Concepts of Programming Languages

Final Project: Standard Corsi Task to study Visuospatial Working Memory

Tai Martinez

18 December 2019

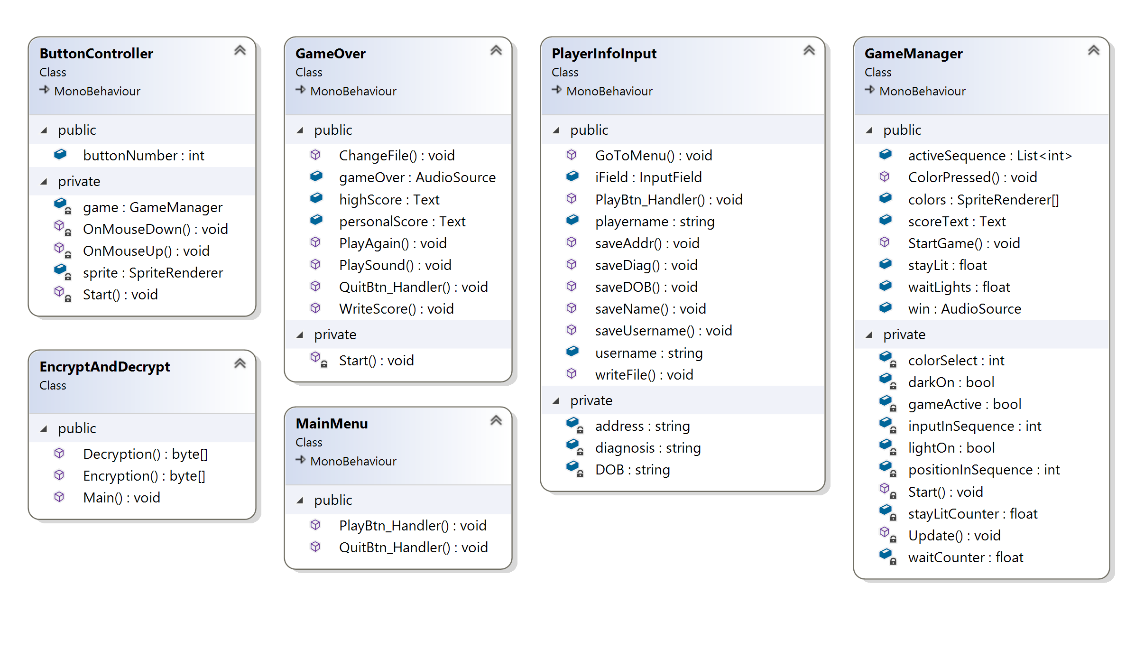
**Abstract**

Cube Commander is a desktop computer game that implements Standard Corsi Task to study Visuospatial Working Memory by assessing visuospatial short-term working memory. The basis of the game consists of nine cubes of different colors which are placed in a fixed three by three square on the screen where a random sequence is presented to players by highlighting cubes sequentially. The colors of the cubes are light blue, red, orange, dark green pink, yellow, purple, blue, and green. Players are asked to touch those cubes in the same order that they are highlighted. The sequence length begins at with one cube so players understand that they must touch the cube after it lights up, and then increases once players correctly select the sequence. The player’s score is displayed throughout the game and the highest player score is the game difficulty level.

**Programming Language Choice**

The programming language that was chosen for this project is C#. There are various pros and cons that helped influence this decision. Initially, I chose C# because I am very familiar with the language when developing games and desktop applications. This is because C# is a high-level object-oriented language which makes it easy and extensive for developing these types of projects. C# is similar to C++ in which it extends C however, C# also further extends C++. A major difference between the two is that C++ complies into machine code whereas C# complies to CLR [1] which is the Common Language Runtime virtual machine component of the .NET framework which is interpreted by the ASP.NET. Due to this reason, C# also runs well with the Unity Game Engine because Unity uses .NET to implement a standard Mono runtime for scripting which natively supports C#. This is a major reason why I choose C# as I wanted to use Unity as a simple yet effective game engine to help me develop my game.

**Memory Management**

Unlike C++ where memory management is the user’s responsibility, C#’s memory management is handled and performed automatically by the .NET garbage collection. This memory management is done through automatically allocating space on the heap. When an object is created during runtime, memory space on the heap memory is allocated for that object. When all actions related to that object are completed, the memory space is collected by the garbage collection to release the memory to be used for something else [2]. The way that this works is when a process is ran at runtime, there is a region of address space that has no storage allocated to it which is known as the managed heap. This heap will also manage a pointer which indicates where the next object can be allocated on the heap [3]. If the application creates a new object but there is not enough address space left to allocate for the object, the heap will add the new object to a pointer pointing to the next object. If this pointer is beyond the end of the address space region then the heap is full, and a garbage collection will be performed. The garbage collector will check to see if it finds any objects in the heap that are no longer being used by the application. If it finds these objects, then the memory is cleared and can be reclaimed.

**C# Binding Mechanism**

C# is primary considered a static binding language however due to the C# compiler performing the binding with the .NET framework, it also can support dynamic binding. For static binding, C# the compiler already knows the objects, methods, and properties because it recognizes them at compile time [4] which creates static objects. For dynamic binding in C# to be achieved, virtual methods must be used, and the “dynamic” keyword must be used when declaring objects. During dynamic binding, the compiler does not recognize the type of object, methods, or properties but rather the type of object is decided based on what data it holds during runtime. Throughout the code used in my game, I have decided to only use static binding due to the fact that it is decreases the number of runtime errors and is typically fast and simpler than dynamic binding.

**Game Code Overview**

*Diagram 1: UML Class Diagram*

The diagram above is a UML class diagram showing the different classes associated with the game. A full-size diagram can be found in the parent directory of the folder that this paper is found in.

There are four scenes incorporated into the game which are Menu, PlayerInput, Main, and GameOver which are controlled by the classes above in the UML class diagram. Menu is a simple overall game menu which includes the game title “Cube Commander”, a Play button, and a Quit button. When the player clicks the Play button it will advance to the next scene which is the PlayerInput scene. If the player clicks the Quit button, the game window will simply be closed. When advanced to the PlayerInput scene, the player will be prompted to input their name, username, date of birth, address, and diagnosis. All those fields are required to be filled out to continue, and they must not exceed twenty characters in length. That player information will be written to a file named nameOfThePlayer\_MM\_dd\_yyy.txt and will be in the OutputFiles folder. Each time a file with player information is created, it is encrypted for player security with RSA encryption which will be described in detail further along in this paper. This screen also has two buttons, one is a Back button which will return to the Menu scene and the other is a Next button which will advance to the next scene which is the Main scene.

On the Main scene, players will see nine colored squares in a three by three pattern. The colors of those squares are light blue, red, orange, dark green, pink, yellow, purple, blue, and green. To start the game play and thus the sequence of light up squares, players must click the “Click to Start!” button. When players correctly click the sequence of lit up squares, a “win” sound will be played, the score will be increased, and the sequence will increase by one. When players incorrectly click the sequence of lit up squares, they will be transitioned to the GameOver scene. While playing the game in the Main scene, players can see their current score, and the overall high score of the game played. When the player is advanced to the GameOver scene, a “game over” sound will be played and players can view the top 5 overall high scores of all players in the game as well as their top 5 personal bests for that username. From this scene players can click the Play Again button which will take them back to the Menu scene or they can click the Quit Game button which will simply close the application.

There are also six different scripts incorporated in the game to handle functionality of these scenes. Those scripts are ButtonController, EncryptAndDecrypt, GameManager, GameOver, MainMenu, and PlayerInfoInput which are all included in the UML class diagram above. An overview description of these scripts is below:

Initially, we have the MainMenu script which simply controls the Menu scene and handles whether the user clicks the Play button or the Quit button. Next, there is the PlayerInfoInput script which is linked to the PlayerInput scene. This script collects the user inputted information including the player name, username, date of birth, address, and diagnosis through individual methods that are populated with the UI text fields on the scene. The PlayerInfoInput script also contains a specific method that handles output to the file that contains all the player information as well as the sequence of events.

The ButtonController script groups the square sprites of the game as button objects and controls what happens to the color of those buttons using imported Unity methods. This script connects to the Unity game through directly referencing a method that saves button-clicked input in the GameManager script which interacts with the game actions that are performed. The main script that controls a majority of the game functionality is the GameManager script. This script is linked to the Main scene and it operates several functions of the game through the Unity Start method, Unity Update method, a StartGame method which is incorporated through the “Click to Start!” button on the scene, and a ColorPressed method. In this script, the main game functionality of lighting up colors in a sequence is done through storing that played sequence in a list. This script controls the light-up of colors through changing the level of alpha color through Unity. Furthermore, when the player clicks the button after the sequence is played, the sequence list is compared to the input to see if the player is correct. If the player is correct, then a “Win” sound is played and the score on the screen is updated. If the player is incorrect, it advances to the GameOver scene. This script also handles file output because it writes the sequence of events to the player information file, writes the high score to the highscore.txt file, and the personal best score to that username’s personal best file.

When the player advances to the GameOver scene, the GameOver script plays the game\_over sound as well as displays the top 5 high scores which are read from the file and the top 5 personal best scores for that username which is also read in from that associated file. Lastly, there is the EncryptAndDecrypt script which as the name suggests, it will encrypt and decrypt the player information file using RSA encryption. This will be described in more detail later in the Encryption section of this paper.

**Modularity**

Modularity is a way to decompose a project into cohesive and loosely coupled components [5]. In my final project game, I incorporated modularity through the various scripts shown above in the UML class diagram and the various functions of those scripts as well as how those scripts related to the game scenes. All the scripts incorporated in this project are loosely coupled meaning they are loosely connected through being broken apart by scene or tasks into simple and clear functionalities. This minimizes the connection between the elements of the project as those elements are completed in their own script and/or scene. Furthermore, there is high cohesion between the elements of the functionality modules meaning that the scripts and scenes implement a clearly defined function with the all elements in that function all connected and contributing to implement that singular function. This can be seen through how the different scripts all implement a different aspect of the game and the elements of that script all are dedicated to ensuring that the game remains playable.

**Inheritance**

A drawback to using C# is the fact that it does not support multiple inheritances through classes. When using Unity, the scripts that I created need to inherit from the base class MonoBehavior which is what every Unity script derives from[6] to be able to connect to the game and implement functionality. Due to this reason, the scripts used in my game solely inherit the properties, methods, operators, and actions from MonoBehavior and do not inherit from each other. This inheritance can be seen in the UML Class Diagram above where the ButtonController, GameOver, MainMenu, PlayerInfoInput, and GameManager all inherit from MonoBehavior because they all specifically connect to the Unity scenes. Without inheriting from MonoBehavior, these scripts would not be able to implement game functionality or game actions into the Unity game engine and thus the game would not be playable.

**Encapsulation**

Encapsulation hides the implementation details in a program. This is included in my game by declaring certain aspects as public where access is not restricted or as private where access is restricted to the containing type for the game. Because the scripts implemented in the game do not inherit from each other as mentioned above, all items including variables, functions, objects, and classes that I declared are private unless they need to be populated in the scene of the Unity game engine in which case I made them public. Certain items in the script need to be populated in the inspector pane of Unity so that it could have the functionality implemented by the code of that script. For example, the AudioSource for the “win” sound in the GameManager script must be public because it must be populated with the actual win.wav asset in Unity. All these public and private variables and methods can be seen in the UML Class Diagram shown above in which they are separated based on their encapsulation. From this diagram, it can be seen that only the public members are accessed by the Unity game engine or from other methods whereas the private members are only needed to be seen by that method they are in.

**Polymorphism**

Polymorphism is the ability to take more than one form or where objects have more than one type. This was incorporated into my game by using Unity sprites as square button objects. By doing this, more functionality could be applied to those sprites to implement game functionality. This includes adding box colliders to detect when the square was clicked, adding all the squares into a list of objects that would implement the game pattern sequence, and being able to access each square when it needed to be randomly lit up for the game to function. The main classes that incorporated this polymorphism was the ButtonController and GameManager class which can be seen in the UML Class Diagram above. This shows how the square button objects are stored as a Unity SpriteRenderer and added to a list of colors.

**Abstraction**

Because C# uses .NET and is a component object-oriented programming language, it uses exchangeable code modules that work on their own without knowing the inner workings to use them. With that being said, I used abstraction through the usage of interfaces in which the knowledge of what is in the implementation is not required in order to use it. An interface is a set of operations or methods that a given object can perform therefore the following C# extended classes under certain namespaces were used in my game as part of interface abstraction.

C# Extended Classes [7]:

* System.Security.Cryptography.RSACryptoServiceProvider: Performs asymmetric encryption and decryption by implementing the RSA algorithm provided by the cryptographic service provider.
* System.Security.Cryptography.RSAParameters: Includes the standard parameters for the RSA encryption algorithm.
* System.Text.UnicodeEncoding: Represents the UTF-16 Unicode characters encoding
* UnityEngine.SceneManager: Scene management at run-time.
* UnityEngine.UI.InputField: Implements a label as an interactable input field
* System.IO.StreamWriter: Implements the ability to write characters to a stream in a particular encoding.
* UnityEngine.AudioSource: Represents an audio source for a game object.
* UnityEngine.Application: Accesses an application’s runtime data.
* System.IO.File: Provides the methods used on a file.
* UnityEngine.UI.Text: Creates font data on the game screen.
* UnityEngine.SpriteRenderer: Renders a sprite for 2D graphics.
* Color.Color: Constructs a new color with given r, g, b, and a components.
* UnityEngine.PlayerPrefs: Stores and accesses player preferences between game sessions.
* UnityEngine.Time.deltaTime: The completion time in seconds since the last frame.
* UnityEngine.Random.Range: Returns a random float number between the declared minimum and maximum.

**Exception Handling**

Similarly to other languages, C# uses try-catch blocks for exception handling. In this method, the try clock identifies a block of code for which the particular exception is activated and it is followed by a catch block which catches the exception and handles the problem accordingly. For my game, I used the try-catch blocks for file handling with the exceptions FileNotFoundExceptin and IOException for when the file could not be created and thus is not found and for when the file could not be written to or read from. I also used try-catch blocks for the encryption and decryption with the exception CryptographicException which would catch any problems with the actual encryption and decryption process.

**Output Files**

There are four different output files that can be found in the “OutputFiles” folder in the root directory of the folder for the game.

* HighScores.txt: This file is continually updated with the high score each time the game played. This file will be read to find the top 5 high scores to be displayed on the GameOver scene.
* UsernamePersonalBest.txt: This file is created each time a player plays the game with a new username. If the player uses the same username as before, the file will just be updated with the personal best scores. When the game is over, the associated personal best file for that username will be read to find the top 5 personal best scores to be displayed on the GameOver scene.
* nameOfThePlayer\_MM\_dd\_yyy.txt: This file will be created for each time the game is played and is saved with the name of the player and the date the game was played. This file will be encrypted unless the commented lines in the EncryptAndDecrypt script for decryption are uncommented. The file will have the following format:

Name: xxxx

UserName: xxxx

Date Of Birth: xxxx

Address (City, State, Country): xxxx

Diagnosis: xxxx

Date Played: xxxx

Game Square Sequence:

Tapped Correctly: Button xx

Tapped Correctly: Button xx

Tapped Correctly: Button xx

…

Tapped Wrong: Button xx

* nameOfThePlayer\_MM\_dd\_yyyDECRYPTED.txt: This file will only be created if the game is running in decryption mode meaning that the commented lines in the EncryptAndDecrypt script for decryption are uncommented. If that is the case, then the file above will be decrypted and saved into this file.

**Encryption and Decryption**

In my game, I used the RSA cryptographic algorithm to encrypt and decrypt my files which is derived from the standard .NET RSA class. The RSA algorithm stands for Rivest, Shamir, and Adleman which are the individuals who developed that algorithm in 1977 [8]. The RSA algorithm is an asymmetric cryptographic algorithm meaning it uses a public key and a private key. The public key is used to encrypt the plain text and the private key is used to decrypt the encrypted text back into plain text. This public/private key pair is generated through the .NET framework RSACryptoServiceProvider. This encryption and decryption is done through the EncryptAndDecrypt class which can be seen in the UML Class Diagram above.

**References**

1. Understanding the Differences Between C#, C++, and C [Internet]. 2018 May 17 [cited 2019 December 9]; Available from: https://csharp-station.com/understanding-the-differences-between-c-c-and-c/
2. Garbage Collection in C# | .NET Framework [Internet]. [cited 2019 December 9]; Available from: https://www.geeksforgeeks.org/garbage-collection-in-c-sharp-dot-net-framework/
3. Agarwal, Tushar. Memory Management in .NET [Internet]. 2003 November 17 [cited 2019 December 9]; Available from: https://www.c-sharpcorner.com/article/memory-management-in-net/
4. Early and late binding in C# [Internet]. [cited 2019 December 9]; Available from: https://www.geeksforgeeks.org/early-and-late-binding-in-c-sharp/
5. El Assas, Akram. Modular Software Development In C# [Internet]. 2019 July 24 [cited 2019 December 9]; Available from: https://www.codeproject.com/Articles/5163072/Modular-Software-Development-In-Csharp
6. MonoBehaviour [Internet]. [cited 2019 December 9]; Available from: https://docs.unity3d.com/ScriptReference/MonoBehaviour.html
7. .NET API Browser [Internet]. [cited 2019 December 9]; Available from: https://docs.microsoft.com/en-us/dotnet/api/?view=netframework-4.8
8. Somani, Sourabh. RSA Algorithm With C# [Internet]. 2013 December 7 [cited 2019 December 9]; Available from: https://www.c-sharpcorner.com/UploadFile/75a48f/rsa-algorithm-with-C-Sharp2/