共享内存通信框架设计文档

1. 设计目标

本框架旨在实现高效的进程间数据传输，支持：

- C++程序之间的数据传输（生产者-消费者模式）
- 支持图像、点云和高度图数据的实时共享
- 高性能的零拷贝数据传输
- 可靠的数据一致性保证

2. 架构设计

2.1 核心组件

1. **共享内存管理器**
   - `ShareMemoryManager`类（支持生产者和消费者模式）
   - 同一个类可以作为生产者或消费者使用
   - 支持异步回调方式接收数据
2. **内存布局**

```
+------------------+
|  Header (固定大小) |
+------------------+
|  Data (可变大小)   |
+------------------+
```

3. **同步机制**
   - 互斥锁：确保独占访问
   - 状态标志：Empty、Writing、Ready、Error

2.2 数据结构

统一数据信息 (DataInfo)

```
struct DataInfo {
    uint32_t width;          // 宽度（图像/高度图）或点数量（点云）
    uint32_t height;         // 高度（图像/高度图）或点维度（点云）
    uint32_t channels;       // 通道数（图像）
    float xSpacing;          // X方向间距（高度图）
    float ySpacing;          // Y方向间距（高度图）
    uint32_t dataType;       // 数据类型（FrameType枚举）
    uint64_t timestamp;      // 时间戳
};
```

共享内存头部 (SharedMemoryHeader)

```
struct SharedMemoryHeader {
    uint32_t Magic;          // 魔数 (0x12345678)
    uint32_t Status;         // 状态
    uint32_t DataSize;       // 数据大小
    uint32_t Checksum;       // 校验和
    uint32_t FrameId;        // 帧ID
    DataInfo info;           // 统一的数据信息
    char ErrorMsg[128];      // 错误信息
};
```

2.3 数据类型支持

1. **图像数据 (FrameType::IMAGE)**

```
width    = 图像宽度 (像素)
height   = 图像高度 (像素)
channels = 图像通道数 (1=灰度图, 3=RGB, 4=RGBA)
数据大小  = width * height * channels
数据格式  = RGBRGBRGB... (交错存储, 每个像素的RGB值连续存储)
           示例: R1G1B1 R2G2B2 R3G3B3...
```

2. **点云数据 (FrameType::POINTCLOUD)**

```
width    = 点的数量
height   = 点维度 (3=XYZ)
数据大小  = width * height * sizeof(float)
数据格式  = XYZXYZXYZ... (交错存储, 每个点的XYZ坐标连续存储)
           示例: X1Y1Z1 X2Y2Z2 X3Y3Z3...
           数据类型: float (32位浮点数)
```

3. **高度图数据 (FrameType::HEIGHTMAP)**

```
width     = 宽度方向的点数
height    = 高度方向的点数
xSpacing  = X方向的采样间距 (米)
ySpacing  = Y方向的采样间距 (米)
数据大小   = width * height * sizeof(float)
数据格式   = ZZZ... (行优先存储, 每个点的高度值连续存储)
           示例: Z11Z12Z13... Z21Z22Z23... (Zij表示第i行第j列的高度值)
           数据类型: float (32位浮点数)
```

3. 工作流程

3.1 写入流程（生产者）

1. 获取互斥锁
2. 检查内存状态（必须为Empty）
3. 设置状态为Writing
4. 写入数据和元信息
5. 计算校验和
6. 设置状态为Ready
7. 释放互斥锁

3.2 读取流程（消费者）

1. 获取互斥锁
2. 检查内存状态（必须为Ready）
3. 验证魔数
4. 读取数据
5. 验证校验和
6. 设置状态为Empty
7. 释放互斥锁

4. 使用方法

4.1 生产者模式

```
// 创建生产者
SharedMemory::ShareMemoryManager producer("TestSharedMemory", 1024 * 1024 * 10);
if (!producer.Initialize()) {
    // 处理错误
}

// 1. 发送图像数据示例
{
    DataInfo info = {};
    info.width = 640;
    info.height = 480;
    info.channels = 3;  // RGB图像
    info.dataType = static_cast<uint32_t>(FrameType::IMAGE);
    info.timestamp = std::chrono::duration_cast<std::chrono::milliseconds>(
        std::chrono::system_clock::now().time_since_epoch()
    ).count();

    std::vector<uint8_t> imageData = GenerateTestImage(info.width, info.height,
info.channels);
    producer.WriteData(imageData.data(), imageData.size(), info);
}

// 2. 发送点云数据示例
{
```

```
    DataInfo info = {};
    info.width = 1000;    // 点的数量
    info.height = 3;      // XYZ三个维度
    info.dataType = static_cast<uint32_t>(FrameType::POINTCLOUD);
    info.timestamp = std::chrono::duration_cast<std::chrono::milliseconds>(
        std::chrono::system_clock::now().time_since_epoch()
    ).count();

    std::vector<uint8_t> pointCloudData = GenerateTestPointCloud(info.width);
    producer.WriteData(pointCloudData.data(), pointCloudData.size(), info);
}

// 3. 发送高度图数据示例
{
    DataInfo info = {};
    info.width = 200;
    info.height = 200;
    info.xSpacing = 0.1f;  // X方向采样间距（米）
    info.ySpacing = 0.1f;  // Y方向采样间距（米）
    info.dataType = static_cast<uint32_t>(FrameType::HEIGHTMAP);
    info.timestamp = std::chrono::duration_cast<std::chrono::milliseconds>(
        std::chrono::system_clock::now().time_since_epoch()
    ).count();

    std::vector<float> heightData = GenerateTestHeightMap(info.width,
info.height);
    std::vector<uint8_t> data(heightData.size() * sizeof(float));
    memcpy(data.data(), heightData.data(), data.size());
    producer.WriteData(data.data(), data.size(), info);
}
```

4.2 消费者模式

```
// 创建消费者
SharedMemory::ShareMemoryManager consumer("TestSharedMemory", 1024 * 1024 * 10);
if (!consumer.Initialize()) {
    // 处理错误
}

// 设置数据接收回调
consumer.SetDataReceivedCallback([](const uint8_t* data, size_t size,
    uint32_t dataType, uint32_t width, uint32_t height) {
    switch (static_cast<FrameType>(dataType)) {
        case FrameType::IMAGE: {
            // 处理图像数据
            const uint8_t* imageData = data;
            ProcessImage(imageData, width, height);
            break;
        }
        case FrameType::POINTCLOUD: {
            // 处理点云数据
```

```
                const float* pointCloud = reinterpret_cast<const float*>(data);
                ProcessPointCloud(pointCloud, width);
                break;
            }
            case FrameType::HEIGHTMAP: {
                // 处理高度图数据
                const float* heightMap = reinterpret_cast<const float*>(data);
                ProcessHeightMap(heightMap, width, height);
                break;
            }
        }
    });

    // 启动监听线程
    consumer.StartMonitoring();

    // ... 主程序逻辑 ...

    // 停止监听
    consumer.StopMonitoring();
```

5. 错误处理

5.1 主要错误类型

1. 初始化错误
    - 互斥锁创建失败
    - 共享内存创建失败
    - 内存映射失败
2. 运行时错误
    - 内存未初始化
    - 数据大小超限
    - 校验和不匹配
    - 状态不一致

5.2 错误恢复

1. 使用`ClearMemory()`方法重置共享内存状态
2. 重新初始化共享内存
3. 记录详细日志用于问题诊断

6. 性能优化

1. 零拷贝传输：直接在共享内存中读写数据
2. 校验和计算优化：使用简单快速的算法
3. 状态检查优化：避免不必要的日志记录
4. 互斥锁超时设置：防止死锁

7. 注意事项

1. 确保生产者和消费者使用相同的共享内存名称
2. 正确处理程序异常退出的情况
3. 定期检查和清理共享内存状态
4. 合理设置缓冲区大小，避免内存浪费
5. 注意32位和64位程序的兼容性

8. 调试方法

1. 使用日志文件
    - producer_log.txt：生产者日志
2. 状态监控
    - 使用`LogStatus()`方法查看当前状态
    - 检查帧ID连续性
    - 监控数据大小变化
3. 常见问题排查
    - 检查魔数是否匹配
    - 验证数据大小是否合理
    - 确认互斥锁是否正常释放
    - 查看校验和计算是否正确

9. 示例程序

9.1 生产者-消费者测试程序

```
int main()
{
    try
    {
        std::cout << "Shared Memory Test Program Starting..." << std::endl;

        // 创建生产者和消费者
        const std::string memoryName = "TestSharedMemory";
        const size_t memorySize = 1024 * 1024 * 10; // 10MB

        // 创建并初始化生产者和消费者
        ShareMemoryManager producer(memoryName, memorySize);
        ShareMemoryManager consumer(memoryName, memorySize);

        if (!producer.Initialize() || !consumer.Initialize()) {
            std::cerr << "Failed to initialize shared memory" << std::endl;
            return 1;
        }

        // 设置数据接收回调
```

```
        consumer.SetDataReceivedCallback([](const uint8_t* data, size_t size,
            uint32_t dataType, uint32_t width, uint32_t height) {
            std::cout << "\n[Consumer] Received data:"
                      << "\n - Type: " << (dataType == static_cast<uint32_t>
(FrameType::HEIGHTMAP) ? "HeightMap" :
                                          (dataType == static_cast<uint32_t>
(FrameType::IMAGE) ? "Image" : "PointCloud"))
                      << "\n - Size: " << size << " bytes"
                      << "\n - Width: " << width
                      << "\n - Height: " << height
                      << "\n - First byte: 0x" << std::hex << (int)data[0]
                      << std::dec << std::endl;

            // 根据数据类型进行处理
            switch (static_cast<FrameType>(dataType)) {
                case FrameType::IMAGE: {
                    const uint8_t* imageData = data;
                    // 处理图像数据...
                    break;
                }
                case FrameType::POINTCLOUD: {
                    const float* pointCloud = reinterpret_cast<const float*>
(data);
                    // 处理点云数据...
                    break;
                }
                case FrameType::HEIGHTMAP: {
                    const float* heightMap = reinterpret_cast<const float*>(data);
                    // 计算高度范围
                    float minHeight = heightMap[0];
                    float maxHeight = heightMap[0];
                    for (size_t i = 1; i < width * height; i++) {
                        minHeight = std::min(minHeight, heightMap[i]);
                        maxHeight = std::max(maxHeight, heightMap[i]);
                    }
                    std::cout << " - Height range: [" << minHeight << ", " <<
maxHeight << "]" << std::endl;
                    break;
                }
            }
        });

        // 启动消费者监听
        consumer.StartMonitoring();

        std::cout << "Starting test data exchange..." << std::endl;

        int frameCount = 0;
        const int totalFrames = 10; // 发送10帧后退出

        while (frameCount < totalFrames)
        {
            std::vector<uint8_t> data;
            DataInfo info = {};
```

```
            info.timestamp = std::chrono::duration_cast<std::chrono::milliseconds>
(
                std::chrono::system_clock::now().time_since_epoch()
            ).count();

            switch (frameCount % 3) {
                case 0: {  // 发送图像数据
                    info.channels = 3;
                    info.width = 640;
                    info.height = 480;
                    info.dataType = static_cast<uint32_t>(FrameType::IMAGE);
                    data = GenerateTestImage(info.width, info.height,
info.channels);
                    std::cout << "Preparing to write RGB image data..." <<
std::endl;
                    break;
                }
                case 1: {  // 发送点云数据
                    info.width = 1000;  // 点数量
                    info.height = 3;    // XYZ维度
                    info.dataType = static_cast<uint32_t>(FrameType::POINTCLOUD);
                    data = GenerateTestPointCloud(info.width);
                    std::cout << "Preparing to write XYZ point cloud data..." <<
std::endl;
                    break;
                }
                case 2: {  // 发送高度图数据
                    info.width = 200;
                    info.height = 200;
                    info.xSpacing = 0.1f;
                    info.ySpacing = 0.1f;
                    info.dataType = static_cast<uint32_t>(FrameType::HEIGHTMAP);

                    std::vector<float> heightData =
GenerateTestHeightMap(info.width, info.height);
                    data.resize(heightData.size() * sizeof(float));
                    memcpy(data.data(), heightData.data(), data.size());
                    std::cout << "Preparing to write height map data..." <<
std::endl;
                    break;
                }
            }

            // 写入数据
            if (producer.WriteData(data.data(), data.size(), info)) {
                frameCount++;
            }

            // 等待下一帧
            std::this_thread::sleep_for(std::chrono::milliseconds(500));
        }

        // 停止消费者监听
        consumer.StopMonitoring();
```

```
        std::cout << "Test completed successfully." << std::endl;
    }
    catch (const std::exception& e)
    {
        std::cerr << "Program exception: " << e.what() << std::endl;
        return 1;
    }

    return 0;
}
```

9.2 性能测试结果

数据类型	大小	传输延迟	CPU占用
RGB图像 (640x480)	921.6KB	<1ms	<5%
点云 (1000点)	12KB	<1ms	<4%
高度图 (200x200)	160KB	<1ms	<2%