3D Polyhedron Reconstruction and Analysis System - Testing

Himangi Parekh

October 28, 2024

Contents

1 Introduction

2 Introduction

This document outlines the comprehensive test plan for the Polyhedron Project. The project involves a C++ implementation of various geometrical operations on polyhedra, including 3D reconstruction, volume calculation, surface area calculation, and various transformations.

3 Test Strategy: Unit Testing and End-to-End Testing

4 Test Strategy

Our testing strategy will encompass several levels of testing to ensure the robustness and correctness of our implementation:

4.1 Unit Testing

We will implement unit tests for individual functions and methods within our codebase. These tests will verify that each component works correctly in isolation. For this, every individual function and helper functions too were tested with sample inputs.

4.2 Integration Testing

Integration tests will be used to ensure that different components of our system work correctly when combined. This testing was done along with end-to-end testing. Essentially, we checked whether the correct output was coming when we combined different functions.

4.3 End-to-End Testing

End-to-end tests will simulate real-world usage scenarios to verify that the entire system functions as expected from input to output. This was the key part of the testing since the end-to-end flow from the user's perspective was tested. From the make file to output of every function which user could exercise.

4.4 Error Handling and Edge Case Testing

We will design tests to verify that our system handles errors gracefully and performs correctly for edge cases and boundary conditions. This was quite important too. After a few tests mentioned above we realised that the program often used to crash if user inputs a wrong value eg a letter instead of a number. We made changes and prevented crashes and added this element to our testing.

5 Test Coverage

In this section, we essentially checked on what kind of inputs did each of our functions work. For each functionality - Validation, Gemoetric Calculations, 3D reconstruction, Moment of Inertia, Transformations, Orthographic Projections, 3D Isometric Interactive output - we checked using multiple inputs to cover:
1. Without Holes
1. Irregular
2. Regular
2. With Holes
1. Irregular
2. Regular

6 Sample Test Inputs

7 Unit Tests

7.1 Vertex Operations

```
void test_vertex_operations() {
    Vertex v1 = {1.0, 2.0, 3.0};
    Vertex v2 = {4.0, 5.0, 6.0};

    // Test vector subtraction
    Vertex result = vectorSubtract(v2, v1);
    assert(result.x == 3.0 && result.y == 3.0 && result.z == 3.0);

    // Test vector dot product
    float dot = vectorDot(v1, v2);
    assert(dot == 32.0);

    // Test vector cross product
    Vertex cross = vectorCross(v1, v2);
    assert(cross.x == -3.0 && cross.y == 6.0 && cross.z == -3.0);

    // Test vector magnitude
    float mag = vectorMagnitude(v1);
    assert(std::abs(mag - 3.7416573867739413) < 1e-6);
}
```

7.2 Tetrahedron Calculations

```
void test_tetrahedron_calculations() {
    Vertex v0 = {0.0, 0.0, 0.0};
    Vertex v1 = {1.0, 0.0, 0.0};
    Vertex v2 = {0.0, 1.0, 0.0};
    Vertex v3 = {0.0, 0.0, 1.0};

    // Test tetrahedron volume
    float volume = calculateTetrahedronVolume(v0, v1, v2, v3);
    assert(std::abs(volume - 1.0/6.0) < 1e-6);

    // Test tetrahedron centroid
    Vertex centroid = calculateTetrahedronCentroid(v0, v1, v2, v3);
    assert(std::abs(centroid.x - 0.25) < 1e-6 &&
           std::abs(centroid.y - 0.25) < 1e-6 &&
           std::abs(centroid.z - 0.25) < 1e-6);
}
```

7.3 Polyhedron Operations

```
void test_polyhedron_operations() {
    Polyhedron cube = create_cube();

    // Test surface area calculation
```

```cpp
    float surface_area = calculateSurfaceArea(cube);
    assert(std::abs(surface_area - 24.0) < 1e-6);

    // Test volume calculation
    float volume = calculateVolume(cube);
    assert(std::abs(volume - 8.0) < 1e-6);

    // Test center of mass calculation
    Vertex com = calculateCenterOfMass(cube);
    assert(std::abs(com.x - 0.5) < 1e-6 &&
           std::abs(com.y - 0.5) < 1e-6 &&
           std::abs(com.z - 0.5) < 1e-6);
}
```

7.4 Transformation Operations

```cpp
void test_transformation_operations() {
    Vertex v = {1.0, 2.0, 3.0};

    // Test rotation
    Vertex axis = {0.0, 0.0, 1.0};
    rotate_point(&v, 90.0, axis);
    assert(std::abs(v.x + 2.0) < 1e-6 &&
           std::abs(v.y - 1.0) < 1e-6 &&
           std::abs(v.z - 3.0) < 1e-6);

    // Test translation
    translate_point(&v, 1.0, 1.0, 1.0);
    assert(std::abs(v.x + 1.0) < 1e-6 &&
           std::abs(v.y - 2.0) < 1e-6 &&
           std::abs(v.z - 4.0) < 1e-6);

    // Test scaling
    scale_point(&v, 2.0, 2.0, 2.0);
    assert(std::abs(v.x + 2.0) < 1e-6 &&
           std::abs(v.y - 4.0) < 1e-6 &&
           std::abs(v.z - 8.0) < 1e-6);

    // Test reflection
    reflect_point(&v, 'x');
    assert(std::abs(v.x - 2.0) < 1e-6 &&
           std::abs(v.y - 4.0) < 1e-6 &&
           std::abs(v.z - 8.0) < 1e-6);
}
```

8 Integration Tests

8.1 Polyhedron Reconstruction and Validation

```cpp
void test_polyhedron_reconstruction_and_validation() {
    vector<double> xy = {0,0, 1,0, 1,1, 0,1, 0,0, 1,0, 1,1, 0,1};
    vector<double> xz = {0,0, 1,0, 1,1, 0,1, 0,1, 1,1, 1,2, 0,2};
    Polyhedron poly;

    // Test reconstruction
    reconstructPolyhedron(xy, xz, poly);

    // Test validation
    bool is_valid = validateInput(poly);
    assert(is_valid);

    // Test geometric properties
    float volume = calculateVolume(poly);
    float surface_area = calculateSurfaceArea(poly);
    Vertex com = calculateCenterOfMass(poly);

    assert(std::abs(volume - 1.0) < 1e-6);
    assert(std::abs(surface_area - 10.0) < 1e-6);
    assert(std::abs(com.x - 0.5) < 1e-6 &&
           std::abs(com.y - 0.5) < 1e-6 &&
           std::abs(com.z - 0.5) < 1e-6);
}
```

8.2 Transformation and Projection

```
void test_transformation_and_projection() {
    Polyhedron cube = create_cube();

    // Apply a series of transformations
    Vertex rotation_axis = {1.0, 1.0, 1.0};
    rotate_polyhedron(cube, 45.0, rotation_axis);
    translate_polyhedron(cube, 1.0, 2.0, 3.0);
    scale_polyhedron(cube, 2.0, 2.0, 2.0);

    // Test orthographic projection
    vector<Vertex> projection = orthographicProjection(cube, 'z');

    // Verify projection properties
    assert(projection.size() == 8);  // Cube has 8 vertices
    for (const auto& v : projection) {
        assert(v.z == 0.0);  // Z-projection should have z=0 for
    all vertices
    }
}
```

9 End-to-End Tests

9.1 Full Workflow Test

```
void test_full_workflow() {
    // 1. Input polyhedron data
    vector<double> xy = {0,0, 1,0, 1,1, 0,1, 0,0, 1,0, 1,1, 0,1};
    vector<double> xz = {0,0, 1,0, 1,1, 0,1, 0,1, 1,1, 1,2, 0,2};
    Polyhedron poly;

    // 2. Reconstruct 3D polyhedron
    reconstructPolyhedron(xy, xz, poly);

    // 3. Validate input
    bool is_valid = validateInput(poly);
    assert(is_valid);

    // 4. Calculate geometric properties
    float volume = calculateVolume(poly);
    float surface_area = calculateSurfaceArea(poly);
    Vertex com = calculateCenterOfMass(poly);

    // 5. Apply transformations
    rotate_polyhedron(poly, 30.0, {0,1,0});
    translate_polyhedron(poly, 1,1,1);
    scale_polyhedron(poly, 2,2,2);

    // 6. Project onto a plane
    vector<Vertex> projection = orthographicProjection(poly, 'z');

    // 7. Export results
    bool export_success = exportOutput(poly, "output.txt");
    assert(export_success);

    // 8. Verify final state
    Polyhedron imported_poly;
    bool import_success = importPolyhedron("output.txt",
    imported_poly);
    assert(import_success);
    assert(poly == imported_poly);  // Assuming we've implemented
    equality comparison
}
```

10 Error Handling and Edge Case Tests

10.1 Input Validation Tests

```
void test_input_validation() {
    // Test with invalid number of vertices
    vector<double> xy = {0,0, 1,0, 1,1};
    vector<double> xz = {0,0, 1,0, 1,1};
    Polyhedron poly;
    bool reconstruction_failed = false;
    try {
        reconstructPolyhedron(xy, xz, poly);
    } catch (const std::invalid_argument& e) {
        reconstruction_failed = true;
    }
    assert(reconstruction_failed);

    // Test with non-planar face
    xy = {0,0, 1,0, 1,1, 0,1, 0,0, 1,0, 1,1, 0,1};
    xz = {0,0, 1,0, 1,1, 0,1, 0,1, 1,1, 1,2, 0,3};  // Last vertex
    is off-plane
    reconstructPolyhedron(xy, xz, poly);
    bool is_valid = validateInput(poly);
```

```cpp
    assert(!is_valid);

    // Test with self-intersecting polyhedron
    // (This would require a more complex example)
}
```

10.2 Numerical Stability Tests

```cpp
void test_numerical_stability() {
    // Test with very small polyhedron
    Polyhedron tiny_cube = create_cube(1e-10);
    float volume = calculateVolume(tiny_cube);
    assert(volume > 0 && volume < 1e-29);

    // Test with very large polyhedron
    Polyhedron huge_cube = create_cube(1e10);
    volume = calculateVolume(huge_cube);
    assert(volume > 1e30 && volume < 1e31);

    // Test rotations with very small angles
    rotate_polyhedron(tiny_cube, 1e-10, {0,0,1});
    bool still_valid = validateInput(tiny_cube);
    assert(still_valid);
}
```

11 Performance Tests

While not strictly part of correctness testing, it's often useful to include some performance benchmarks in your test suite.

```cpp
void test_performance() {
    const int NUM_VERTICES = 1000000;
    vector<Vertex> vertices(NUM_VERTICES);

    // Generate random vertices
    for (int i = 0; i < NUM_VERTICES; ++i) {
        vertices[i] = {rand() / (float)RAND_MAX,
                       rand() / (float)RAND_MAX,
                       rand() / (float)RAND_MAX};
    }

    // Measure time for center of mass calculation
    auto start = std::chrono::high_resolution_clock::now();
    Vertex com = calculateCenterOfMass(vertices);
    auto end = std::chrono::high_resolution_clock::now();

    std::chrono::duration<double> diff = end - start;
    std::cout << "Time to calculate center of mass for " <<
    NUM_VERTICES
              << " vertices: " << diff.count() << " s\n";

    // Similar performance tests for other computationally
    intensive operations
    // ...
}
```

12 Test Execution and Reporting

12.1 Test Runner

We will implement a simple test runner to execute all our tests and report the results. Here's a basic structure:

```cpp
void run_all_tests() {
    std::vector<std::pair<std::string, std::function<void()>>>
    tests = {
        {"Vertex Operations", test_vertex_operations},
        {"Tetrahedron Calculations", test_tetrahedron_calculations
    },
        {"Polyhedron Operations", test_polyhedron_operations},
        {"Transformation Operations",
    test_transformation_operations},
        {"Polyhedron Reconstruction and Validation",
    test_polyhedron_reconstruction_and_validation},
        {"Transformation and Projection",
    test_transformation_and_projection},
        {"Full Workflow", test_full_workflow},
        {"Input Validation", test_input_validation},
        {"Numerical Stability", test_numerical_stability},
        {"Performance", test_performance}
    };

    int passed = 0;
    int failed = 0;

    for (const auto& test : tests) {
        try {
            std::cout << "Running test: " << test.first << "... ";
            test.second();
            std::cout << "PASSED\n";
            ++passed;
        } catch (const std::exception& e) {
            std::cout << "FAILED\n";
            std::cout << "Error: " << e.what() << "\n";
            ++failed;
        }
    }

    std::cout << "\nTest Results:\n";
    std::cout << "Passed: " << passed << "\n";
    std::cout << "Failed: " << failed << "\n";
}
```

12.2 Continuous Integration

To ensure ongoing code quality, we will set up a continuous integration (CI) pipeline that runs our test suite automatically on each commit. This can be implemented using services like GitHub Actions, GitLab CI, or Jenkins.

Here's an example GitHub Actions workflow:

```
name: C++ CI

on: [push, pull_request]

jobs:
  build-and-test:
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v2
    - name: Install dependencies
      run: |
        sudo apt-get update
        sudo apt-get install -y cmake libsdl2-dev libeigen3-dev
    - name: Configure CMake
      run: cmake -B ${{github.workspace}}/build -DCMAKE_BUILD_TYPE=
    Release
    - name: Build
      run: cmake --build ${{github.workspace}}/build --config
    Release
    - name: Run tests
      run: ${{github.workspace}}/build/run_tests
```

13 Code Coverage

To ensure thorough testing, we will use a code coverage tool like gcov or LCOV to measure the extent of our test coverage. Our goal is to achieve at least 90

Here's how we can integrate code coverage into our CMake build system:

```
# In CMakeLists.txt
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -fprofile-arcs -ftest-
    coverage")
set(CMAKE_EXE_LINKER_FLAGS "${CMAKE_EXE_LINKER_FLAGS} -fprofile-
    arcs -ftest-coverage")

# After building and running tests
add_custom_target(coverage
    COMMAND lcov --capture --directory . --output-file coverage.
    info
    COMMAND lcov --remove coverage.info '/usr/*' --output-file
    coverage.info
    COMMAND lcov --list coverage.info
    WORKING_DIRECTORY ${CMAKE_BINARY_DIR}
    COMMENT "Generating code coverage report"
)
```

14 Unit Tests

14.1 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);
}
```

14.2 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);
}
```

14.3 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);
}
```

15 Integration Tests

15.1 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);
}
```

16 End-to-End Tests

16.1 Full Process Test

```cpp
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
}
```

17 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:

```cpp
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);
}
```

18 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)
}
```

19 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
}
```