**Department of Computing**

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

**Lab 08: Spatial Data Structures**

**CLO 2 -** Apply mathematical and algorithmic principles to implement basic computer graphics techniques, such as line drawing and shading.

**CLO 3-** Develop interactive graphics applications using modern graphics APIs such as OpenGL or DirectX.

**CLO 4 -** Design and implement 2D and 3D graphical solutions for real-world problems.

**Date: 18th Mar 2025**

**Time: 12:00 PM – 14:30 PM**

# Instructor: Dr. Sidra Sutana

# Lab Engineer: Mr. Aftab Farooq

**Name:** Ahmed Mohiuddin Shah

**CMS ID:** 415216

**Section:** BSCS-12-A

**Lab:** 8

**Lab 08: Spatial Data Structures**

### **Introduction:** Spatial data structures like Bounding Volumes and Uniform Grids are essential for managing graphics scenes efficiently. Bounding volumes simplify collision detection by enclosing objects in geometric shapes, while uniform grids accelerate object searches by dividing space into cells. These techniques enhance performance in applications like ray tracing and object culling.

### **Lab Objective:**

## In this lab, students will implement Bounding Volumes (AABB and Bounding Spheres) and Uniform Grids for efficient object management. They will visualize these structures in 2D/3D environments and apply them to optimize collision detection in dynamic scenes.

## 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++
* A programming environment (e.g., Visual Studio, PyCharm, or any IDE of your choice).
* A graphics library (optional, e.g., OpenGL, SDL, or a simple image library for saving images).
* Basic knowledge of Spatial Data Structures.

## Prerequisites :

 2D/3D coordinate systems

 Vector mathematics  
  Collision detection techniques

**Lab Tasks :**

**Task 1: Implementing Bounding Volumes**

* **Objective:** Develop a program that demonstrates the use of bounding volumes for efficient collision detection.
* **Steps:**
  + Create a 2D environment with multiple objects.
  + Implement Bounding Boxes (AABB) and Bounding Circles around the objects.
  + Allow users to move objects and visualize collision detection using these bounding volumes.

**Solution:**

**Code:**

#include <GL/glut.h>

#include <vector>

#include <cmath>

#include <iostream>

// Struct for a 2D point

*struct* Point {

*float* x, y;

};

// Struct for Axis-Aligned Bounding Box (AABB)

*struct* AABB {

   Point min, max;

*bool* intersects(const AABB& *other*) const {

       return !(max.x < other.min.x || min.x > other.max.x ||

           max.y < other.min.y || min.y > other.max.y);

   }

};

// Struct for Spherical Bounding Box

*struct* BoundingCircle {

   Point center;

*float* radius;

*bool* intersects(const BoundingCircle& *other*) const {

*float* dx = center.x - other.center.x;

*float* dy = center.y - other.center.y;

*float* distanceSquared = dx \* dx + dy \* dy;

*float* radiusSum = radius + other.radius;

       return distanceSquared <= radiusSum \* radiusSum;

   }

};

// Struct for Shapes

*struct* Shape {

   std::vector<Point> vertices;

   AABB aabb;

   BoundingCircle circle;

*float* r, g, b;

*void* updateBoundingVolumes() {

       if (vertices.empty()) return;

*float* minX = vertices[0].x, maxX = vertices[0].x;

*float* minY = vertices[0].y, maxY = vertices[0].y;

       for (const *auto*& v : vertices) {

           if (v.x < minX) minX = v.x;

           if (v.x > maxX) maxX = v.x;

           if (v.y < minY) minY = v.y;

           if (v.y > maxY) maxY = v.y;

       }

       aabb = { {minX, minY}, {maxX, maxY} };

       // Calculate bounding circle

       circle.center = { (minX + maxX) / 2, (minY + maxY) / 2 };

       circle.radius = std::sqrt((maxX - minX) \* (maxX - minX) + (maxY - minY) \* (maxY - minY)) / 2;

   }

};

std::vector<Shape> shapes;

*int* selectedShapeIndex = 0;

// Draw a shape

*void* drawShape(const Shape& *shape*) {

   glColor3f(shape.r, shape.g, shape.b);

   glBegin(GL\_POLYGON);

   for (const *auto*& v : shape.vertices)

       glVertex2f(v.x, v.y);

   glEnd();

}

// Draw AABB

*void* drawAABB(const AABB& *aabb*, *bool* *collided* = false) {

   glColor3f(0, 1, 0);

   if (collided) {

       glColor3f(1, 0, 0);

   }

   glBegin(GL\_LINE\_LOOP);

   glVertex2f(aabb.min.x, aabb.min.y);

   glVertex2f(aabb.max.x, aabb.min.y);

   glVertex2f(aabb.max.x, aabb.max.y);

   glVertex2f(aabb.min.x, aabb.max.y);

   glEnd();

}

// Draw Bounding Circle

*void* drawBoundingCircle(const BoundingCircle& *circle*, *bool* *collided* = false) {

   glColor3f(0, 1, 0);

   if (collided) {

       glColor3f(1, 0, 0);

   }

   glBegin(GL\_LINE\_LOOP);

   for (*int* i = 0; i < 360; i += 10) {

*float* theta = i \* 3.14159f / 180;

       glVertex2f(circle.center.x + cos(theta) \* circle.radius, circle.center.y + sin(theta) \* circle.radius);

   }

   glEnd();

}

// Render function

*void* display() {

   glClear(GL\_COLOR\_BUFFER\_BIT);

   for (*size\_t* i = 0; i < shapes.size(); i++) {

       drawShape(shapes[i]);

       for (*size\_t* j = 0; j < shapes.size(); j++) {

           if (i != j) {

               if (shapes[i].aabb.intersects(shapes[j].aabb)) {

                   drawAABB(shapes[j].aabb, true);

               }

               else {

                    drawAABB(shapes[j].aabb);

               }

                if (shapes[i].circle.intersects(shapes[j].circle)) {

                    drawBoundingCircle(shapes[j].circle, true);

                }

               else {

                   drawBoundingCircle(shapes[j].circle);

               }

           }

       }

   }

   glutSwapBuffers();

}

// Keyboard input

*void* keyboard(*unsigned* *char* *key*, *int* *x*, *int* *y*) {

   switch (key) {

   case 27: exit(0); break;

   case ' ': // Switch shape selection

       selectedShapeIndex = (selectedShapeIndex + 1) % shapes.size();

       break;

   case 'w':

       for (*auto*& v : shapes[selectedShapeIndex].vertices) v.y += 0.05f;

       break;

   case 's':

       for (*auto*& v : shapes[selectedShapeIndex].vertices) v.y -= 0.05f;

       break;

   case 'a':

       for (*auto*& v : shapes[selectedShapeIndex].vertices) v.x -= 0.05f;

       break;

   case 'd':

       for (*auto*& v : shapes[selectedShapeIndex].vertices) v.x += 0.05f;

       break;

   }

   shapes[selectedShapeIndex].updateBoundingVolumes();

   glutPostRedisplay();

}

*void* initShapes() {

   Shape triangle = { {{-0.1f, -0.1f}, {0.1f, -0.1f}, {0.0f, 0.1f}}, {}, {}, 1.0f, 0.6f, 0.0f };

   triangle.updateBoundingVolumes();

   shapes.push\_back(triangle);

    Shape hexagon = { {{-0.1f, -0.2f}, {0.1f, -0.2f}, {0.2f, 0.0f}, {0.1f, 0.2f}, {-0.1f, 0.2f}, {-0.2f, 0.0f}}, {}, {}, 1.0f, 0.6f, 0.0f };

    hexagon.updateBoundingVolumes();

    shapes.push\_back(hexagon);

}

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

   glutInit(&argc, argv);

   glutInitDisplayMode(GLUT\_DOUBLE | GLUT\_RGB);

   glutInitWindowSize(500, 500);

   glutCreateWindow("Task1 - Bounding Volumes");

   glMatrixMode(GL\_PROJECTION);

   glLoadIdentity();

   gluOrtho2D(-1, 1, -1, 1);

   initShapes();

   glutDisplayFunc(display);

   glutKeyboardFunc(keyboard);

   glutMainLoop();

   return 0;

}

**Output:**

**WASD to move  
SpaceBar to Select**

**A screenshot of a computer

AI-generated content may be incorrect.**

**A screenshot of a computer

AI-generated content may be incorrect.** A screenshot of a computer

AI-generated content may be incorrect.

**Task 2: Visualizing Uniform Grid**

* **Objective:** Implement a uniform grid structure for space partitioning.
* **Steps:**
  + Create a grid-based environment where multiple objects are placed.
  + Assign objects to grid cells based on their coordinates.
  + Visualize which objects fall in each cell and highlight potential collision regions.

**Solution:**

**Code:**

#include <GL/glut.h>

#include <vector>

#include <iostream>

const *int* GRID\_SIZE = 10; // Size of the grid (10x10)

const *int* CELL\_SIZE = 50; // Size of each cell in pixels

const *int* WINDOW\_WIDTH = GRID\_SIZE \* CELL\_SIZE;

const *int* WINDOW\_HEIGHT = GRID\_SIZE \* CELL\_SIZE;

*struct* Object {

*int* x, y; // Position in the grid

*int* width, height; // Size of the object

*bool* selected; // Whether the object is selected

};

std::vector<Object> objects; // List of objects in the scene

std::vector<std::vector<std::vector<Object\*>>> grid(GRID\_SIZE, std::vector<std::vector<Object\*>>(GRID\_SIZE)); // Uniform grid

*int* selectedObjectIndex = -1; // Index of the currently selected object

// Function to draw a rectangle

*void* drawRect(*int* *x*, *int* *y*, *int* *width*, *int* *height*) {

    glBegin(GL\_QUADS);

    glVertex2i(x, y);

    glVertex2i(x + width, y);

    glVertex2i(x + width, y + height);

    glVertex2i(x, y + height);

    glEnd();

}

// Function to draw the grid

*void* drawGrid() {

    glColor3f(0.5, 0.5, 0.5); // Gray color for the grid

    for (*int* i = 0; i <= GRID\_SIZE; ++i) {

        glBegin(GL\_LINES);

        glVertex2i(i \* CELL\_SIZE, 0);

        glVertex2i(i \* CELL\_SIZE, WINDOW\_HEIGHT);

        glVertex2i(0, i \* CELL\_SIZE);

        glVertex2i(WINDOW\_WIDTH, i \* CELL\_SIZE);

        glEnd();

    }

}

// Function to draw objects

*void* drawObjects() {

    for (const *auto*& obj : objects) {

        if (obj.selected) {

            glColor3f(0.0, 1.0, 0.0); // Green color for selected object

        }

        else {

            glColor3f(0.0, 0.0, 1.0); // Blue color for other objects

        }

        drawRect(obj.x, obj.y, obj.width, obj.height);

    }

}

// Function to check for collisions and highlight them

*void* checkCollisions() {

    for (*int* i = 0; i < GRID\_SIZE; ++i) {

        for (*int* j = 0; j < GRID\_SIZE; ++j) {

            if (grid[i][j].size() > 1) { // More than one object in the cell

                for (*auto* obj : grid[i][j]) {

                    glColor3f(1.0, 0.0, 0.0); // Red color for collisions

                    drawRect(obj->x, obj->y, obj->width, obj->height);

                }

            }

        }

    }

}

// Function to assign objects to grid cells

*void* assignObjectsToGrid() {

    // Clear the grid

    for (*int* i = 0; i < GRID\_SIZE; ++i) {

        for (*int* j = 0; j < GRID\_SIZE; ++j) {

            grid[i][j].clear();

        }

    }

    // Assign objects to grid cells

    for (*auto*& obj : objects) {

*int* gridX = obj.x / CELL\_SIZE;

*int* gridY = obj.y / CELL\_SIZE;

        if (gridX >= 0 && gridX < GRID\_SIZE && gridY >= 0 && gridY < GRID\_SIZE) {

            grid[gridX][gridY].push\_back(&obj);

        }

    }

}

// Function to handle keyboard input

*void* keyboard(*unsigned* *char* *key*, *int* *x*, *int* *y*) {

    if (selectedObjectIndex != -1) {

        Object& selectedObject = objects[selectedObjectIndex];

        switch (key) {

        case 'w': // Move up

            selectedObject.y += 10;

            break;

        case 'a': // Move left

            selectedObject.x -= 10;

            break;

        case 's': // Move down

            selectedObject.y -= 10;

            break;

        case 'd': // Move right

            selectedObject.x += 10;

            break;

        case ' ': // Deselect object

            selectedObject.selected = false;

            selectedObjectIndex = -1;

            break;

        }

        // Reassign objects to grid after movement

        assignObjectsToGrid();

        glutPostRedisplay(); // Request redraw

    }

}

// Function to handle mouse click for object selection

*void* mouse(*int* *button*, *int* *state*, *int* *x*, *int* *y*) {

    if (button == GLUT\_LEFT\_BUTTON && state == GLUT\_DOWN) {

        // Convert screen coordinates to grid coordinates

*int* gridX = x / CELL\_SIZE;

*int* gridY = (WINDOW\_HEIGHT - y) / CELL\_SIZE;

        // Check if there is an object in the clicked cell

        if (gridX >= 0 && gridX < GRID\_SIZE && gridY >= 0 && gridY < GRID\_SIZE) {

            for (*auto* obj : grid[gridX][gridY]) {

                if (x >= obj->x && x <= obj->x + obj->width && (WINDOW\_HEIGHT - y) >= obj->y && (WINDOW\_HEIGHT - y) <= obj->y + obj->height) {

                    // Deselect previously selected object

                    if (selectedObjectIndex != -1) {

                        objects[selectedObjectIndex].selected = false;

                    }

                    // Select new object

                    obj->selected = true;

                    selectedObjectIndex = std::distance(objects.data(), obj);

                    break;

                }

            }

        }

        glutPostRedisplay(); // Request redraw

    }

}

// Display callback function

*void* display() {

    glClear(GL\_COLOR\_BUFFER\_BIT);

    drawGrid();

    drawObjects();

    checkCollisions();

    glutSwapBuffers();

}

// Initialize objects and grid

*void* init() {

    // Add some objects to the scene

    objects.push\_back({ 100, 100, 30, 30, false });

    objects.push\_back({ 120, 120, 30, 30, false });

    objects.push\_back({ 300, 300, 30, 30, false });

    objects.push\_back({ 320, 320, 30, 30, false });

    // Assign objects to grid cells

    assignObjectsToGrid();

}

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

    glutInit(&argc, argv);

    glutInitDisplayMode(GLUT\_DOUBLE | GLUT\_RGB);

    glutInitWindowSize(WINDOW\_WIDTH, WINDOW\_HEIGHT);

    glutCreateWindow("Task2 - Uniform Grid");

    glMatrixMode(GL\_PROJECTION);

    glLoadIdentity();

    gluOrtho2D(0, WINDOW\_WIDTH, 0, WINDOW\_HEIGHT);

    init();

    glutDisplayFunc(display);

    glutKeyboardFunc(keyboard); // Register keyboard callback

    glutMouseFunc(mouse); // Register mouse callback

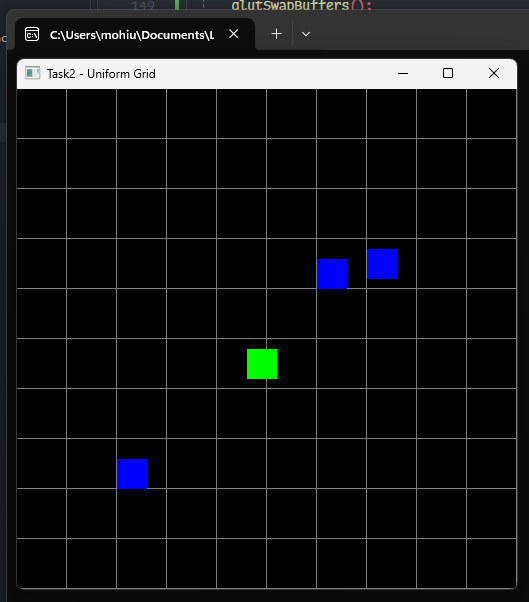
    glutMainLoop();

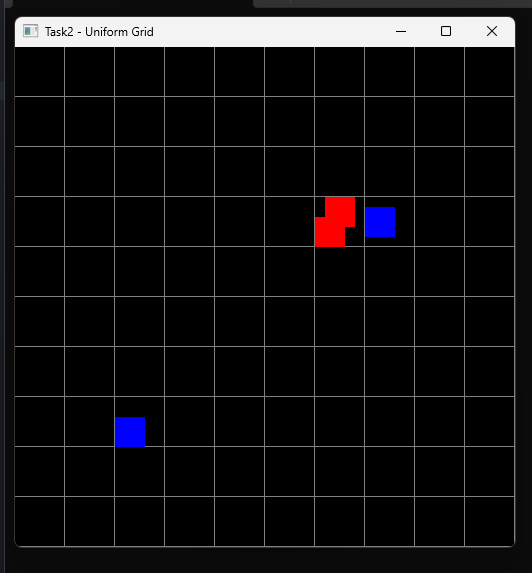
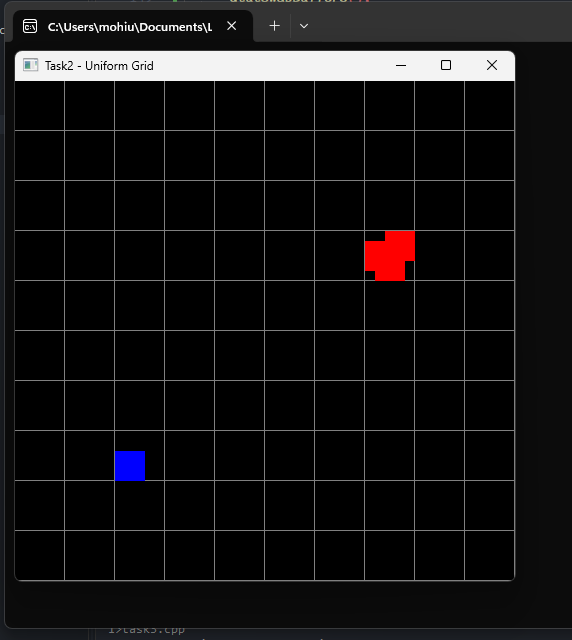
    return 0;

}

**Output:**

**Click to Select  
WASD to MOVE**

****

**** ****

**Task 3: Combining Techniques**

* **Objective:** Combine bounding volumes and uniform grids for optimized collision detection.
* **Steps:**
  + Create a dynamic scene with multiple moving objects.
  + Use uniform grids to narrow down collision checks, and bounding volumes for precise detection.
  + Display grid cell updates and bounding volume adjustments in real-time.

**Solution:**

**Code:**

#include <GL/glut.h>

#include <vector>

#include <iostream>

#include <cmath>

const *int* GRID\_SIZE = 10; // Size of the grid (10x10)

const *int* CELL\_SIZE = 50; // Size of each cell in pixels

const *int* WINDOW\_WIDTH = GRID\_SIZE \* CELL\_SIZE;

const *int* WINDOW\_HEIGHT = GRID\_SIZE \* CELL\_SIZE;

*struct* Object {

*int* x, y; // Position in the grid

*int* width, height; // Size of the object

*float* radius; // Radius for spherical bounding box

*float* vx, vy; // Velocity

*bool* selected; // Whether the object is selected

};

std::vector<Object> objects; // List of objects in the scene

std::vector<std::vector<std::vector<Object\*>>> grid(GRID\_SIZE, std::vector<std::vector<Object\*>>(GRID\_SIZE)); // Uniform grid

*int* selectedObjectIndex = -1; // Index of the currently selected object

// Function to draw a rectangle

*void* drawRect(*int* *x*, *int* *y*, *int* *width*, *int* *height*) {

    glBegin(GL\_QUADS);

    glVertex2i(x, y);

    glVertex2i(x + width, y);

    glVertex2i(x + width, y + height);

    glVertex2i(x, y + height);

    glEnd();

}

// Function to draw a circle

*void* drawCircle(*int* *x*, *int* *y*, *float* *radius*) {

    glBegin(GL\_TRIANGLE\_FAN);

    glVertex2f(x, y); // Center of the circle

    for (*int* i = 0; i <= 360; i++) {

*float* angle = i \* 3.14159 / 180;

        glVertex2f(x + cos(angle) \* radius, y + sin(angle) \* radius);

    }

    glEnd();

}

// Function to draw the grid

*void* drawGrid() {

    glColor3f(0.5, 0.5, 0.5); // Gray color for the grid

    for (*int* i = 0; i <= GRID\_SIZE; ++i) {

        glBegin(GL\_LINES);

        glVertex2i(i \* CELL\_SIZE, 0);

        glVertex2i(i \* CELL\_SIZE, WINDOW\_HEIGHT);

        glVertex2i(0, i \* CELL\_SIZE);

        glVertex2i(WINDOW\_WIDTH, i \* CELL\_SIZE);

        glEnd();

    }

}

// Function to draw objects

*void* drawObjects() {

    for (const *auto*& obj : objects) {

        if (obj.selected) {

            glColor3f(0.0, 1.0, 0.0); // Green color for selected object

        }

        else {

            glColor3f(0.0, 0.0, 1.0); // Blue color for other objects

        }

        drawRect(obj.x, obj.y, obj.width, obj.height);

        // Draw spherical bounding box

        glColor3f(1.0, 0.0, 0.0); // Red color for spherical bounding box

        drawCircle(obj.x + obj.width / 2, obj.y + obj.height / 2, obj.radius);

    }

}

// Function to check for AABB collisions

*bool* checkAABBCollision(const Object& *obj1*, const Object& *obj2*) {

    return (obj1.x < obj2.x + obj2.width &&

        obj1.x + obj1.width > obj2.x &&

        obj1.y < obj2.y + obj2.height &&

        obj1.y + obj1.height > obj2.y);

}

// Function to check for spherical collisions

*bool* checkSphericalCollision(const Object& *obj1*, const Object& *obj2*) {

*float* dx = (obj1.x + obj1.width / 2) - (obj2.x + obj2.width / 2);

*float* dy = (obj1.y + obj1.height / 2) - (obj2.y + obj2.height / 2);

*float* distance = sqrt(dx \* dx + dy \* dy);

    return distance < (obj1.radius + obj2.radius);

}

// Function to handle collisions and reflections

*void* handleCollisions() {

    for (*size\_t* i = 0; i < objects.size(); ++i) {

        for (*size\_t* j = i + 1; j < objects.size(); ++j) {

            if (checkAABBCollision(objects[i], objects[j])) {

                // Reflect velocities for AABB collision

                std::swap(objects[i].vx, objects[j].vx);

                std::swap(objects[i].vy, objects[j].vy);

            }

            if (checkSphericalCollision(objects[i], objects[j])) {

                // Reflect velocities for spherical collision

                std::swap(objects[i].vx, objects[j].vx);

                std::swap(objects[i].vy, objects[j].vy);

            }

        }

        // Reflect off walls

        if (objects[i].x < 0 || objects[i].x + objects[i].width > WINDOW\_WIDTH) {

            objects[i].vx = -objects[i].vx;

        }

        if (objects[i].y < 0 || objects[i].y + objects[i].height > WINDOW\_HEIGHT) {

            objects[i].vy = -objects[i].vy;

        }

    }

}

// Function to assign objects to grid cells

*void* assignObjectsToGrid() {

    // Clear the grid

    for (*int* i = 0; i < GRID\_SIZE; ++i) {

        for (*int* j = 0; j < GRID\_SIZE; ++j) {

            grid[i][j].clear();

        }

    }

    // Assign objects to grid cells

    for (*auto*& obj : objects) {

*int* gridX = obj.x / CELL\_SIZE;

*int* gridY = obj.y / CELL\_SIZE;

        if (gridX >= 0 && gridX < GRID\_SIZE && gridY >= 0 && gridY < GRID\_SIZE) {

            grid[gridX][gridY].push\_back(&obj);

        }

    }

}

// Function to update object positions

void update(int value) {

    for (auto& obj : objects) {

        obj.x += obj.vx;

        obj.y += obj.vy;

    }

    handleCollisions();

    assignObjectsToGrid();

    glutPostRedisplay();

    glutTimerFunc(16, update, 0); // 60 FPS

}

// Function to handle keyboard input

void keyboard(unsigned char key, int x, int y) {

    if (selectedObjectIndex != -1) {

        Object& selectedObject = objects[selectedObjectIndex];

        switch (key) {

        case 'w': // Move up

            selectedObject.y += 10;

            break;

        case 'a': // Move left

            selectedObject.x -= 10;

            break;

        case 's': // Move down

            selectedObject.y -= 10;

            break;

        case 'd': // Move right

            selectedObject.x += 10;

            break;

        case ' ': // Deselect object

            selectedObject.selected = false;

            selectedObjectIndex = -1;

            break;

        }

        // Reassign objects to grid after movement

        assignObjectsToGrid();

        glutPostRedisplay(); // Request redraw

    }

}

// Function to handle mouse click for object selection

void mouse(int button, int state, int x, int y) {

    if (button == GLUT\_LEFT\_BUTTON && state == GLUT\_DOWN) {

        // Convert screen coordinates to grid coordinates

        int gridX = x / CELL\_SIZE;

        int gridY = (WINDOW\_HEIGHT - y) / CELL\_SIZE;

        // Check if there is an object in the clicked cell

        if (gridX >= 0 && gridX < GRID\_SIZE && gridY >= 0 && gridY < GRID\_SIZE) {

            for (auto obj : grid[gridX][gridY]) {

                if (x >= obj->x && x <= obj->x + obj->width && (WINDOW\_HEIGHT - y) >= obj->y && (WINDOW\_HEIGHT - y) <= obj->y + obj->height) {

                    // Deselect previously selected object

                    if (selectedObjectIndex != -1) {

                        objects[selectedObjectIndex].selected = false;

                    }

                    // Select new object

                    obj->selected = true;

                    selectedObjectIndex = std::distance(objects.data(), obj);

                    break;

                }

            }

        }

        glutPostRedisplay(); // Request redraw

    }

}

// Display callback function

void display() {

    glClear(GL\_COLOR\_BUFFER\_BIT);

    drawGrid();

    drawObjects();

    glutSwapBuffers();

}

// Initialize objects and grid

void init() {

    // Add some objects to the scene

    objects.push\_back({ 100, 200, 30, 30, 20, 2, 2, false });

    objects.push\_back({ 120, 120, 30, 30, 20, -2, 2, false });

    objects.push\_back({ 300, 100, 30, 30, 20, 2, -2, false });

    objects.push\_back({ 320, 320, 30, 30, 20, -2, -2, false });

    objects.push\_back({ 200, 200, 30, 30, 20, 1, 1, false });

    objects.push\_back({ 250, 250, 30, 30, 20, -1, 1, false });

    // Assign objects to grid cells

    assignObjectsToGrid();

}

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

    glutInit(&argc, argv);

    glutInitDisplayMode(GLUT\_DOUBLE | GLUT\_RGB);

    glutInitWindowSize(WINDOW\_WIDTH, WINDOW\_HEIGHT);

    glutCreateWindow("Task3 - Uniform Grid AABB and Spherical");

    glMatrixMode(GL\_PROJECTION);

    glLoadIdentity();

    gluOrtho2D(0, WINDOW\_WIDTH, 0, WINDOW\_HEIGHT);

    init();

    glutDisplayFunc(display);

    glutKeyboardFunc(keyboard); // Register keyboard callback

    glutMouseFunc(mouse); // Register mouse callback

    glutTimerFunc(0, update, 0); // Start the update loop

    glutMainLoop();

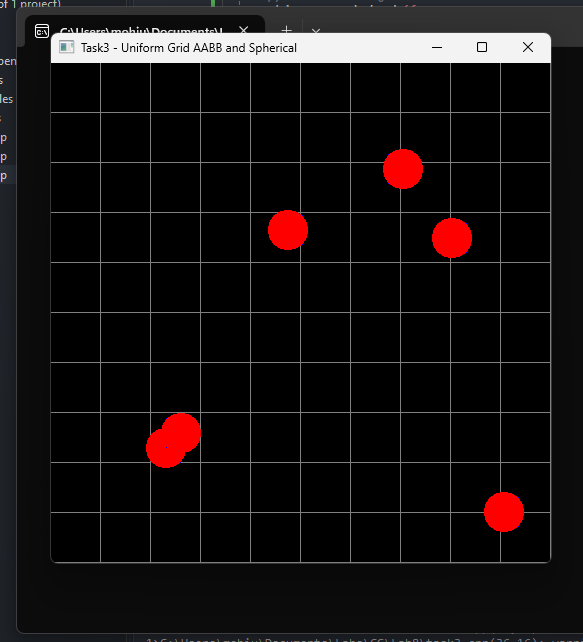
    return 0;

}

**Output:**

**A screenshot of a computer screen

AI-generated content may be incorrect.**

****

### **Deliverables:**

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

 Include screenshots of the program outputs.

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

# Lab Rubrics

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Lab Rubrics for (Lab-08: Spatial Data Structures) | | | | | |
|  | | | | | |
| **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**  **(CLO3, PLO5)** | 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**  **(CLO4-PLO3)** | 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. |