Implementation Report for *DELTA: CONCEPTS OF PROGRAMMING*

Fall 2017, Implementation Phase
Software Design Methods – SW301
November 27, 2017

Developers:
Jack Crowley
Peter Julian
Chris Kelly
Matt Richardson
Nick Zazula

Table of Contents

Purpose .... Page 3
Implementation IDE .... Page 3
Implementation Language .... Page 3
Coding Styles .... Page 4
Purpose .... Page 4
Game Structure .... Page 4
File Names .... Page 4
Script File Structure .... Page 4
Comments .... Page 5
Single Line Comments .... Page 5
Multi-Line Comments .... Page 6
Naming Conventions .... Page 7
Method Parameters .... Page 8
Control Structures .... Page 8
Miscellaneous Coding Conventions .... Page 9
Naming Conventions .... Page 9
White Space and Indentation Conventions .... Page 9
Curly Brace Conventions .... Page 10
An Example .... Page 10
Appendix .... Page 12
General Scripts .... Page 12
Level 1 Scripts .... Page 12
Level 2 Scripts .... Page 12

Purpose

The purpose of this game is to teach users the fundamentals of programming across the various programming languages. We are not aiming to teach the exact syntax of languages, such as Java and Python, but to explain the theory behind various programming concepts. In the game, we will attempt to teach the user about control statements, sorting algorithms, data structures, and the importance of commenting. We will tackle this challenge by presenting the user with various challenges to complete, while also explaining how these relate to the world of programming.

Implementation IDE

After being assigned a game to teach users the fundamentals of programming, our team had to decide how we would implement the software. We chose to use the Unity Game Engine. Unity allows the programmers to create a 2D or 3D game, depending on the end goal of the project, across various platforms. This is one of the main reasons why we chose to implement the Unity IDE. For example, with only a few minor changes to our program, our game will be able to run on both OS X, Linux, and Windows computers and on mobile devices with iOS and Android operating systems. Another advantage to using Unity is the fact that it has its own Asset Store. Here we had access to various free and/or cheap components for our games, including sprites and other art. Finally, for our purposes Unity was free to use, which was also a big plus. Each of these facts made our decision easy, as the Unity Game Engine would allow us to design and implement our game with relative ease.

While the Unity IDE was focused more on the graphics and visual aspects of how the game functioned, we would still need an IDE to write the necessary scripts for our project. Based on the fact that we were planning on using Unity, this decision was extremely easy to make. When downloading Unity onto one’s computer, there is also a prompt that asks if he or she would also like to download Visual Studio, as the two work hand in hand with each other. Therefore, we chose to implement our code using the Microsoft Visual Studio IDE.

Implementation Language

Based on our decision to use the Unity IDE and Visual Studio IDE, we decided to implement the C# programming language. We all had experience with Java before this class, so we believed that C# would be relatively easy to learn based on this knowledge. In addition to this, Visual Studio and C# were both developed by Microsoft, so they would be extremely compatible with one another. Another reason we chose to implement C# was due to its vast Collections Framework, which we implement in nearly every script in our project. Finally, of the programming languages that the Unity Game Engine successfully interacts with, C# gave us the most flexibility with how we wanted to create our game.

Coding Styles

Purpose

The purpose of this section is to describe the coding standards used throughout the game that we have developed.

Game Structure

All of the files that we used to develop the game can be found in the “Assets” folder in the main project file. “Assets” contains all of the folders containing the files we have created to run the game. The four main folders that we used were the “Prefabs,” “Scenes,” “Scripts” and “Sprites” folders. The “Prefabs” folder contains our saved game objects. The “Scenes” folder contains the graphics for each of the game levels we have created so far. Currently, we have a *HomeTown* scene, *Start* scene and scenes for the first three levels .The “Sprites” folder contains images that we used to create the scenes for the game. This include the background and characters in the game. The “Scripts” folder contains all of the files that make the game run. The main folder contains the scripts that control how the character and camera move, the *LevelManager* script, which opens the level connected with a certain building in the Hometown, and two dialog controlling scripts. Finally, in this are two subfolders for levels 1, 2 and 3.

File Names

C# source files use the suffix *.cs*
C# metadata files use the suffix *.meta*
Unity Scene files use the suffix *.unity*
Sprite image files use the suffix *.png*

Script File Structure

All of the script files in this game follow the same format as one another. First, at the top of the file, are the import statements. These can be found by the “using” keyword. Many of our classes use all four of the following imports or a subset of them:

- System.Collections
- System.Collections.Generic
- UnityEngine.UI
- UnityEngine

The first two allow us to import the C# Collections Framework and all of the data structures that is contains. The latter two imports allow our C# files to import the necessary data for our scripts to successfully interact with the Unity Game Engine.

After the import statements, is the class declaration statement, which start with “public class.” This is followed by the name of our class. Finally, at the end of the is a colon followed by the “MonoBehaviour” keyword. “MonoBehaviour” is a class in the UnityEngine that inherits from the “Behaviour” class. Essentially, this means that all of our classes in the game inherit from the “MonoBehaviour” and “Behaviour” classes in the UnityEngine.

After the class declaration, the main body of our code can be found. At the top of this section, our variables used in the class can be found. We use a variety of private and public variables. In addition to this, we use primitive data types, such as “int” and bool,” along with abstract data types, including “Sprite,” “Vector,” “GameObject,” and “AvatarController.” Finally, below our variables are our methods, which either have parameters or do not depending on the function of the method.

Comments

Our game scripts contains a multitude of comments throughout. This is based on the principle that self-documenting code does not exist under any circumstances. All parts of the code have been sufficiently commented, as the goal is to have other users be able to read through the code and understand fairly quickly what each class does and how they interact with one another.

Throughout the scripts we developed, we implemented two different types of commenting. The first type is single line comments, which are depicted with two slashes (//). These are often used to describe one specific line of code and are often found either on their own line or at the end of line with some code. Examples of these can be found below:

Single Line Comments

Example 1: AvatarController Class

```
 else
{
        // Whenever user is not moving.
        Stationary(savedDirection);
 }
```

This example explains why the Stationary method is called. As the user can see, this is an else statement following a string of “if” and/or “else-if” statements. It helps the user to determine when the Stationary method would be called. As the comment states, it will be called when the user is not moving in any direction.

Example 2: CameraLock Class

```
void start()
 // Runs at start and will determine Avatar's Current Position;then go there
{
        targetPos = transform.position;
}
```

This example describes when and how this method will run. First, the comment states that the method will run as soon the games is started. Next it will determine the player’s current position on the map, and it will lock the camera on this location and move to it.

The second type of comments implemented in the game are multi-line comments. They are denoted in the following format:

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

As opposed to the single line comments, these multi-line ones are used to explain the purpose of a method and summarize what it does. This is extremely helpful to users, as they will not need to read through hundreds of lines of code in order to understand how the game operates.

Multi-Line Comments

Example 1: AvatarController Class

```
public void loadLevel(int levelNumber)
/* Input: int levelNumber, to tell apart levels.
 * Purpose: Allow the user to teleport from Level A to Level B smoothly.
 */
 {
         Application.loadLevel("Level"+levelNumber);
}
```

This example demonstrates the structure of the multi-line comments used throughout the game's scripts. First, is the necessary input that a method needs to run. In this case, it needs an "int" value. Next, the purpose of the level is stated. This method takes an "int" and loads the corresponding game level for the user playing the game.

Example 2: DialogueController Class

```
 public void showDialogue()
 /* Input: N/A (Ran from DialogHolder Class).
 * Purpose: It will stop the avatar from moving, force the Stationary method
 *  to run then open the dialog box, and display the dialog text.
  */
   {
       avatarController.canMove = false;
       avatarController.stationary(avatarController.savedDirection);
       dialogueBox.setActive(true);
       dialogActive = true;
   }
```

Again, this multi-line comment follows the same structure as the previous example. First, this method requires no parameters and the comment informs the reader that this method is automatically from the DialogHolder Class. The comment then explains that the purpose of this method is to force the avatar to become stationary when the user interact with an NPC and then opens up the dialog box.

Naming Conventions

Identifier	Convention	Examples
Class Names	The first letter of each word is capitalized, no spaces present	DialogHolder, CameraLock, BoxController
Class Member and Methods	The first word is all lowercase; for each successive word the first letter is capitalized	dialogueText, avatarController, loadLevel(), update(),

As demonstrated above, we followed the normal naming conventions for general best practice programming.

Method Parameters

We have employed two types of methods in our project: those that do not require arguments and those that do. The latter can be subdivided into methods with primitive arguments and methods with an object parameter.

Example of Method with no parameter: BoxController Class

Method Signature: void start()
How to call: start()

Example of Method with primitive parameter: LevelManager Class

Method Signature: void loadLevel(int levelNumber)
How to call: loadLevel(2)

Example of Method with an object parameter: LightSwitcher Class

Method Signature: void onTriggerEnter2D(Collider2D col)
How to call: onTriggerEnter2D(colliderObject)

Control Structures

Throughout our scripts, the only control structures we needed to implement are “if” and “if-else” statements. In the former, there is only one condition that is checked in a method, while in the latter multiple conditions can be checked before taking action. Two examples of these come from the *LightSwitcher Class:*

Example 1: Single “if” Statement

```
if (col.gameObject.name == "Avatar")
        {
            lightChanger();
        }
```

Example 2: Multiple "if-else if – else" Statements

```
 if (light.color == Color.clear)
        {
            light.color = Color.yellow;
        }
        if (light.color == Color.yellow)
        {
            light.color = Color.blue;
        }
        else if (light.color == Color.blue)
       {
            light.color = Color.red;
       }
        else if (light.color == Color.red)
       {
            light.color = Color.yellow;
       }
```

Miscellaneous Coding Conventions

Naming Conventions

Good practice states that variables, methods and classes should be named with meaningful names. Names such as a, b, c are extremely unhelpful when trying to determine what that is responsible for doing. Throughout each of our scripts, we have tried to name all the parts of our scripts in a meaningful way.

White Space and Indentation Conventions

Good practice states that there should be a sufficient amount of white space to improve readability in the code. It is especially important in C# because, like Java, there is no need for white space, as the compiler does not interpret white space to run. On the other hand, Python is a language that requires the user to enter white space for the interpreter to run correctly on the program. In our code, we used white space (1 space) on each side of an arithmetic or Boolean operator. In addition to this, the class declaration is left aligned, method signatures and variable declarations are indented 1 tab in, and the bodies of methods are indented at least 2 tabs. If there is an "if" statement in the body, addition indentations were made.

Curly Brace Conventions

Curly braces (“{“ and “}”) were used an enormous amount in our code, so it was important to adhere to best practice standards to make our code the most readable it can be. All curly braces were placed on their own line, so that the opening brace could be lined up easily with the closing brace. This makes it easy to understand where “if” statements and methods start and end.

An Example

Below is the *LevelManager Class*, which demonstrates each of these conventions in our code. Comments and imports have been removed.

```
public class LevelManager : MonoBehaviour
{
    public int levelNumber;
    private AvatarController avatarController;
    void start ()
    {
        avatarController = FindObjectOfType<AvatarController>();
    }
    void update ()
    {
    }
    void onTriggerEnter2D(Collider2D col)
    {
        if(this.levelNumber != 0)
        {
            loadLevel(this.levelNumber);
         }
         else
         {
              loadLevel();
          }
     }
   public void loadLevel(int levelNumber)
   {
         Application.loadLevel("Level"+levelNumber);
   }
   public void loadLevel()
   {
         if (avatarController.keyAccess == true)
         {
               Application.loadLevel("HomeTown");
```

```
                avatarController.keyAccess = false;
            }
        }
    }
```

Appendix

This contains each of the C# class files and a short description of what each class does in the scope of the game.

General

- AvatarController
  Allows the user to control their player in the game with the directional buttons and controls how the avatar moves through the various themes
- CameraLock
  Locks the camera onto the position of the avatar and follows its movements throughout the game
- DialogHolder
  Called by the DialogManager class to display various messages to the user
- DialogManager
  Manages the dialog that is displayed to the user by the NPCs in the game
- LevelManager
  Allows the player to move between levels in the game

Level 1 Scripts

- ChestInteraction
  Allows the user to interact with the chest in the center of the room; it is locked at first, but when the user completes the challenge it becomes unlocked
- EntryDialog1
  Controls the dialog of the NPC when the user enters the room
- ExitToTown
  Allows the user to go back into the town after they complete the challenge and unlock the door
- LightSwitcher
  Controls the color of the lights
- TownExit
  Allows the user to enter Level 1 after they pass through the door to the building

Level 2 Scripts

- ArrayController
  Allows the player to successfully move the boxes
- ArrayManager
  Controls the location the boxes are moved
- Exit
  Allows the user to re-enter the town.

Level 3 Scripts

- BoxController
  Allows the player to successfully move the boxes
- BoxManager

Controls the location the boxes are moved to