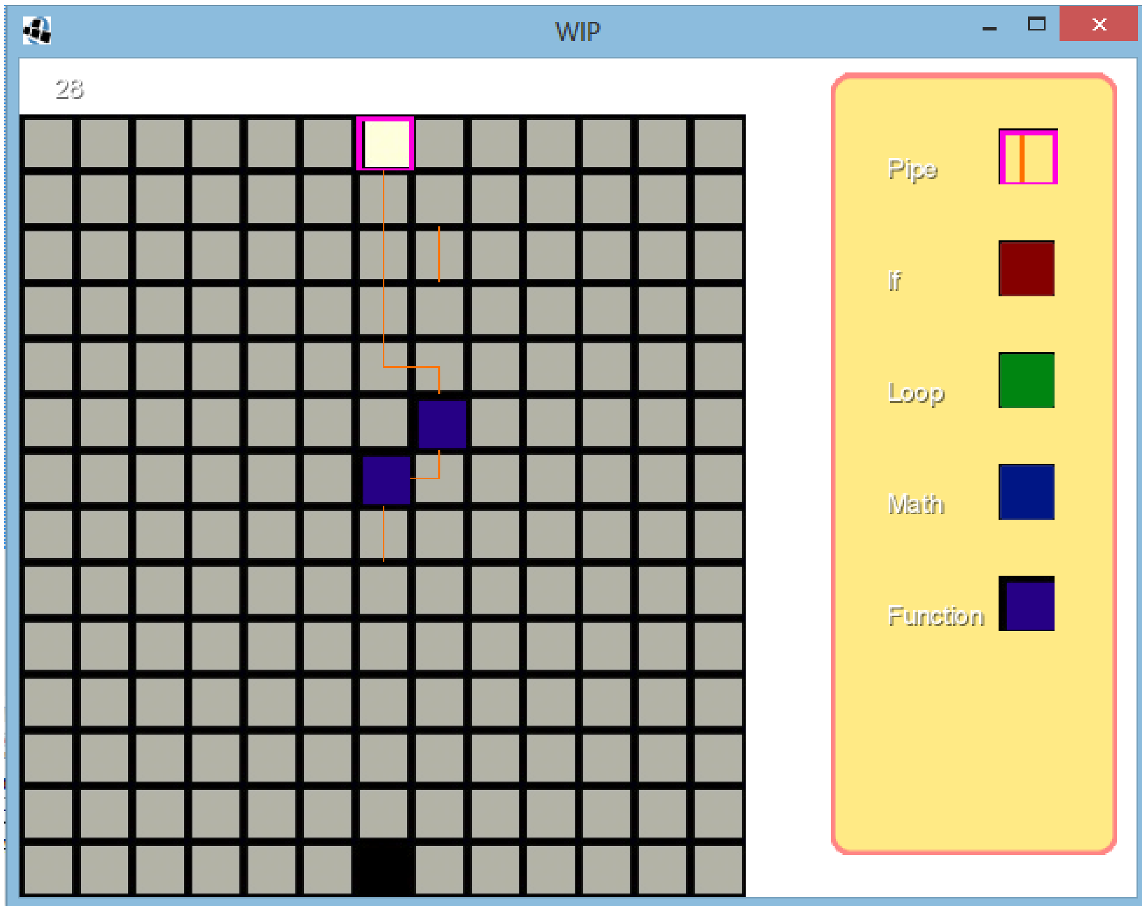


Illustration 9: A Screenshot of WIP just before Post-Production

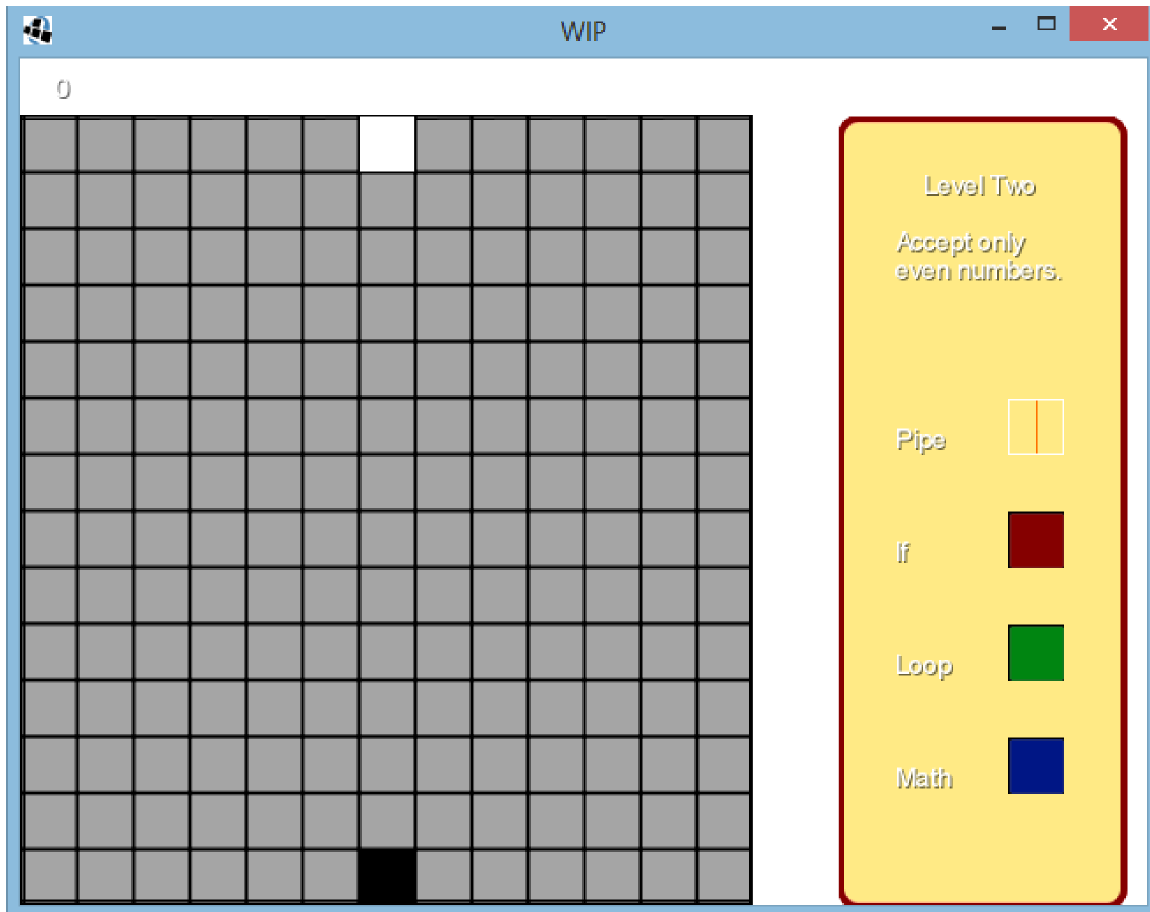


Illustration 10: A Screenshot of WIP after a Visual Update

3: Results

3.1: Results Overview

The goal of this research was to develop a video game that would teach beginning programmers the fundamental tools and concepts of programming. Specifically, the game would introduce the programming concepts of sequential, conditional, and iterative operations. WIP successfully does this. In the following sections, we will detail what WIP does and how it successfully fulfills the research goal.

3.2: WIP Content Overview

When WIP is booted up, the player is introduced to a main menu. From here, he or she may select whichever level they so wish to play. See Illustration 11 for a screenshot of the main menu.

Illustration 11: WIP at its Main Menu

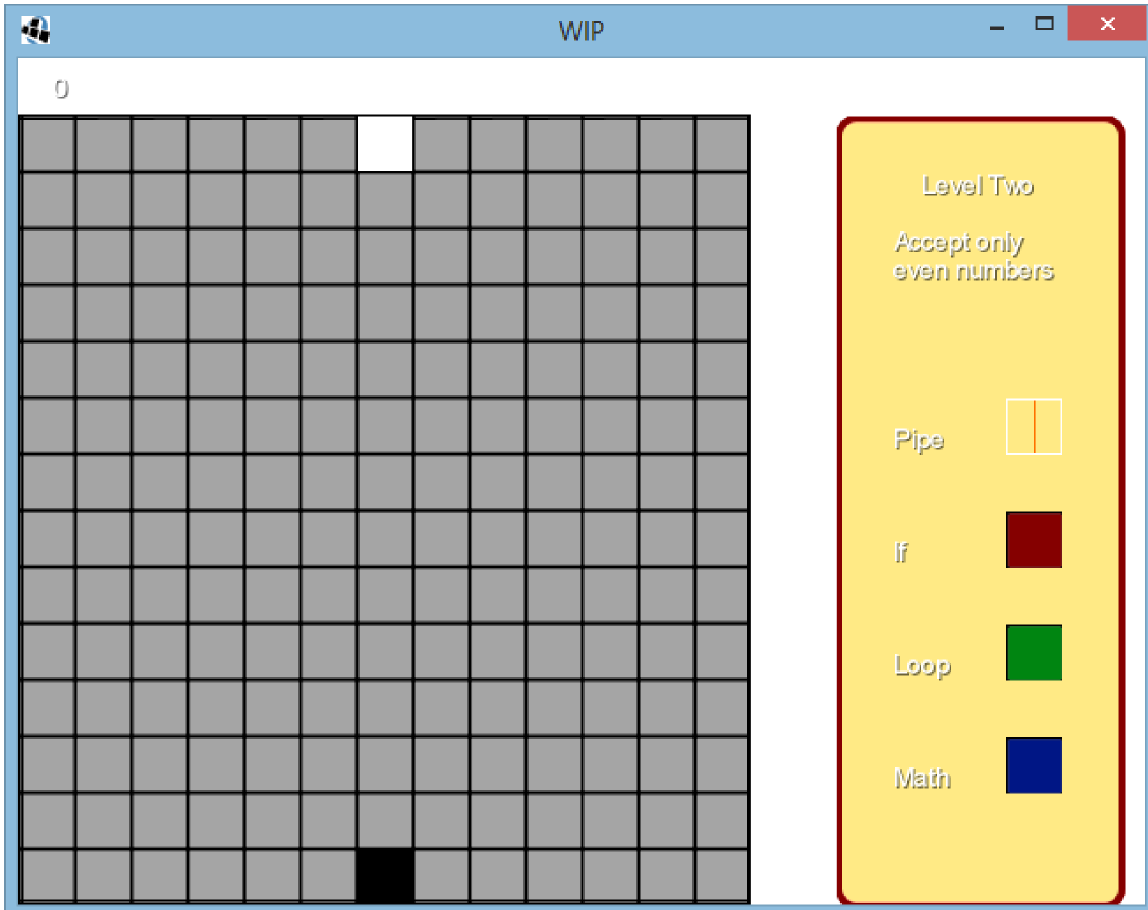


Illustration 12: A Screenshot of an Empty Level

The above (Illustration 12) is what the player first sees when they load a level. Here, he or she may select one of four tools (Pipe, If, Loop, and Math) and place them on the grid. Their objective is to connect the Input tool (the white block) to the Output tool (the black block).

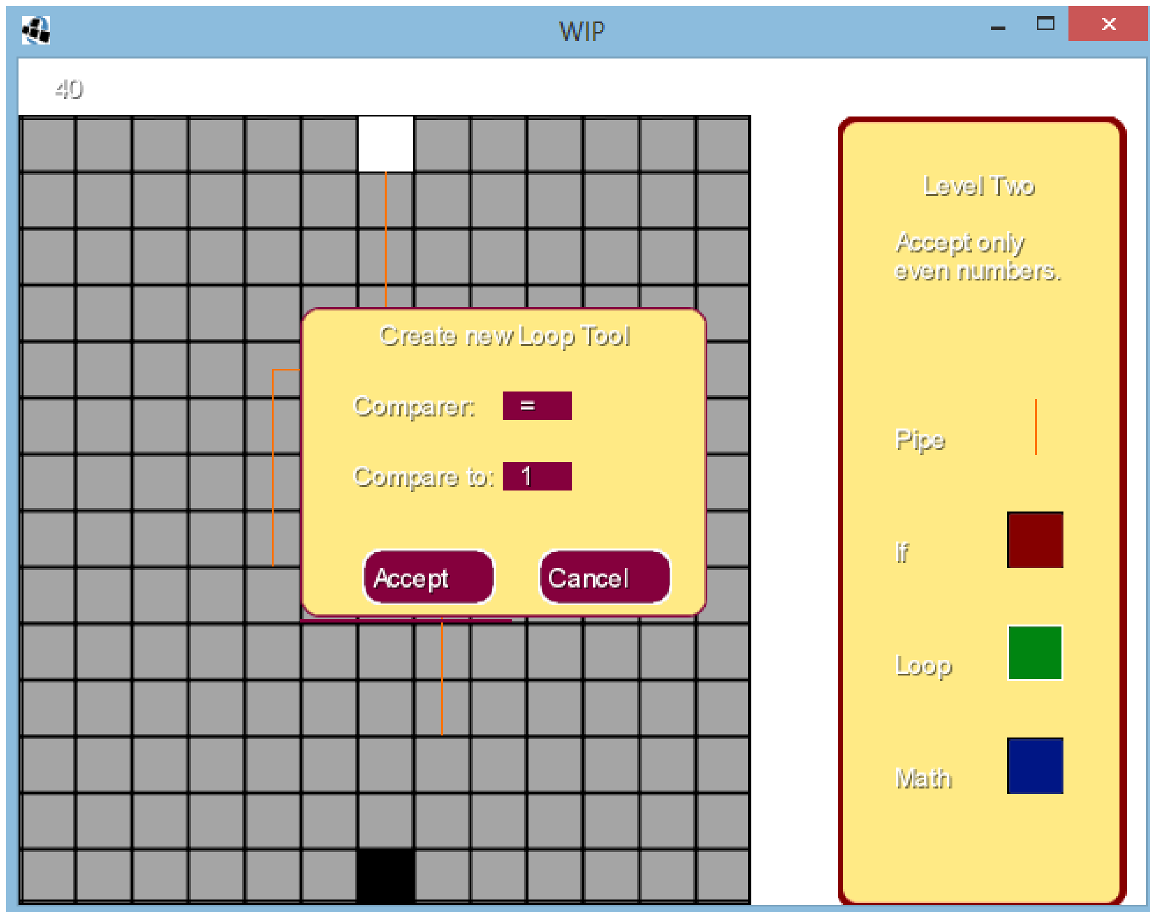


Illustration 13: A Screenshot During the Placement of a New Tool

There are three levels in WIP. Each level is designed to introduce a tool and its uses by having a level objective that require that tool. In doing so, the player learns the uses and value of each tool implicitly.

Level One serves as the introduction the MathTool. This done by the level requesting the player return the value of the input incremented by one. In order for the player to solve this puzzle, the player must create a MathTool and set the settings to add one.

Level Two introduces the IfTool. This is done by the level objective being that the player must deliver only the input if the inputs value is even. In order to determine whether the input can be accepted, the player must check to see if the input is even. This is done by first using the MathTool with the attributes of module and two. The resulting number will either be a one or a zero depending on if it is odd or not. The player must then create an IfTool to check if the value that comes out of the MathTool is zero. Depending on the result they will either accept or reject the input.

The final level, Level Three, serves as an introduction to the LoopTool. Here, the level objective is to return the number one hundred. To this the player must employ the LoopTool. When the tool is created (an example of this process can be seen in Illustration 13), it must be set to check if the level input is less than one hundred. If it is, then it must send the level input to an adjacent MathTool to be incremented by one. Afterward, the level input returns to the LoopTool to be checked again. When the LoopTool's comparision comes out false, the player can then send the level input to be accepted.

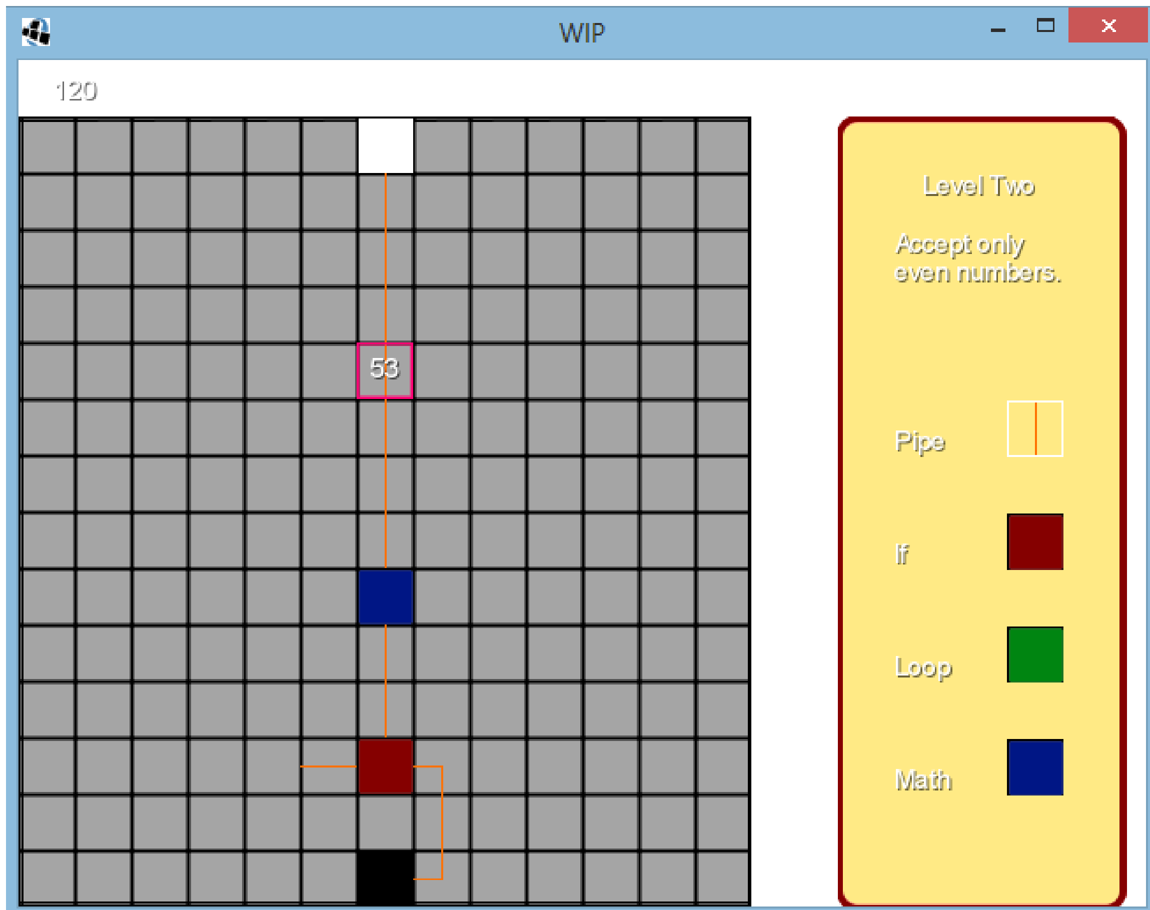


Illustration 14: The Player's Solution Being Tested

The player is free to place tools however they wish for as long as they want. When her or she is ready they may execute his or her solution and watch as the level input is either rejected or accepted, by the player's hopes, successfully. During this process, the player is able to watch as the level input slowly works through their solution. In this, the player can see what happens each step of the way. This is akin to stepping through a program during a debugging mode an IDE. By allowing the player to watch, he or she can see where things go right or wrong with their solution and adjust if need be.

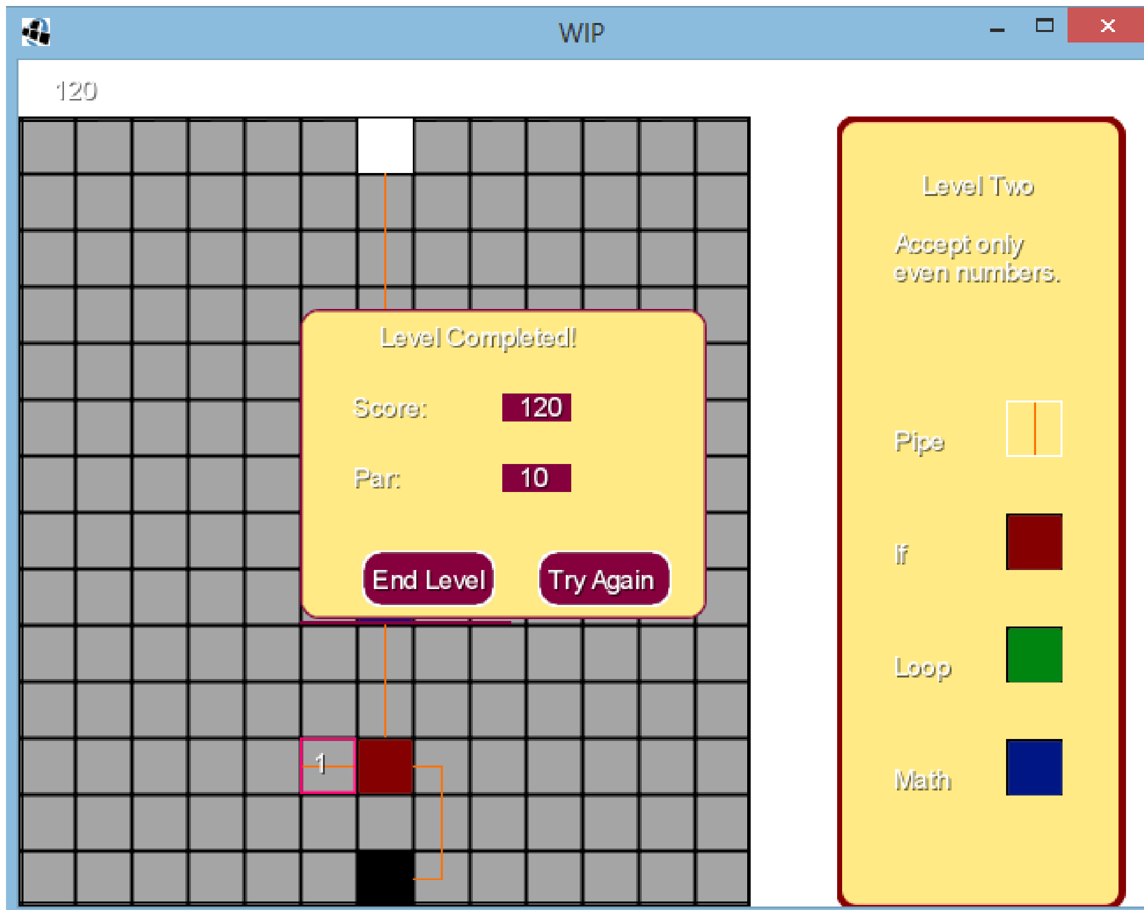


Illustration 15: The Player Successfully Completes the Level

Upon success, the player is asked whether they would like to play the level again or return back to the menu, this can be seen in Illustration 15. In addition, the player is given their score (which calculated by the tools used in the solution) and is informed of a par score (the most optimal solution we could come up with). This comparison is very important. To start, the player is given a challenge to try and find the best solution for the puzzle. This creates game replayability which gives the player more exposure to the game and the subtle lessons it contains. In addition this introduces the player to the idea that there is often more than one solution to a problem in

game and, by extension, in programming. Some solutions are better and worse than others.

4: Conclusions and Future Work

4.1: Conclusions

It can first be stated that this research and WIP are a success. The research solution was implemented and all objectives were completed. However, there is room for further work. This future work may be divided into two categories: studies and feature expansion. The following sections detail these categories.

4.2: Future Work: Studies

While WIP accomplishes the research of objective, experiments to test the game's effectiveness at teaching would be something of great interest. There are a number of questions to ask ranging from how much did players learn to whether their interest in programming rose or not.

4.3: Future Work: Feature Expansion

In this section, we will describe some of the possible features that could be added to WIP. In particular, we explore the idea of additional tools, smoothing the level curve, and adding additional levels.

In terms of gameplay, there are many interesting programming concepts that do not yet exist in WIP. Two concepts that we find most interesting are the variable and the function. With a variable tool the user would be able to store information on the fly. And with a function tool, that would allow the user to explore the concept code compartmentalization and modularization. Between these two, incredible puzzles could arise that could explore concepts like recursion and global/local variables.

Additionally, there would be a focus on smoothing out the learning curve in the game. Currently, there is not dedicated tutorial to introduce a tool. The result is the player having to blindly learn how tools work, a trial and error process that can frustrate.

Finally, with the inclusion of new tools, there could thematic level sets. These sets could be simply themed levels like using only loops and functions. Through these sets the player could be thoroughly introduced to all the possible uses of a tool in conjunction with another tool. They could also help the player practice as more material means more opportunity to challenged.

References

Bowman, R.F. 1982. A Pac-Man theory of motivation. Tactical implications for classroom instruction. Educational Technology 22(9), 14-17.

Squire 2005 http://files.eric.ed.gov/fulltext/ED497016.pdf

Rosas, Ricardo, Miguel Nussbaum, Patricio Cumsille, Vladimir Marianov, Monica Correa, Patricia Flores, Valeska Grau, Francisca Lagos, Ximena Lopez, Veronica Lopez, Patricio Rodriguez, and Marcela Salinas. "Beyond Nintendo: Design and Assessment of Educational Video Games for First and Second Grade Students." (2002): n. pag. Elsevier Science Ltd, 2002. Web. 20 Apr. 2015. <http://www.psiucv.cl/wp-content/uploads/2012/11/Beyond-Nintendo.pdf>.

Jenkins, Tony. "On the Difficulty of Learning to Program." *On the Difficulty of Learning to Program*. University of Leeds, 2002. Web. 20 Apr. 2015. <http://www.psy.gla.ac.uk/~steve/localed/jenkins.html>.

Gomes, Anabela, and A.J. Mendes. "Learning to Program - Difficulties and Solutions." ICEE, 2007. Web. <http://ineer.org/Events/ICEE2007/papers/411.pdf>.

"Dot Wars." 1UpOrPoison, 13 Nov. 2013. Web. 25 Apr. 2015.