NODE BASED IMAGE EDITOR

A PROJECT REPORT

SUBMITTED IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR

THE AWARD OF THE DEGREE OF

BACHELOR OF TECHNOLOGY

IN

MATHEMATICS AND COMPUTING

SUBMITTED BY :

ADITYA HARSH (2K20/MC/012)

ANNU YADAV (2K20/MC/024)

UNDER THE SUPERVISION OF :

DR. GOONJAN JAIN

DEPARTMENT OF MATHEMATICS AND COMPUTING

DELHI TECHNOLOGICAL UNIVERSITY (Formerly Delhi College of Engineering)

Bawana Road, Delhi-110042

NOVEMBER 2021

DELHI TECHNOLOGICAL UNIVERSITY
(Formerly Delhi College of Engineering)
Bawana Road, Delhi-110042

CANDIDATE'S DECLARATION

We ADITYA HARSH 2K20/MC/12 and ANNU YADAV 2K20/MC/24 hereby declare that the work presented in this project titled "Node Band Image Editor", submitted to the B.Tech. (MCE) Delhi Technological University, Delhi for the award of the *Bachelor of Technology* degree in (MCE) , is our original work and done under the guidance of DR GOONJAN JAIN.

PLACE:DELHI

ADITYA HARSH 2K20/MC/12

ANNU YADAV 2K20/MC/24

DELHI TECHNOLOGICAL UNIVERSITY
(Formerly Delhi College of Engineering)
Bawana Road, Delhi-110042

ABSTRACT

Using the concepts of Graph theory to build an image editor with basic and intermediate level of functionalities to read, write and view the image, apply various photo editing changes and convert to different image formats. Each node in the graph represents an image editing functionality and the edge represents flow of image as a data on which operations are being applied.

But the outstanding feature of the editor which makes it stand out is its ability to preserve each operation individually.

CERTIFICATE

I hereby certify that the project dissertation titled “Node Based Image Editor ” which is submitted by ADITYA HARSH (2K20/MC/12) and ANNU YADAV (2K20/MC/24) in MCE, Delhi Technological University, Delhi in partial fulfilment of the requirement for the completion of the third semester of their degree, is a record of the project work carried out by the students under my supervision. To the best of my knowledge, this work has not been submitted in part or full for any other project

Place: Delhi

Dr. Goonjain Jain

DELHI TECHNOLOGICAL UNIVERSITY
(Formerly Delhi College of Engineering)
Bawana Road, Delhi-110042

OBJECTIVE

To build a node-based image editor utilising graph theory where each node represents an image operation.

MOTIVATION AND INTRODUCTION

There are two radically different digital compositing workflows: node-based compositing and layer-based compositing. Node-based image compositing represents an entire composite as a directed acyclic graph, linking media objects and effects in a procedural map, intuitively laying out the progression from source input to final output. This type of compositing interface allows great flexibility, including the ability to modify the parameters of an earlier image processing step "in context" (while viewing the final composite). However, only video editing softwares are available commercially which utilise this. Although, node-based compositing has some dis-advantages as well, but only in video editing. This makes it a perfect candidate for use in image editing.

Every operation is preserved individually and is editable at any point in time.

DISCRETE MATHEMATICS CONCEPTS

Graph Theory

In mathematics, graph theory is the study of graphs, which are mathematical structures used to model pairwise relations between objects. A graph in this context is made up of vertices (also called nodes or points) which are connected by edges (also called links or lines).

Directed Acyclic Graph

In mathematics, particularly graph theory, a directed acyclic graph (DAG) is a directed graph with no directed cycles. That is, it consists of vertices and edges (also called arcs), with each edge directed from one vertex to another, such that there is no way to start at any vertex v and follow a consistently-directed sequence of edges that eventually loops back to v again. Equivalently, a DAG is a directed graph that has a topological ordering, a sequence of the vertices such that every edge is directed from earlier to later in the sequence.

Depth First Search(DFS)

Depth-first search is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking. So the basic idea is to start from the root or any arbitrary node and mark the node and move to the adjacent unmarked node and continue this loop until there is no unmarked adjacent node. Then backtrack and check for other unmarked nodes and traverse them.

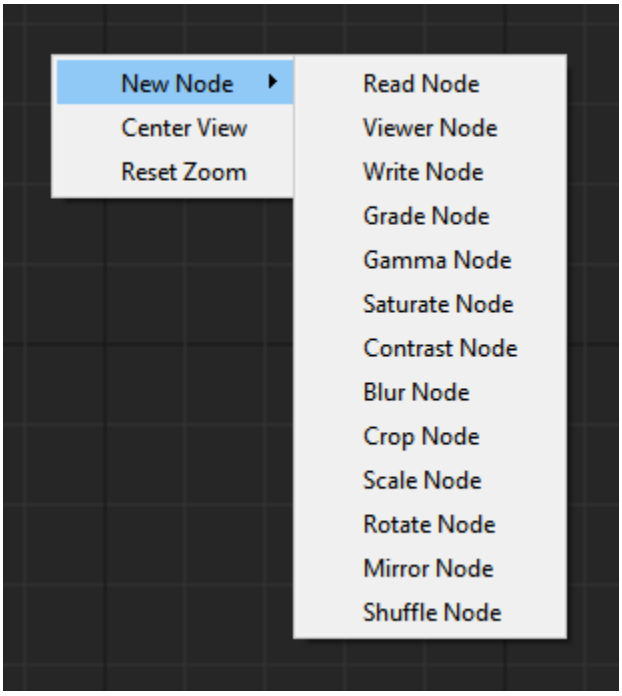
TOOLS AND TECHNOLOGIES USED

We are using Qt Framework For C++.

Qt is a free and open-source widget toolkit for creating graphical user interfaces as well as cross-platform applications that run on various software and hardware platforms such as Linux, Windows, macOS, Android or embedded systems with little or no change in the underlying codebase while still being a native application with native capabilities and speed.

Simple Usage Tips:

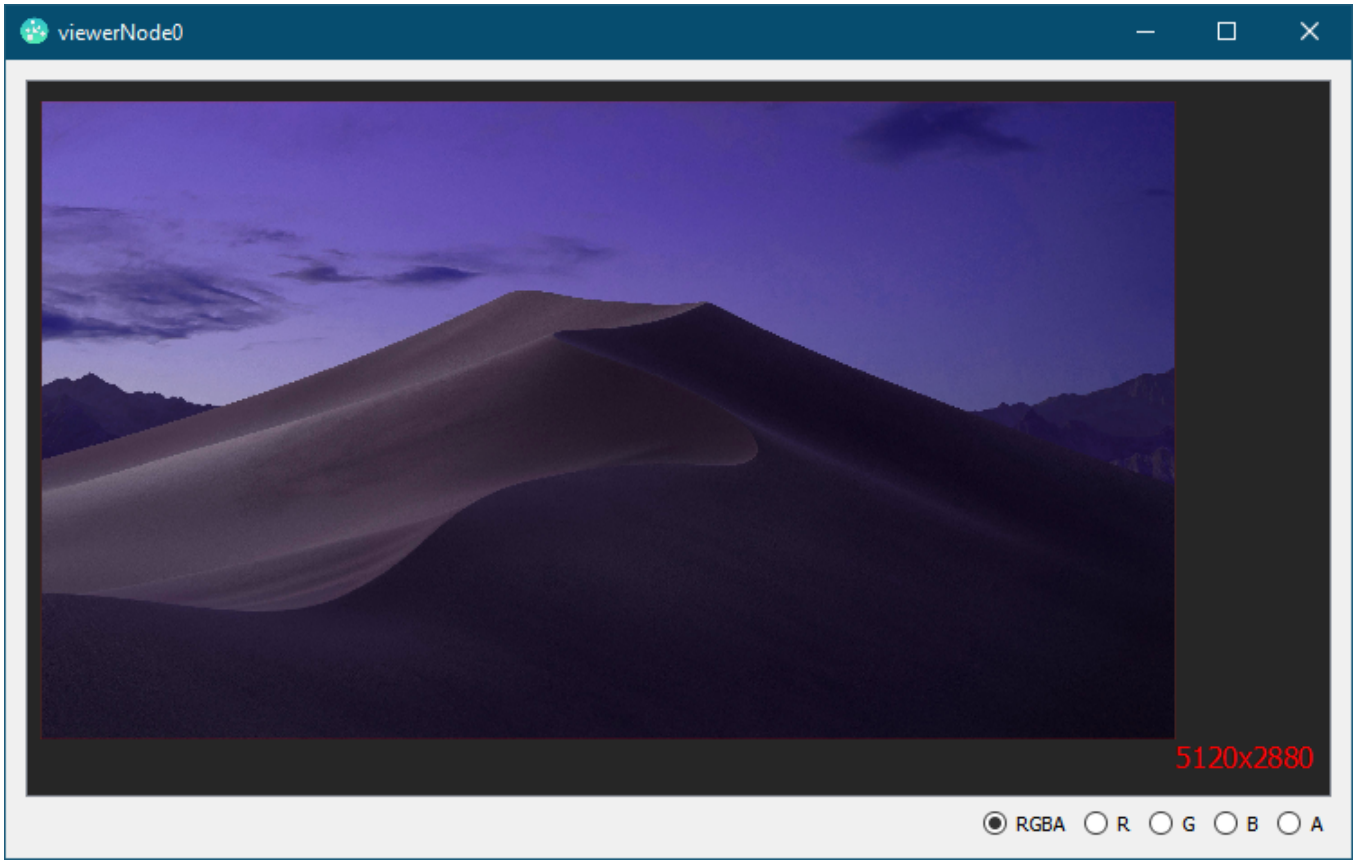
- Right click on the empty canvas and go to the 'New Node' menu to create that type of node.



- Once the nodes are created, right click on each node's connector (a small white box) to connect the input and output of various nodes sequentially. Right-click for creating edges, middle button for deleting edges. In the middle of connecting the 2nd node, press right-click on the canvas to cancel the connection.
- Right-click on each node for properties of each node namely the properties window and delete the node.

Node Functionalities In Brief:

- Read node: Read the image file from your system.
- Viewer node: Provides a viewer window to view the image stream. The viewer window also gives the ability to view the individual colour channels. 2
- Write node: To save the modified image with desired quality factor.



All the other nodes are image transformation nodes.

- Grade node: Used to define the blackpoints and whitepoints of the image.
- Gamma node: Applies a constant gamma value to the image. This affects the mid-tones of the image.
- Saturate node: Alters the intensity or purity of a color as displayed in an image. The higher the saturation of a color, the more vivid and intense it is. The lower a color's saturation, the closer it is to pure gray on the grayscale.
- Contrast node: To alter the difference in brightness between objects or regions. High contrast photos pop out, show textures in the subject and give a feeling of edginess, high energy and strength.
- Blur node: Adds box blur to the image with the specified radius.
- Crop node: Crops the image to the specified coordinates.
- Scale node: Scales the image to the specified resolution.
- Rotate node: Rotates the image with the specified angle in degrees.
- Mirror node: Mirrors the image across horizontal as well as vertical axis
- Shuffle node: Shuffle node rearranges different color channels of the image into other color channels.

PREREQUISITES

A color can be represented as a combination either:
1)Red, green and blue colors
2)Hue, saturation, and value
This can be easily implemented by storing 3 values and creating appropriate functions. However, this functionality is already provided by the Qt framework, by providing a class QColor which also deals with inter-conversion between these 2 types. We will later see in saturation node, how this conversion will help. Apart from either triplet, another channel is required named as alpha which defines transparency. Our editor can read alpha channel but doesn't allow its direct modification due to the limited time available for the project. Our editor works in 8-bit images which allocate 8 bit for each channel i.e 32 bits per pixel. Although, it can read higher depth images, it will convert them down to 8-bit images.

An image can be represented as a 2D matrix of pixels. This can be easily implemented by storing a 2D array and creating appropriate getters and setters. However, this functionality is provided by the Qt framework by providing QImage class.

IMPLEMENTATION

A class hierarchy has been setup for the graph nodes. Firstly, a node class is created which deals with both the graphics and functionality part of a graph node. In graphics, it deals with the shape, colors, text, edges between nodes, etc. In functionality, it deals with how information is traversed through the graph edges. All the specialized nodes inherit this node class and overrides the necessary virtual functions to implement its functionality. There are 2 types of depth first search traversals involved for traversal of information across the graph, first on the graph and second on the transpose of the graph. Whenever, a change is done on a node, either changes to its linkage or its properties, a DFS traversal starts from this node. This traversal end when either the traversal reaches a node with 0 out-degree or some specific nodes which store their own copy of image(viewer, write, crop and scale nodes). The aim of this traversal is to inform the connected nodes that a change has been done and the changes need to be updated. The nodes mentioned above keep a copy of an image which represents the result up to that node. Thus changes done before that node must be reflected in those copies of the image. Whenever the DFS traversal reaches these specific nodes, DFS traversal on the transpose of the graph starts from that node. This is another type of traversal which is done in order to actually update the copies of the images stored. This traversal carries with itself the reference to the stored image, to change it. Whenever

this traversal on the transpose of the graph ends and returns back to the starting node, the old traversal which was suspended continues further. How this functionality is implemented with code is discussed below.

node class:

This class defines all the graphics, based on the number of inputs and outputs which have to be specified in the constructor. It inherits QObject and QGraphicsItem classes both provided by the Qt framework. The discussion of classed provided by Qt framework is discussed at then end of the report. A virtual function refresh() has been defined in this class responsible for DFS traversal on the graph.

Pseudo code for refresh:

```
void refresh(){
  for all connected nodes: refresh();
}
```

As in a typical DFS traversal code, it calls the same function on all its connected vertices. Since, this function is virtual, it will be overridden in the children classes which

define the actions taken when the traversal reaches that node unlike nothing defined in the node class. Another virtual function imageCalcualte() is defined in this class which is responsible for the DFS traversal on the transpose of the graph.

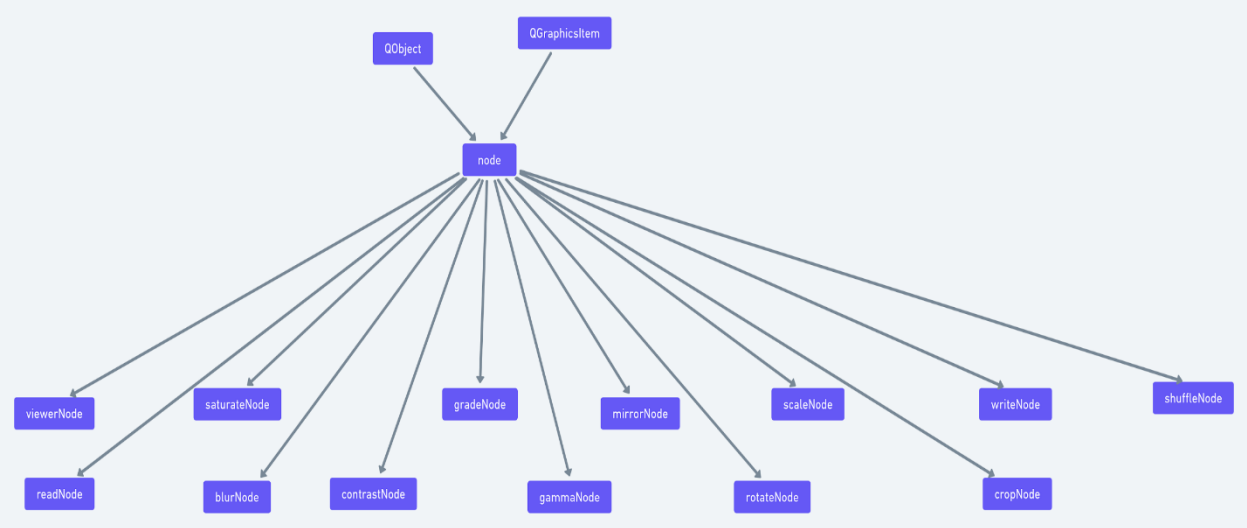
Pseudo code for imageCalculate():

```
Bool imageCalculate(image reference){
  Return false;
}
```

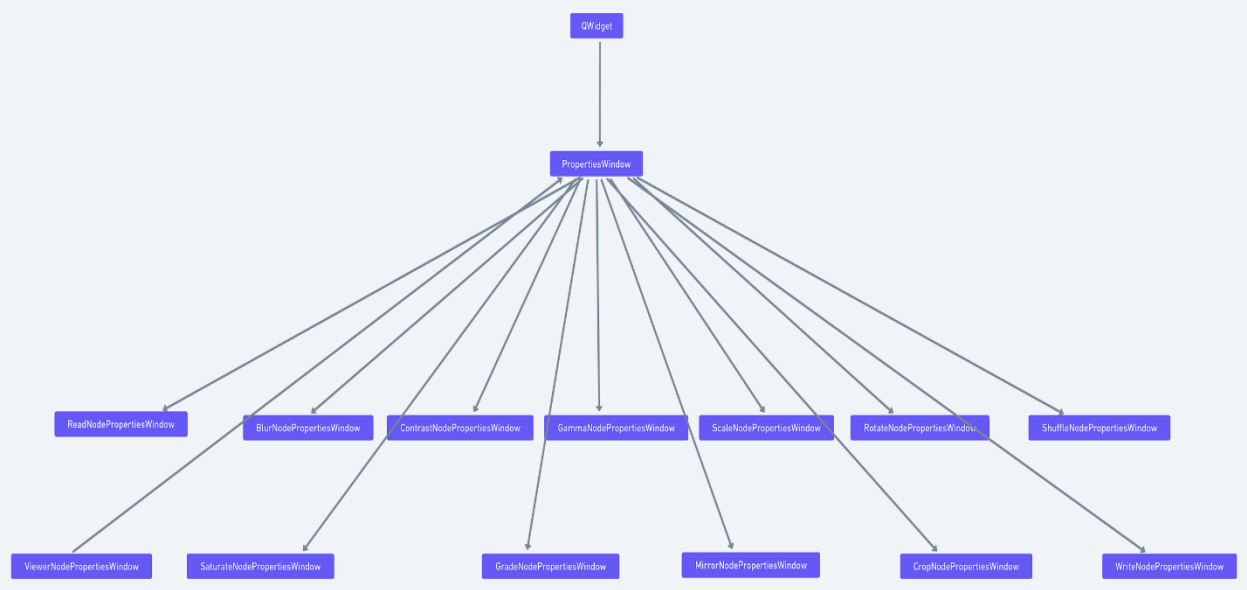
This function is supposed to call the 2nd typed of DFS traversal from this node along with reference to an image which is either received from another node or its own if it itself calls this function when it receives a refresh call. It is supposed to return a boolean value if the calculation procedure was successfully executed or not. As this is the base node class, this actual function is never called but the overridden functions defined in children classes are. It will be more clear below how the boolean value returned by this function influences.

Also, a node is supposed to have many properties which are specific to it which can be edited through a Properties Window whose pointer is also stored in the node and is created along with the node. propertiesWindow class is also created which is also inherited by windows of specific nodes. propertiesWindow class inherits QWidget class. All the types of nodes discussed in brief above are implemented be creating classes which inherit the node class. A class hierarchy is formed as a result for both node class and propertiesWindow.

node Class Hierarchy:



propertiesWindow Class Hierarchy:



readNode class:

This node is used for reading images from file. It overrides imageCalculate function as follows:

Pseudo code for imageCalculate():

```
bool imageCalculate(image reference){
  bool ans=image.load(filename);
image.convertToFormat(8-bit RGBA Image);
return ans;
}
```

This function now loads the image with the specified file and then converts it to 8-bit RGBA image and returns if the image was successfully loaded or not.

viewerNode class:

This node is responsible for letting the user view the image stream. It stores a copy of image which is supposed to be shown to the user. It has an updateViewer() function which as the name suggests updates the viewer window and overridden refresh() and imageCalculate() functions.

updateViewer() updates the viewer by setting only the checked channels in the viewer window in the image. It also adds resolution of the image if overlay is toggled.

Pseudo code for refresh():

```
void refresh(){
  imageCalculate(image reference);
}
```

refresh() in viewerNode only calls the imageCalculate function with the reference to the image stored in

viewerNode. No further calls are required as viewerNode cannot have out-degree greater than 0.

Pseudo code for imageCalculate:

```
bool imageCalculate(image reference){
  if(input==NULL) return false;
  bool ans=input->imageCalculate(image);
  if(!ans) return false;
  updateViewer();
return true;
}
```

If the image is calculated successfully, the viewer is updated.

blurNode class:

This node allows users to blur the image by specifying the blur radius. imageCalculate function is overridden.

Pseudo code for blur:

```
void blur(image reference, radius){
  for each pixel in image:
    r=0,g=0,b=0,num=0;
    for each position in square of side 2*radius+1:
      r=r+r';
      g=g+g';
      b=b+b';
      num=num+1;
  r=r/num;
  g=g/num;
  b=b/num;
  setPixel(r,g,b);
}
```

Pseudo code for imageCalculate:

```
bool imageCalculate(image reference){
  if(input==NULL) return false;
  bool ans=input->imageCalculate(image);
  if(!ans) return false;
  blur(image,radius);
  return true;
}
```

saturateNode class:

This node allows users to saturate the image.

imageCalculate function is overridden.

Pseudo code for saturate:

```
void saturate(image reference, value){
  for each pixel in image:
    if(value>=0) setSaturation(s+(1-s)value);
    else setSaturation(s(1-abs(value)));
}
```

Pseudo code for imageCalculate:

```
bool imageCalculate(image reference){
  if(input==NULL) return false;
  bool ans=input->imageCalculate(image);
  if(!ans) return false;
  saturate(image,value);
  return true;
}
```

We can now get a fair idea of how imageCalculate function overrides are modifying the image. It is redundant to discuss its override for each node from now.

contrastNode class:

Pseudo code for contrast:

```
void contrast(image reference, value){
  f=259(value+255)/255(259-c);
  for each pixel in image:
    setRed(f(r-128)+128);
    setGreen(f(g-128)+128);
    setBlue(f(b-128)+128);
}
```

gradeNode class:

Pseudo code for grade:

```
void grade(image reference,lift,gain,offset){
  for each pixel in image:
    setRed(r*(gain-lift)+lift+offset);
    setGreen(g*(gain-lift)+lift+offset);
    setBlue(b*(gain-lift)+lift+offset);
}
```

gammaNode class:

Pseudo code for gammaCorrect:

```
void gammaCorrect(image reference, gamma){
  for each pixel in image:
    setRed(255(r/255)^(1/gamma));
    setGreen(255(g/255)^(1/gamma));
    setBlue(255(b/255)^(1/gamma));
}
```

mirrorNode class:

Pseudo code for mirror_img:

```
Void mirror_imgH(image reference){
  for each (x,y) in image:
    swap(pixel(x,y), pixel(width-1-x,y));
}
```

Pseudo code for mirror_imgV:

```
Void mirror_imgV(image reference){
  Swap(pixel(x,y), pixel(x,height-1-y));
}
```

rotateNode class:

Rotation is implemented using the functions provided directly by the framework which utilises affine transformation. It also provides the option of using bilinear filtering implemented in the rotation function itself. QTransform class is provided by Qt framework which deals with a 3x3 matrix containing values for translation, scale, shear, rotate etc. We can set its rotate value to the one specified by the user. QImage class which we have used for implementing images has a function of applying a QTransform object to itself. This now results to a rotated image.

scaleNode class:

This is implemented using scaled function available for QImage class provided by the Qt framework.

cropNode class:

This is implemented using copy function available for QImage class provided by the Qt framework.

IMPORTANT CLASSES USED

1. QGraphicsScene - Provides a surface for managing a large number of 2D graphical items.
2. QGraphicswidget is the base class for all widget items in a QGraphicsScene.
3. QImage - Provides a hardware-independent image representation that allows direct access to the pixel data, and can be used as a paint device.
4. QCharts - Manages the graphical representation of the chart's series, legends, and axes. (QScatterSeries and QValueAxis are some other classes)
5. QColor - Class provides colors based on RGB and HSV values.
6. QMenu - Provides a menu widget for use in menu bars, context menus, and other popup menus.
7. QApplication class - Manages the GUI application's control flow and main settings.
8. The QGraphicsView class provides a widget for displaying the contents of QGraphicsScene.

Future Scope

There is plethora of different types of image modifications that can be done. The code base was written taking into consideration that a node can have more than 1 input. Due to a limited time-frame, only nodes having 1 input have been implemented. Nodes such as shuffleCopy(shuffle node for 2 inputs), Graph Visualiser(for analysing output vs. input images graphically) can be implemented. Painting on an image can also be implemented. Sharpening of image, more types of blur such as gaussian blur and machine learning algorithms can also be implemented

BIBLIOGRAPHY

- Qt For Beginners
- Documentation

DEPLOYED

We have deployed our application for all operating systems(namely Windows, MacOS, Linux) users, which makes it publicly available to be used for everyone.