3 ФУНКЦИОНАЛЬНОЕ ПРОЕКТИРОВАНИЕ

Рассмотрим подробнее функционирование программного модуля. В предыдущем разделе были описаны все блоки программного модуля. В этом разделе будут описаны классы, с помощью которых будет реализован функционал каждого модуля.

Диаграмма классов программного модуля для построения трёхмерных моделей объектов по изображениям приведена на чертеже ГУИР.400201.040 РР.1.

Далее рассмотрим каждый блок программного модуля и классы, которые в нём реализованы.

3.1 Функциональное проектирование блока декодирования изображений

Данный блок предназначен для преобразования изображений из формата, в котором оно было сохранено, в формат, пригодный для применения алгоритмов обработки.

Приложение будет поддерживать достаточно большое количество форматов изображений: bmp, dib, jpeg, jpg, jpe, jp2, png, webp, pbm, pgm, ppm pxm, pnm, sr, ras, tiff, tif, exr, hdr, pic. Этот список форматов во многом обусловлен форматами, которые поддерживаются в библиотеке OpenCV в модуле imgcodecs.

Особое внимание стоит обратить на форматы JPEG(jpeg, jpg, jpe) и JPEG2000(jp2). Так как декодирование изображений в OpenCV выполняется на CPU, оно не обладает большой скоростью. Для форматов JPEG и JPEG2000 компания Nvidia выпустила библиотеки, которые выполняют декодирование на GPU. Данные библиотеки называются NvJPEG и NvJPEG2000.

Сравним скорость декодирования изображений с помощью OpenCV и NvJPEG. Для этого выполним декодирование нескольких изображений и измерим время декодирования одного изображения (см. таблицу 3.1).

Таблица 3.1 – Время декодирования изображений OpenCV и NvJPEG

Номер изображения	Время декодирования OpenCV, мкс.	Время декодирования NvJPEG, мкс.
1	8299	2820
2	8431	2811
3	8517	2830
4	9921	2717
5	8625	2802
6	8669	2605
7	8852	2543
8	8773	2452

Данный эксперимент проводился на CPU Intel Core i9-9900 и GPU Nvidia RTX 3080. Среднее время декодирования изображения средствами OpenCV – 8761 мкс, средствами NvJPEG – 2698 мкс. Как можно заметить, скорость декодирования с помощью NvJPEG выше более чем в 3 раза.

Для NvJPEG2000 можно добиться примерно того же результата.

Исходя из полученных результатов, следует отдельно выделить классы для декодирования изображений, которые используют GPU. При этом в силу небольшого числа форматов, которые можно декодировать на графическом процессоре, остальные форматы можно декодировать одним классом, основанным на OpenCV.

В таблице 3.2 приведены классы данного блока и их назначение.

Таблица 3.2 – Классы блока декодирования изображений и их назначение

Название клсса	Назначение
IImageDecoder	Базовый класс для декодеров изображений.
ImageDecoderFactory	Класс, реализующий шаблон проектирования «фабрика» для создания различных декодеров.
NvJPEG2kImageDecoder	Декодер изображений в формате JPEG2000.
NvJPEGHardwareImageDecoder	Декодер изображений в формате JPEG, использующий аппаратное декодирование изображений. Доступно только для видеокарты Nvidia A100.
NvJPEGImageDecoder	Декодер изображений в формате JPEG.
OpenCVImageDecoder	Универсальный декодер.

В таблицах 3.3 – 3.7 приведены поля и методы классов, реализованных в данном блоке.

Таблица 3.3 – Поля и методы класса IImageDecoder

Сигнатура поля/метода	Назначение
`IImageDecoder()`	Конструктор по умолчанию.
`virtual ~IImageDecoder() noexcept(false)`	Виртуальный деструктор по умолчанию.
`virtual bool Decode(const unsigned char* data, unsigned long long size,CUDAImage& decodedImage)`	Виртуальный метод, предназначенный для декодирования изображения.
`virtual void Initialize()`	Виртуальный метод, выполняющий инициализацию декодера.
`virtual bool IsInitialized()`	Виртуальный метод, проверяющий, инициализирован ли декодер.

Данный класс является базовым для всех декодеров.

Таблица 3.4 – Поля и методы класса ImageDecoderFactory

Сигнатура поля/метода	Назначение
`static std::unique_ptr<IImageDecoder> Create(DecoderType type, bool useCUDAStream, void* cudaStream)`	Данный метод возвращает указатель на созданный объект декодера, тип которого передаётся как аргумент данной функции.

Таблица 3.5 – Поля и методы класса NvJPEGImageDecoder

Сигнатура поля/метода	Назначение
1	2
`explicit NvJPEGImageDecoder(cudaStream_t cudaStream)`	Конструктор.
`~NvJPEGImageDecoder() noexcept(false) override`	Деструктор.
`bool Decode(const unsigned char* data, unsigned long long size, DataStructures::CUDAImage& decodedImage) override`	Метод, предназначенный для декодирования изображения. Основан на методе DecodeInternal.
`void Initialize() override`	Метод, выполняющий инициализацию декодера.
`bool IsInitialized() override`	Метод, проверяющий, инициализирован ли декодер.
`bool DecodeInternal(const unsigned char* data, unsigned long long size, DataStructures::CUDAImage& image)`	Метод, выполняющий декодирование изображение в формате JPEG на GPU.
`virtual void AllocateBuffer(int width, int height, int channels)`	Метод, выделяющий буфер для изображение в памяти GPU.
`nvjpegJpegState_t state_`	Хранит промежуточную информацию этапов декодирования.
`nvjpegJpegState_t decoupledState_`	Внутреннее состояние декодера.

Продолжение таблицы 3.5

Сигнатура поля/метода	Назначение
1	2
`nvjpegHandle_t handle_`	Используется для внутренних нужд NvJPEG.
`size_t bufferSize_`	Размер выделенного буфера.
`bool initialized_`	Флаг инициализации декодера.
`bool InitDecoder()`	Метод, инициализирующий декодер.
`nvjpegBufferPinned_t pinnedBuffer_`	Используется для внутренних нужд NvJPEG.
`nvjpegBufferDevice_t deviceBuffer_`	Используется для внутренних нужд NvJPEG.
`nvjpegDecodeParams_t decodeParams_`	Используется для внутренних нужд NvJPEG.
`nvjpegJpegDecoder_t decoder_`	NvJPEG декодер.
`nvjpegJpegStream_t jpegStream_`	Хранит информацию о декодируемом изображении.
`cudaStream_t cudaStream_`	Используется в асинхронных операциях копирования.
`nvjpegImage_t imageBuffer_`	Буфер для изображения в памяти GPU.

Таблица 3.6 – Поля и методы класса OpenCVImageDecoder

Сигнатура поля/метода	Назначение
`OpenCVImageDecoder()`	Конструктор.
`~OpenCVImageDecoder() noexcept(false) override`	Деструктор.
`bool Decode(const unsigned char* data, unsigned long long size, DataStructures::CUDAImage& decodedImage) override`	Виртуальный метод, предназначенный для декодирования изображения.
`void Initialize() override`	Метод, выполняющий инициализацию декодера.
`bool IsInitialized() override`	Метод, проверяющий, инициализирован ли декодер.

Таблица 3.7 – Поля и методы класса NvJPEG2kImageDecoder

Сигнатура поля/метода	Назначение
`explicit NvJPEG2kImageDecoder(cudaStream_t cudaStream)`	Конструктор.
`std::vector<unsigned char*> bufferChannels_`	Хранит указатели на участки памяти GPU для буферизации изображений
`std::vector<size_t> bufferChannelsPitches_`	Флаг инициализации декодера.
`std::vector<size_t> bufferChannelsSizes_`	Метод, инициализирующий декодер.
`bool initialized_`	Флаг инициализации декодера.
`~NvJPEG2kImageDecoder() noexcept(false) override`	Деструктор.
`bool Decode(const unsigned char* data, unsigned long long size, DataStructures::CUDAImage& decodedImage) override`	Виртуальный метод, предназначенный для декодирования изображения.
`bool DecodeInternal(const unsigned char* data, unsigned long long size, DataStructures::CUDAImage& outputImage)`	Метод, выполняющий декодирование изображение в формате JPEG2000 на GPU.
`void AllocateBuffer(int width, int height, int channels, size_t elementSize)`	Метод, выделяющий буфер для изображение в памяти GPU.
`bool InitDecoder()`	Инициализирует NvJPEG2000 декодер.
`cudaStream_t cudaStream_`	Используется для асинхронных операций на GPU.
`nvjpeg2kHandle_t handle_`	Используется для внутренних нужд NvJPEG2000.
`nvjpeg2kDecodeState_t decodeState_`	Хранит промежуточную информацию этапов декодирования.
`nvjpeg2kStream_t`	Хранит информацию о декодируемом изображении.

3.2 Функциональное проектирование блока управления графическими процессорами

В этом блоке будут реализованы классы, которые содержат функционал выбора активного графического процессора.

Классы данного блока перечислены в таблице 3.8.

Таблица 3.8 – Классы блока декодирования изображений и их назначение

Название клсса	Назначение
GPU	Класс, хранящий информацию о графическом процессоре.
GpuManager	Класс, отвечающий за выбор активного графического процессора.

Поля и методы вышеперечисленных классов расписаны в таблицах 3.9 и 3.10

Таблица 3.9 – Поля и методы класса GPU

Сигнатура поля/метода	Назначение
`int deviceId_`	Индекс GPU.
`std::string name_`	Название GPU.
`int computeCapabilityMajor_`	Старшая цифра версии Compute Capability.
`int computeCapabilityMinor_`	Младшая цифра версии Compute Capability.
`int multiprocessorsAmount_`	Количество потоковых мультипроцессоров на GPU.
`std::size_t memoryTotal_`	Объём глобальной памяти на GPU.
`double memoryBandwidth_`	Пропускная способность памяти GPU.
`int maxThreadsPerMultiprocessor_`	Максимальное количество потоков, одновременно выполняемых на одном мультипроцессоре.
`int maxThreadsPerBlock_`	Максимальное количество потоков в CUDA блоке.
`std::size_t sharedMemPerBlock_`	Максимальный объём разделяемой памяти в блоке.

Данный класс описывает все CUDA-совместимые устройства системы.

Далее будет описан класс GpuManager – класс, отвечающий за выбор активного GPU для исполнения алгоритмов фотограмметрии.

Таблица 3.10 – Поля и методы класса GpuManager

Сигнатура поля/метода	Назначение
`void UpdateCUDACapableDevicesList()`	Обновляет список графических процессоров и записывает данные в cudaCapableDevices_.
`unsigned int GetCUDACapableDevicesAmount()`	Определяет число графических процессоров.
`const std::vector<GPU>& GetCUDACapableDevicesList()`	Предоставляет доступ к элементу cudaCapableDevices_.
`GPU& SelectMatchingGPU(const std::shared_ptr<Config::JsonConfig>& serviceConfig)`	Выбирает GPU с учётом заданной политики.
`void SetDevice(GPU& gpu)`	Назначает выбранный GPU активным.
`const std::shared_ptr<GPU>& GetCurrentGPU()`	Возвращает указатель на текущий активный GPU.
`std::vector<GPU> cudaCapableDevices_`	Массив, содержащий информацию обо все GPU системы.
`std::shared_ptr<GPU> selectedGPU_`	Указатель на текущий активный GPU.

3.3 Функциональное проектирование блока инициализации

Данный блок будет состоять из 1 класса, который будет реализовывать шаблон проектирования «фасад». Поля и методы данного класса приведены в таблице 3.11.

Таблица 3.11 – Поля и методы класса ServiceSDK

Сигнатура поля/метода	Назначение
1	2
`ServiceSDK(int argc, char** argv)`	Конструктор.
`~ServiceSDK()`	Деструктор.
`void Initialize()`	Инициализирует все блоки программы.
`void Start()`	Переводит потоки управляющих единиц в режим исполнения заданных алгоритмов.
`void InitializeConfigFolderPath()`	Определяет путь к папке с конфигурацией сервиса.

Продолжение таблицы 3.11

Сигнатура поля/метода	Назначение
1	2
`void InitializeProcessingQueues(const std::shared_ptr<Config::JsonConfig>& serviceConfig)`	Инициализирует очереди для хранения результатов работы исполняющих единиц.
`void InitializeProcessors(const std::shared_ptr<Config::JsonConfig>& serviceConfig)`	Инициализирует исполняющие единицы программного модуля.
`std::shared_ptr<Config::JsonConfig> GetServiceConfig()`	Обращается к блоку конфигурирования, получает указатель на конфигурацию сервиса, в которой указано, сколько необходимо создать исполняющих единиц и очередей, как соединить созданные компоненты между собой.
`static void ValidateServiceConfiguration(const std::shared_ptr<Config::JsonConfig>& serviceConfig)`	Валидирует конфигурацию сервиса на предмет соответствия формату.
`constexpr const inline static char* organizationName`	Константа, хранящая название организации-разработчика.
`constexpr const inline static char* productName`	Константа, хранящая название программного продукта.
`constexpr const inline static char* version`	Константа, хранящая версию программного продукта.
`std::unique_ptr<Config::JsonConfigManager> configManager_`	Указатель на управляющий класс в блоке конфигурирования.
`std::unique_ptr<GPU::GpuManager> gpuManager_`	Указатель на управляющий класс в блоке управления графическими процессорами.
`std::unique_ptr<DataStructures::ProcessingQueueManager> queueManager_`	Указатель на класс, управляющий очередями.
`std::unique_ptr<Processing::ProcessorManager> processorManager_`	Указатель на класс, управляющий исполняющими единицами.

Продолжение таблицы 3.11

Сигнатура поля/метода	Назначение
1	2
`void InitializeConfigManager()`	Инициализирует блок конфигурации.
`void InitializeGpuManager()`	Инициализирует блок управления GPU.
`void InitializeServiceGPU(const std::shared_ptr<Config::JsonConfig>& serviceConfig)`	Производит обращение к блоку управления GPU, получает активный графический процессор.
`std::string configPath_`	Путь к папке, содержащей конфигурационные файлы всех компонентов пограммы.

Стоит отметить, что данный класс является одним из основных классов проекта. Данный класс содержит в себе указатели на объекты управляющих классов большинства блоков программного модуля, что позволяет упростить управление всей системой в целом, используя для этого средства одного класса.

3.4 Функциональное проектирование блока управления исполняющими единицами

Данный блок содержит набор классов, отвечающих за выполнение алгоритмов на CPU и GPU. Классы данного блока перечислены в таблице 3.12.

Таблица 3.12 – Классы блока управления исполняющими единицами

Название клсса	Назначение
IProcessor	Базовый класс исполняющей единицы.
CpuProcessor	Исполняющая единица, задействующая только CPU.
GpuProcessor	Исполняющая единица, задействующая CPU и GPU.
IThread	Базовый класс исполняющего потока.
Thread	Класс потока, обеспечивающего однократное выполнение задачи.
EndlessThread	Класс потока, обеспечивающего бесконечное выполнение задачи.
ProcessorManager	Класс, управляющий исполняющими единицами.

В таблицах 3.13 – 3.15 приведены описания полей и методов перечисленных классов.

Таблица 3.13 – Поля и методы класса IProcessor

Сигнатура поля/метода	Назначение
1	2
`IProcessor(const std::shared_ptr<Config::JsonConfig>& config, const std::unique_ptr<DataStructures::ProcessingQueueManager>& queueManager)`	Конструктор.
`virtual ~IProcessor() = default`	Виртуальный деструктор.
`virtual void Process() = 0`	Виртуальный метод для запуска выполнение команд.
`virtual void Stop() = 0`	Виртуальный метод для остановки выполнения команд.
`virtual void InitializeAlgorithms(const std::unique_ptr<IAlgorithmFactory>& algorithmFactory, const std::unique_ptr<JsonConfigManager>& configManager, const std::unique_ptr<GPU::GpuManager>& gpuManager) = 0`	Виртуальный метод, инициализирующий список исполняемых алгоритмов.
`virtual void Initialize() = 0`	Виртуальный метод инициализации.
`virtual bool IsStarted() = 0`	Виртуальный метод, проверяющий состояние объекта.
`const std::string& GetName()`	Возвращает имя исполняющей единицы.
`void SetName(const std::string& name)`	Задаёт имя исполняющей единицы.
`std::string name_`	Имя исполняющей единицы.
`std::shared_ptr<ProcessingQueue<std::shared_ptr<ProcessingData>>> inputQueue_`	Указатель на очередь, из которой будут браться данные для обработки.
`std::vector<std::unique_ptr<Algorithms::IAlgorithm>> processAlgorithms_`	Список выполняемых алгоритмов.

Продолжение таблицы 3.13

Сигнатура поля/метода	Назначение
1	2
`std::shared_ptr<DataStructures::ProcessingQueue<std::shared_ptr<DataStructures::ProcessingData>> outputQueue_`	Указатель на очередь, в которую будут помещаться данные после обработки.

Таблица 3.14 – Поля и методы класса CpuProcessor

Сигнатура поля/метода	Назначение
1	2
`CpuProcessor(const std::shared_ptr<Config::JsonConfig>& config, const std::unique_ptr<DataStructures::ProcessingQueueManager>& queueManager)`	Конструктор.
`~CpuProcessor() override`	Деструктор.
`void Process() override`	Метод для запуска выполнение команд.
`void Stop() override`	Метод для остановки выполнения команд.
`bool IsStarted() override`	Метод, проверяющий, происходит ли выполнение алгоритмов в данный момент.
`void InitializeAlgorithms(const std::unique_ptr<Algorithms::IAlgorithmFactory>& algorithmFactory, const std::unique_ptr<Config::JsonConfigManager>& configManager, const std::unique_ptr<GPU::GpuManager>& gpuManager) override`	Метод, инициализирующий список исполняемых алгоритмов.

Продолжение таблицы 3.14

Сигнатура поля/метода	Назначение
1	2
`std::shared_ptr<DataStructures::ProcessingQueue<std::shared_ptr<DataStructures::ProcessingData>>> outputQueue_`	Указатель на очередь, в которую будут помещаться данные после обработки.
`std::vector<std::unique_ptr<Algorithms::IAlgorithm>> processingAlgorithms_`	Список выполняемых алгоритмов.
`void Initialize() override`	Метод инициализации.
`static void ValidateAlgorithmConfig(const std::shared_ptr<Config::JsonConfig>& algorithmConfig)`	Метод, проверяющий конфигурацию объекта на соответствие формату.
`EndlessThread thread_`	Исполняющий поток.

Таблица 3.15 – Поля и методы класса GpuProcessor

Сигнатура поля/метода	Назначение
1	2
`CpuProcessor(const std::shared_ptr<Config::JsonConfig>& config, const std::unique_ptr<DataStructures::ProcessingQueueManager>& queueManager)`	Конструктор.
`~CpuProcessor() override`	Деструктор.
`void Process() override`	Метод для запуска выполнение команд.
`void Stop() override`	Метод для остановки выполнения команд.
`bool IsStarted() override`	Метод, проверяющий, происходит ли выполнение алгоритмов в данный момент.
`cudaStream_t cudaStream_`	Очередь команд для GPU.
`EndlessThread thread_`	Исполняющий поток.
`void Initialize() override`	Метод инициализации.

Продолжение таблицы 3.15

Сигнатура поля/метода	Назначение
1	2
`void InitializeAlgorithms(const std::unique_ptr<Algorithms::IAlgorithmFactory>& algorithmFactory, const std::unique_ptr<Config::JsonConfigManager>& configManager, const std::unique_ptr<GPU::GpuManager>& gpuManager) override`	Метод, инициализирующий список исполняемых алгоритмов.
`static void ValidateAlgorithmConfig(const std::shared_ptr<Config::JsonConfig>& algorithmConfig)`	Метод, проверяющий конфигурацию объекта на соответствие формату.