Comprehensive Test Plan for Polyhedron Project

Your Name

October 19, 2024

Contents

1 Introduction

This document outlines a comprehensive test plan for the polyhedron project, which includes functionality for handling regular polyhedrons, objects with holes, and irregular polyhedrons. The plan covers both unit testing and end-to-end testing strategies, aiming to achieve high test coverage and ensure robust error handling.

2 Test Strategy

Our testing strategy will employ a combination of the following test types:

1. Unit Testing: To verify the correctness of individual functions and methods.
2. Integration Testing: To ensure different components of the system work correctly together.
3. End-to-End Testing: To validate the entire system's functionality from input to output.
4. Edge Case Testing: To verify the system's behavior under extreme or unusual conditions.
5. Performance Testing: To assess the system's performance under various loads.

3 Test Coverage

We aim to achieve high test coverage for the following key components of our system:

1. Input Validation
2. 3D Reconstruction
3. Geometric Calculations
4. Transformations
5. Projections
6. Visualization
7. Error Handling

4 Unit Tests

4.1 Input Validation Tests

```
void test_input_validation() {
    // Test valid input
    Polyhedron valid_poly = create_valid_test_polyhedron();
    assert(validateInput(valid_poly) == true);

    // Test invalid edge length
    Polyhedron invalid_edge_poly = create_invalid_edge_test_polyhedron();
    assert(validateInput(invalid_edge_poly) == false);

    // Test collinear points
    Polyhedron collinear_poly = create_collinear_test_polyhedron();
    assert(validateInput(collinear_poly) == false);

    // Test non-planar face
    Polyhedron non_planar_poly = create_non_planar_test_polyhedron();
    assert(validateInput(non_planar_poly) == false);

    // Test unclosed polyhedron
    Polyhedron unclosed_poly = create_unclosed_test_polyhedron();
    assert(validateInput(unclosed_poly) == false);
}
```

4.2 Geometric Calculation Tests

```
void test_geometric_calculations() {
    Polyhedron test_poly = create_test_cube();

    // Test surface area calculation
    float expected_surface_area = 6.0f;  // For a unit cube
    assert(abs(calculateSurfaceArea(test_poly) - expected_surface_area) < 1e-6);

    // Test volume calculation
    float expected_volume = 1.0f;  // For a unit cube
    assert(abs(calculateVolume(test_poly) - expected_volume) < 1e-6);

    // Test center of mass calculation
    Vertex expected_com = {0.5f, 0.5f, 0.5f};  // For a unit cube
    Vertex calculated_com = calculateCenterOfMass(test_poly);
    assert(abs(calculated_com.x - expected_com.x) < 1e-6);
    assert(abs(calculated_com.y - expected_com.y) < 1e-6);
    assert(abs(calculated_com.z - expected_com.z) < 1e-6);
}
```

4.3 Transformation Tests

```
void test_transformations() {
    Polyhedron test_poly = create_test_cube();

    // Test rotation
    Vertex axis = {1.0f, 0.0f, 0.0f};
    rotate_point(&test_poly.vertices[0], 90.0, axis);
    assert(abs(test_poly.vertices[0].y) < 1e-6);
    assert(abs(test_poly.vertices[0].z - 1.0f) < 1e-6);

    // Test translation
    translate_point(&test_poly.vertices[0], 1.0, 1.0, 1.0);
    assert(abs(test_poly.vertices[0].x - 1.0f) < 1e-6);
    assert(abs(test_poly.vertices[0].y - 1.0f) < 1e-6);
    assert(abs(test_poly.vertices[0].z - 2.0f) < 1e-6);

    // Test scaling
    scale_point(&test_poly.vertices[0], 2.0, 2.0, 2.0);
    assert(abs(test_poly.vertices[0].x - 2.0f) < 1e-6);
    assert(abs(test_poly.vertices[0].y - 2.0f) < 1e-6);
    assert(abs(test_poly.vertices[0].z - 4.0f) < 1e-6);
}
```

5 Integration Tests

5.1 3D Reconstruction Test

```
void test_3d_reconstruction() {
    vector<double> xy = {0,0, 1,0, 1,1, 0,1};
    vector<double> xz = {0,0, 1,0, 1,1, 0,1};
    Polyhedron reconstructed_poly = reconstruct_3d(xy, xz);

    assert(reconstructed_poly.vertices.size() == 4);
    assert(reconstructed_poly.faces.size() == 1);
    assert(reconstructed_poly.faces[0].edges.size() == 4);

    // Verify the reconstructed vertices
    assert(abs(reconstructed_poly.vertices[0].x) < 1e-6);
    assert(abs(reconstructed_poly.vertices[0].y) < 1e-6);
    assert(abs(reconstructed_poly.vertices[0].z) < 1e-6);

    assert(abs(reconstructed_poly.vertices[1].x - 1.0f) < 1e-6);
    assert(abs(reconstructed_poly.vertices[1].y) < 1e-6);
    assert(abs(reconstructed_poly.vertices[1].z) < 1e-6);

    assert(abs(reconstructed_poly.vertices[2].x - 1.0f) < 1e-6);
    assert(abs(reconstructed_poly.vertices[2].y - 1.0f) < 1e-6);
    assert(abs(reconstructed_poly.vertices[2].z - 1.0f) < 1e-6);

    assert(abs(reconstructed_poly.vertices[3].x) < 1e-6);
    assert(abs(reconstructed_poly.vertices[3].y - 1.0f) < 1e-6);
    assert(abs(reconstructed_poly.vertices[3].z - 1.0f) < 1e-6);
}
```

6 End-to-End Tests

6.1 Full Process Test

```
void test_full_process() {
    // Create test input
    vector<double> xy = {0,0, 1,0, 1,1, 0,1};
    vector<double> xz = {0,0, 1,0, 1,1, 0,1};

    // Reconstruct 3D polyhedron
    Polyhedron poly = reconstruct_3d(xy, xz);

    // Validate input
    assert(validateInput(poly) == true);

    // Perform geometric calculations
    float surface_area = calculateSurfaceArea(poly);
    float volume = calculateVolume(poly);
    Vertex com = calculateCenterOfMass(poly);

    // Perform a transformation
    rotate_point(&poly.vertices[0], 45.0, {1.0f, 0.0f, 0.0f});

    // Project the transformed polyhedron
    orthographicProjection(poly, 'z');

    // Verify results (you would need to define expected values)
    assert(abs(surface_area - expected_surface_area) < 1e-6);
    assert(abs(volume - expected_volume) < 1e-6);
    assert(abs(com.x - expected_com.x) < 1e-6);
    assert(abs(com.y - expected_com.y) < 1e-6);
    assert(abs(com.z - expected_com.z) < 1e-6);

    // Verify projection (this would depend on your implementation
    of orthographicProjection)
    // You might check if certain vertices are where you expect
    them to be after projection
}
```

7 Testing Objects with Holes

To test objects with holes, we need to create specific test cases that include polyhedrons with internal cavities. Here's an example of how we might approach this:

```
Polyhedron create_cube_with_hole() {
    Polyhedron poly;
    // Create outer cube
    // ... (code to create outer cube vertices and faces)

    // Create inner cube (hole)
    // ... (code to create inner cube vertices and faces)

    // Mark inner faces as internal
```

```
    for (int i = 6; i < 12; i++) {  // Assuming inner cube faces start at index 6
        poly.faces[i].is_internal = true;
    }

    return poly;
}

void test_object_with_hole() {
    Polyhedron hollow_cube = create_cube_with_hole();

    // Test volume calculation
    float expected_volume = 0.875f;  // Assuming outer cube is 1x1x1 and inner cube is 0.5x0.5x0.5
    assert(abs(calculateVolume(hollow_cube) - expected_volume) < 1e-6);

    // Test surface area calculation
    float expected_surface_area = 7.5f;  // Outer surface area + inner surface area
    assert(abs(calculateSurfaceArea(hollow_cube) - expected_surface_area) < 1e-6);

    // Test center of mass calculation
    Vertex expected_com = {0.5f, 0.5f, 0.5f};  // Should be the same as a solid cube
    Vertex calculated_com = calculateCenterOfMass(hollow_cube);
    assert(abs(calculated_com.x - expected_com.x) < 1e-6);
    assert(abs(calculated_com.y - expected_com.y) < 1e-6);
    assert(abs(calculated_com.z - expected_com.z) < 1e-6);
}
```

8 Testing Irregular Polyhedrons

For testing irregular polyhedrons, we need to create test cases with non-uniform shapes. Here's an example:

```
Polyhedron create_irregular_polyhedron() {
    Polyhedron poly;
    // Create an irregular shape, e.g., a tetrahedron with unequal faces
    // ... (code to create vertices and faces of an irregular tetrahedron)
    return poly;
}

void test_irregular_polyhedron() {
    Polyhedron irregular_poly = create_irregular_polyhedron();

    // Test volume calculation
    float expected_volume = calculate_expected_volume(irregular_poly);
    assert(abs(calculateVolume(irregular_poly) - expected_volume) < 1e-6);

```

```
    // Test surface area calculation
    float expected_surface_area = calculate_expected_surface_area(
    irregular_poly);
    assert(abs(calculateSurfaceArea(irregular_poly) -
    expected_surface_area) < 1e-6);

    // Test center of mass calculation
    Vertex expected_com = calculate_expected_com(irregular_poly);
    Vertex calculated_com = calculateCenterOfMass(irregular_poly);
    assert(abs(calculated_com.x - expected_com.x) < 1e-6);
    assert(abs(calculated_com.y - expected_com.y) < 1e-6);
    assert(abs(calculated_com.z - expected_com.z) < 1e-6);

    // Test transformations
    transform_polyhedron(irregular_poly);
    // Verify the transformation results
    // ... (code to verify the transformation)
}
```

9 Performance Testing

To ensure the system performs well with large or complex polyhedrons:

```
void test_performance() {
    Polyhedron large_poly = create_large_complex_polyhedron
    (1000000);  // 1 million vertices

    auto start = std::chrono::high_resolution_clock::now();

    calculateVolume(large_poly);
    calculateSurfaceArea(large_poly);
    calculateCenterOfMass(large_poly);
    transform_polyhedron(large_poly);
    orthographicProjection(large_poly, 'z');

    auto end = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::
    milliseconds>(end - start);

    assert(duration.count() < 5000);  // Ensure all operations
    complete within 5 seconds
}
```

10 Conclusion

This test plan provides a comprehensive strategy for validating the functionality, correctness, and performance of the polyhedron project. By implementing these tests, we can ensure that the system correctly handles regular polyhedrons, objects with holes, and irregular polyhedrons, while maintaining high performance even with complex shapes.

Regular execution of these tests throughout the development process will help maintain the integrity and reliability of the system as new features are added or existing ones are modified.