Assignment 2 Visualizing the Piano Simulation

# Objectives

* Learn the MonoGame framework
* Reuse existing code and library from a previous project
* Separate game logic from simulation logic

# Instructions

In this project, a MonoGame visualization of the Piano Simulation from assignment 1. The graphical user interface (GUI) will show a piano keyboard, where notes pressed on the keyboard are reflected on the onscreen keyboard. Additionally, when pressing a note, the sound will play, and the corresponding note letter will be shown on screen.

A picture containing text, music, piano

Description automatically generated

Figure Possible piano keyboard design, showing the note D being played

## Setting up the Project

Create a new mono game project using ‘dotnet new mgdesktopgl --framework netcoreapp3.1’, called InteractivePiano. Note a mgdesktopgl project is a cross platform application that uses OpenGL and should run on both Windows and Linux.

Create a new class called InteractivePianoGame that derives from Game. Note, the template might create a Game class that you can rename instead. Inside Main in Program.cs invoke your InteractivePianoGame.

Copy your existing PianoSimulation and PianoSimulationTests projects. Add a reference to the PianoSimulation project in your InteractivePiano project.

## Modifying Audio.cs

The previous Audio.cs file used in Assignment 1 will be reused in Assignment 2 to play sound. Since there should only be a single instance of the WaveOutEvent and it needs to be correctly disposed at the end of the program, the Audio class needs to be modified. Additionally, some helper methods will be added to allow to work with Tasks.

Make the following changes to the Audio class:

1. Transform Audio into a Singleton, where only a single instance of Audio is allowed to exist at a time.
2. Define a new method in Audio called Reset
   1. This method should clear the \_bufferCount and clear the audio buffer in the \_bufferedWaveProvider
   2. This method will clear the current sound that is playing allowing a new sound to be played in its place
3. Have Audio implement the IDisposable pattern
   1. Since Audio is now a singleton, Dispose is a good place to clean up the singleton
   2. Additionally, the BufferedWaveProvider and WaveOutEvent are unmanaged resources and need to be cleaned up
      1. Ensure you call Stop() on the WaveOutEvent and dispose of the objects

## Handling user input

Users will be able to interact with the game using their keyboard, like Assignment 1. Modify the InteractivePianoGame class to handle user input from the keyboard. The user should be able to press a key and hear the appropriate note to play. Reuse the Audio.cs class to play the sound.

## Creating the Virtual Piano

To visualize the virtual piano, sprites must be created to draw the piano keyboard on screen. Create a sprite that represents the white keys and black keys. Note that both white keys and black keys share common functionality, and a base class might be necessary. Recall that a MonoGame sprite needs a 2D texture. Use paint or another tool to create the required textures.

When a key is pressed, change the color of the key, and draw text on screen indicating the letter of the note.

Hint, it will be useful to keep a list of all the different sprites representing each piano key.

## Keep the GUI responsive

To ensure the GUI remains responsive, code running the Main thread (UI Thread) must be fast and efficient. Currently, when we play a note, our PianoSimulation generates a series of doubles in a loop. While these doubles are being generated, the UI Thread cannot react to other user input.

To resolve this and ensure that the UI Thread is always ready to handle user input, place the work of generating the piano sound double and playing the note in Audio in a Task. You can use an anonymous Task for this.

Since playing the audio now occurs in a Task, the buffer in Audio.Play() must be protected as multiple tasks could be accessing it at once. Add a critical section to the Play method of the Audio class ensuring that only one Task at a time can access the buffer.

After making these changes, the user should be able to quickly press keys one after another and hear the different notes, without crashing the Audio class.

# Deliverable

Create a private Gitlab repo to track your code. Add your instructor as a maintainer to the repo (section 1 swetha411, section 2 ddubois1). When developing, feel free to use the feature branch model.

The deliverable will consist of a Merge Request (Pull request) from your main branch to a branch called Submission. Note, in this case you are merging from main into the Submission branch you have created. There must be no commits on the Submission branch, and the change list should show all the code you have modified in main. Submit the URL for the Merge Request to Moodle.

Proper code standards should be followed when developing the solution. However, students have the freedom to design graphics and alter the presentation of the game as they see fit, provided it meets the basic requirements of the assignment.

# Resources

* 2D MonoGame graphics <http://rbwhitaker.wikidot.com/monogame-2d-tutorials>
* Locks in C# <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/statements/lock>
* IDisposable in C# <https://docs.microsoft.com/en-us/dotnet/api/system.idisposable?view=netcore-3.1>
* Using statement in C# <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/using-statement>
* Keyboard in MonoGame <https://docs.monogame.net/api/Microsoft.Xna.Framework.Input.Keyboard.html>
* Fonts in MonoGame <https://docs.monogame.net/articles/content/adding_ttf_fonts.html>