TRIBHUVAN UNIVERSITY
INSTITUTE OF ENGINEERING
PULCHOWK CAMPUS

VOXELOOP: MUSICAL LOOP GENERATION IN VOXEL WORLD

A COURSE PROJECT SUBMITTED TO THE DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE PRACTICAL COURSE ON DATA STRUCTURES AND ALGORITHMS [CT552]

Submitted by:
Prajwol Pradhan (PUL076BEI023)
Rujal Acharya (PUL076BEI029)

Submitted to:
Department of Electronics and Computer Engineering
Pulchowk Campus, Institute of Engineering,
Tribhuvan University
Lalitpur, Nepal

March 9, 2022

Abstract

The knowledge of data structures and algorithms is a must for any good programmer in order to make their program optimized and write some good programs. Not only proper knowledge of data structures and algorithms, but the skill to apply them to real projects is also equally necessary. So in order to showcase the knowledge gained during the whole semester, a small project is presented which shows how we could apply the knowledge gained. Different data structures like Stack, Vector, Linked List and Binary Search Tree were applied in order to create a simple 3D world, where one could play a music sample and loop it.

***Key Words**: Algorithms, Binary Search Tree, Data Structures, Linked List, Music, Stack, Vector*

Acknowledgements

First of all, we would like to extend our sincere gratitude to our instructor Mr. Prakash Chandra Prasad for his methodology and guidance while teaching us the theoretical aspect of the course. It really prepared us to work on this project. We would also like to thank the Department of Electronics and Computer Engineering for this opportunity to work on a project that would enable us to comprehend the subject matter of the course more easily.

While acknowledging everybody, we must not miss to thank the authors/collaborators of the different libraries that we have used in our project. The major ones are: GLFW (for OpenGL framework), Dear ImGui (for Graphical User Interface (GUI) and visual components), miniaudio (for audio support) and stb (for texture support). We are also grateful to the authors/collaborators of different software products that we have used to build our project. Similarly, we are thankful to the authors of various projects that we have taken reference from while writing our code. Without any of them we would not have been able to create this project in the form it is currently. There are many more people that have been a great source of inspiration and assistance for us, and we are equally grateful towards all of them.

Lastly, we would like to conclude by thanking our friends and seniors for their incredible support and feedback throughout the project. It was certainly very essential to keep our motivation up during the hard times while we were dealing with errors in our code as well as during the times of joy because something was working as expected.

Table of Contents

Abstract i

Acknowledgements ii

List of Figures iv

List of Abbreviations and Acronyms v

1 INTRODUCTION 1

1 Background . . . 1

2 Objectives . . . 1

2 FLOWCHART 2

1 Flowchart for the Main Program Flow . . . 2

2 Flowchart for Play State . . . 3

3 IMPLEMENTATION 6

1 Summary: . . . 6

2 State Machine . . . 7

3 Managers . . . 7

4 Looping of Cubes (Using Linked List) . . . 7

5 Looping of Audio (Using Binary Search Tree) . . . 8

6 Code . . . 8

4 RESULTS AND DISCUSSION 9

1 Results . . . 9

1.1 Screenshots of Voxeloop . . . 9

2 Limitations . . . 11

3 Further Improvements . . . 12

List of Figures

2.1 Flowchart for the main program loop 2
2.2 Flowchart for the Menu Screen (MenuState) 3
2.3 Flowchart for Play Screen (PlayState) 3
2.4 Flowchart for Flattening BST 4
2.5 Flowchart for Tree Traversal 5

4.1 Menu Screen 9
4.2 Play Screen 10
4.3 Settings Screen 10
4.4 About Screen 11

List of Abbreviations and Acronyms

GUI Graphical User Interface ii

BST Binary Search Tree 6

AVL Adelson-Velsky and Landis 8

Chapter 1

INTRODUCTION

1 Background

We wanted to make a program through which we can create and play musical loops using our keyboard as a launchpad.

First of all, we have to upload some beats – which are just some chops of music split (chopped) at certain timings – in the program. Then, we can play those beats with the assigned keys of keyboard.

The playing of the beats is to be visualized in a 3D world of voxels, which gives the user a unique visual experience of playing musical beats. Moreover, this can be looped and combined with more beats, which we refer to as musical loops.

2 Objectives

- To create a 3D world where one can render a voxel on keyboard button press with musical feedback.
- To create a system where we can loop the music played using appropriate data structure.
- To develop a program with multiple states (or screens), managed using a suitable data structure.
- To apply the knowledge gained about data structures and algorithms in a real project.

Chapter 2

FLOWCHART

1 Flowchart for the Main Program Flow

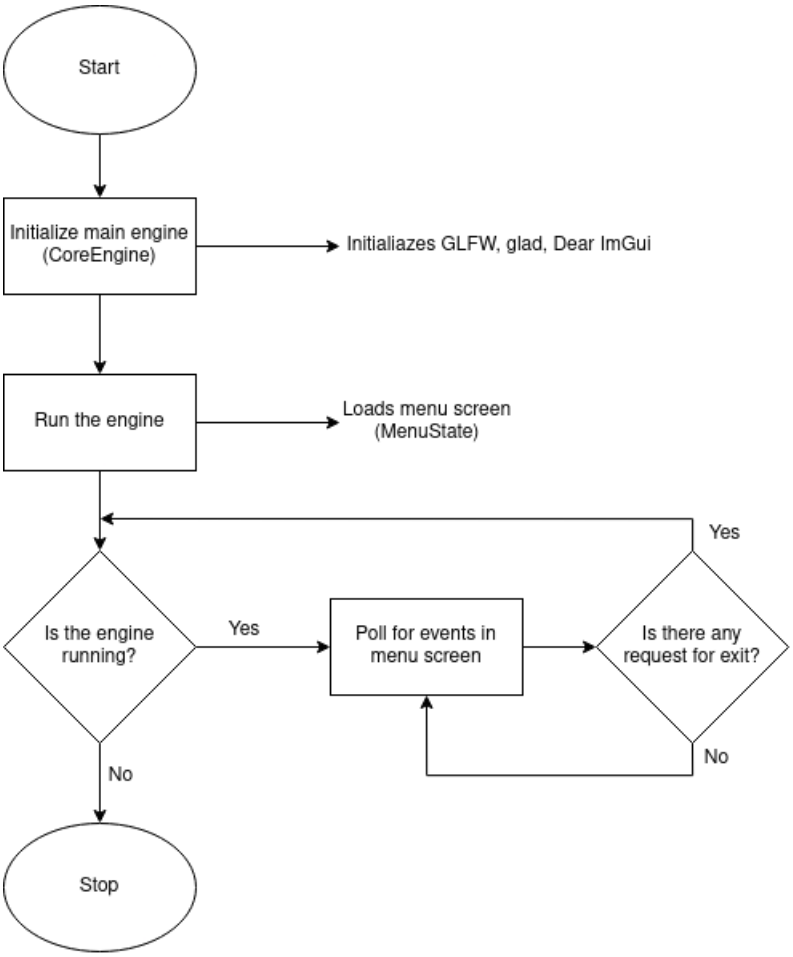

fig. 2.1: *Flowchart for the main program loop*

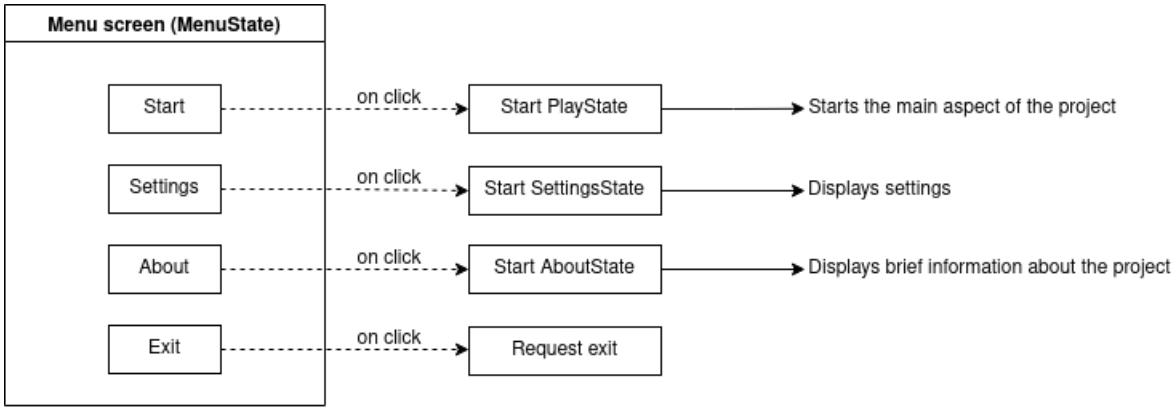

fig. 2.2: *Flowchart for the Menu Screen (MenuState)*

2 Flowchart for Play State

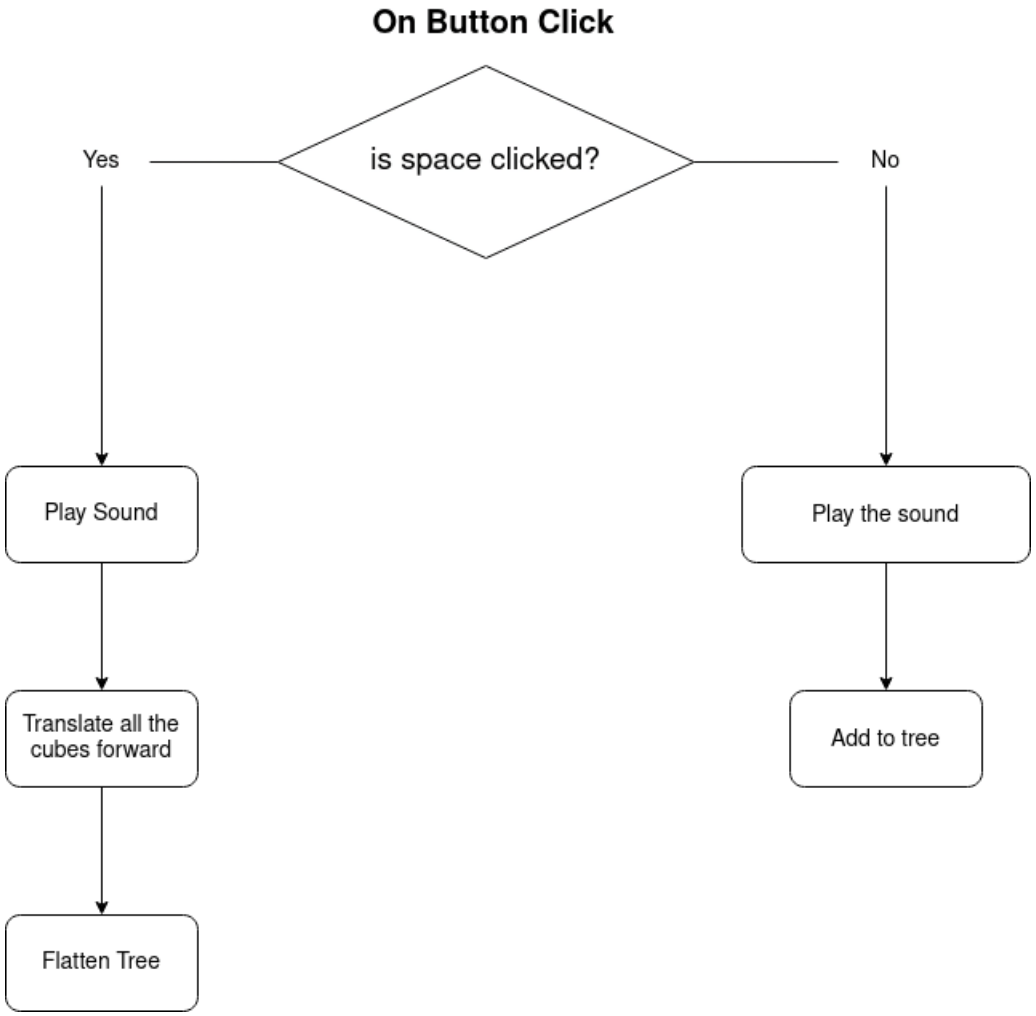

fig. 2.3: *Flowchart for Play Screen (PlayState)*

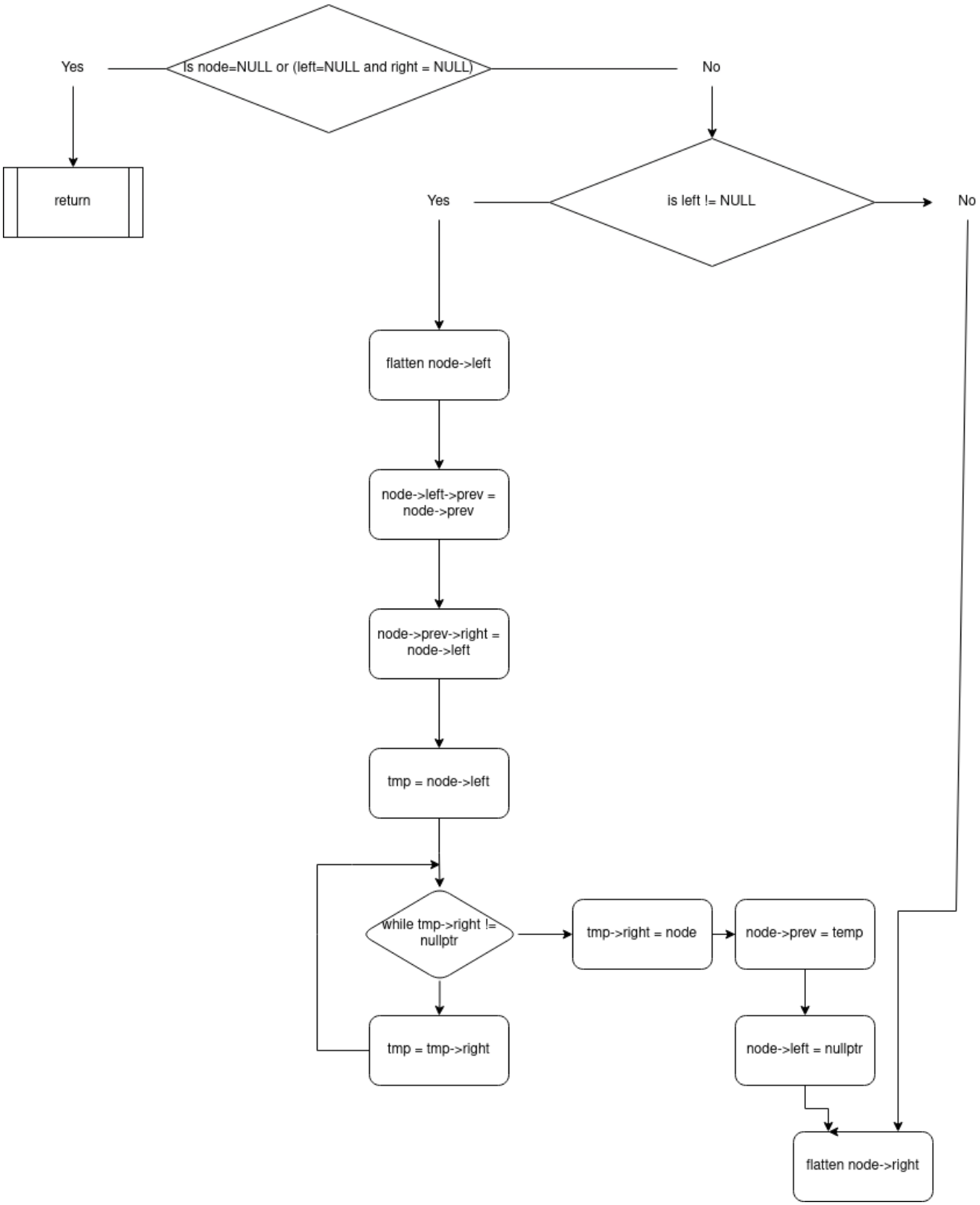

fig. 2.4: *Flowchart for Flattening BST*

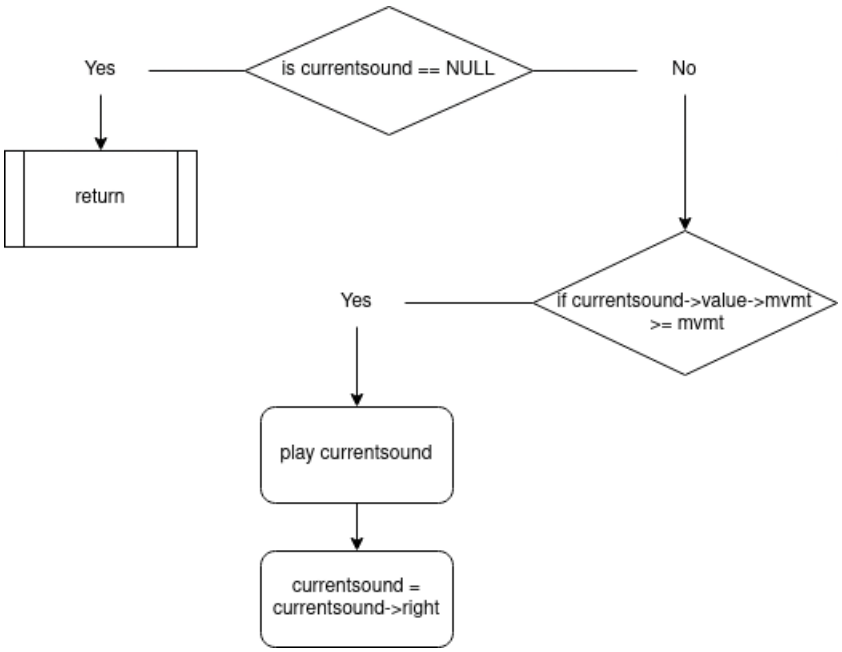

fig. 2.5: *Flowchart for Tree Traversal*

Chapter 3

IMPLEMENTATION

1 Summary:

Language used: C++
Major libraries used:

- GLFW: for OpenGL
- GLAD: to load OpenGL functions
- Dear ImGui: for GUI
- stb: for loading textures
- miniaudio: for audio support

Data Structures and Algorithms:

- Vector (in the form of Stack): to handle the state changes
- Linked List: to store and traverse cubes
- Map: to read audio data from file and map it to keyboard
- Binary Search Tree (BST): to store the audio data and loop it

Other noteworthy libraries/products used:

- CMake: for cross-platform build
- Doxygen: for documentation generation

2 State Machine

State machine, in the sense of this program, is a system designed to handle the states. The states refer to the screens, but all of them are inherited from a base state machine class. The states pass data between them using a shared pointer. The data can be anything from a 'window' to the assets. This architecture makes switching between the screens easier and sharing data between them more organized.

The underlying state machine uses `std::vector` in the form of 'stack' data structure and so it implements the mechanism of the stack.

The main program flow is shown by using a flowchart in fig. 2.1. This flowchart shows what happens as soon as the program starts running. The main part is the loop that keeps checking (polling) for any events in the `MenuState`.

In the beginning of program (and after CoreEngine initializes), the state is changed to `MenuState`. In `MenuState`, if any button but the '`EXIT`' button is pressed, the respective state is pushed onto the stack. Then immediately the state at the top of the stack initializes. Now in the `draw` function of each state everything that is to be displayed on the screen is drawn. When the '`EXIT`' button is pressed, the program logic requests the window manager to close and exit the program.

The flowchart in fig. 2.2 shows the flow of actions in the `MenuState`.

3 Managers

In order to simply window/context management, GUI management, GUI draw events, audio management, etc., different types of managers have been implemented. There is `WindowManager` to manage the window creation and destruction. It also initializes the libraries GLFW and GLAD. The other manager is the `GUIManager`, which initializes the GUI in the program. This manager is responsible for drawing the visual elements in different screens. However, the voxels in `PlayState` are handled by a different set of managers that render cubes. Similarly, there is `Audio` to manage initializing, loading, playing and destroying anything related to audio.

4 Looping of Cubes (Using Linked List)

In order to render cubes, a linked list is implemented. When a new cube is rendered on keyboard input, it is immediately added to a linked list. In every frame, the linked list is traversed and the cubes are drawn in the window. The window to viewport clipping of the cubes is handled by OpenGL itself.

5 Looping of Audio (Using Binary Search Tree)

The core of the program i.e looping of the audio is implemented using binary search tree. On adding a new node, it is also added in the binary search tree based on its `movement`, which is a value given to it based on its position in the 3D world.

When the audio is looped, the BST thus formed is flattened. This flattened BST gives resemblance to the inorder traversal of the BST which is then added to a doubly linked list. Now, the flattened list is traversed based on the timing of the audio which gives the user feeling of looping of audio clip exactly from where they had been placed in the 3D world.

The inclusion of audio position in BST and then inorder traversal of the tree simply is an algorithm for tree sort. The time complexity of tree sort is $O(n^2)$. If we had used self balancing trees like Adelson-Velsky and Landis (AVL) Tree or Red Black Tree, the time complexity could be reduced to $O(nlog(n))$. But since we need to flatten the tree afterwards, balancing it would be meaningless and waste of resources. Similarly, in our case, most of the nodes have left node as null, the process of flattening to the right side would be faster than balanced trees whose both left and right nodes are occupied.

6 Code

The code for this project is available in GitHub:
https://github.com/jarp0l/Voxeloop

The doxygen generated documentation of this project is available at:
https://jarp0l.github.io/Voxeloop

Chapter 4

RESULTS AND DISCUSSION

1 Results

After days of work, the project is finally ready for submission. However, there still exist room for improvement not just in logic, but in design of the whole system as well.

1.1 Screenshots of Voxeloop

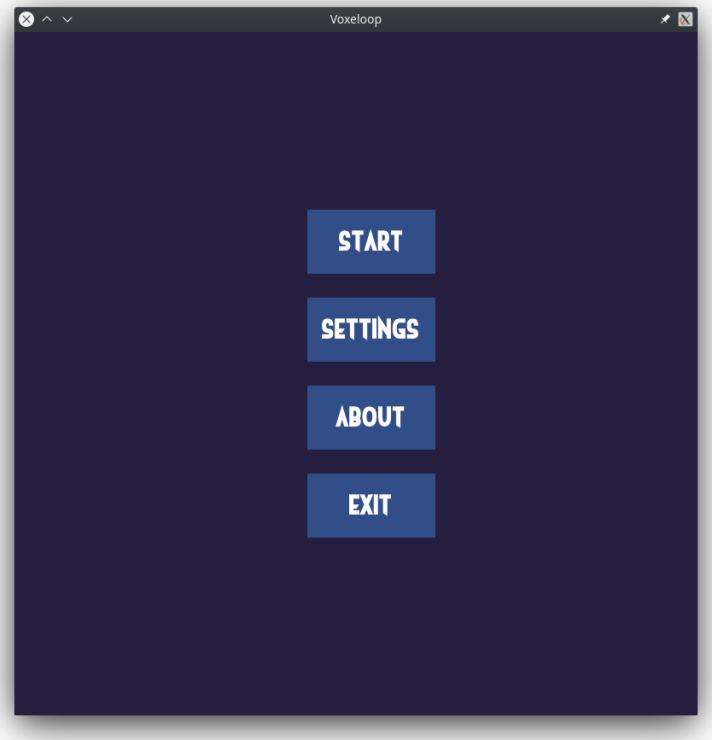

fig. 4.1: *Menu Screen*

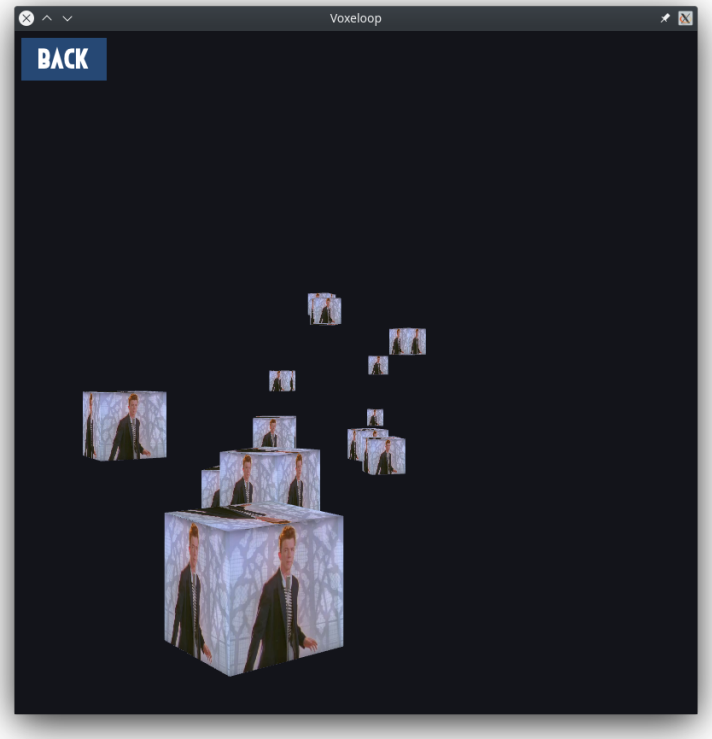

fig. 4.2: Play Screen

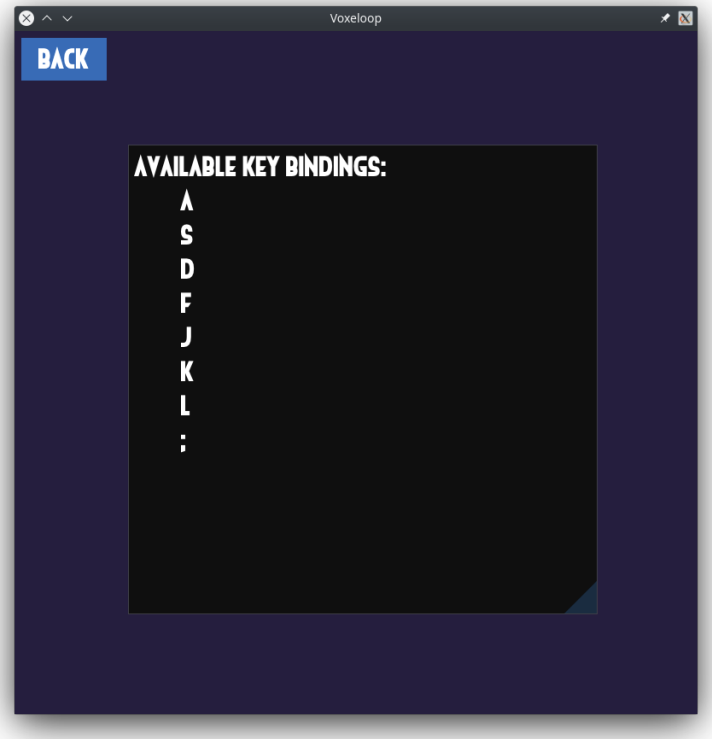

fig. 4.3: Settings Screen

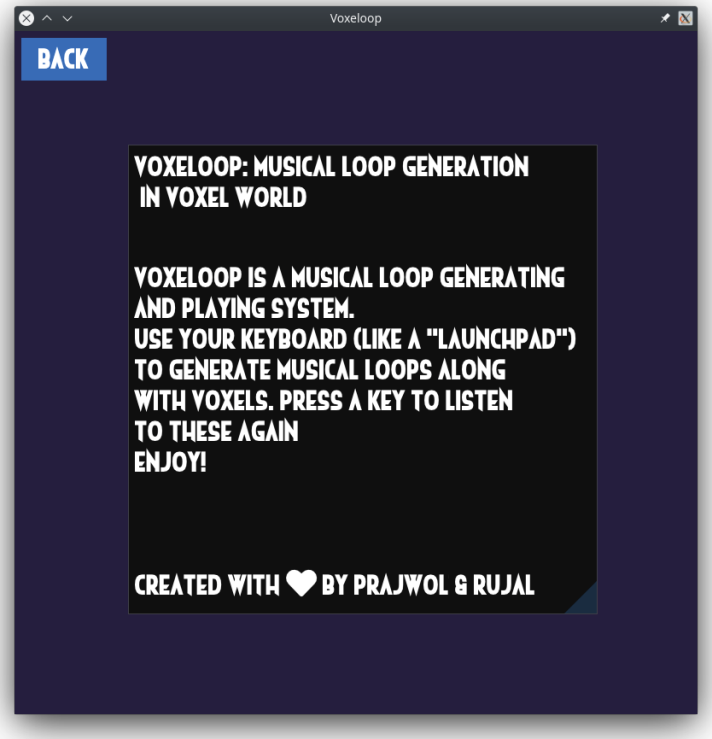

fig. 4.4: *About Screen*

2 Limitations

As discussed earlier, there are some limitations in the program as of now. Some of them are noted below:

- The beats have to be configured manually and the program has to be compiled every time after doing so.
- Other aspects also need to be changed manually and then have to be compiled to reflect the changes; there is no GUI for that.
- The voxels have just an image, which makes them look not so visually pleasing.
- The manually developed 'stack' data structure is not as much efficient as the well-tested ones (such as `std::stack`).
- Only the audio files that have the extension ".wav" can be used for now.
- The camera is static for now.

3 Further Improvements

- A computer-readable file format such as JSON can be implemented to easily change settings/configurations and store them for later, which also avoids the hassle of compiling the program every time a change is made.
- The voxels can display some solid color and incorporate lighting and shading to make them look better visually.
- The manually developed 'stack' can be further optimized taking reference of how the standard data structure (`stack`) has been designed.
- Support for more audio file extensions can be added.
- Camera movement support can be implemented.