Test Plan for Polyhedron Project

[Your Name]

October 19, 2024

Contents

1 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.

2 Test Strategy

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

2.1 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.

2.2 Integration Testing

Integration tests will be used to ensure that different components of our system work correctly when combined.

2.3 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.

2.4 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.

3 Test Coverage

We aim to achieve high test coverage across our codebase. The following key components will be thoroughly tested:

- Vertex and Edge data structures
- Face and Polyhedron classes
- Geometric calculations (volume, surface area, center of mass)
- Transformation operations (rotation, translation, scaling, reflection)
- Projection functions

- Input validation and error handling
- File I/O operations

4 Unit Tests

4.1 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);
}
```

4.2 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);
}
```

4.3 Polyhedron Operations

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

    // Test surface area calculation
    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);
}
```

4.4 Transformation Operations

```
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);
}
```

5 Integration Tests

5.1 Polyhedron Reconstruction and Validation

```
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);
}
```

5.2 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
    }
}
```

6 End-to-End Tests

6.1 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
}
```

7 Error Handling and Edge Case Tests

7.1 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);
    assert(!is_valid);

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

7.2 Numerical Stability Tests

```
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);
}
```

8 Performance Tests

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

```
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
    // ...
}
```

9 Test Execution and Reporting

9.1 Test Runner

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

```
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";
}
```

9.2 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
```

10 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"
)
```

11 Conclusion

This comprehensive test plan outlines our strategy for ensuring the correctness, robustness, and performance of our Polyhedron project. By implementing a combination of unit tests, integration tests, end-to-end tests, and performance benchmarks, we aim to catch potential issues early and maintain high code quality throughout the development process.

Regular execution of these tests, coupled with continuous integration and code coverage analysis, will help us identify and address problems quickly, leading to a more reliable and efficient implementation of our geometric algorithms and data structures.

As the project evolves, this test plan should be regularly reviewed and updated to reflect new features, edge cases, or potential areas of concern that may arise during development.