**Department of Computing**

**CS-361: Computer Graphics  
  
Class: BSCS-12ABC & SE12AB**

**Lab 02: Rasterization Basics and Line Drawing Algorithms**

# CLO-02: Develop 2D and 3D graphical applications using programming libraries and tools.

# CLO-03: Implement algorithms for rendering, transformations, and animations.

**Date: 04th Feb 2025**

**Time: 02:00 PM – 04:50 PM**

# Instructor: Dr. Sidra Sutana

# Lab Engineer: Mr. Aftab Farooq

# Name: Ahmed Mohiuddin Shah

# CMS ID: 415216

# Section: BSCS-12-A

# CMS ID: 415216

**Lab 02: Rasterization Basics and Line Drawing Algorithms**

### **Lab Objective:**

This lab is designed to:

* Reinforce the concepts of rasterization by drawing basic shapes.
* Implement and compare line drawing algorithms (DDA and Bresenham’s) for a better understanding of their differences.

## Tools/Software Requirement:

* **Operating System:**
  + Windows / macOS / Linux (Ubuntu recommended)
* **Development Environment:**
  + **Windows:** [Code::Blocks](http://www.codeblocks.org/) or [Visual Studio](https://visualstudio.microsoft.com/)
  + **macOS:** [Xcode](https://developer.apple.com/xcode/)
  + **Linux:** GCC and g++ compilers
* **Graphics Libraries:**
  + **OpenGL** (built-in on macOS and Linux, available in Windows IDEs)
  + **GLUT** (OpenGL Utility Toolkit)
  + **GLEW** (OpenGL Extension Wrangler Library)
* **Package Manager (for macOS/Linux):**
  + **Homebrew** (macOS): brew install freeglut glew
  + **APT** (Linux): sudo apt-get install freeglut3-dev glew-utils
* **Compilers:**
  + **Windows:** MinGW (for Code::Blocks) or Microsoft C++ Compiler (for Visual Studio)
  + **macOS/Linux:** GCC/G++

**Task 1: Drawing Basic Shapes**:  
 **1.** **Draw a Rectangle**: Write a function to draw a rectangle on the screen by specifying its corner coordinates. Ensure that you calculate and draw each side as a line.

* **Input**: Coordinates of two opposite corners of the rectangle.
* **Output**: A rectangle displayed on the screen using OpenGL.

**2.** **Draw a Circle Using Midpoint Circle Algorithm**: Implement the **Midpoint Circle Algorithm** to draw a circle.

* **Input**: Center point and radius.
* **Output**: A circle rendered using the Midpoint Circle Algorithm.
* **Hint**: Use symmetry properties of circles to minimize calculations.

**Solution:**

**Code:**

#include <GL/freeglut.h>

// Bresenham’s Line Algorithm

void drawLine(int x1, int y1, int x2, int y2)

{

    int dx = abs(x2 - x1), dy = abs(y2 - y1);

    int sx = (x1 < x2) ? 1 : -1;

    int sy = (y1 < y2) ? 1 : -1;

    int err = dx - dy;

    glBegin(GL\_POINTS);

    while (x1 != x2 || y1 != y2)

    {

        glVertex2i(x1, y1);

        int e2 = 2 \* err;

        if (e2 > -dy)

        {

            err -= dy;

            x1 += sx;

        }

        if (e2 < dx)

        {

            err += dx;

            y1 += sy;

        }

    }

    glEnd();

    glFlush();

}

void drawRectangle(int top\_left\_x, int top\_left\_y, int bottom\_right\_x, int bottom\_right\_y)

{

    if (top\_left\_x < 0 || top\_left\_y < 0 || bottom\_right\_x < 0 || bottom\_right\_y < 0)

    {

        // coordinates shouldn't be negative

        return;

    }

    // top line

    drawLine(top\_left\_x, top\_left\_y, top\_left\_x + bottom\_right\_x, top\_left\_y);

    drawLine(top\_left\_x + bottom\_right\_x, top\_left\_y, top\_left\_x + bottom\_right\_x, top\_left\_y + bottom\_right\_y);

    drawLine(top\_left\_x + bottom\_right\_x, top\_left\_y + bottom\_right\_y, top\_left\_x, top\_left\_y + bottom\_right\_y);

    drawLine(top\_left\_x, top\_left\_y, top\_left\_x, top\_left\_y + bottom\_right\_y);

}

void drawCirclePoints(int x, int y, int center\_x, int center\_y)

{

    // We used the 8 point symmetry of the circle to save on number of calculations

    glBegin(GL\_POINTS);

    glVertex2i(center\_x + x, center\_y + y);

    glVertex2i(center\_x - x, center\_y + y);

    glVertex2i(center\_x + x, center\_y - y);

    glVertex2i(center\_x - x, center\_y - y);

    glVertex2i(center\_x + y, center\_y + x);

    glVertex2i(center\_x - y, center\_y + x);

    glVertex2i(center\_x + y, center\_y - x);

    glVertex2i(center\_x - y, center\_y - x);

    glEnd();

}

void drawCircle(int center\_x, int center\_y, int radius)

{

    int x = 0, y = radius;

    int p = 1 - radius; // Midpoint decision parameter

    drawCirclePoints(x, y, center\_x, center\_y);

    while (x < y)

    {

        x++;

        if (p < 0)

            p += 2 \* x + 1;

        else

        {

            y--;

            p += 2 \* x - 2 \* y + 1;

        }

        drawCirclePoints(x, y, center\_x, center\_y);

    }

    glFlush();

}

void display()

{

    glClear(GL\_COLOR\_BUFFER\_BIT);

    // Drawing Green Rectangles

    glColor3f(0.0f, 1.0f, 0.0f); // Set Line Color to green

    drawRectangle(10, 10, 300, 250);

    drawRectangle(20, 20, 50, 40);

    // Drawing red Circles

    glColor3f(1.0f, 0.0f, 0.0f);

    drawCircle(400, 400, 100);

    drawCircle(200, 400, 60);

}

int main(int argc, char \*\*argv)

{

    glutInit(&argc, argv);

    glutInitDisplayMode(GLUT\_SINGLE | GLUT\_RGB);

    glutInitWindowSize(700, 700); // Here we set the window size

    glutCreateWindow("Task 1");

    glClearColor(0.0f, 0.0f, 0.0f, 1.0f); // Set background color to black

    glMatrixMode(GL\_PROJECTION);          // Switch to projection matrix

    glLoadIdentity();                     // Reset the matrix

    gluOrtho2D(0, 700, 700, 0);           // Set orthographic projection

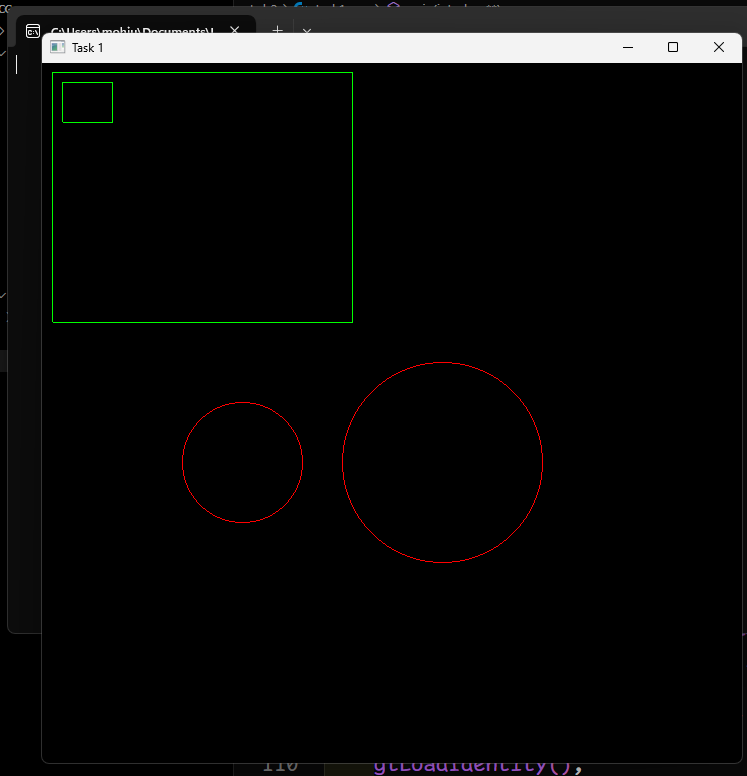
    glutDisplayFunc(display);

    glutMainLoop();

    return 0;

}

**Output:**

****

#### **Task 2: Implement DDA Line Drawing for Multiple Lines**

1. **Draw a Star Shape Using DDA**: Using the DDA algorithm, write a program that draws a **star shape**. A star can be constructed by connecting multiple line segments between points on a circle or any predefined coordinates.
   * **Input**: Coordinates for the star points (e.g., 5-pointed or 6-pointed star).
   * **Output**: A star rendered by drawing multiple lines.
2. **Draw a Polygon Using DDA**: Implement the DDA algorithm to draw a **polygon** with n sides.
   * **Input**: Number of sides and radius.
   * **Output**: A regular polygon with specified sides rendered by connecting lines between points.

**Solution:**

**Code:**

#include <GL/freeglut.h>

#include <math.h>

#define PI 3.14159265358979323846

// DDA Algorithm for Line Drawing

void drawLineDDA(float x1, float y1, float x2, float y2)

{

    float dx = x2 - x1;

    float dy = y2 - y1;

    float steps = (abs(dx) > abs(dy)) ? abs(dx) : abs(dy);

    float xIncrement = dx / steps;

    float yIncrement = dy / steps;

    float x = x1, y = y1;

    glBegin(GL\_POINTS);

    for (int i = 0; i <= steps; i++)

    {

        glVertex2i(round(x), round(y));

        x += xIncrement;

        y += yIncrement;

    }

    glEnd();

    glFlush();

}

void drawStarPolygonal(float cx, float cy, float radius)

{

    const int numPoints = 5;          // 5-pointed star

    float angleStep = PI / numPoints; // 36 degrees per step

    float innerRadius = radius / 2.5; // Smaller radius for inner points

    float points[10][2];

    // Compute the outer and inner points

    for (int i = 0; i < 10; i++)

    {

        float r = (i % 2 == 0) ? radius : innerRadius;           // Alternate between outer and inner radius

        points[i][0] = cx + r \* cos(i \* angleStep \* 2 - PI / 2); // Adjusted angle for correct orientation

        points[i][1] = cy + r \* sin(i \* angleStep \* 2 - PI / 2);

    }

    // Correct star connections (every second point)

    for (int i = 0; i < 10; i++)

    {

        drawLineDDA(points[i][0], points[i][1], points[(i + 1) % 10][0], points[(i + 1) % 10][1]);

    }

}

void drawStarCircle(int N, float cx, float cy, float radius)

{

    float points[N][2];           // Store coordinates of N points

    float angleStep = 2 \* PI / N; // Angle step between points

    // Compute N equidistant points on the circle

    for (int i = 0; i < N; i++)

    {

        points[i][0] = cx + radius \* cos(i \* angleStep);

        points[i][1] = cy + radius \* sin(i \* angleStep);

    }

    // Connect every non-adjacent pair

    for (int i = 0; i < N; i++)

    {

        for (int j = i + 2; j < N; j++)

        { // Skip adjacent points (i+1 and i-1)

            if (j != (i - 1 + N) % N)

            { // Avoid connecting last point to first

                drawLineDDA(points[i][0], points[i][1], points[j][0], points[j][1]);

            }

        }

    }

}

void drawPolygon(int sides, float cx, float cy, float radius)

{

    float angleStep = 2 \* PI / sides;

    float points[sides][2];

    for (int i = 0; i < sides; i++)

    {

        points[i][0] = cx + radius \* cos(i \* angleStep);

        points[i][1] = cy + radius \* sin(i \* angleStep);

    }

    for (int i = 0; i < sides; i++)

    {

        drawLineDDA(points[i][0], points[i][1], points[(i + 1) % sides][0], points[(i + 1) % sides][1]);

    }

}

void display()

{

    glClear(GL\_COLOR\_BUFFER\_BIT);

    // Drawing Green Polygons

    glColor3f(0.0f, 1.0f, 0.0f);  // Set Line Color to green

    drawPolygon(5, 70, 70, 60);   // Five Sided

    drawPolygon(6, 300, 300, 50); // Six Sided

    // Drawing red Star

    glColor3f(1.0f, 0.0f, 0.0f);

    drawStarPolygonal(400, 500, 100); // Five Point Star

    drawStarCircle(5, 200, 400, 50);  // Five Point star using circle method

    drawStarCircle(6, 70, 500, 60);  // Six Point star using circle method

}

int main(int argc, char \*\*argv)

{

    glutInit(&argc, argv);

    glutInitDisplayMode(GLUT\_SINGLE | GLUT\_RGB);

    glutInitWindowSize(700, 700); // Here we set the window size

    glutCreateWindow("Task 1");

    glClearColor(0.0f, 0.0f, 0.0f, 1.0f); // Set background color to black

    glMatrixMode(GL\_PROJECTION);          // Switch to projection matrix

    glLoadIdentity();                     // Reset the matrix

    gluOrtho2D(0, 700, 700, 0);           // Set orthographic projection

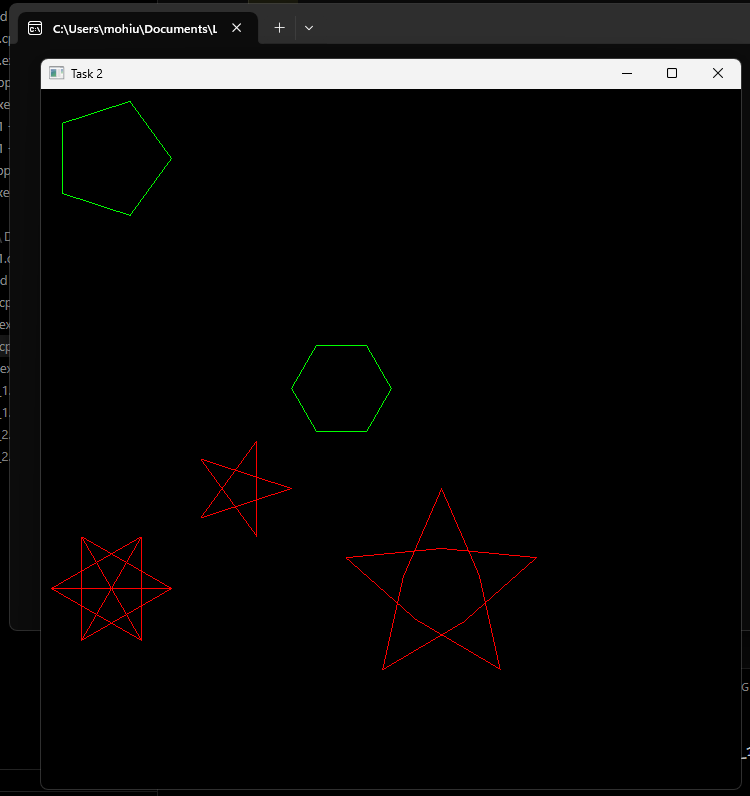
    glutDisplayFunc(display);

    glutMainLoop();

    return 0;

}

**Output:**

****

#### **Task 3: Compare DDA and Bresenham’s Line Algorithm**

1. **Draw Parallel Lines Using DDA and Bresenham**: Create a program that draws **two parallel lines**: one using the DDA algorithm and the other using Bresenham’s Line Algorithm. Compare the visual results, speed, and pixel-level differences.
   * **Input**: Start and end points of both lines.
   * **Output**: Parallel lines drawn using DDA and Bresenham’s algorithms.
2. **Draw a Triangle with DDA and Bresenham**: Write code to draw a triangle with DDA for one side and Bresenham’s algorithm for the other two sides. Analyze the visual consistency and performance between the two algorithms.
   * **Input**: Coordinates for the three vertices of the triangle.
   * **Output**: A triangle drawn using a combination of both algorithms.

**Solution:**

From the results we obtains we can see negligible differences in the quality of our line that are rendered but int speed and time complexity the Bresenham’s Line Algorithm wins a it took on average 0 ms of time to render the line whereas the DDA Algortihm took considerably much more time to render the line as evidenced by the times we recorded.

**Code:**

**For Parallel Lines:**

#include <GL/freeglut.h>

#include <math.h>

#include <iostream>

#include <chrono>

template <typename Func, typename... Args>

double calculateExecutionTime(Func func, Args... args)

{

    auto start = std::chrono::high\_resolution\_clock::now();

    func(args...);

    auto end = std::chrono::high\_resolution\_clock::now();

    std::chrono::duration<double, std::milli> duration = end - start;

    return duration.count(); // Return time in milliseconds

}

// DDA Algorithm for Line Drawing

void drawLineDDA(float x1, float y1, float x2, float y2)

{

    float dx = x2 - x1;

    float dy = y2 - y1;

    float steps = (abs(dx) > abs(dy)) ? abs(dx) : abs(dy);

    float xIncrement = dx / steps;

    float yIncrement = dy / steps;

    float x = x1, y = y1;

    glBegin(GL\_POINTS);

    for (int i = 0; i <= steps; i++)

    {

        glVertex2i(round(x), round(y));

        x += xIncrement;

        y += yIncrement;

    }

    glEnd();

    glFlush();

}

// Bresenham’s Line Algorithm

void drawLineBL(int x1, int y1, int x2, int y2)

{

    int dx = abs(x2 - x1), dy = abs(y2 - y1);

    int sx = (x1 < x2) ? 1 : -1;

    int sy = (y1 < y2) ? 1 : -1;

    int err = dx - dy;

    glBegin(GL\_POINTS);

    while (x1 != x2 || y1 != y2)

    {

        glVertex2i(x1, y1);

        int e2 = 2 \* err;

        if (e2 > -dy)

        {

            err -= dy;

            x1 += sx;

        }

        if (e2 < dx)

        {

            err += dx;

            y1 += sy;

        }

    }

    glEnd();

    glFlush();

}

// Function to draw parallel lines using DDA and Bresenham

void drawParallelLines(int x1, int y1, int x2, int y2, int spacing)

{

    glColor3f(1, 0, 0); // Red - DDA

    int dda\_time = calculateExecutionTime(drawLineDDA, x1, y1, x2, y2);

    std::cout << "DDA Time: " << dda\_time << "ms\n";

    glColor3f(0, 0, 1); // Blue - Bresenham

    int bl\_time = calculateExecutionTime(drawLineBL, x1, y1 + spacing, x2, y2 + spacing);

    std::cout << "BL Time: " << bl\_time << "ms\n";

}

void display()

{

    glClear(GL\_COLOR\_BUFFER\_BIT);

    drawParallelLines(100, 200, 400, 350, 20);

}

int main(int argc, char \*\*argv)

{

    glutInit(&argc, argv);

    glutInitDisplayMode(GLUT\_SINGLE | GLUT\_RGB);

    glutInitWindowSize(700, 700); // Here we set the window size

    glutCreateWindow("Task 1");

    glClearColor(0.0f, 0.0f, 0.0f, 1.0f); // Set background color to black

    glMatrixMode(GL\_PROJECTION);          // Switch to projection matrix

    glLoadIdentity();                     // Reset the matrix

    gluOrtho2D(0, 700, 700, 0);           // Set orthographic projection

    glutDisplayFunc(display);

    glutMainLoop();

    return 0;

}

**Output:**

**A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated**

**For Triangle:**

#include <GL/freeglut.h>

#include <math.h>

#include <iostream>

#include <chrono>

template <typename Func, typename... Args>

double calculateExecutionTime(Func func, Args... args)

{

    auto start = std::chrono::high\_resolution\_clock::now();

    func(args...);

    auto end = std::chrono::high\_resolution\_clock::now();

    std::chrono::duration<double, std::milli> duration = end - start;

    return duration.count(); // Return time in milliseconds

}

// DDA Algorithm for Line Drawing

void drawLineDDA(float x1, float y1, float x2, float y2)

{

    float dx = x2 - x1;

    float dy = y2 - y1;

    float steps = (abs(dx) > abs(dy)) ? abs(dx) : abs(dy);

    float xIncrement = dx / steps;

    float yIncrement = dy / steps;

    float x = x1, y = y1;

    glBegin(GL\_POINTS);

    for (int i = 0; i <= steps; i++)

    {

        glVertex2i(round(x), round(y));

        x += xIncrement;

        y += yIncrement;

    }

    glEnd();

    glFlush();

}

// Bresenham’s Line Algorithm

void drawLineBL(int x1, int y1, int x2, int y2)

{

    int dx = abs(x2 - x1), dy = abs(y2 - y1);

    int sx = (x1 < x2) ? 1 : -1;

    int sy = (y1 < y2) ? 1 : -1;

    int err = dx - dy;

    glBegin(GL\_POINTS);

    while (x1 != x2 || y1 != y2)

    {

        glVertex2i(x1, y1);

        int e2 = 2 \* err;

        if (e2 > -dy)

        {

            err -= dy;

            x1 += sx;

        }

        if (e2 < dx)

        {

            err += dx;

            y1 += sy;

        }

    }

    glEnd();

    glFlush();

}

// Function to draw a triangle with DDA for one side and Bresenham for the other two

void drawTriangle(int x1, int y1, int x2, int y2, int x3, int y3)

{

    glColor3f(1, 0, 0); // Red - DDA

    int dda\_time = calculateExecutionTime(drawLineDDA, x1, y1, x2, y2);

    std::cout << "DDA Time: " << dda\_time << "ms\n";

    glColor3f(0, 1, 0); // Green - Bresenham

    int bl\_time = calculateExecutionTime(drawLineBL, x2, y2, x3, y3);

    std::cout << "BL Time 1: " << bl\_time << "ms\n";

    drawLineBL(x2, y2, x3, y3);

    glColor3f(0, 0, 1); // Blue - Bresenham

    bl\_time = calculateExecutionTime(drawLineBL, x3, y3, x1, y1);

    std::cout << "BL Time 2: " << bl\_time << "ms\n";

}

void display()

{

    glClear(GL\_COLOR\_BUFFER\_BIT);

    drawTriangle(200, 300, 100, 100, 300, 100);

}

int main(int argc, char \*\*argv)

{

    glutInit(&argc, argv);

    glutInitDisplayMode(GLUT\_SINGLE | GLUT\_RGB);

    glutInitWindowSize(700, 700); // Here we set the window size

    glutCreateWindow("Task 1");

    glClearColor(0.0f, 0.0f, 0.0f, 1.0f); // Set background color to black

    glMatrixMode(GL\_PROJECTION);          // Switch to projection matrix

    glLoadIdentity();                     // Reset the matrix

    gluOrtho2D(0, 700, 700, 0);           // Set orthographic projection

    glutDisplayFunc(display);

    glutMainLoop();

    return 0;

}

**Output:**

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

### **Deliverables:**

 Compile a single word document by filling in the solution part and submit this Word file on LMS

 Submit your code files (.cpp) for both the DDA and Bresenham’s algorithms.

 Include screenshots of the program outputs.

 Submit your Lab Word File and code files seperately on submission link.

# Lab Rubrics

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Lab Rubrics for Lab 2 (Rasterization Basics and Line Drawing Algorithms) | | | | | |
|  | | | | | |
| **Sr.**  **No.** | **Assessment** | **Unacceptable (0 Marks)** | **Does Not Meet Expectations (1/2 Marks)** | **Meets Expectations (3/4 Marks)** | **Exceeds Expectations (5 Marks)** |
| **1** | **Illustrating the basic understanding of semantics and syntax**  **(CLO2, PLO3)** | The student did not submit any work.  OR  The student plagiarized the solution and/or used unfair means. | The student is unable to demonstrate the understanding of syntax of C language and is unable to write an executable code.  The student is not able to understand the structure of a program at all. | The student demonstrates some understanding of syntax of C language and is able to write a code with few errors.  The student is able to understand the structure but still learning the syntax. | The student demonstrates good understanding of syntax of C language and is able to write executable code without help  The student is able to understand the structure and is able to identify problems in the code  when introduced |
| **2** | **Software Tool Usage**  **(CLO3-PLO5)** | The student demonstrates a lack of understanding of tool usage.  Implementation has syntax/semantic/runtime errors, and the student is unable to debug and correct the errors.  The code has inadequate comments and variable names and does not adhere to the coding standards.  No Error handling has been performed.  Documentation is poorly structured. | The student demonstrates some understanding of tool usage.  The codes are correct in terms of their syntax, however, the program output is not always correct in all test cases.  The code has limited comments and inconsistent variable names and may not adhere to the coding standards.  Some Error handling has been performed.  Documentation is adequately structured. | The student demonstrates a good understanding of tool usage.  Furthermore, his/her coding is complete and functional, and the program output is correct in all test cases.  The code has sufficient comments and consistent variable names and reasonably adhere to the coding standards.  Adequate Error handling has been performed.  Documentation is well structured. |