**Chapter 1**

**Preamble**

* 1. **Introduction to Computer Graphics**

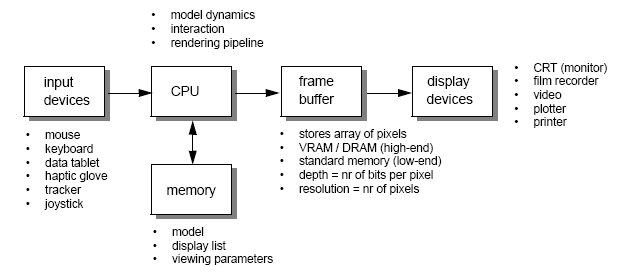
Computer graphics are pictures and films created using computers. Usually, the term refers to computer-generated image data created with help from specialized graphical hardware and software. It is a vast and recent area in computer science. The phrase was coined in 1960, by computer graphics researchers 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](https://en.wikipedia.org/wiki/User_interface_design), [sprite graphics](https://en.wikipedia.org/wiki/Sprite_(graphics)), [vector graphics](https://en.wikipedia.org/wiki/Vector_graphics), [3D modelling](https://en.wikipedia.org/wiki/3D_modeling), [shaders](https://en.wikipedia.org/wiki/Shader), [GPU](https://en.wikipedia.org/wiki/GPU) design, [implicit surface](https://en.wikipedia.org/wiki/Implicit_surface) visualization with [ray tracing](https://en.wikipedia.org/wiki/Ray_tracing_(graphics)), and [computer vision](https://en.wikipedia.org/wiki/Computer_vision), among others. The overall methodology depends heavily on the underlying sciences of [geometry](https://en.wikipedia.org/wiki/Geometry), [optics](https://en.wikipedia.org/wiki/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](https://en.wikipedia.org/wiki/Animation), [movies](https://en.wikipedia.org/wiki/Movies), [advertising](https://en.wikipedia.org/wiki/Advertising), [video games](https://en.wikipedia.org/wiki/Video_game), and [graphic design](https://en.wikipedia.org/wiki/Graphic_design) generally [1].

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: [1]

* the representation and manipulation of image data by a computer
* the various [technologies](https://en.wikipedia.org/wiki/Technologies) used to create and manipulate images
* the sub-field of [computer science](https://en.wikipedia.org/wiki/Computer_science) which studies methods for digitally synthesizing and manipulating visual content, see [study of computer graphics](https://en.wikipedia.org/wiki/Computer_graphics_(computer_science))

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](https://en.wikipedia.org/wiki/Chart)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](https://en.wikipedia.org/wiki/3D_computer_graphics) have become more common, but [2D computer graphics](https://en.wikipedia.org/wiki/2D_computer_graphics) are still widely used. Computer graphics has emerged as a sub-field of [computer science](https://en.wikipedia.org/wiki/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](https://en.wikipedia.org/wiki/Information_visualization), and [scientific visualization](https://en.wikipedia.org/wiki/Scientific_visualization) more concerned with "the visualization of [three dimensional](https://en.wikipedia.org/wiki/Three-dimensional_space) phenomena (architectural, meteorological, medical, [biological](https://en.wikipedia.org/wiki/Biological_Data_Visualization), etc.), where the emphasis is on realistic renderings of volumes, surfaces, illumination sources, and so forth, perhaps with a dynamic (time) component". [5]



**Figure 1.1**: Graphics Pipeline

* 1. **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. [1]

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.[1]

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]

* 1. **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 [6]

* 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. **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. [2]

* 1. **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. [5]

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. **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. [2]

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. [2]

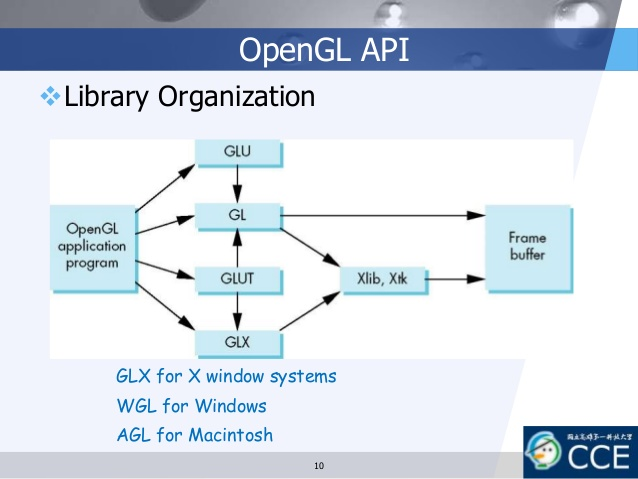
The two aims of GLUT are to allow the creation of rather portable code between operating systems (GLUT is cross-platform) and to make learning OpenGL easier. Getting started with OpenGL programming while using GLUT often takes only a few lines of code and does not require knowledge of operating system–specific windowing APIs. All GLUT functions start with the glut prefix , for example, glutPostRedisplay marks the current window as needing to be redrawn. [5]

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. [2]
* 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().

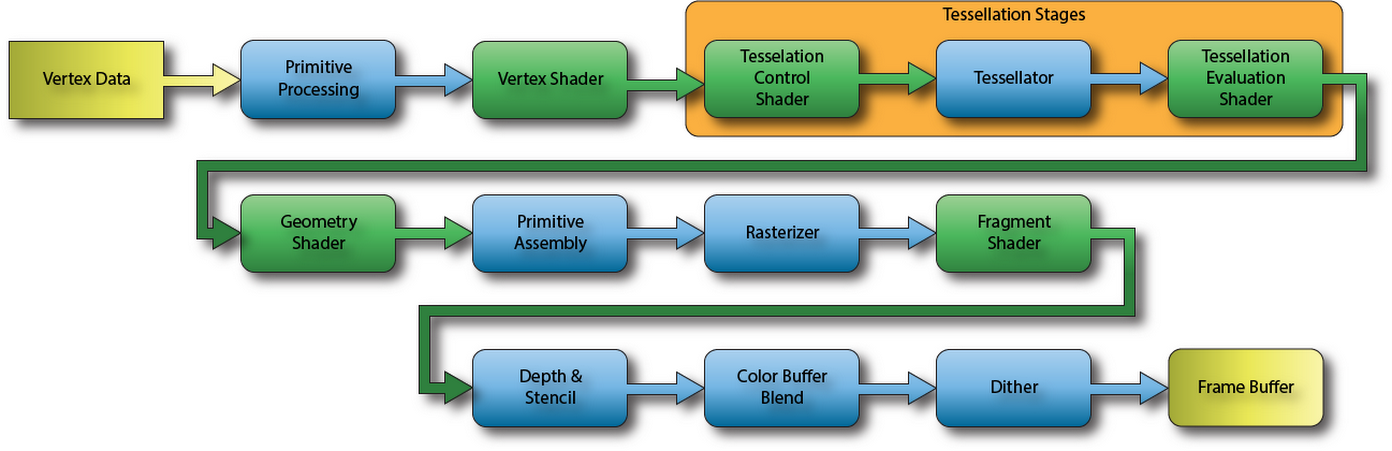


**Figure 1.2**: Library Organization of OpenGL APIs

* 1. **OpenGL Rendering Pipeline**

The OpenGL rendering pipeline works in the following order:

1. Vertex Specification: Prepare vertex array data, and then render it
2. Vertex Processing:
   * 1. Each vertex is acted upon by a Vertex Shader. Each vertex in the stream is processed in turn into an output vertex.
     2. Optional primitive tessellation stages.
     3. Optional Geometry Shader primitive processing. The output is a sequence of primitives.
3. Vertex Post-Processing: The outputs of the last stage are adjusted or shipped to different locations. It performs the following tasks -
   * 1. 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.
     2. 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
4. 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. [3]
5. 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. [3]
6. 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. [3]
7. 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: [3]
   * 1. 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.
     2. Scissor Test: When enabled, the test fails if the fragment's pixel lies outside of a specified rectangle of the screen.
     3. 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.
     4. [C](https://www.khronos.org/opengl/wiki/Blending)olor 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.
     5. Masking operations: They allow the user to prevent writes to certain values.

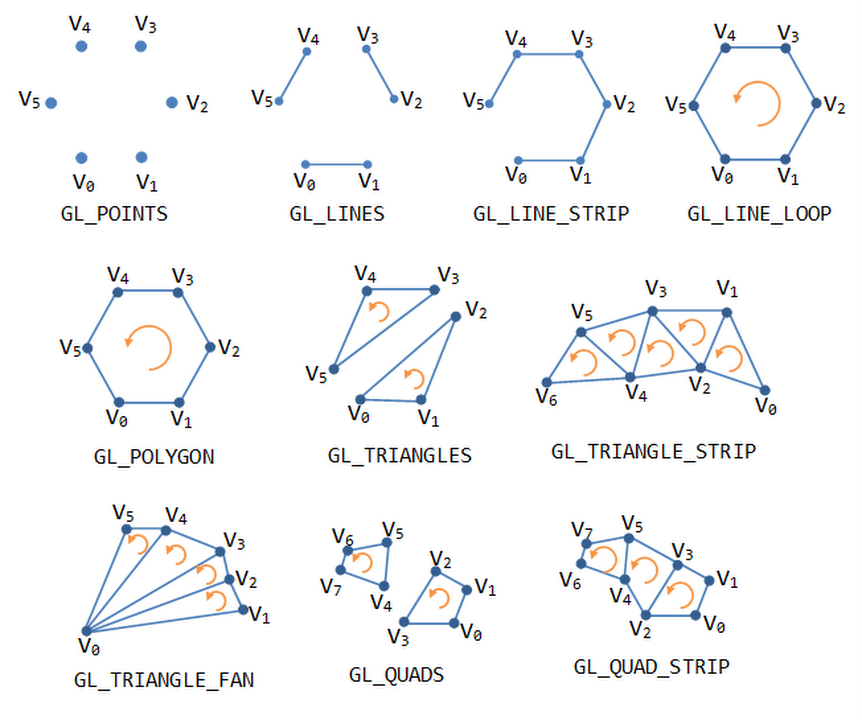


**Figure 1.3**: OpenGL rendering Pipeline

* 1. **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. **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

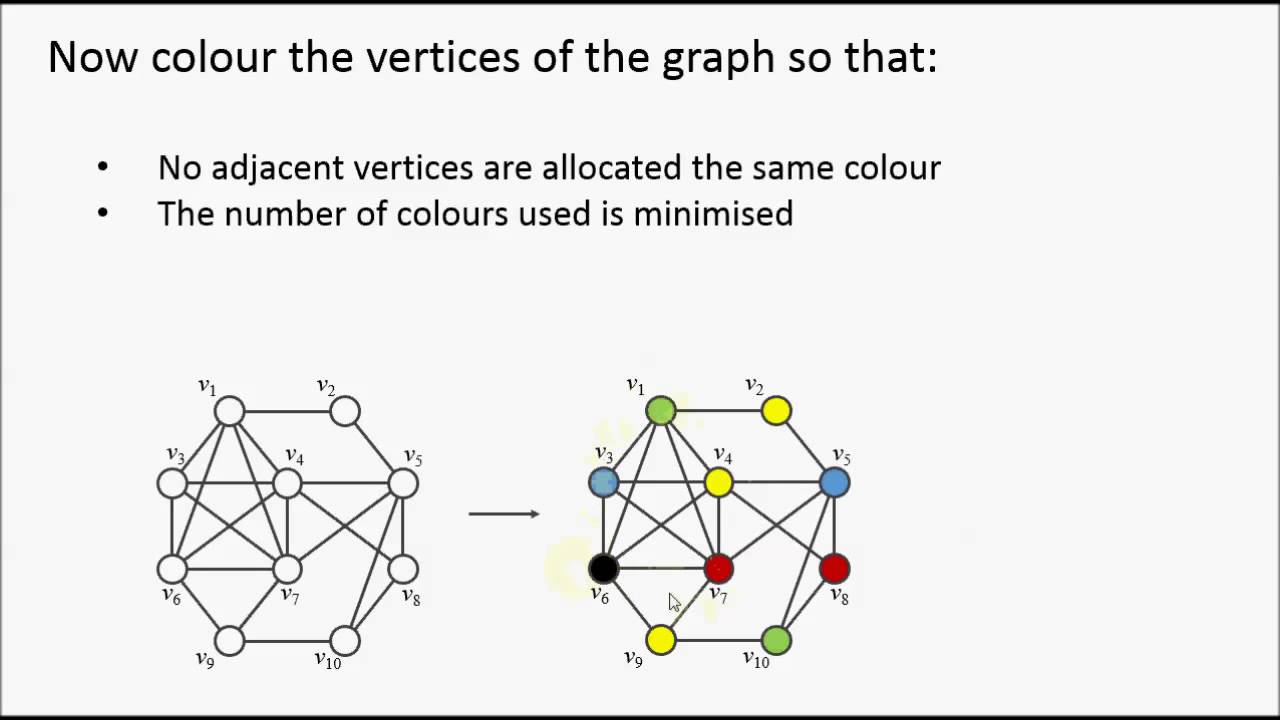
* 1. **Introduction to Graph Coloring**

In graph theory, a graph coloring is an assignment of colors to certain objects in a graph subject to certain constraints. The simplest form of graph coloring, called the proper or legal vertex coloring, is to assign colors to the vertices of a graph such that no two adjacent vertices share the same color. Similarly, a proper edge coloring assigns colors to the edges such that no two incident edges share the same color, while a proper face coloring of a planar graph assigns colors to the faces or regions such that no two adjacent faces share the same color. Typically, other coloring problems can be transformed into a vertex coloring version. For example, an edge coloring of a graph is just a vertex coloring of its adjoint graph, while a face coloring of a planar graph is just a vertex coloring of -its planar dual graph. When used without any further specification, a coloring of a graph is always assumed to be a proper vertex coloring. However, non-vertex coloring problems are usually stated and studied as they are to keep things in their perspective. Graph coloring problem has found a number of applications such as scheduling, compiler allocation, frequency assignment and pattern matching.

A proper coloring of a graph using at most k colors is called a k-coloring. Clearly, finding a k-coloring of a graph is equivalent to the problem of partitioning the vertices into k or fewer independent sets. The least number of colors needed to color a graph is called its chromatic number χ. A graph is called k-colorable if there exists a k-coloring of the graph and is called k-chromatic if its chromatic number is exactly k.

Theorem 1: Let G = (V, E) be a graph. We have

1. χ(G) = 1 if and only if G is totally disconnected;
2. χ(G) ≥ 3 if and only if G is not bipartite, or equivalently, has an odd cycle;
3. χ(G) ≥ ω(G), where ω(G) is the order of a largest clique in G;
4. χ(G) ≤ 4(G) + 1, where 4(G) is the largest degree over all vertices;
5. (Brook’s Theorem) χ(G) ≤ 4(G) if G is not Kn or C2n+1 where Kn is the complete graph with n vertices and C2n+1 is the odd cycle with 2n + 1 vertices.
6. χ(G) ≤ 4 for any planar graph G. Given a k-colorable graph G = (V, E), finding a k-coloring for G is solvable in polynomial time for k = 2 but NP-hard for k ≥ 3. The decision version of graph covering problem, i.e., whether or not is there a k-coloring, is one of Karp’s 21 NP-complete problems. As shown by Garey et al. [9], it remains to be NP-complete even on planar graphs with node degree at most 4 although it is trivial for k ≥ 4 on planar graphs due to the four-color theorem.. Johnson [12] showed that a version of the greedy algorithm gives an O(n/ log n)-approximation algorithm for k-colorings where n is the number of vertices in the graph. Wigderson [21] developed an algorithm to find a O( √ n)-coloring for any 3-colorable graph in polynomial time, which can be extended to find O(n 1−1/(k−1))-colorings for general k. Blum [3] provided a combinatorial algorithm for coloring a 3-colorable graph using O(n 3/8 log8/5 n) colors, which can be generalized to color a k-colorable graph with O(n 1−1/(k−4/3) log8/5 n) colors. Karger et al. [15] presented a semidefinite programming based O(n 1/4 log1/2 n)-coloring for 3-colorable graphs and O(n 1−3/(k+1) log1/2 n)-coloring for k-colorable graphs. Blum and Karger [4] combined the techniques in Blum [3] and Karger et al. [15] and improved the bound to O˜(n 3/14) for 3- colorable graphs where the notation O˜ is used to hide lower-order multiplicative terms, such as log n. So far, the best known approximation for 3-colorable graphs is due to Arora et al. [1], no polynomial time algorithm can approximate the chromatic number of a graph to within a ratio of n unless P = NP. Feige and Kilian [8] and 2 H˙astad [11] showed that approximating the chromatic number to within n 1−δ for any δ > 0 would imply NP = RP where RP is the class of probabilistic polynomial time algorithms making one sided error.



**Figure 1.5**: Example of a graph whose vertex is colored using graph coloring

* 1. **Objectives**

The main objectives of this project include:

* To demonstrate the use of OpenGL APIs to create graphics
* To provide a visual representation of Graph Coloring Algorithm
* Apply the programming knowledge to design the software
* Animate real world problems using OpenGL
* Apply the concepts of computer graphics
  1. **Organization of the Report**

Chapter 1 provides the information about the basics of computer graphics, the history of OpenGL, the major OpenGL libraries and about the basics of Selective Repeat Protocol. 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. **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 to Graph Coloring Algorithm. It even includes the Graphics pipeline diagram, the diagrams demonstrating graph coloring for some standard graphs. 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 : Windows 10

Compiler used : C++

Programming language : C ++

Editor : Visual Studio

Graphics library : GL and GLU / GLUT

**2.2 Hardware Requirements**

Processor : Intel I3/I5/I7

Processor speed : 1GHz 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**

**Graph Coloring Algorithm**

This mini project on **Graph Coloring Algorithm** demonstrates how, the algorithm works. It employs a greedy algorithm to find a solution.

\* Color first vertex with first colour.

\* Do following for remaining V-1 vertices

Consider the currently picked vertex

Colour it with the lowest numbered colour

that has not been used on any previously colored vertices

adjacent to it

If all previously used colors appear on vertices adjacent to v, assign a new color to it .

The program is interactive, uses mouse functions to select the options. It is also possible to switch between different menus by pressing specific keys. Menus for allowing the user to perform specific actions like entering the vertices, drawing edges among the vertices, and finding the solution is provided.

.

**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.

* 1. **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 interface. Initialize the input and calculate functions for various mathematical calculations. Initialize the window position and size and create the window to display the output.

* 1. **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.

MAIN

INITIALIZE CALLBACK FUNCTIONS

MAIN SCREEN DISPLAYED

END OF INPUT

KEYBOARD

CALCULATE

MOUSE INTERUPT

READING INPUT STRING AND ANIMATE MOUSE CLICK

DISPLAY OUTPUT

* 1. **OpenGL APIs used/Built-in functions**
* **glRasterPos3f()**
  + - 1. Specifies the raster position for pixel operations.
* **glutBitmapCharacter()**
  + - 1. Renders a bitmap charater using OpenGL from the specified array of characters, and in the specified font style.
* **glutPostRedisplay()** 
  + - 1. Marks the current window as needing to be redisplayed.
* **glClearColor ()**
  + - 1. Specifies clear values for the color buffers.
* **glShadeModel ()**
  + - 1. Select flat or smooth shading. Specifies a symbolic value representing a shading technique. Accepted values are GL\_FLAT and GL\_SMOOTH.
* **glEnable()**
  + - 1. Enables the OpenGL capabilities, Specifies the conditions under which the pixels will be drawn.
* **glLightfv()**
  + - 1. Creates a new light source with specified parameter values.
* **gluQuadricDrawStyle()**
  + - 1. Specifies the draw style required for quadrics.
* **glMatrixMode ()**
  + - 1. Specifies which matrix is the current matrix.
* **glLoadIdentity()**
  + - 1. Pushes the identity matrix to the top of the matrix stack.
* **glutSwapBuffers()**
  + - 1. Swaps the buffers of the current window if double buffered.
* **glViewport()**
  + - 1. Sets the viewport.
* **glutInitDisplayMode ()**
  + - 1. Sets the initial display mode.
* **glutInitWindowSize ()** and **glutInitWindowPosition ()**
  + - 1. Set the initial window size and position respectively.
* **glutCreateWindow()**
  + - 1. Creates a top level window with the window name as specified.
* **glutAddMenuEntry()**
  + - 1. Adds a menu entry to the bottom of the current menu.
* **glutAttachMenu()**
  + - 1. Attaches a mouse button for the current window to the identifier of the current menu**.**
* **glutDisplayFunc()**
  + - 1. Sets the display callback for the current window.
* **glutReshapeFunc()**
  + - 1. Sets the reshape callback for the current window.
* **glutMainLoop()**
  + - 1. 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**

Graph Coloring Algorithm

**isValid(vertex, colorList, col)**

**Input:**Vertex, colorList to check, and color, which is trying to assign.

**Output:** True if the color assigning is valid, otherwise false.

Begin

   for all vertices v of the graph, do

      if there is an edge between v and i, and col = colorList[i], then

         return false

   done

   return true

End

**graphColoring(colors, colorList, vertex)**

**Input:**Most possible colors, the list for which vertices are colored with which color, and the starting vertex.

**Output:** True, when colors are assigned, otherwise false.

Begin

   if all vertices are checked, then

      return true

   for all colors col from available colors, do

      if isValid(vertex, color, col), then

         add col to the colorList for vertex

         if graphColoring(colors, colorList, vertex+1) = true, then

            return true

         remove color for vertex

   done

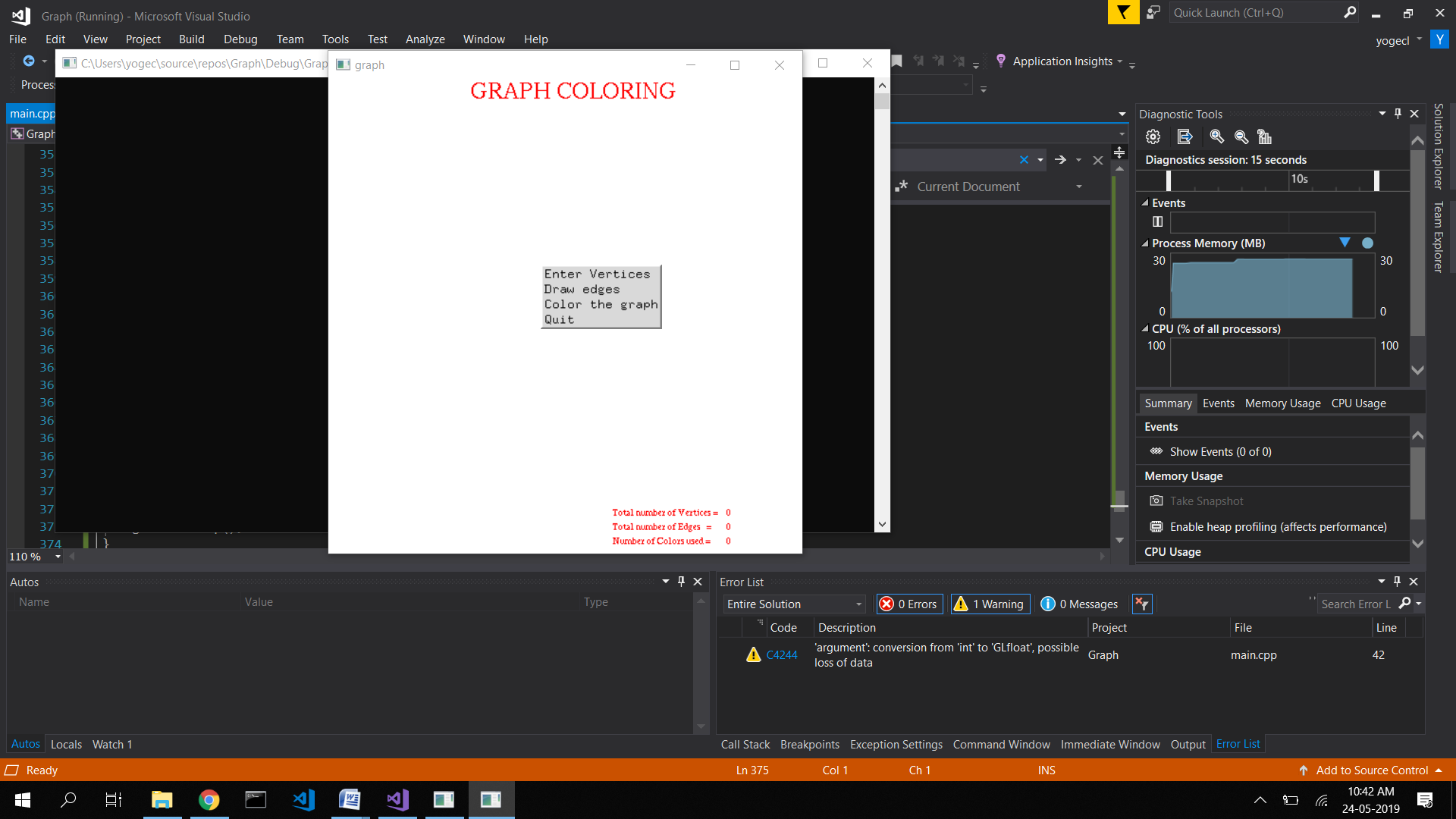
   return false

End

**Chapter 5**

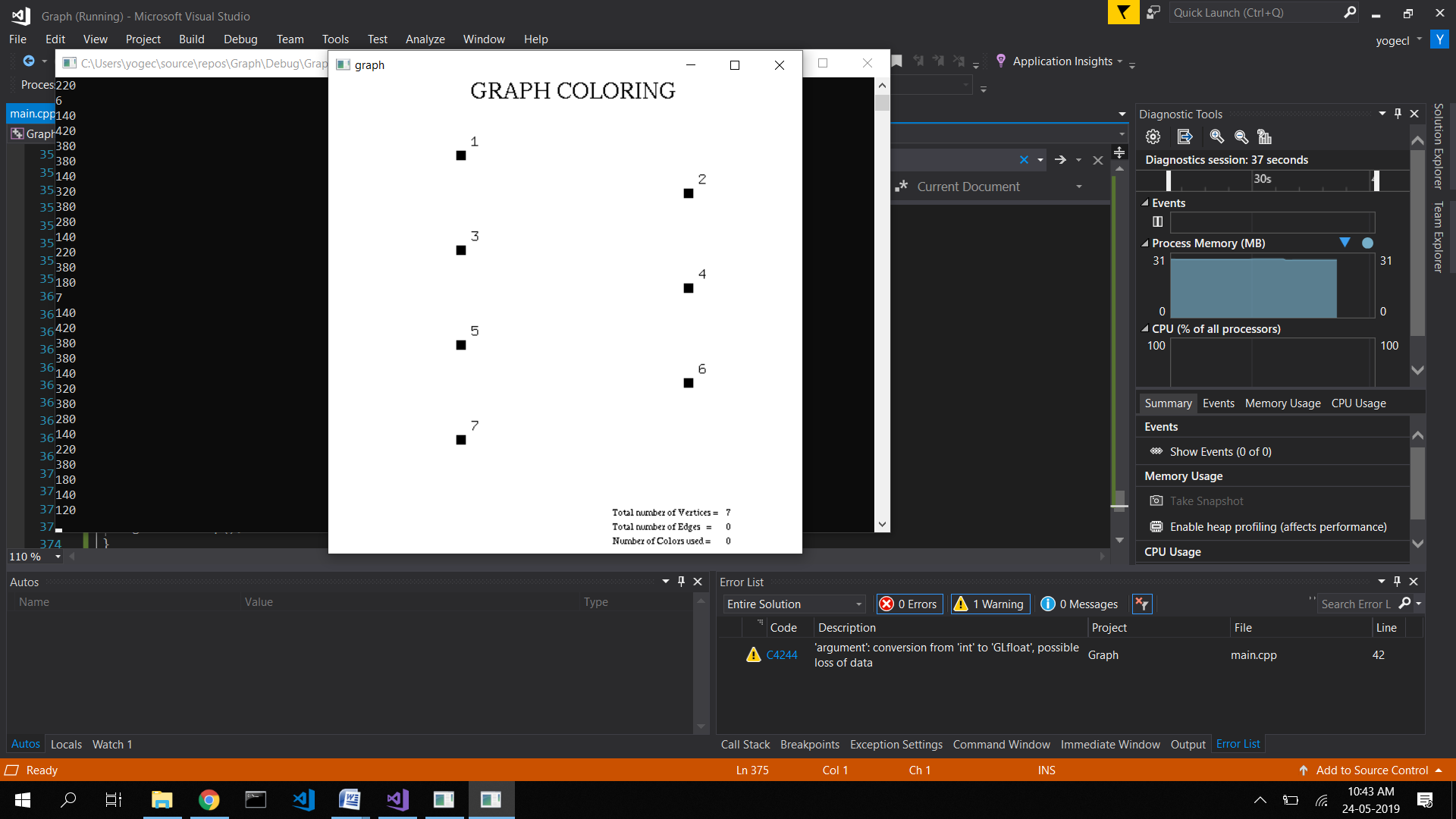
**Results and Discussions**

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



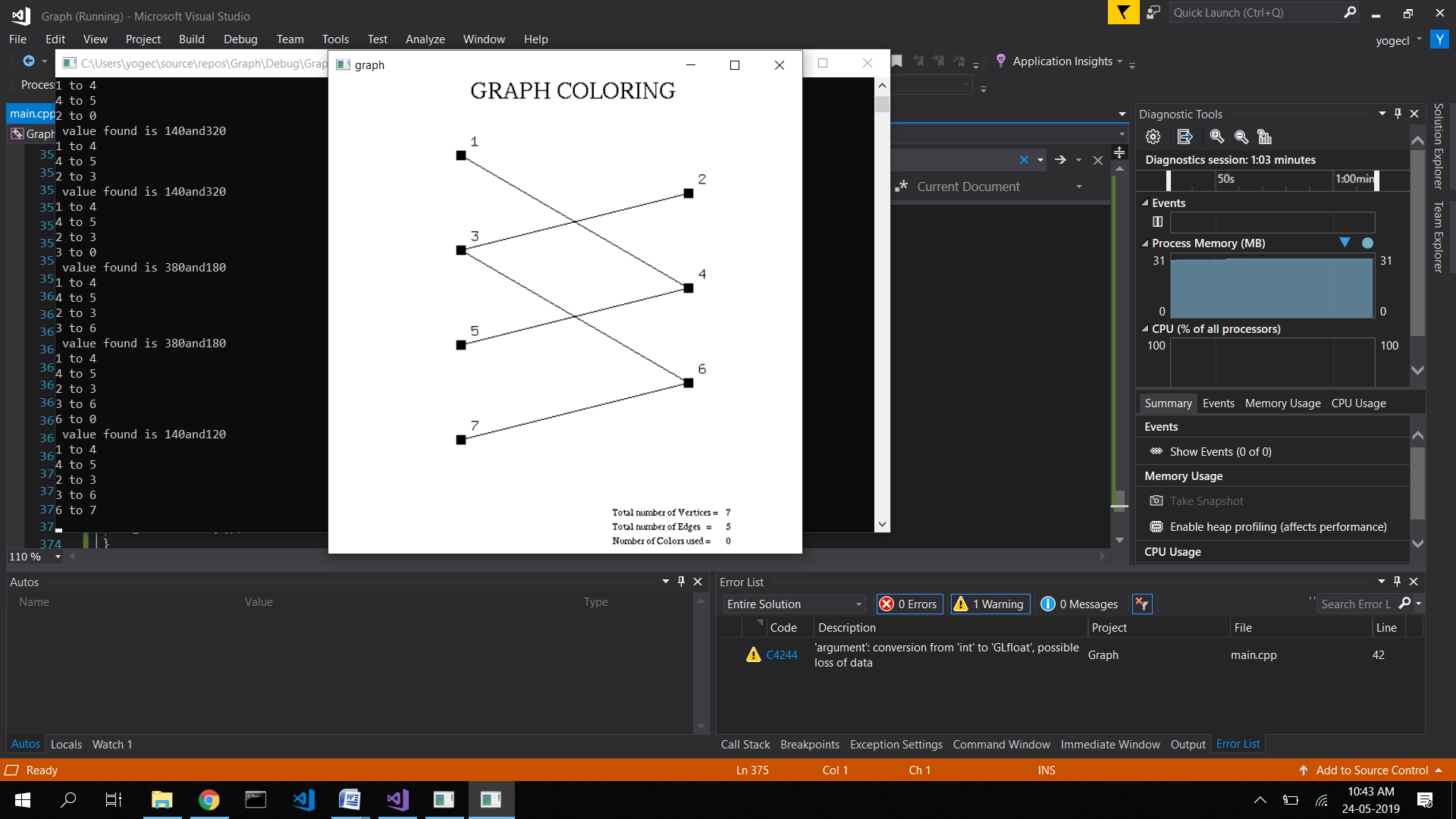
**Figure 5.1:** Main Menu

Figure5.1 shows the main menu of the program. From here the user can navigate to various sections of the program by selecting one of the available options.



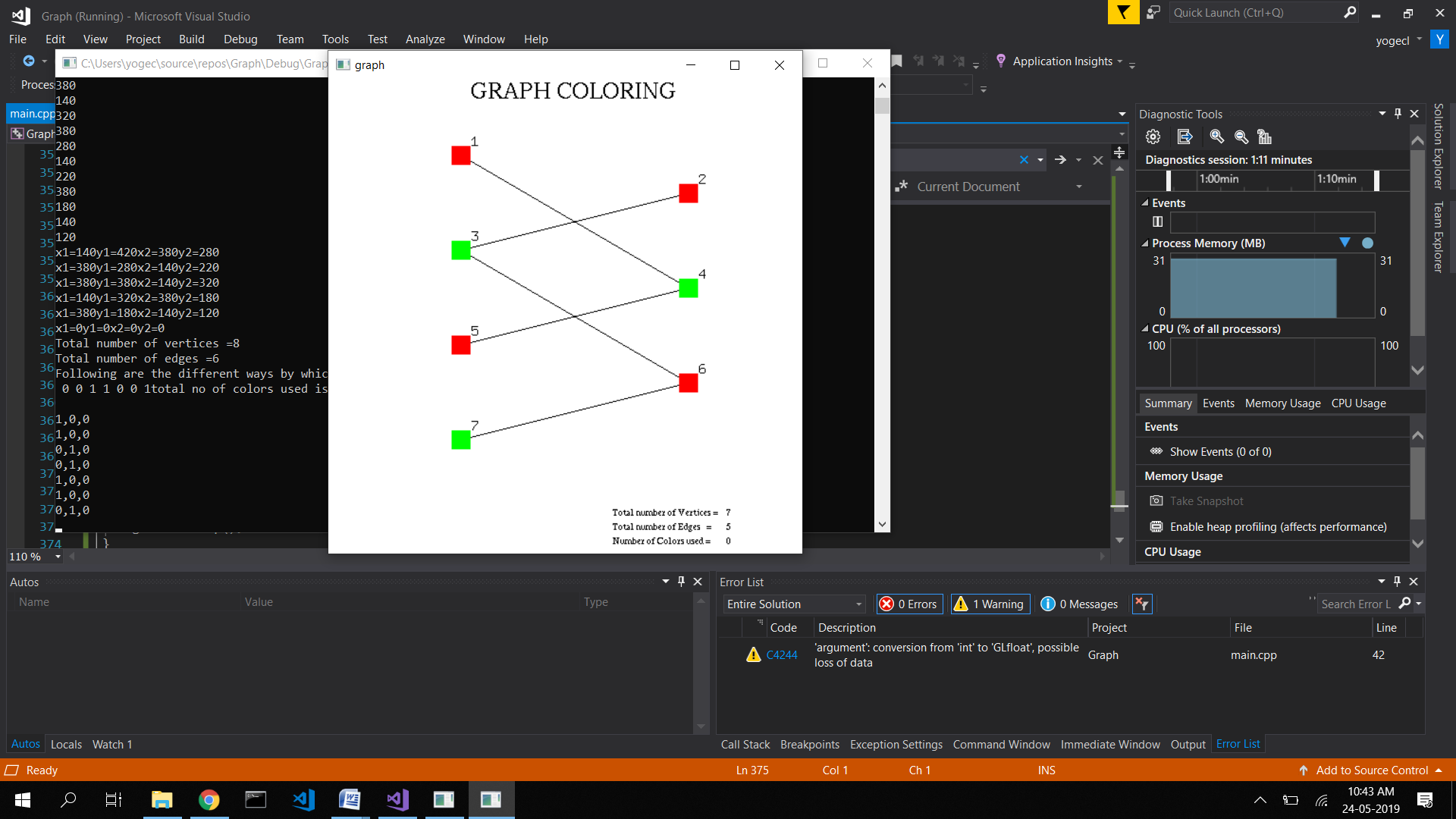
**Figure 5.2:** Main window where vertices are drawn

Figure 5.2 gives a visual representation of the main window where vertices are drawn.

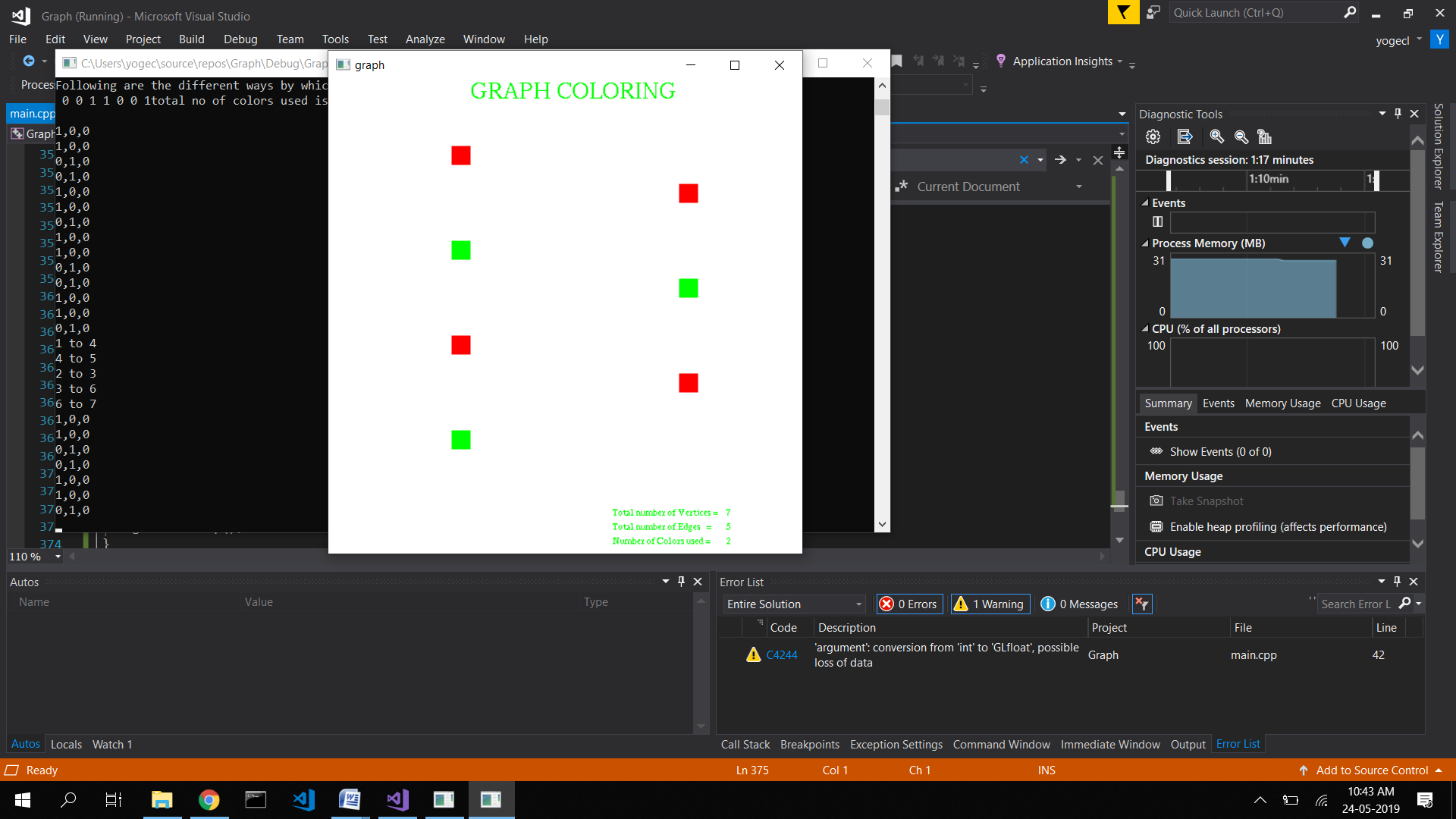


**Figure 5.3:** Connecting edges among the vertices

Figure 5.3 is a screenshot which describes the bipartite graph with edges connected

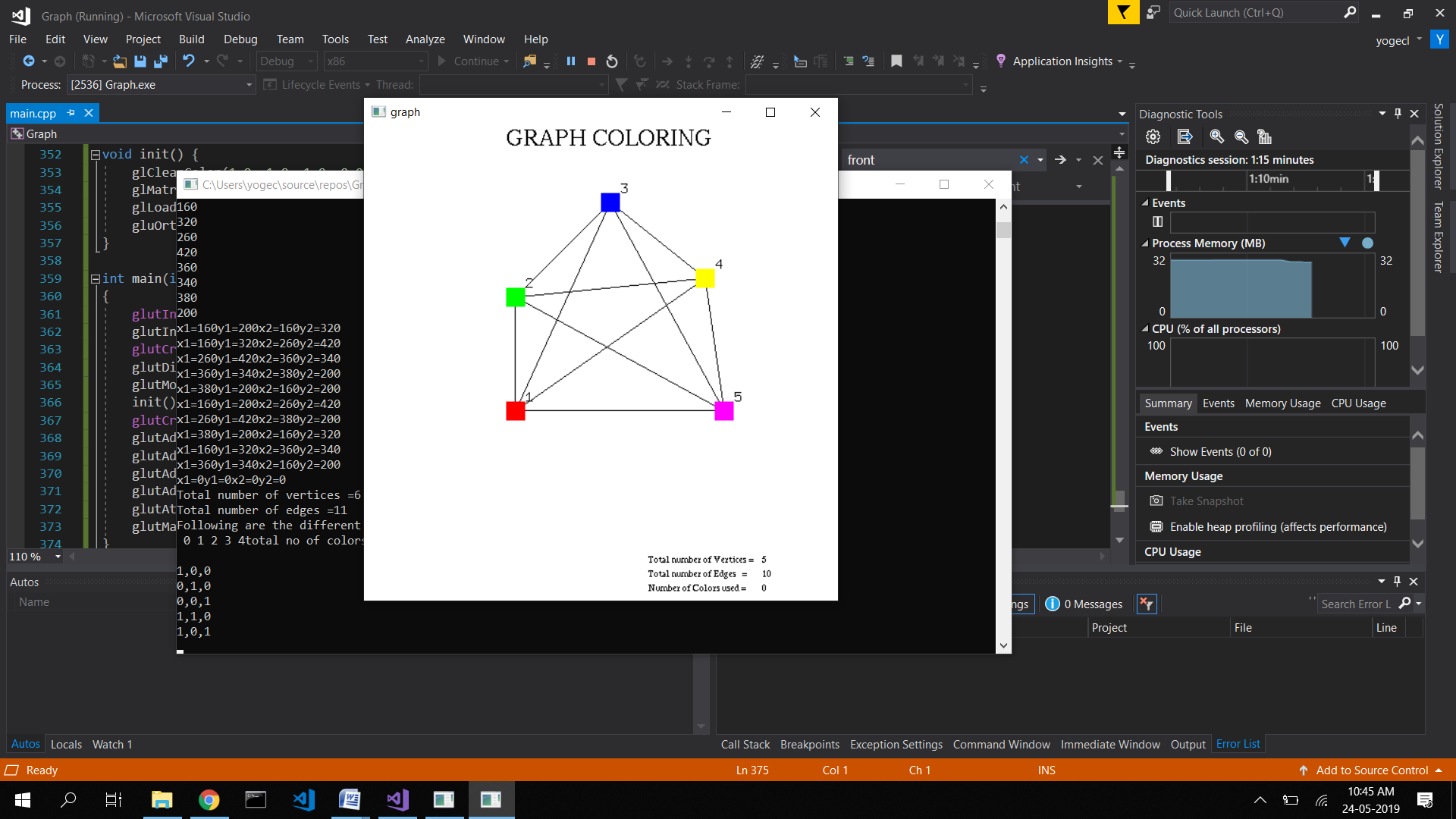


**Figure 5.4(a):** Graph Coloring

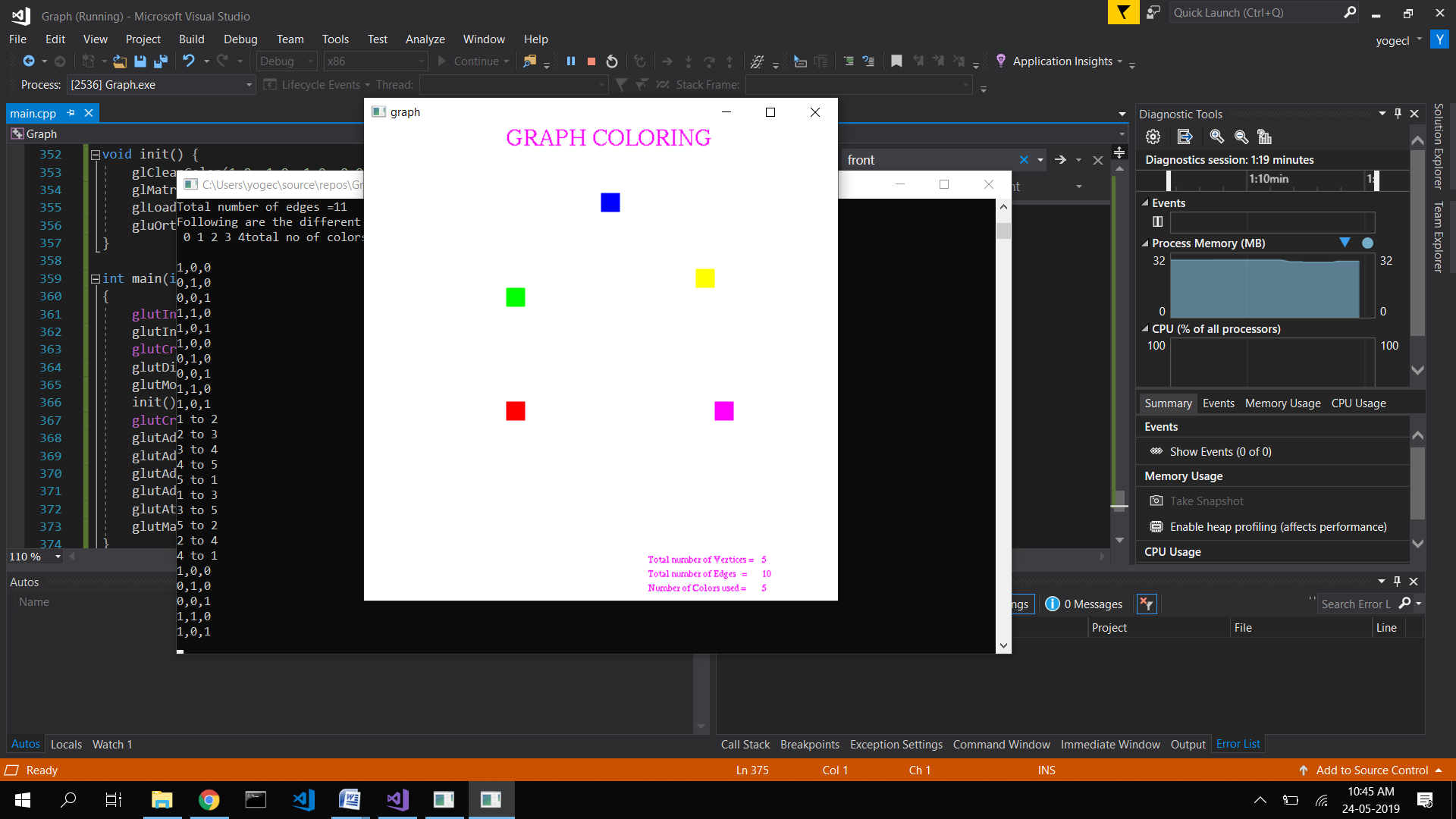


**Figure 5.4(b):** Displaying the chromatic number

Figure 5.4 shows successful demonstration of graph coloring of bipartite graph



**Figure 5.5(a):** Graph Coloring of complete graph with 5 vertices

****

**Figure 5.5(b):** Displaying the chromatic number

In Figure5.5 shows successful demonstration of graph coloring of complete graph with 5 vertices

**Chapter 6**

**Conclusion and Future Enhancements**

**6.1 Conclusion**

This mini project on Graph Coloring Algorithm using OpenGL is a reliable graphics package that provides the user with the basic working of the protocol. It provides a visual representation of the algorithm for a few standard graphs. The user-friendly interface allows the user to interact with it very effectively.

**6.2 Future Enhancements**

The future scope of this project is vast. Using this project as base, similar problems such as edge coloring can be implemented with very little modifications to the original code.

**References**

**Web Sources**

[1]:<https://en.wikipedia.org/wiki/Computer_graphics#Concepts_and_principles>

[2]:<https://en.wikipedia.org/wiki/OpenGL_Utility_Toolkit>

[3]: <https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview>

[4]: <https://www.geeksforgeeks.org/graph-coloring-set-2-greedy-algorithm/>

**Books**

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

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