Chapter 1

Preamble

1.1 Introduction to Computer Graphics

Computer graphics are pictures and films created using computers. Usually, the term refers to computer-generated image data created with help from specialized graphical hardware and software. It is a vast and recent area in computer science. The phrase was coined in 1960, by computer graphics researchers Verne Hudson and William Fetter of Boeing. It is often abbreviated as CG, though sometimes erroneously referred to as computer-generated imagery (CGI).Some topics in computer graphics include user interface design, sprite graphics, vector graphics, 3D modelling, shaders, GPU design, implicit surface visualization with ray tracing, and computer vision, among others. The overall methodology depends heavily on the underlying sciences of geometry, optics, and physics. Computer graphics is responsible for displaying art and image data effectively and meaningfully to the user. It is also used for processing image data received from the physical world. Computer graphic development has had a significant impact on many types of media and has revolutionized animation, movies, advertising, video games, and graphic design generally

The term computer graphics has been used in a broad sense to describe "almost everything on computers that is not text or sound". Typically, the term *computer graphics* refers to several different things:

- the representation and manipulation of image data by a computer
- the various technologies used to create and manipulate images
- the sub-field of computer science which studies methods for digitally synthesizing and manipulating visual content, see study of computer graphics

Today, computer graphics is widespread. Such imagery is found in and on television, newspapers, weather reports, and in a variety of medical investigations and surgical procedures. A well-constructed graph can present complex statistics in a form that is easier to

understand and interpret. In the media "such graphs are used to illustrate papers, reports, theses", and other presentation material.

Many tools have been developed to visualize data. Computer generated imagery can be categorized into several different types: two dimensional (2D), three dimensional (3D), and animated graphics. As technology has improved, 3D computer graphics have become more common, but 2D computer graphics are still widely used. Computer graphics has emerged as a sub-field of computer science which studies methods for digitally synthesizing and manipulating visual content. Over the past decade, other specialized fields have been developed like information visualization, and scientific visualization more concerned with "the visualization of three dimensional phenomena (architectural, meteorological, medical, biological, etc.), where the emphasis is on realistic renderings of volumes, surfaces, illumination sources, and so forth, perhaps with a dynamic (time) component".

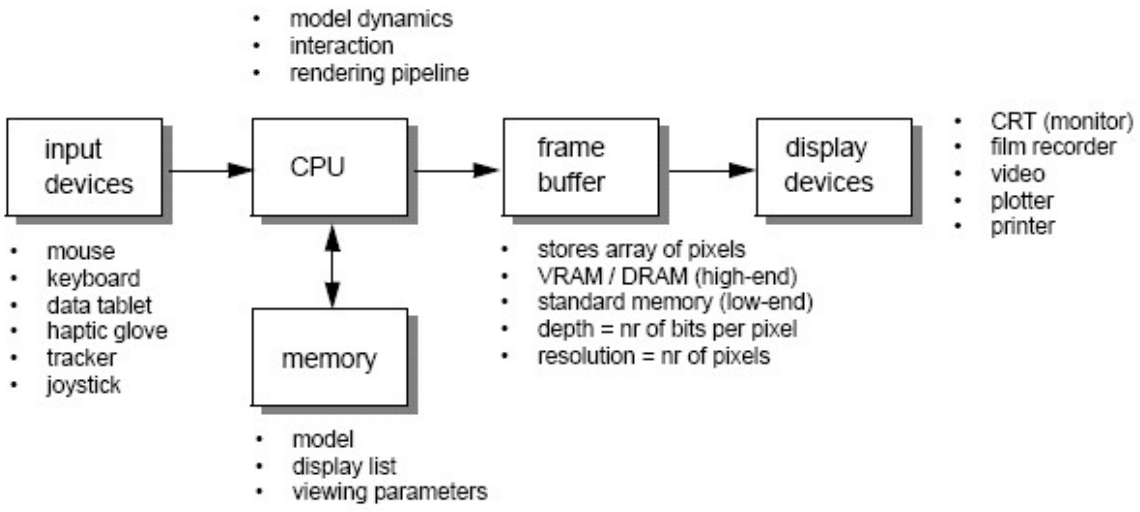

Figure 1.1: Graphics Pipeline

1.2 History of Computer Graphics

The phrase Computer Graphics was coined in 1960 by William Fetter, a graphic designer for Boeing. The field of Computer Graphics developed with the emergence of computer graphics hardware. Early projects like the Whirlwind and SAGE projects introduced the CRT as a viable display and interaction interface and introduced the light pen as an input device.

Further advances in computing led to greater advancements in interactive computer graphics. In 1959, the TX-2 computer was developed at MIT's Lincoln Laboratory. A light pen could be used to draw sketches on the computer using Ivan Sutherland's revolutionary Sketchpad software.

Also, in 1961 another student at MIT, Steve Russell, created the first video game, Spacewar! E. E. Zajac, a scientist at Bell Telephone Laboratory (BTL), created a film called "Simulation of a two-gyro gravity attitude control system" in 1963. In this computer-generated film, Zajac showed how the attitude of a satellite could be altered as it orbits the Earth. Many of the most important early breakthroughs in computer graphics research occurred at the University of Utah in the 1970s.

The first major advance in 3D computer graphics was created at UU by these early pioneers, the hidden-surface algorithm. In order to draw a representation of a 3D object on the screen, the computer must determine which surfaces are "behind" the object from the viewer's perspective, and thus should be "hidden" when the computer creates (or renders) the image.

Graphics and application processing were increasingly migrated to the intelligence in the workstation, rather than continuing to rely on central mainframe and mini-computers. 3D graphics became more popular in the 1990s in gaming, multimedia and animation. Computer graphics used in films and video games gradually began to be realistic to the point of entering the uncanny valley. Examples include the later *Final Fantasy* games and animated films like *The Polar Express*.

1.3 Applications of Computer graphics

The development of computer graphics has been driven both by the needs of the user community and by advances in hardware and software. The applications of computer graphics are many and varied. We can however divide them into four major areas

- Display of information: More than 4000 years ago, the Babylonians developed floor plans of buildings on stones. Today, the same type of information is generated by architects using computers. Over the past 150 years, workers in the field of statistics have explored techniques for generating plots. Now, we have computer plotting packages. Supercomputers now allow researchers in many areas to solve previously intractable problems. Thus, Computer Graphics has innumerable applications.
- Design: Professions such as engineering and architecture are concerned with design. Today, the use of interactive graphical tools in CAD, in VLSI circuits, characters for animation have developed in a great way.
- Simulation and animation: One of the most important uses has been in pilots' training. Graphical flight simulators have proved to increase safety and reduce expenses. Simulators can be used for designing robots, plan it's path, etc. Video games and animated movies can now be made with low expenses.
- User interfaces: Our interaction with computers has become dominated by a visual paradigm. The users' access to internet is through graphical network browsers. Thus Computer Graphics plays a major role in all fields.

1.4 Open Graphics Library (OpenGL)

OpenGL is a software interface to graphics hardware. This interface consists of about 150 distinct commands that are used to specify the objects and operations needed to produce interactive three-dimensional applications. OpenGL is designed as a streamlined hardware-independent interface to be implemented on many different hardware platforms. [2]

These are certain characteristics of OpenGL:

- OpenGL is a better documented API.
- OpenGL is much easier to learn and program.
- OpenGL has the best demonstrated 3D performance for any API.

The OpenGL specification describes an abstract API for drawing 2D and 3D graphics. Although it's possible for the API to be implemented entirely in software, it's designed to be implemented mostly or entirely in hardware.

In addition to being language-independent, OpenGL is also platform-independent. The specification says nothing on the subject of obtaining, and managing, an OpenGL context, leaving this as a detail of the underlying windowing system. For the same reason, OpenGL is purely concerned with rendering, providing no APIs related to input, audio, or windowing.

OpenGL is an evolving API. New versions of the OpenGL specification are regularly released by the Khronos Group, each of which extends the API to support various new features. In addition to the features required by the core API, GPU vendors may provide additional functionality in the form of *extensions*. Extensions may introduce new functions and new constants and may relax or remove restrictions on existing OpenGL functions. Vendors can use extensions to expose custom APIs without needing support from other vendors or the Khronos Group as a whole, which greatly increases the flexibility of OpenGL. All extensions are collected in, and defined by, the OpenGL Registry.

1.5 OpenGL Utility Library (GLU)

The OpenGL Utility Library (GLU) was a computer graphics library for OpenGL.It consists of a number of functions that use the base OpenGL library to provide higher-level drawing routines from the more primitive routines that OpenGL provides. It is usually distributed with the base OpenGL package. GLU is not implemented in the embedded version of the OpenGL package, OpenGL ES.

Among these features are mapping between screen- and world-coordinates, generation of texture mipmaps, drawing of quadric surfaces, NURBS, tessellation of polygonal primitives, interpretation of OpenGL error codes, an extended range of transformation routines for setting up viewing volumes and simple positioning of the camera, generally in more human-friendly terms than the routines presented by OpenGL. It also provides additional primitives for use in OpenGL applications, including spheres, cylinders and disks.

All GLU functions start with the glu prefix. An example function is gluOrtho2D which defines a two-dimensional orthographic projection matrix.

1.6 OpenGL Utility Toolkit (GLUT)

The OpenGL Utility Toolkit (GLUT) is a library of utilities for OpenGL programs, which primarily perform system-level I/O with the host operating system. Functions performed include window definition, window control, and monitoring of keyboard and mouse input. Routines for drawing a number of geometric primitives (both in solid and wireframe mode) are also provided, including cubes, spheres and the Utah teapot. GLUT also has some limited support for creating pop-up menus.

GLUT was written by Mark J. Kilgard, author of OpenGL Programming for the X Window System and The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics, while he was working for Silicon Graphics Inc.

The two aims of GLUT are to allow the creation of rather portable code between operating systems (GLUT is cross-platform) and to make learning OpenGL easier. Getting started with OpenGL programming while using GLUT often takes only a few lines of code and does not require knowledge of operating system–specific windowing APIs. All GLUT functions start with the glut prefix , for example, glutPostRedisplay marks the current window as needing to be redrawn.

The toolkit supports:

- Multiple windows for OpenGL rendering and callback driven event processing
- Sophisticated input devices
- An 'idle' routine and timers
- A simple, cascading pop-up menu facility
- Utility routines to generate various solid and wire frame objects
- Support for bitmap and stroke fonts
- Miscellaneous window management functions

Some of the more notable limitations of the original GLUT library by Mark Kilgard include:

- The library requires programmers to call glutMainLoop(), a function which never returns. This makes it hard for programmers to integrate GLUT into a program or library which wishes to have control of its own event loop. A common patch to fix this is to introduce a new function, called glutCheckLoop() (macOS) or glutMainLoopEvent() (FreeGLUT/OpenGLUT), which runs only a single iteration of the GLUT event loop. Another common workaround is to run GLUT's event loop in a separate thread, although this may vary by operating system, and also may introduce synchronization issues or other problems: for example, the macOS GLUT implementation requires that glutMainLoop() be run in the main thread.

- The fact that glutMainLoop() never returns also means that a GLUT program cannot exit the event loop. FreeGLUT fixes this by introducing a new function, glutLeaveMainLoop().
- The library terminates the process when the window is closed; for some applications this may not be desired. Thus, many implementations include an extra callback, such as glutWMCloseFunc().

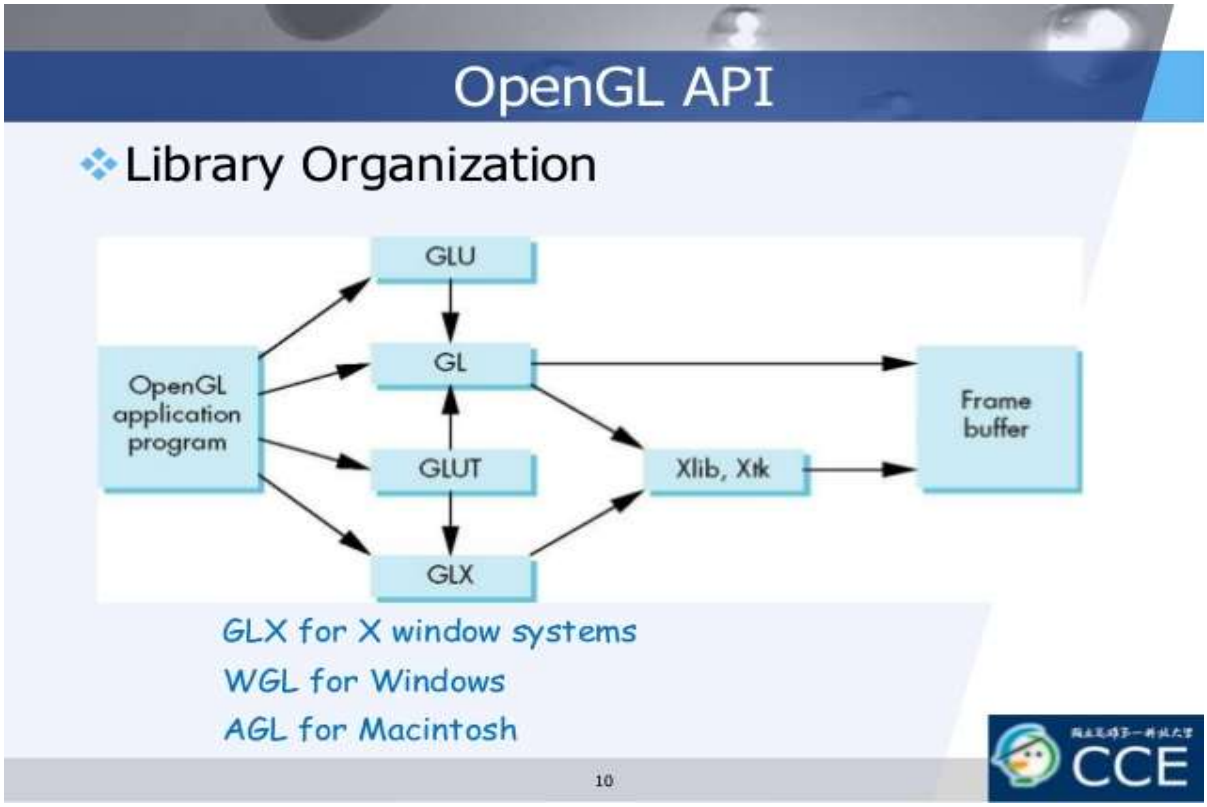

Figure 1.2: Library Organization of OpenGL APIs

1.7 OpenGL Rendering Pipeline

The OpenGL rendering pipeline works in the following order:

a) Vertex Specification: Prepare vertex array data, and then render it

b) Vertex Processing:

i. Each vertex is acted upon by a Vertex Shader. Each vertex in the stream is processed in turn into an output vertex.

ii. Optional primitive tessellation stages.

iii. Optional Geometry Shader primitive processing. The output is a sequence of primitives.

c) Vertex Post-Processing: The outputs of the last stage are adjusted or shipped to different locations. It performs the following tasks -

i. Transform Feedback: the outputs of the geometry shader or primitive assembly are written to a series of buffer objects that have been setup for this purpose, thus allowing the user to transform data via vertex and geometry shaders, then hold on to that data for use later.

ii. Primitive Clipping: it means that primitives that lie on the boundary between the inside of the viewing volume and the outside are split into several primitives, such that the entire primitive lies in the volume. The vertex positions are transformed from clip-space to window space via the Perspective Divide and the Viewport Transform

d) Primitive Assembly: Primitive assembly is the process of collecting a run of vertex data output from the prior stages and composing it into a sequence of primitives. The type of primitive the user rendered with determines how this process works. The output of this

process is an ordered sequence of simple primitives (lines, points, or triangles). If the input is a triangle strip primitive containing 12 vertices, for example, the output of this process will be 10 triangles.

e)Rasterization: Primitives that reach this stage are then rasterized in the order in which they were given. The result of rasterizing a primitive is a sequence of Fragments. A fragment is a set of state that is used to compute the final data for a pixel (or sample if multi-sampling is enabled) in the output framebuffer.

f)Fragment Processing: The data for each fragment from the rasterization stage is processed by a fragment shader. The output from a fragment shader is a list of colors for each of the color buffers being written to, a depth value, and a stencil value. Fragment shaders are not able to set the stencil data for a fragment, but they do have control over the color and depth values.

g)Per-Sample Processing: The fragment data output from the fragment processor is then passed through a sequence of steps. The first step is a sequence of culling tests; if a test is active and the fragment fails the test, the underlying pixels/samples are not updated (usually). Many of these tests are only active if the user activates them. The tests are:

i. Pixel ownership test: Fails if the fragment's pixel is not "owned" by OpenGL (if another window is overlapping with the GL window). Always passes when using a Framebuffer Object. Failure means that the pixel contains undefined values.

ii. Scissor Test: When enabled, the test fails if the fragment's pixel lies outside of a specified rectangle of the screen.

iii. Stencil Test: When enabled, the test fails if the stencil value provided by the test does not compare as the user specifies against the stencil value from the underlying sample in the stencil buffer. Depth Test: When enabled, the test fails if the fragment's depth does not compare

as the user specifies against the depth value from the underlying sample in the depth buffer.

iv. Color Blending: For each fragment color value, there is a specific blending operation between it and the color already in the framebuffer at that location. Logical Operations may also take place in lieu of blending, which perform bitwise operations between the fragment colors and framebuffer colors.

v. Masking operations: They allow the user to prevent writes to certain values.

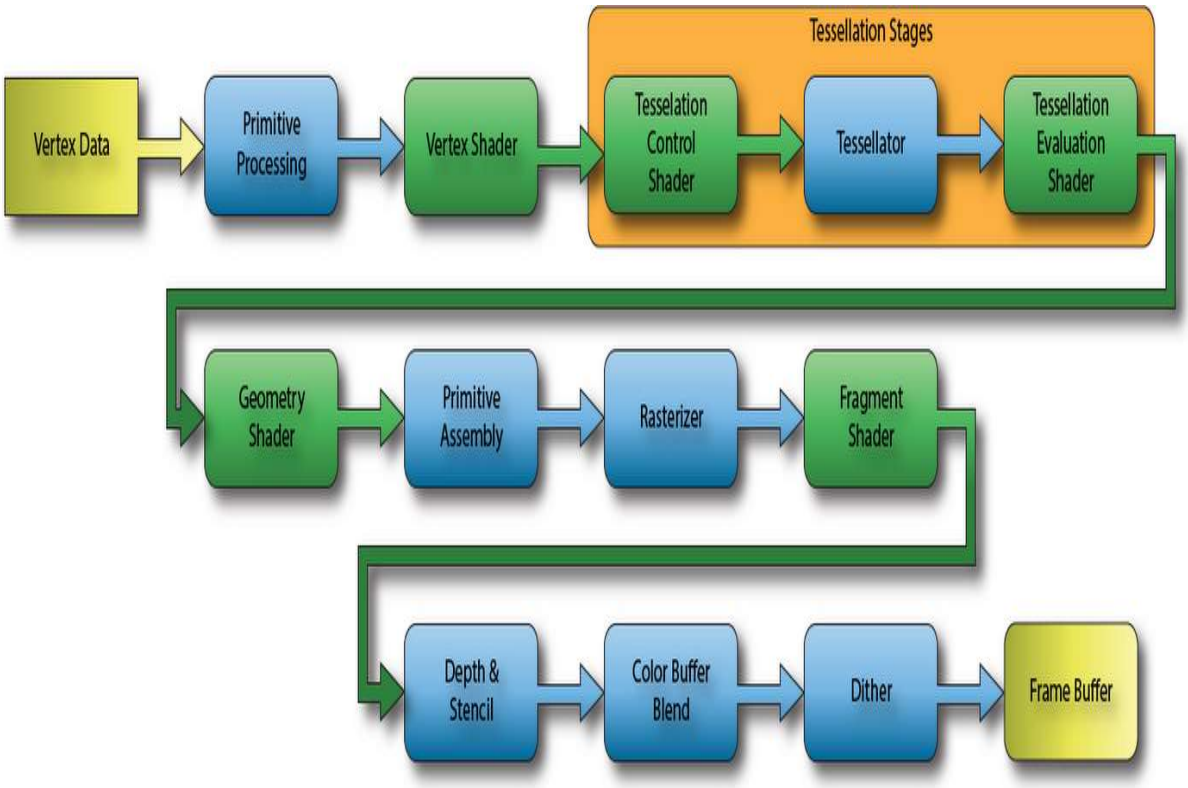

Figure 1.3: OpenGL rendering Pipeline

1.8 Applications of OpenGL

- OpenGL (Open Graphics Library) is a cross-language, multi-platform API for rendering 2D and 3D computer graphics.
- The API is typically used to interact with a GPU, to achieve hardware-accelerated rendering.
- It is widely used in CAD, virtual reality, scientific visualization, information visualization and flight simulation.
- It is also widely used in the development of video games for different platforms such as PC , consoles or smartphones.

1.9 OpenGL Primitives

OpenGL supports two classes of primitives:

- Geometric Primitives: Geometric primitives are specified in the problem domain and include points, line segments, polygons, curves and surfaces.
- Image(Raster) Primitives : Raster primitives, such as arrays of pixels pass through a separate parallel pipeline on their way to the frame buffer.

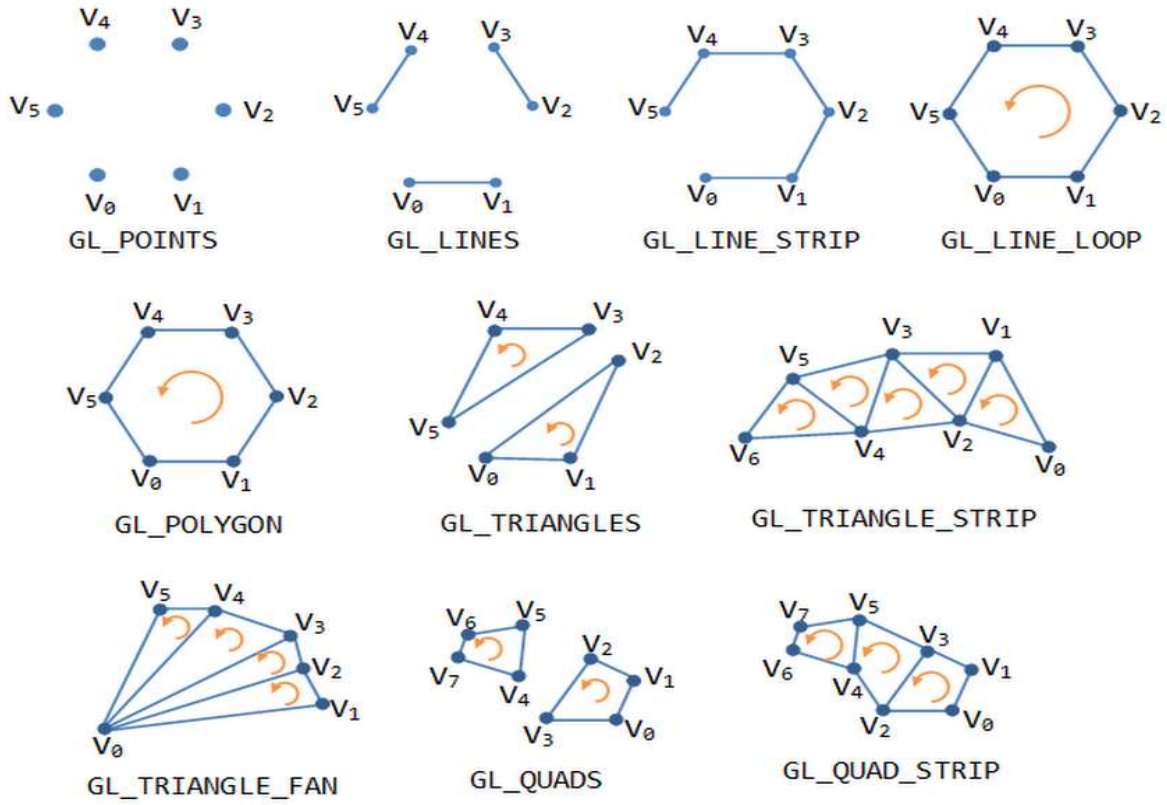

Figure 1.4: OpenGL Geometric Primitives

1.10 Introduction to *k*-nearest neighbors algorithm

In pattern recognition, the ***k*-nearest neighbors algorithm** (***k*-NN**) is a non parametric method used for classification and regression.[1]In both cases, the input consists of the *k* closest training examples in the feature space. The output depends on whether *k*-NN is used for classification or regression:

K-Nearest Neighbours is one of the most basic yet essential classification algorithms in Machine Learning. It belongs to the supervised learning domain and finds intense application in pattern recognition, data mining and intrusion detection.

It is widely disposable in real-life scenarios since it is non-parametric, meaning, it does not make any underlying assumptions about the distribution of data (as opposed to other algorithms such as GMM, which assume a Gaussian distribution of the given data).
We are given some prior data (also called training data), which classifies coordinates into groups identified by an attribute.

1.11 Objectives

The main objectives of this project include:

- To demonstrate the use of OpenGL APIs to create graphics
- To provide a visual representation of *k*-nearest neighbors algorithm (*k*-NN)
- Apply the programming knowledge to design the software
- Animate real world problems using OpenGL
- Apply the concepts of computer graphics

1.12 Organization of the Report

Chapter 1 provides the information about the basics of computer graphics, the history of OpenGL, and the major OpenGL libraries and about the basics of *k*-nearest neighbors algorithm (*k*-NN). Chapter 2 provides the hardware and software specifications required by a system to run this project. Chapter 3 gives the idea of the project. Chapter 4 discusses about the algorithm used to develop the program and its actual implementation. Chapter 5 provides various screenshots of the program in run time. Chapter 6 concludes by giving directions for future enhancements.

1.13 Summary

The chapter discussed before is an overview about Computer graphics, its history, the various utility tools to develop this project and also a brief introduction on *k*-nearest neighbors algorithm (*k*-NN). The scope of study and objectives of the project are mentioned clearly. The organization of the report has been pictured to increase the readability. Further, coming up chapters discuss about the overall working of the project.

Chapter 2

System Specifications

2.1 Software Requirements

Operating system	:	Ubuntu 16.x or above
Compiler used	:	GCC
Programming language	:	C language
Editor	:	gedit/notepadqq
Graphics library	:	GL and GLU / GLUT

2.2 Hardware Requirements

Processor	:	Intel I3/I5/I7
Processor speed	:	1 GHz or more
Hard disk	:	40 GB or more
RAM size	:	1 GB or more
Display	:	800x600 or higher resolution display with 256 colours
Mouse	:	Standard serial mouse
Keyboard	:	Standard QWERTY keyboard
GPU	:	Intel HD Graphics 5000 or better

Chapter 3

K-NEAREST NEIGHBORS ALGORITHM (*K*-NN)

K-Nearest Neighbours is one of the most basic yet essential classification algorithms in Machine Learning. It belongs to the supervised learning domain and finds intense application in pattern recognition, data mining and intrusion detection.

It is widely disposable in real-life scenarios since it is non-parametric, meaning, it does not make any underlying assumptions about the distribution of data (as opposed to other algorithms such as GMM, which assume a Gaussian distribution of the given data).
We are given some prior data (also called training data), which classifies coordinates into groups identified by an attribute.

As an example, consider the following table of data points containing two features:

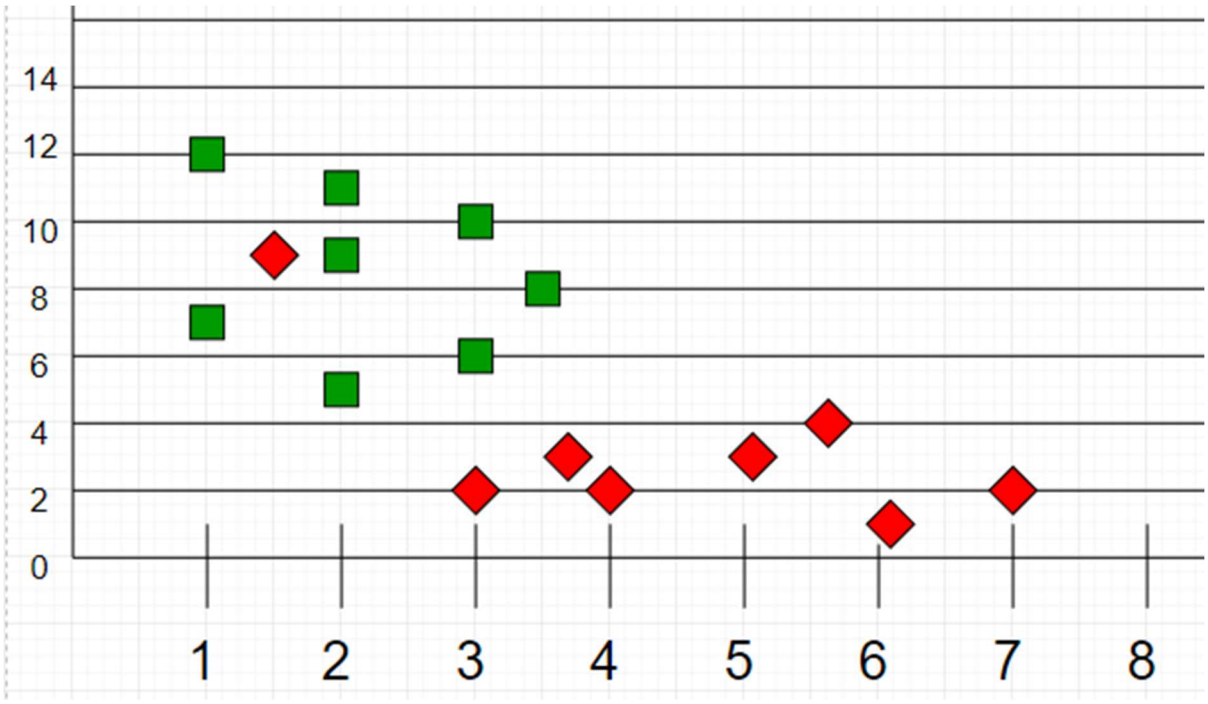

Fig 3.1 Training datasets

Now, given another set of data points (also called testing data), allocate these points a group by analyzing the training set. Note that the unclassified points are marked as ‘White’.

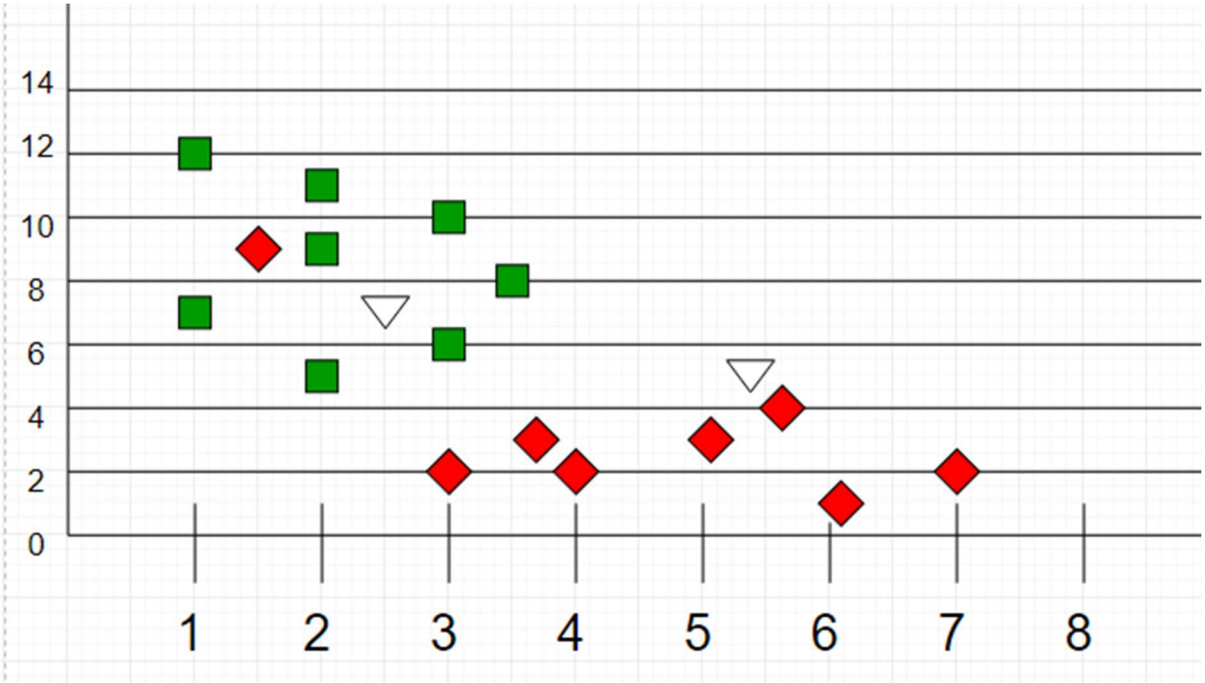

Fig 3.2 Testing datasets

Intuition

If we plot these points on a graph, we may be able to locate some clusters, or groups. Now, given an unclassified point, we can assign it to a group by observing what group its nearest neighbours belong to. This means, a point close to a cluster of points classified as ‘Red’ has a higher probability of getting classified as ‘Red’.

Intuitively, we can see that the first point (2.5, 7) should be classified as ‘Green’ and the second point (5.5, 4.5) should be classified as ‘Red’.

Algorithm

Let m be the number of training data samples. Let p be an unknown point.

1. Store the training samples in an array of data points arr[]. This means each element of this array represents a tuple (x, y).
2. For i=0 to m:
3. Calculate Euclidean distance d(arr[i],p).
4. Make set S of K smallest distances obtained. Each of these distances correspond to an already classified data point.
5. Return the majority label among S.

K can be kept as an odd number so that we can calculate a clear majority in the case where only two groups are possible (e.g. Red/Blue). With increasing K, we get smoother, more defined boundaries across different classifications. Also, the accuracy of the above classifier increases as we increase the number of data points in the training set.

KNN can be used for both classification and regression predictive problems. However, it is more widely used in classification problems in the industry. To evaluate any technique we generally look at 3 important aspects:

1. Ease to interpret output
2. Calculation time
3. Predictive Power

Chapter 4

System Design and Implementation

4.1 Introduction

Systems design is the process or art of defining the architecture, components, modules, interfaces, and data for a system to satisfy specified requirements. One could see it as the application of systems theory to product development.

4.2 Initialization

Initialize the interaction with the windows. Initialize the display mode- double buffer and depth buffer. Initialize the various call back functions for drawing and redrawing, for mouse and keyboard interfaces. Initialize the input and calculate functions for various mathematical calculations. Initialize the window position and size and create the window to display the output.

4.3 Flow of control

The flow of control in the below flow chart is with respect to the Texture Package. For any of the program flow chart is compulsory to understand the program. We consider the flow chart for the texture project in which the flow starts from start and proceeds to the main function after which it comes to the initialization of call back functions and further it proceeds to mouse and keyboard functions, input and calculation functions. Finally, it comes to quit, the end of flow chart.

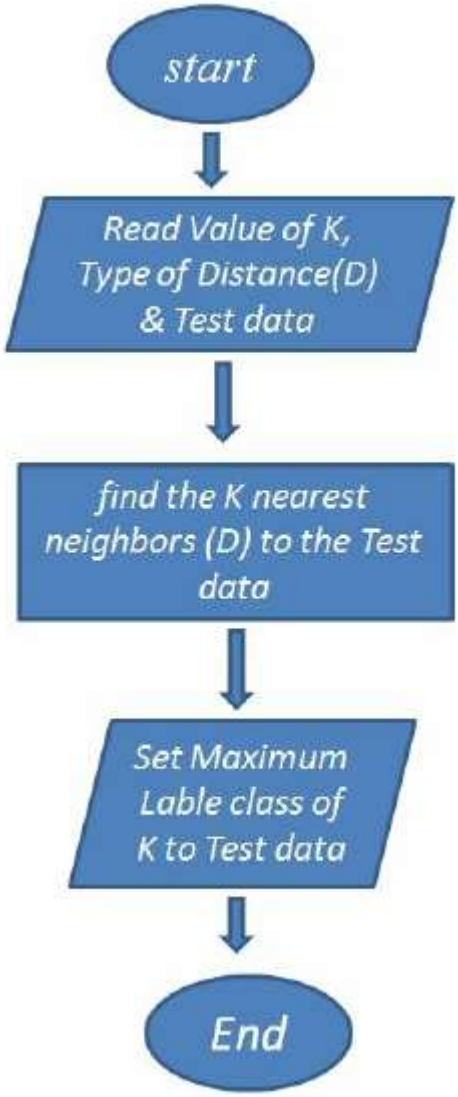

Figure 4.1: Project Design

4.4 OpenGL APIs used/Built-in functions

- glRasterPos3f()

  Specifies the raster position for pixel operations.

- glutBitmapCharacter()

  Renders a bitmap charater using OpenGL from the specified array of characters, and in the specified font style.

- glutPostRedisplay()

  Marks the current window as needing to be redisplayed.

- glutTimerFunc( )

  Registers a timer callback to be triggered in a specified number of milliseconds.

- glClearColor ()

  Specifies clear values for the color buffers.

- glShadeModel ()

  Select flat or smooth shading. Specifies a symbolic value representing a shading technique. Accepted values are GL_FLAT and GL_SMOOTH.

- glEnable()

  Enables the OpenGL capabilities, Specifies the conditions under which the pixels will be drawn.

- glLightfv()

  Creates a new light source with specified parameter values.

- gluQuadricDrawStyle()

  Specifies the draw style required for quadrics.

- glPushMatrix() and glPopMatrix()

Push and pop the current matrix stack.

- glTranslatef() and glRotatef()

  Multiplies current matrix by Translation and Rotation matrix respectively.

- glMatrixMode ()

  Specifies which matrix is the current matrix.

- glLoadIdentity()

  Pushes the identity matrix to the top of the matrix stack.

- gluLookAt()

  Defines a viewing transformation.

- glutSwapBuffers()

  Swaps the buffers of the *current window* if double buffered.

- glViewport()

  Sets the viewport.

- glutInitDisplayMode ()

  Sets the initial display mode.

- glutInitWindowSize () and glutInitWindowPosition ()

  Set the initial window size and position respectively.

- glutCreateWindow()

  Creates a top level window with the window name as specified.

- glutAddMenuEntry()

  Adds a menu entry to the bottom of the *current menu*.

- glutAttachMenu()

  Attaches a mouse button for the *current window* to the identifier of the *current menu.*

- glutDisplayFunc()

  Sets the display callback for the *current window.*

- glutReshapeFunc()

  Sets the reshape callback for the *current window.*

- glutMainLoop()

  Enters the GLUT event processing loop. This routine should be called at most once in a GLUT program. Once called, this routine will never return. It will call as necessary any callbacks that have been registered.

4.5 Pseudo Codes/Algorithms

4.5.1 KNN algorithm:

1. Calculate "d(x, x_i)" i =1, 2, ….., **n**; where **d** denotes the Euclidean distance between the points.
2. Arrange the calculated **n** Euclidean distances in non-decreasing order.
3. Let **k** be a +ve integer, take the first **k** distances from this sorted list.
4. Find those **k**-points corresponding to these **k**-distances.
5. Let $\mathbf{k}_i$ denotes the number of points belonging to the i^{th} class among **k** points i.e. $k \geq 0$
6. If $k_i > k_j \; \forall \; i \neq j$ then put x in class i.

Chapter 5

Results and Discussions

The project is executed using C++ language. We have put in few screen shots to show the working of Knn algorithm.

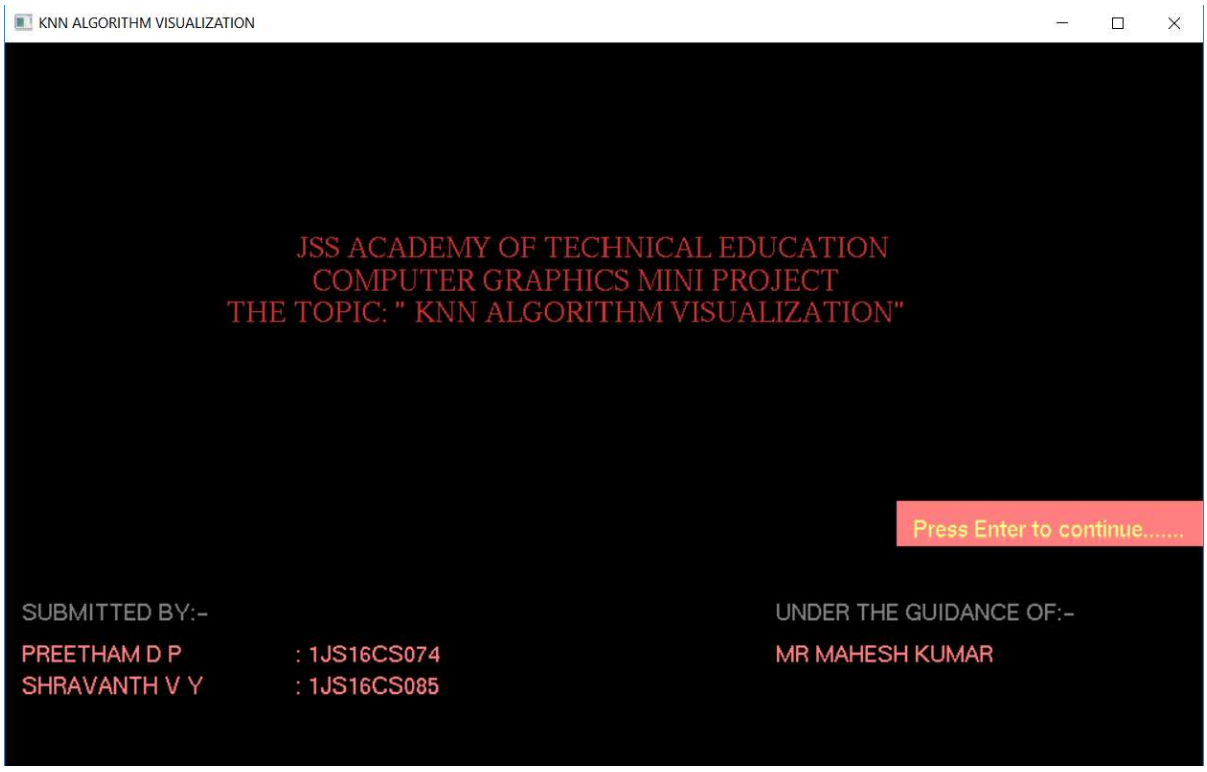

Figure 5.1: Home Screen

Figure 5.1 shows the home page of the program. It contains information about the title of the project, identities of the students participating in it and the teachers who guided them. By pressing 'ENTER' on the keyboard, the user can move on to the next screen of the program.

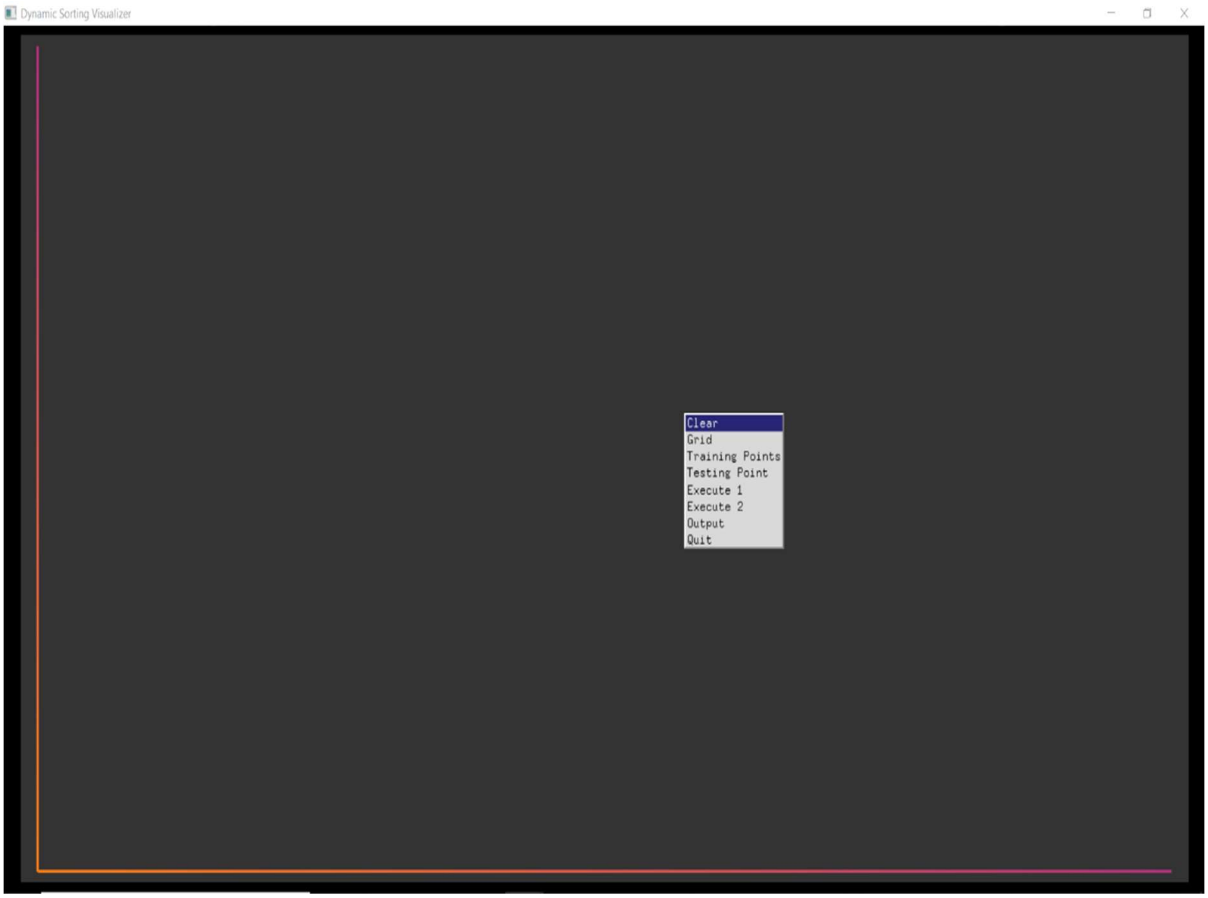

Figure 5.2: Right click menu

Figure 5.2 shows the initial screen of the program. Right click will display menu to select as shown in above picture.

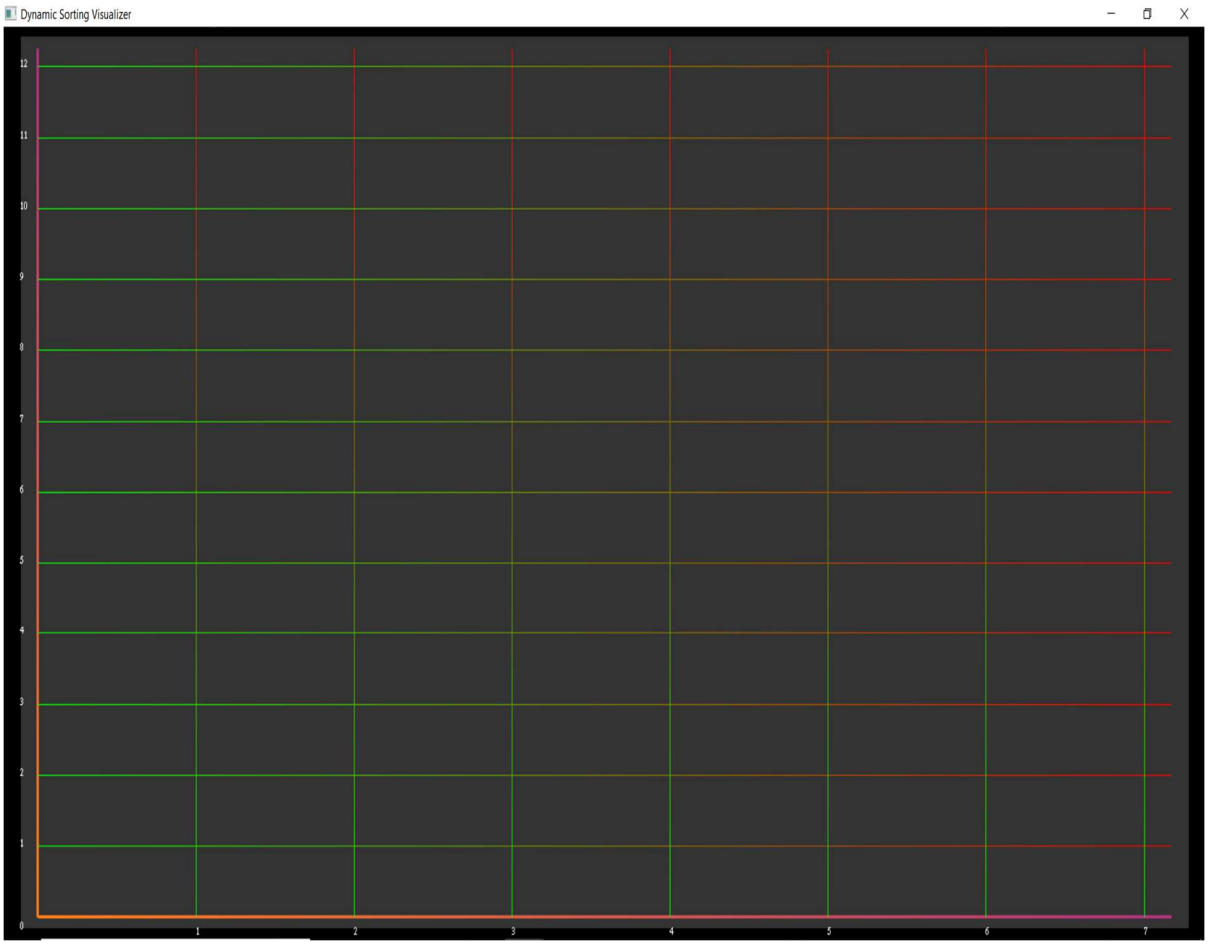

Figure 5.3: Grid enables.

Figure 5.3 Gives a visual representation of the grid when selected. This option is optional the algorithm can be visualized without enabling this, but grids helps us to understand datapoints better.

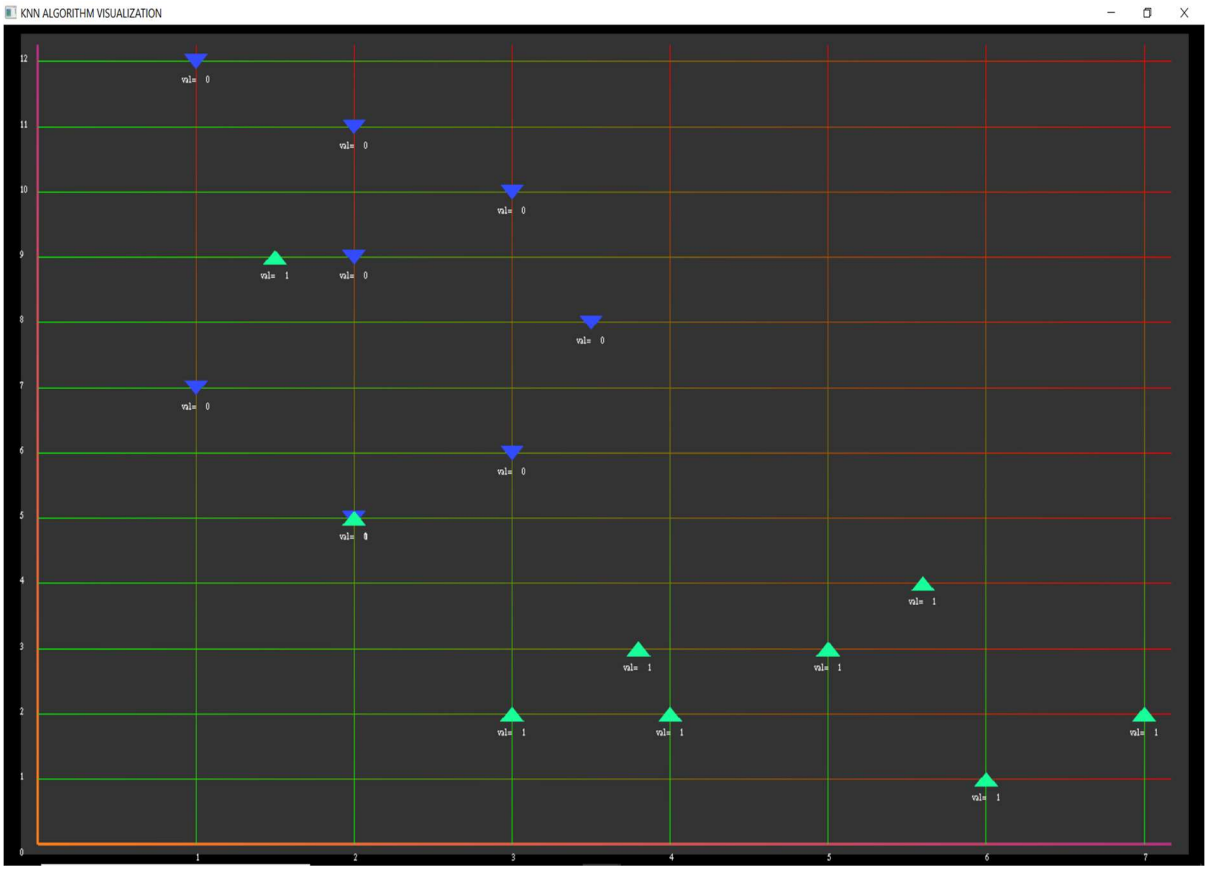

Figure 5.4: Visual Representation of the training data points.

Figure 5.4: Gives a visual representation of the training dataset. These can be changed in the input file as desired and the resulting datapoints will be displayed here.

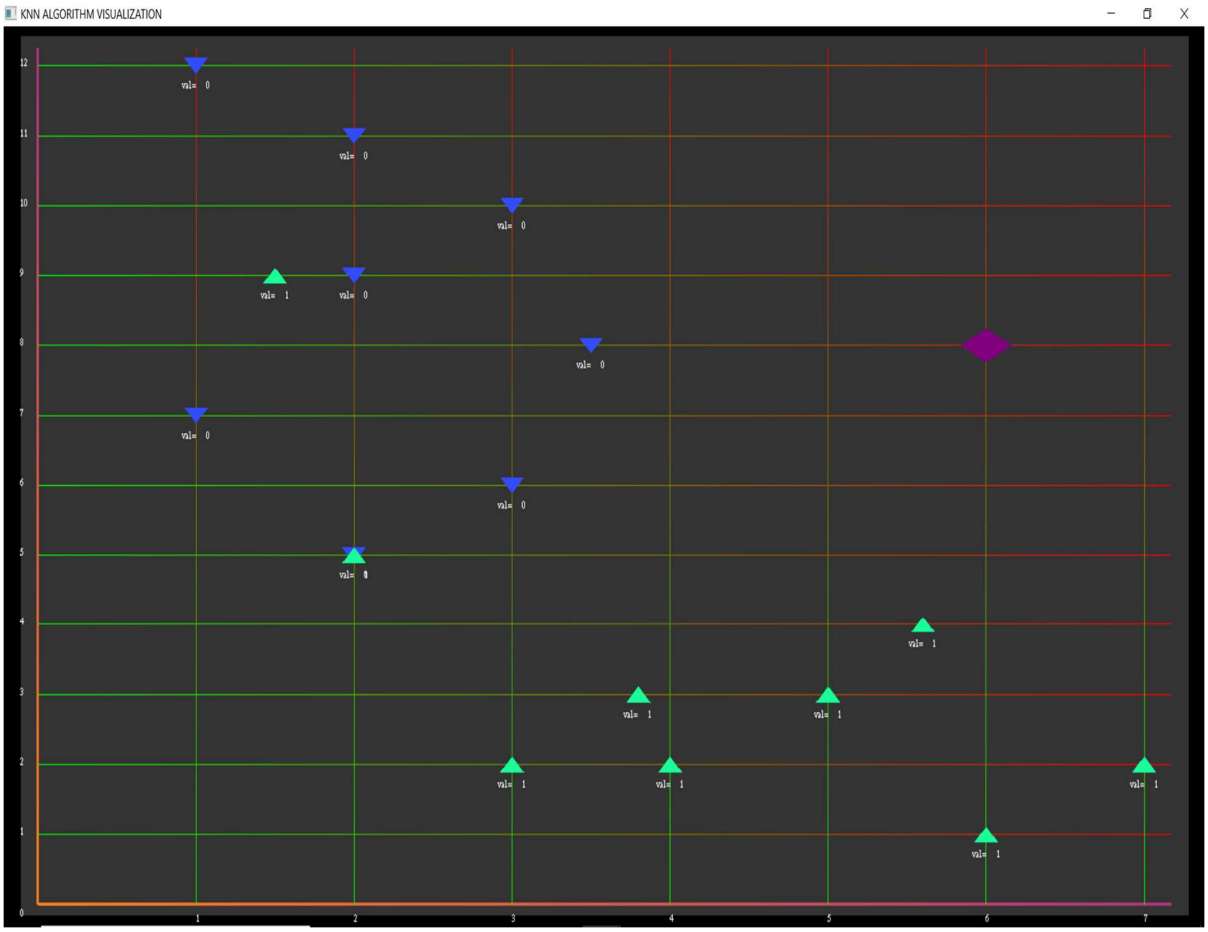

Figure 5.5: Visual Representation of the testing data which user will enter.

Figure 5.5 Shows the testing point along with the training points(datapoints). The testing point is given by the user at the beginning of the execution of the program.

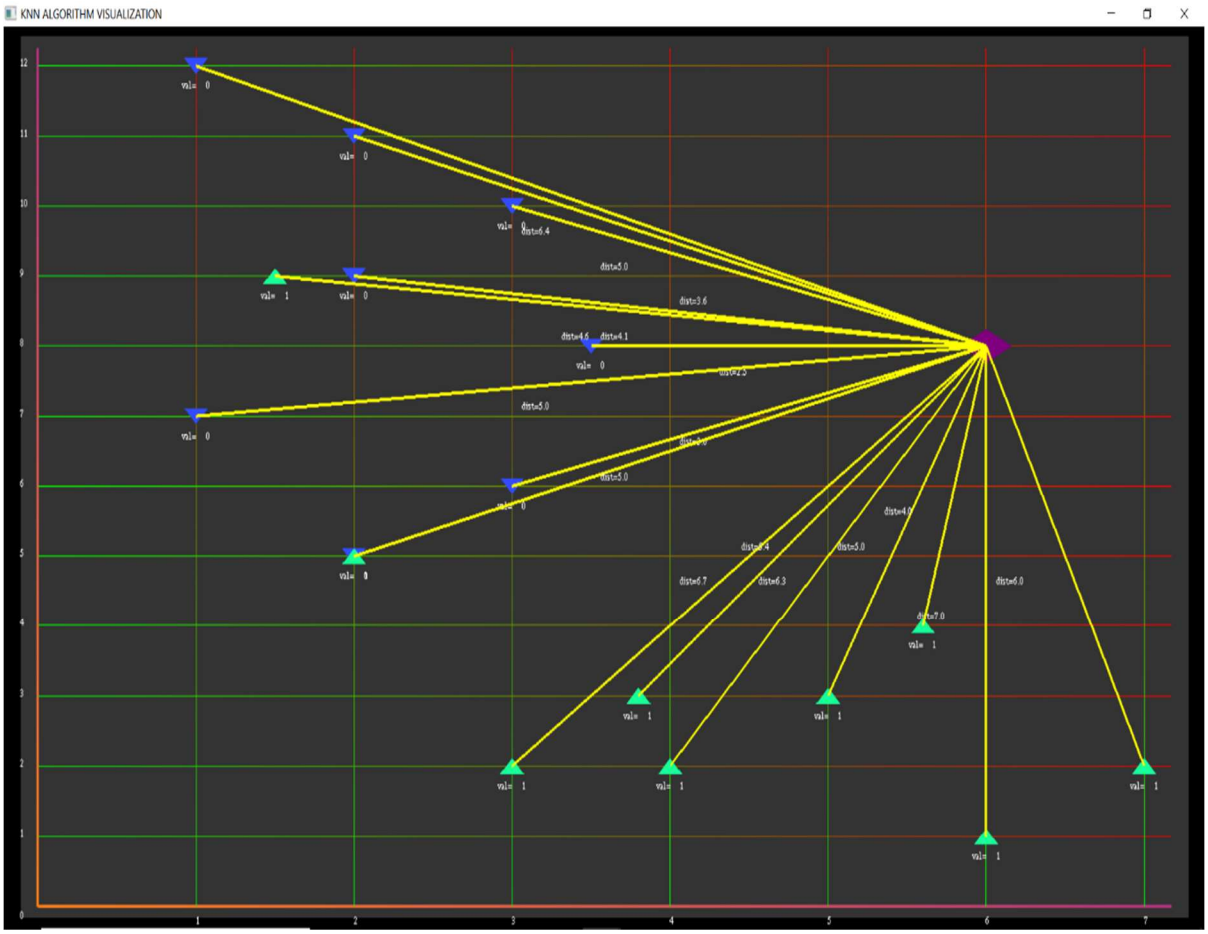

Figure 5.6: Calculation of Euclidian distance.

In Figure 5.6: The line represents the calculation of distance between testing point and the training points the distance is printed below lines. This can done by selecting Execute1 in menu. To select this option the training points and testing points should be displayed first.

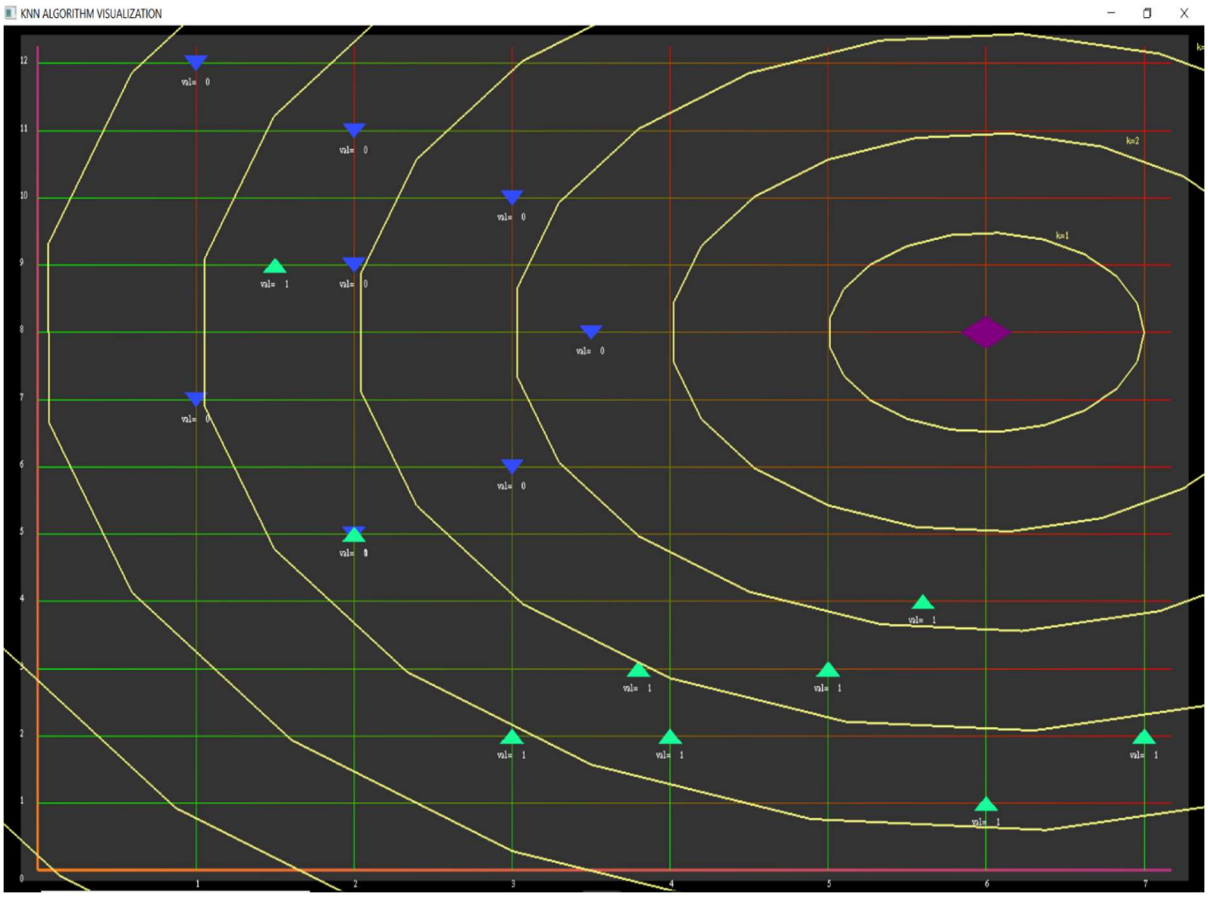

Figure 5.7: Another way to see how algorithm works

Figure 5.7: The circles helps to visualize how many data points will fall inside that circle and how that affects the output.

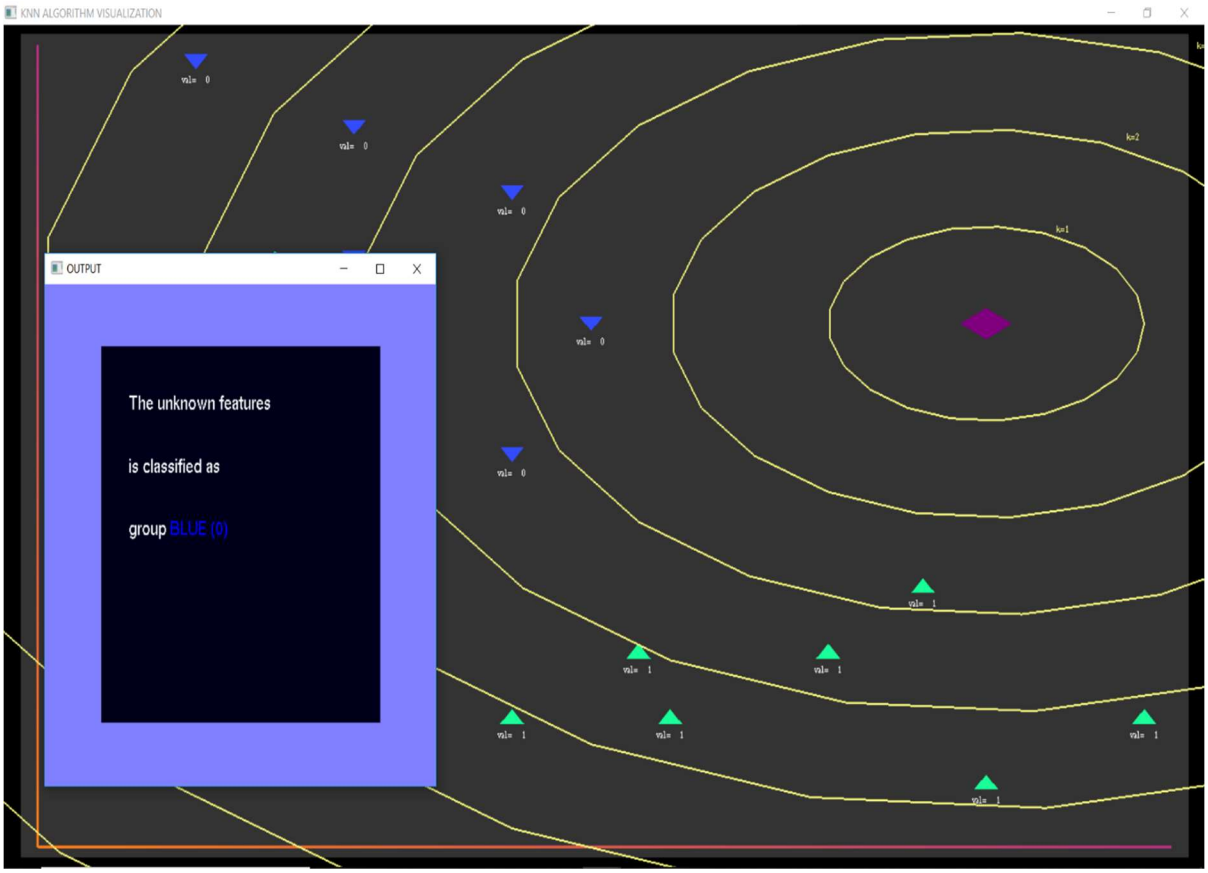

Figure 5.8: The output.

Figure 5.8: Here it can be clearly seen it belongs to group blue. It is clear that there is one from each group in circle three bit two from blue and one from green in circle four. According to knn the testing point is nearest to the blue ones.

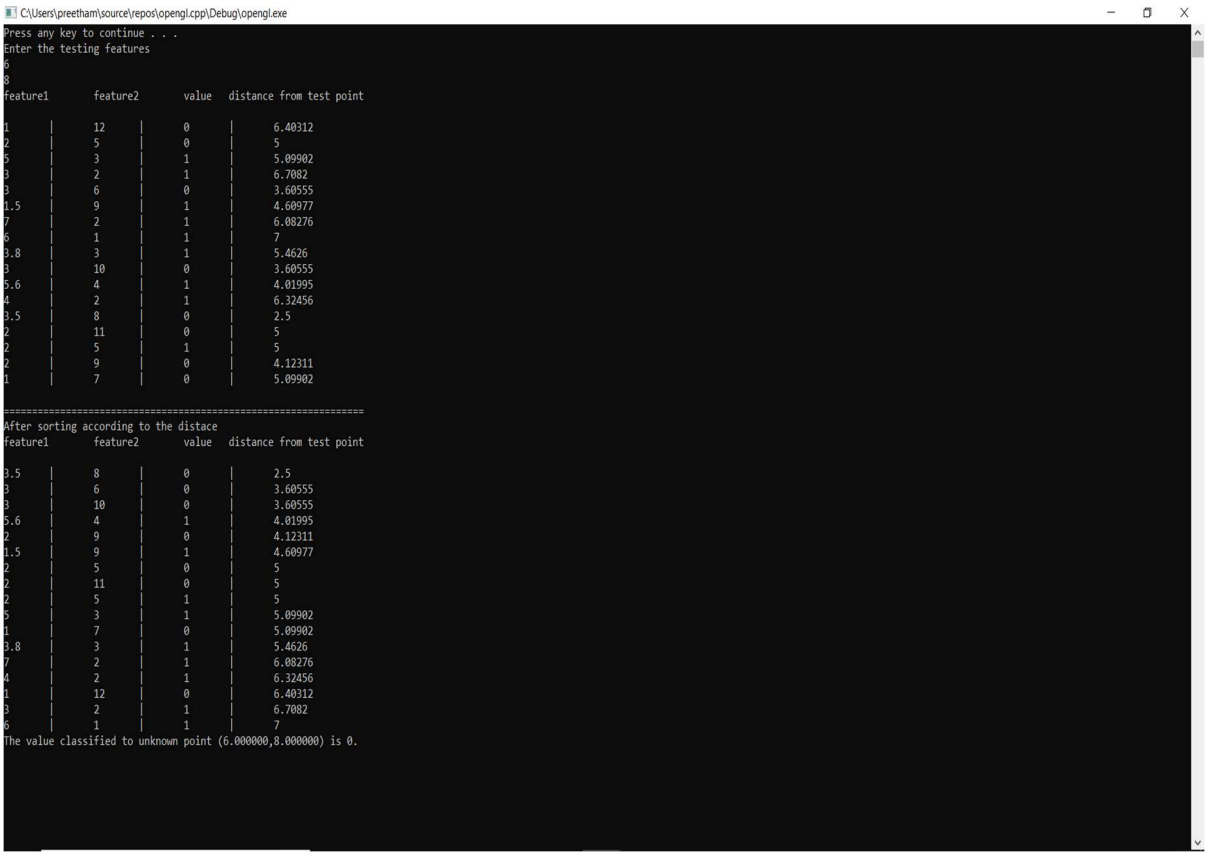

Figure 5.9: Initial user input and output.

In Figure 5.9: This shows us the user input i.e. 6,8 and the output to which group it belongs and the distances between testing and all the training points.

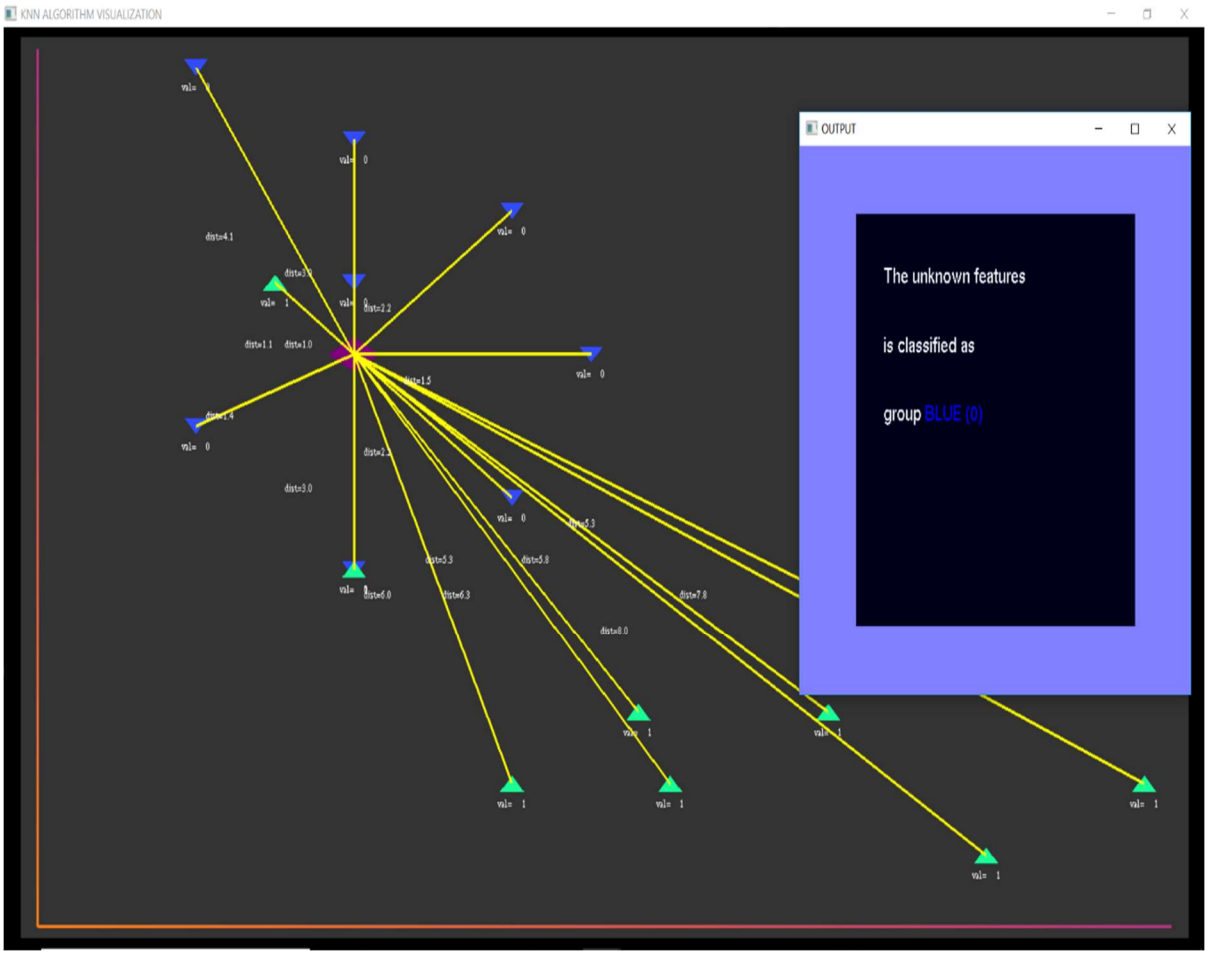

Figure 5.10: Another example.

Figure 5.10: Here is another example with different testing point. this time the point is classified as blue. Here the testing point is (2,8).

Chapter 6

Conclusion and Future Enhancements

6.1 Conclusion

K-Nearest neighbor classification is a general technique to learn classification based on instance and do not have to develop an abstract model from the training data set. However the classification process could be very expensive because it needs to compute the similarly values individually between the test and training examples. K-nearest neighbor classifier also suffers from the scaling issue. It computes the proximity among the test example and training examples to perform classification. If the attributes have different scales, the proximity distance might be dominated by one of the attributes, which is not good.

6.2 Future Enhancements

As a scope for future enhancement,

- Visualization can be better if it was in 3D, hence it can be enhanced.
- Can use the existing models to visuals other algorithms which are similar to knn by giving more option and making few changes to the existing code.

References

Web Sources

[1]:https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm

[2]: https://www.geeksforgeeks.org/k-nearest-neighbours/

Books

[3] Donald Hearn & Pauline Baker: Computer Graphics with OpenGL Version,3rd / 4th Edition, Pearson Education,2011

[4] Edward Angel: Interactive Computer Graphics- A Top Down approach with OpenGL,5th edition. Pearson Education, 2008