СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Шейдерные программы – инструменты, с помощью которых можно добиться невероятных результатов в компьютерной графике [7]. Профессиональные программисты, работающие с графикой, с помощью шейдеров создают всевозможные эффекты: добиваются эффекта фотореалистичности итогового изображения, поражают эффектами из миллиардов частиц, акцентируют внимание замыливанием заднего фона, придают плоскостям эффект объёмности и многое другое. Главной целью написание шейдеров является изменение конвейера рендеринга графики для получения желаемого результата. Современные компьютеры способны в реальном времени рассчитывать 60 раз в секунду динамические картины, близкие к максимальному фотореализму, в высоком разрешении, что ещё 20 лет назад было невозможно, и всё это благодаря шейдерам. Получив возможность менять графический конвейер[1], программисты смогли увеличить производительность систем построения изображений, перекладывая всё больше вычислений на графические ускорители. Ранее такой подход был невозможен, так как порядок построения изображения и функциональные возможности были строго ограничены и не могли быть изменены, из-за чего приходилось выполнять некоторые расчёты, такие как освещение, на центральном процессоре. Появление возможности писать шейдеры и встраивать их в графический конвейер изменило мир компьютерной графики в сторону гибкости и производительности. Сегодня шейдеры являются актуальной темой исследований, так как они являются важной частью любой программы, которая работает с графикой и использует для этого аппаратное ускорение, ведь благодаря им возможно быстро преобразовать геометрию, рассчитать промежуточные данные и вычислить итоговый цвет сразу для нескольких сотен пикселей в итоговом изображении. Инструментом для создания шейдера может быть любой текстовый редактор, однако не у всех есть достаточные знания для написания шейдеров, способных выдать желаемый результат. Некоторые программы стараются предоставить простой интерфейс для настройки заранее подготовленных программ, однако такой имеет ряд ограничений и не всегда оправдывает ожидания.

Главной проблемой при написании шейдеров является то, что сам процесс является нетривиальным. Одной из причин является существование разных популярных графических API, а также существование нескольких языков программирования шейдеров. Это создаёт некоторые проблемы при написании шейдеров для разных графических API, так как каждый API использует свои языки программирования шейдеров. Каким должен быть инструмент разработки шейдеров, который устранит эти недостатки?

Объектом исследования являются шейдеры, их структура и способы их применения для решения практических задач. Предметом исследования является процесс написания шейдеров, особенности их реализации в разных API и способы упрощения разработки шейдеров.

Целью дипломной работы является разработка программы для создания шейдеров, с помощью которой можно будет реализовать низкоуровневую и высокоуровневую логику шейдеров, используя для этого только унифицированные графические элементы. Основные задачи, которые должны быть решены для достижения цели:

— анализ исходных данных;
— выбор инструментальных средств для реализации;
— проектирование программного обеспечения;
— разработка графического интерфейса;
— реализация функциональных частей;
— тестирование результатов.

Предполагается, что в результате выполнения дипломной работы будет реализована программа, функциональные и графические особенности которой позволят упростить и ускорить процесс разработки шейдеров для разных API.

1 АНАЛИЗ ИСХОДНЫХ ДАННЫХ И ПОСТАНОВКА ЗАДАЧ

При разработке программного обеспечения важно ещё в начале пути разработать архитектуру, которая будет одновременно гибкой, масштабируемой и производительной. Разработка такой архитектуры может занять много времени, если подходить к процессу без каких-либо знаний о предметной области и существующих решений. Это значит, что перед началом проектирования программы необходимо собрать и тщательно проанализировать все доступные данные. Полученные результаты необходимо использовать для полного описания предметной области. Проанализировав аналоги и прототипы разрабатываемого программного продукта будут получены данные для подготовки к проектированию с учётом всех плюсов и минусов, выявленных в аналогичных продуктах.

1.1 Описание предметной области

Растеризация – процесс, в результате которого получается растровое изображение.[1] В компьютерной графике растеризация может быть выполнена двумя способами:

— Software – цвет каждой точки результирующего изображения вычисляется на центральном процессоре;

— Hardware acceleration (аппаратное ускорение) – цвет каждой точки вычисляется на отдельном устройстве – графическом ускорители.

С распространение и удешевлением видеокарт аппаратное ускорение стало активно применяться в компьютерных играх для достижения более качественных и реалистичных результатов с максимально возможной скоростью. Для этого часть работы по построению изображения перекладывалась с процессора на видео ускоритель. В конце 2000 года компания Microsoft осуществила релиз нового API – DirectX 8.0. Основным нововведением нового API стало понятие «Шейдерная модель» – новый подход к разработке, при котором программисту давалась возможность самостоятельно определить поведение видео ускорителя на каждом этапе построения конечного изображения. Для получения желаемого результата программисту предлагается написать шейдер, который затем будет выполняться на виде ускорители и обрабатывать данные так, как это было задумано самим программистом.

Шейдер – специальная программа, экземпляры которой выполняются на графическом процессоре параллельно. На сегодняшний день существует несколько видов шейдеров, которые выполняются на разном этапе в графическом конвейере:

— Vertex shaders – шейдер, в котором происходит обработка вершин геометрии;

— Geometry shaders – шейдер, в котором происходит генерация новых примитивов на основе уже существующих вершин;

— Fragment shaders – шейдер, в котором высчитывается итоговый цвет каждого фрагмента в итоговом изображении;

— Tessellation shaders – шейдер, который используется для изменения сетки объекта в зависимости от определённых условий, которые определяются в этом типе шейдеров;

— Compute shaders – используются для вычисления информации любого желаемого типа, например, анимации или особой формы освещения.

Основными языками программирования шейдеров являются GLSL и HLSL. GLSL используется чаще всего совместно с программами, которые для построения изображения используют OpenGL API. HLSL чаще всего используется в программах, использующих DirectX API. Оба языка могут использоваться для написания шейдеров с последующей их компиляцией для использования совместно с Vulkan API.

OpenGL и Vulkan являются кроссплатформенными API, благодаря чему они могут использоваться на любой платформе без необходимости полностью переписывать программный код. Для OpenGL существует несколько разных спецификаций: OpenGL, OpenGL ES, WebGL и другие. Отличия между ними минимальны и основные подходы к разработки очень сильно похожи, однако есть и различия, определяемые спецификой целевой платформы. DirectX является разработкой компании Microsoft, поэтому данный API разрабатывается только для операционной системы Windows.

Визуальное программирование – способ создания программ путём манипулирования графическими объектами вместо написания её текста. Преимуществом является визуальная составляющая данного подхода, так как глядя на графические объекты чаще всего проще понять логику программы.

1.2 Анализ аналогов и прототипов

Для более продуктивной и результативной разработки программного продукта необходимо найти и проанализировать аналоги и прототипы проектируемого продукта. Это делается с целью выявления плюсов и минусов существующих решений, что в дальнейшем будет влиять на всю архитектуру и конечные возможности готового программного продукта. В результате поиска инструментов для графического программирования шейдеров были найдены следующие решения: ShaderFrog, GSN Composer, прочие. Прочие программные продукты не рассматриваются, так как в них отсутствует визуальное программирование.

ShaderFrog – WEB приложение для разработки шейдеров на языке программирования шейдеров GLSL. Основным преимуществом является кроссплатформенность, которая достигается за счёт использования браузера в качестве платформы. Присутствует как текстовый, так и графический редакторы, однако минусом является то, что в визуальном редакторе можно использовать только готовые компоненты, написанные в текстовом редакторе. Это создаёт ряд сложностей для активного творчества неподготовленных пользователей. В качестве дополнительной возможности присутствует возможность экспорта наработок для использования в других системах: Three.js, IOS, Unity, однако последние доступны только обладателям платной подписки.

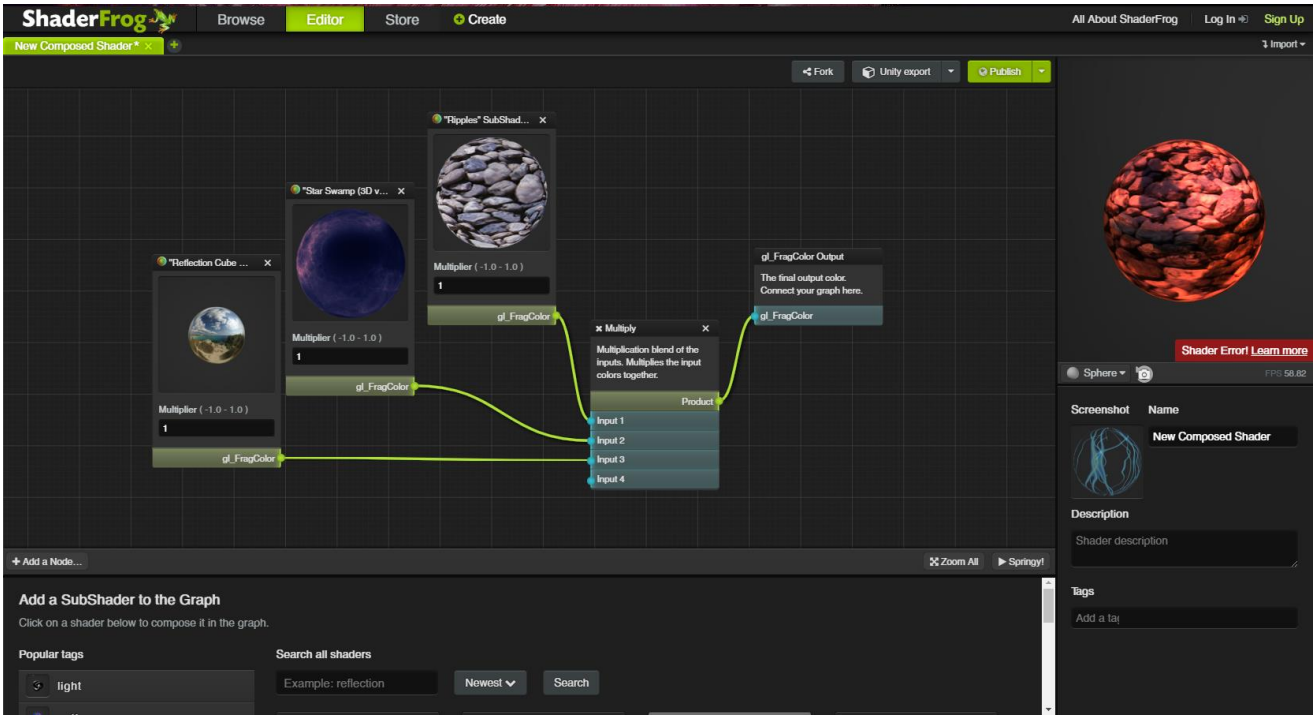

Рисунок 1.1 – Интерфейс программы ShaderFrog

GSN Composer – так же является WEB приложением, однако, в отличии от ShaderFrog, основной способ разработки шейдеров – визуальное программирование с использованием узлов. Разработчику доступно большое количество узлов, выполняющих разные задачи, которые могут быть использованы для достижения поставленной цели. Присутствует возможность экспорта проекта в виде WEB содержимого для встраивания результата на свои HTML страницы.

Минусом этих решений является то, что они, в основном, ориентированы на генерацию кода для программ, использующих API OpenGL. Так же у них нет полностью бесплатной возможности экспортировать проект в другие среды, что было бы очень удобно в некоторых случаях. Одним из таких случаев может стать разработка шейдера, который будет

использоваться как OpenGL API в приложениях для всех систем, так и, например, в приложениях, использующих API DirectX, написанных для операционной системы Windows.

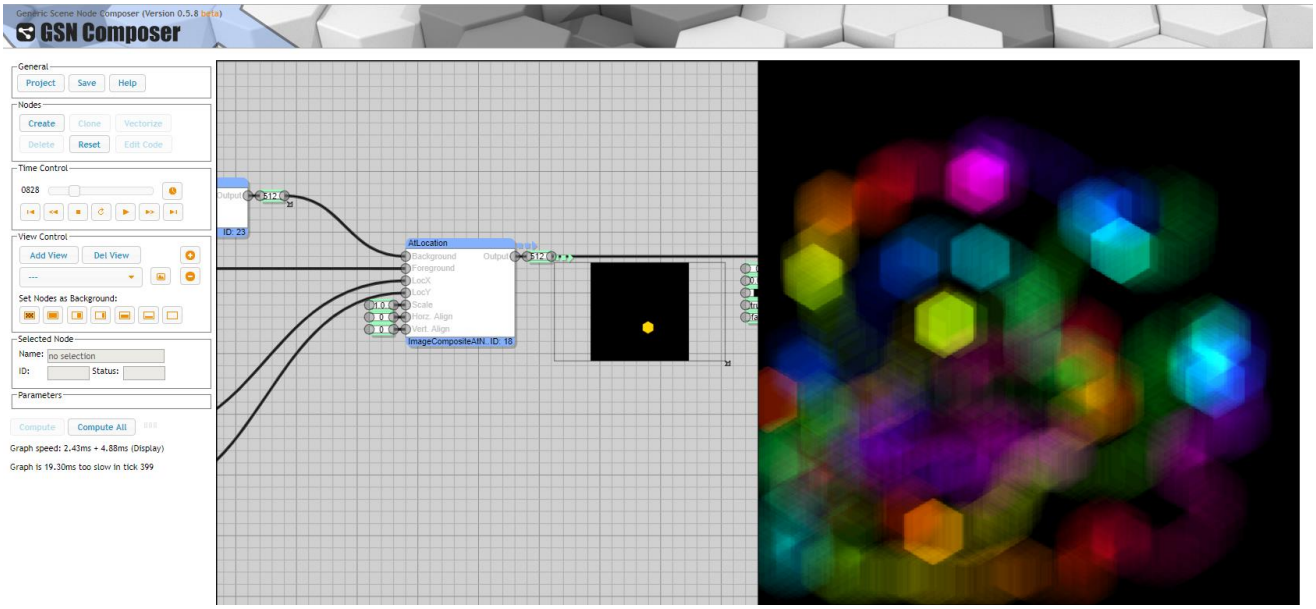

Рисунок 1.2 – Интерфейс программы GSN Composer

На основе полученной информации можно произвести сравнение аналогов с разрабатываемым программным обеспечением. Сравнение разрабатываемого программного обеспечения и его аналогов приведено в таблице 1.1

Таблица 1.1 – Сравнение разрабатываемого программного обеспечения с аналогами

Критерий	**ShaderFrog**	**GSN Composer**	**Разрабатываемое программное обеспечение**
Поддержка узлов	Да	Да	Да
Нет необходимо писать текст программ	Частично	Частично	Да
Поддержка экспорта на разные платформы	Платно	Нет	Да
Расширяемость	Нет	Нет	Да
Магазин готовых шейдеров	Да	Нет	Нет
Пользователю не нужно подстраиваться под результат	Да	Да	Нет

Таким образом, на основе таблицы 1.1 можно сделать вывод, что разрабатываемое программное обеспечение выделяется на фоне аналогов и предлагает больше удобств, в отличии от аналогов.

1.3 Постановка задачи проектирования

Цель данной работы – реализация удобного и простого в использовании графического приложения для разработки шейдерных программ с использованием визуально программирования.

При проектирование программного продукта важно определить приоритетные задачи, невыполнение которых станет критической ошибкой в проектировании. Для достижения цели необходимо решить следующие задачи:

— изучение и систематизация информации;

— выбор языка программирования;

— выбор среды разработки;

— проектирование функциональной структуры программы;

— разработка алгоритмической структуры приложения;

— создание приложения, выполняющего все определенные функции;

— разработка интуитивно понятного пользовательского интерфейса;

— разработка программной документации.

Разрабатываемое программное обеспечение должно решить следующие основные задачи:

— удобный пользовательский интерфейс для проектирования логики шейдера;

— компиляция шейдерной логики в другие форматы представления шейдеров;

— визуализация полученных результатов.

1.4 Анализ требований к проекту

Разрабатываемое приложение должно выполнять рад требований:

— разработка логики шейдерных программ должна быть максимально интерактивной;

— графический интерфейс должен быть интуитивно понятный и адаптивный;

— программа должна работать стабильно и максимально быстро;

— программа должна быть расширяема путём изменения исходного кода, а также путём дополнения её внешними компонентами.

1.5 Выбор среды и средств разработки дипломного проекта

Для разработки целевого программного обеспечения выбран язык программирования C++. Данный язык программирования является одновременно гибким и мощным, что является плюсом при проектировании системы, в которой производительность имеет значение. Недостатком является то, что для C++ нет стандартной библиотеки для разработки графических программ. Для решения этой проблемы подойдёт Qt Framework. Qt – кроссплатформенный framework, инструментарий которого не ограничивается оконными виджетами. В Qt есть всё, что может понадобиться во время разработки: коллекции, контейнеры, классы для работы с JSON, классы для работы с сетью, а та же реализованный паттерн “Observer”, использование которого допускается со специальным препроцессором MOC.

Для автоматического разворачивания проекта на разных операционных системах и для использования с разными компиляторами необходимо использовать автоматическую кроссплатформенную систему сборки программного обеспечения. CMake – проверенная временем система сборки, которая стала негласным стандартом при разработке кроссплатформенных проектов на языках C/C++. Благодаря CMake выбор IDE для разработки целевого программного обеспечения не имеет значения, так как в конечном итоге проект надо будет приготовить в CMake и скомпилировать компилятором. Современные IDE, такие как MSVS, Qt Creator, CLION, Visual Studio Code и другие умеют работать с CMake, что ещё больше упрощает процесс разработки программного обеспечения на языках программирования C/C++.

Для проверки результата своей работы пользователь будет смотреть в модуль вывода, где по умолчанию будет виден стандартный объект, на который применяется шейдер. Чтобы предоставить пользователю возможность загружать объекты для тестирования своих программ, необходимо знать множество форматов, в которых эти объекты могут храниться. Среди огромного количества форматов сложно выбрать наиболее удобный и гибкий формат, однако в этом нет необходимости. Assimp – библиотека, разработанная для получения геометрии объектов и сцен из множества поддерживаемых форматов. Assimp может разобрать более 20 форматов файлов с геометрией, а также автоматически выполнить оптимизации, добавить данные о вершинах и исправить ошибки в модели.

Главной задачей, которую необходимо будет решить – сбор информации о логике программы для её дальнейшего преобразования в исходный код шейдера. Для этих целей оптимально будет использовать JSON-объекты. Для работы с JSON-объектами в Qt реализованы специальные классы, использование которых поможет построить схему логики программы в единый JSON-объект, который затем можно будет передать и обработать в компиляторе.

2 ПРОЕКТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

2.1 Функциональная структура

Техническое задание в приложении А описывает функциональные возможности, которые можно выделить в три основных модуля:

— модуль редактирования логики шейдера;

— модуль компиляции разработанной логики в целевой формат;

— модуль визуализации результатов работы.

Каждый из этих модулей будет выполнять определённый в приложении А набор функций.

Модуль редактирования логики шейдера выполняет все функции, связанные с разработкой логики шейдера, а именно:

— программа должна предоставлять графический инструмент редактирования, основанный на узлах;

— инструмент редактирования, основанный на узлах позволяет создавать и удалять узлы и соединения между ними;

— инструмент редактирования, основанный на узлах позволяет изменять положение каждого узла в произвольном порядке;

— создание, редактирование и удаление пользовательских переменных;

— отображение пользовательских переменных;

— отображение узлов в специальном окне;

— перенос узлов из специального окна на поле редактирования;

— инструмент редактирования должен предоставить набор узлов, достаточный для разработки простых шейдерных программ;

— сохранение результата разработки в удобном для чтения формате.

Модуль компиляции выполняет все функции, связанные с преобразованием разработанной шейдерной логики в целевой формат:

— преобразование разработанной логики в выбранный пользователем вид;

— преобразование не должно мешать работе пользователя;

— в случае ошибки преобразования необходимо сообщить об этом пользователю.

Модуль визуализации выполняет все функции, связанные с визуальным представление результатов работы пользовательской шейдерной логики:

— установка пользовательских 3D моделей;

— вращение сцены с 3D моделью;

— применение разработанных шейдерных эффектов к установленной 3D моделей.

Для организации всех этих модулей необходимо реализовать главный модуль программы, который будет объединять и связывать функционал всех разработанных модулей. На рисунке 2.1 представлена схема функциональной структуры разрабатываемого программного обеспечения.

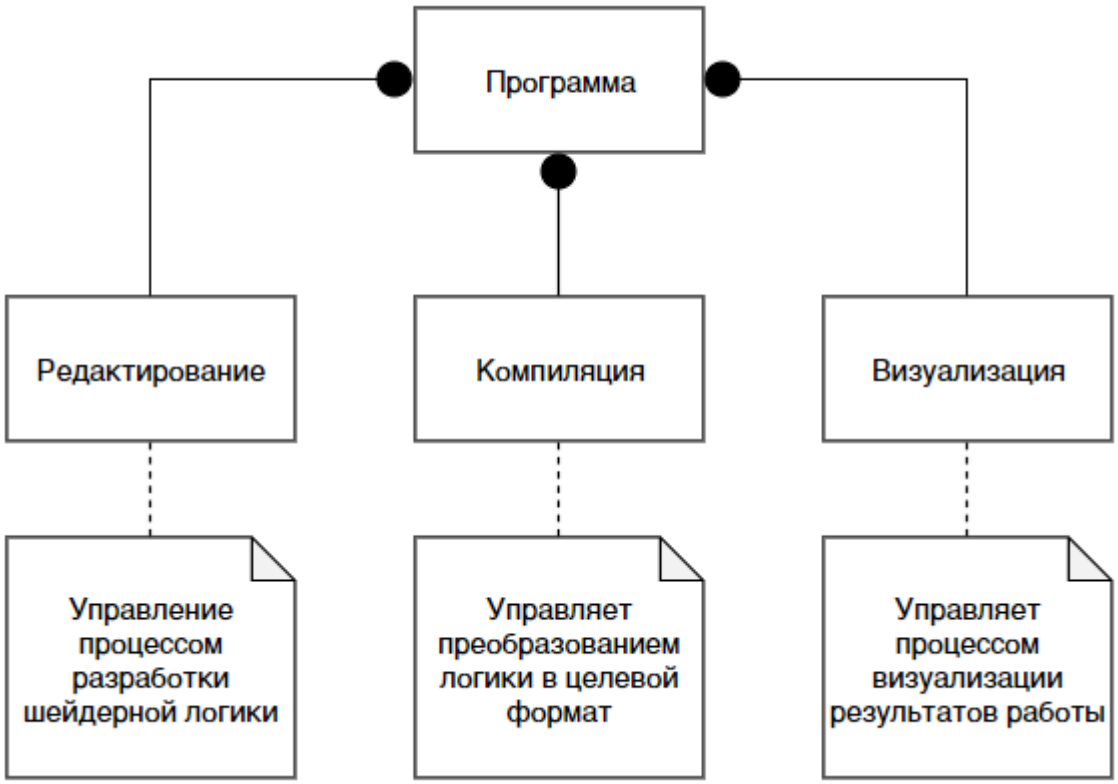

Рисунок 2.1 – Функциональная структура программы

2.2 Варианты использования

В силу модульности разрабатываемого программного обеспечения, где каждый модуль, кроме главного, является независимым от других, все диаграммы вариантов использования (Use case diagram) будут представлены в виде четырёх диаграмм.

Диаграмма вариантов использования модуля редактирования, представленная в приложении Д.1 описывает возможности модуля редактирования, доступные пользователю, при разработке шейдерной логики.

Диаграмма вариантов использования модуля компиляции, представленная в приложении Д.2 описывает возможности модуля компиляции, доступные пользователю, при преобразовании шейдерной логики в целевой результат.

Диаграмма вариантов использования модуля визуализации, представленная в приложении Д.3 описывает возможности модуля визуализации, доступные пользователю, при просмотре результатов работы разработанной логики.

Диаграмма вариантов использования модуля программы, представленная в приложении Д.4 описывает возможности модуля программы, доступные пользователю, для управления всеми остальными модулями в разрабатываемом программном обеспечении.

2.3 Описание классов приложения

Каждый модуль разрабатываемого программного обеспечения должен реализовать свой набор классов, с помощью которых происходит управление состоянием каждого разрабатываемого модуля.

Диаграмма классов, представленная в приложение Е.1, содержит классы и их отношения в модуле редактирования. Ниже будут описаны все классы, входящие в модуль редактирования.

EditorController – класс, являющийся посредником между модулем редактирования и другими модулями разрабатываемой программы. Конструирует все дочерние объекты и предоставляет интерфейс для доступа к данным модуля.

VariablesController – класс, управляющий всеми пользовательскими переменными, а также предоставляющий графический интерфейс для этих целей.

VariableDataModel – структура, описывающая переменную.

VariableDataModelTemplate – шаблонная структура, реализующая хранение значений разных типов данных.

VariableDataModelsFactory – класс-фабрика, специализирующийся на сборке объектов запрашиваемого типа

NodeDataTypeFactory – класс-фаборика, специализирующийся на предоставлении объекта типа данных по его имени.

NodeDataTypeSerializer – класс, выполняющий сереализацию и десереализацию значений пользовательских переменных в Java script object notation (JSON) формат.

VariablesControllerWidget – класс, предоставляющий графический интерфейс для управления переменными.

VariablesEditorWidget – класс, предоставляющий графический интерфейс для редактирования переменных.

VariablesListWidget – класс, предоставляющий графический интерфейс для создания, удаления и просмотра пользовательских переменных.

VariableValueEditor – базовый класс. Является основой для классов, предоставляющих графический интерфейс для редактирования значений переменных.

IntegerValueEditor – реализует графический интерфейс для редактирования значений целочисленных переменных.

UnsignedIntegerValueEditor – реализует графический интерфейс для редактирования значений беззнаковых целочисленных переменных.

FloatValueEditor – реализует графический интерфейс для редактирования значений переменных с плавающей точкой одинарной точности.

DoubleValueEditor – реализует графический интерфейс для редактирования значений переменных с плавающей точкой двойной точности.

BooleanValueEditor – реализует графический интерфейс для редактирования значений логических переменных.

Vector2ValueEditor – реализует графический интерфейс для редактирования значений переменных двухмерных векторов.

Vector3ValueEditor – реализует графический интерфейс для редактирования значений переменных трёхмерных векторов.

Vector4ValueEditor – реализует графический интерфейс для редактирования значений переменных четырёхмерных векторов.

VariableValueEditorsFactory – класс-фабрика, предоставляющая редактор переменной по имени типа данных этой переменной.

NodeStoreWidget – класс, предоставляющий графический интерфейс для представления пользователю списков доступных узлов.

NodeWidget – класс, являющийся графическим представление узла в списке класса NodeStoreWidget.

DefaultDataModelRegistry – класс, выполняющий регистрацию доступных пользователю узлов.

Диаграмма классов, представленная в приложение Е.2, содержит классы и их отношения в модуле компиляции. Ниже будут описаны все классы, входящие в модуль компиляции.

CompilerController – класс, являющийся интерфейсом между внутренней структурой модуля и внешними модулями.

CompilerWrapper – класс, реализующий управление процессом компиляции.

ICompiler – абстрактный класс, являющийся интерфейсом для всех реализаций компиляторов.

DefaultGlslCompiler – класс, являющийся встроенным в модуль стандартным компилятором логики шейдерной программы. Преобразует входные данные в текст шейдера на языке программирования шейдеров GLSL[9].

Диаграмма классов, представленная в приложение Е.3, содержит классы и их отношения в модуле визуализации. Ниже будут описаны все классы, входящие в модуль визуализации.

ICamera – интерфейс, описывающий набор методов камеры, которые должны быть реализованы-наследниками.

ITargetSceneRenderer – интерфейс, описывающий набор методов визуализатора сцены, которые должны быть реализованы всеми классами-наследниками.

LookCentralObjectCamera – класс камеры, реализующий камеру, смотрящую в цент сцены.

OpenGLScene – класс визуализации, реализующий визуализацию сцены, используя для этого API OpenGL.

TargetObject – структура, описывающая геометрию объекта, располагаемого в сцене.

TargetSceneWrapperWidget – класс, реализующий графический интерфейс, включающий в себя интерфейс визуализатора сцены и некоторые элементы управления трёхмерной сценой.

VisualizerController – класс, предоставляющий интерфейс взаимодействия внешних модулей с компонентами модуля визуализации.

Диаграмма классов, представленная в приложение Е.4, содержит классы и их отношения в модуле программы. Ниже будут описаны все классы, входящие в модуль программы.

Application – класс, являющийся главным классом разрабатываемой программы. Реализует связь между всеми модулями внутри программы.

WelcomeWindowDialog – класс, описывающий графический интерфейс диалога открытия и создания пользовательских проектов.

2.4 Диаграмма развёртывания приложения

При разработке диаграмм развертывания преследуются следующие цели:

— специфицировать физические узлы, необходимые для размещения на них исполнимых компонентов программной системы;

— показать физические связи между узлами реализации системы на этапе ее исполнения;

— выявить узкие места системы и реконфигурировать ее топологию для достижения требуемой производительности.

Разрабатываемое программное обеспечение позиционируется как кроссплатформенное программное обеспечение, основными платформами для которого являются Windows, MacOS и Linux.

Диаграмма развёртывания разрабатываемого программного обеспечения представлена в приложение Ж.

2.5 Алгоритмическое представление решаемых задач

Главной задачей разрабатываемого программного обеспечения является процесс преобразования пользовательской логики в целевое представление. Для достижения этой цели предполагается реализация компилятора, выполняющего функцию преобразования одних данных в другие.

В разрабатываемое программное обеспечение должен быть включен базовый компилятор, которого будет достаточно для преобразования логики в исходный код шейдера, который затем может быть использован для визуализации. Для решения проблемы кроссплатформенности базовый компилятор должен преобразовывать исходную логику в текст шейдерной программы на языке программирования шейдеров GLSL.

Данный компилятор, для получения результата, будет выполнять определённый порядок действий:

1 Кэширование информации о всех переменных.

2 Кэширование информации об узлах.

3 Кэширование информации о зависимостях.

4 Взвешивание всех узлов.

5 Сортировка взвешенных узлов.

6 Подготовка заголовка результирующего шейдера и объявление всех входящих переменных.

7 Генерация текстов программы.

8 Объединение текстов программы.

9 Завершение компиляции.

Под кэшированием информации понимается сбор всех данных об объектах сцены из исходных данных и сохранение их в удобном для обработки виде.

Ключевым моментом в этой последовательности является взвешивание узлов, так как благодаря этому этапу становится возможным организация правильной последовательности выполнения всех действий. Схема алгоритма взвешивания представлена на рисунке 2.2.

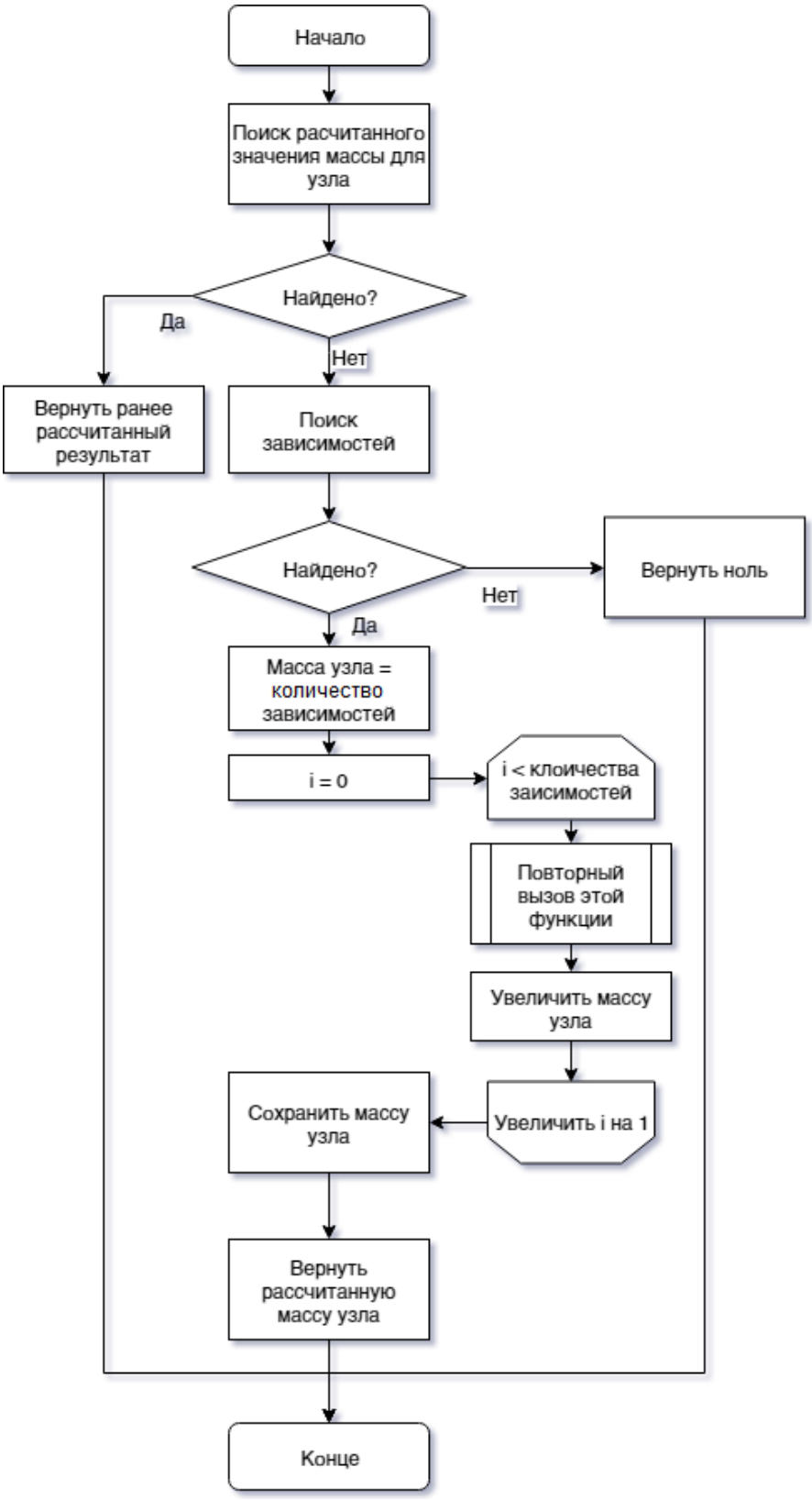

Рисунок 2.2 – Схема алгоритма взвешивания узлов

После взвешивания узлы сортируются в порядке возрастания их массы. Отсортированные узлы учавствуют в процессе формирования последовательности текстовой информации. Схема преобразования узлов в текст представлена на рисунке 2.3.

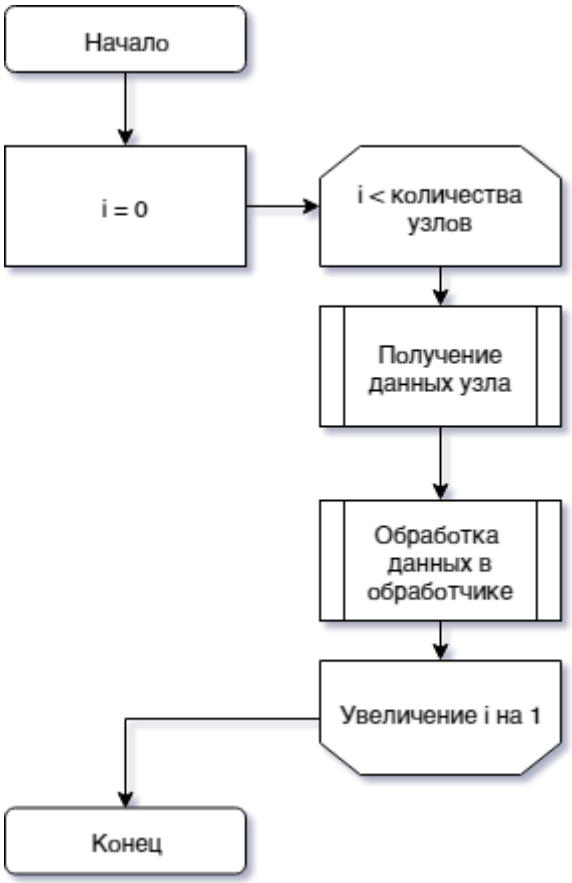

Рисунок 2.3 – Схема обработки отсортированных узлов

После успешной генерации текстовой информации и её объединения в единый блок, результат работы отправляется пользователю.

2.6 Проектирование пользовательского интерфейса

Пользовательский интерфейс – это удобный и понятный для пользователя способ взаимодействия с программой. Графический интерфейс множество преимуществ, среди которых главными преимуществом перед, например, текстовыми интерфейсами, является интуитивность, удобство и наглядность.

Каждый компонент разрабатываемого программного обеспечения должен предоставлять графический интерфейс для управления компонентом, если в этом есть необходимость. Графический интерфейс в разрабатываемом программном обеспечении нужен всем модулям кроме модуля компиляции.

Для управления модулем редактирования необходимо спроектировать интерфейс пользователя для всех его компонент. Графический редактор представляет из себя управляемую сцену с узлами, установленными пользователем. Ну сцене не должно находится сторонних объектов. Макет редактора узлов представлен на рисунке 2.4.

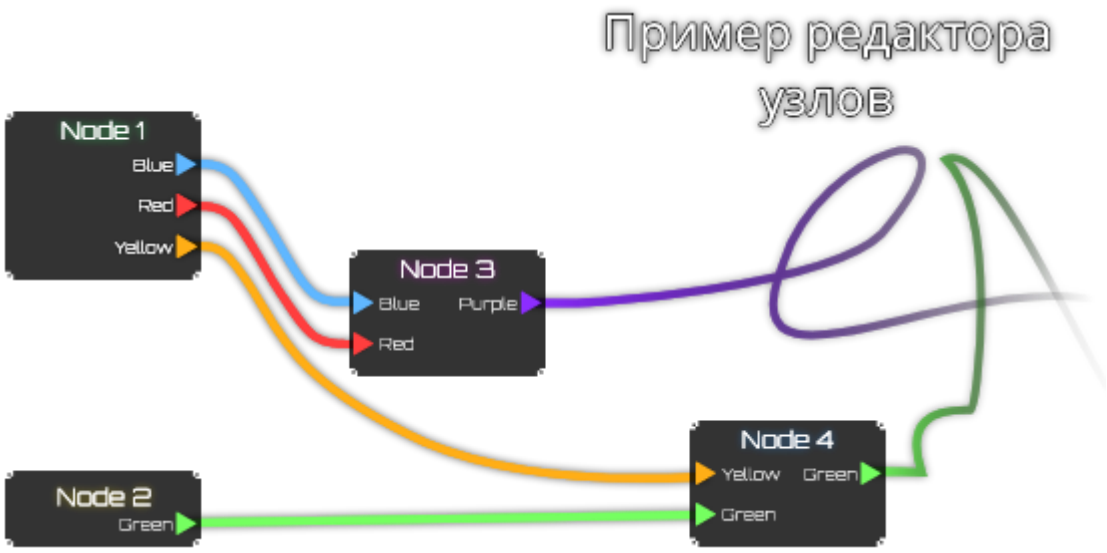

Рисунок 2.4 – Макет графического редактора узлов

Для управления переменными необходимо реализовать графический интерфейс со списком переменных с элементами управления этими переменными, а также графический интерфейс с элементами управления для редактирования параметров переменной. Макет списка переменных представлен на рисунке 2.5. Макет редактора переменных представлен на рисунке 2.6.

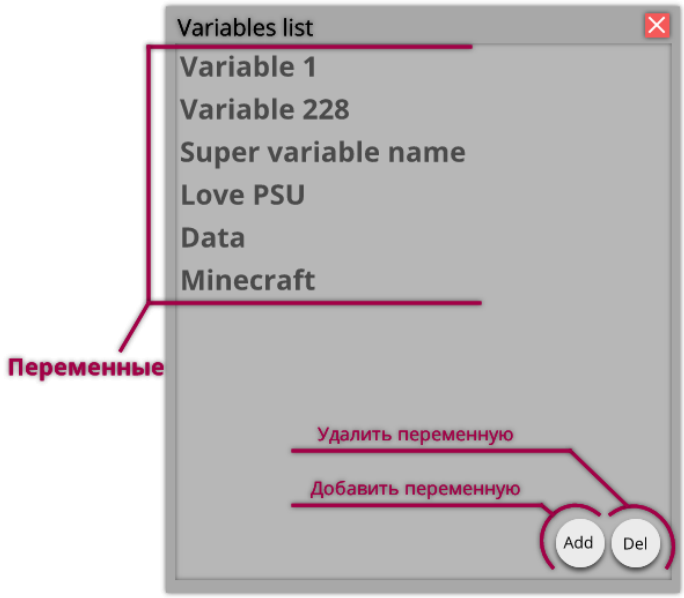

Рисунок 2.5 – Макет списка переменных

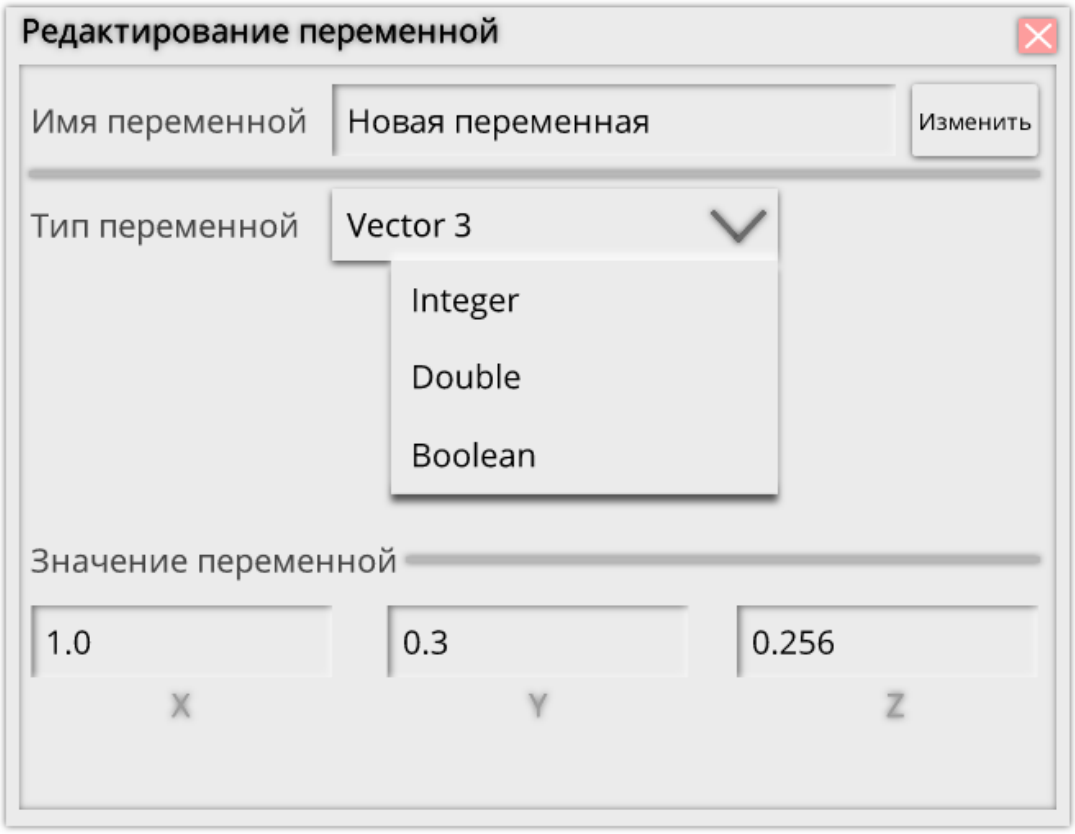

Рисунок 2.6 – Макет редактора значений переменных

Для визуализации достаточно реализовать окно, в котором будет несколько кнопок и представление для визуализации сцены. Макет окна визуализации представлен на рисунке 2.8.

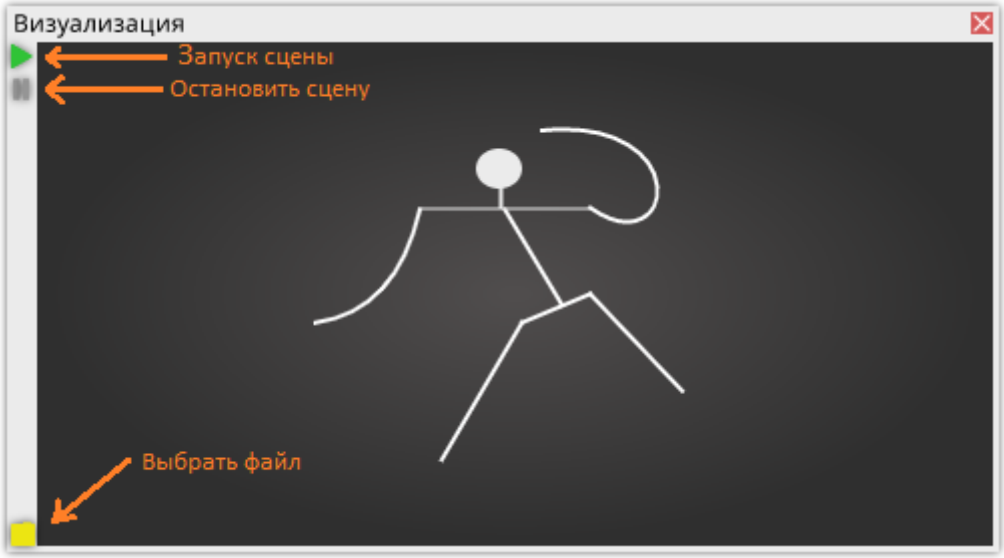

Рисунок 2.8 – Макет окна визуализации

3 РЕАЛИЗАЦИЯ И ТЕСТИРОВАНИЕ

После проектирования программного обеспечения начинается процесс его разработки. Разработка программной составляющей программного обеспечения делится на два этапа: программирование и тестирование. Во время программирования программы реализуются все архитектурные решения, заложенные на этапе проектирования. На этапе тестирования выявляются слабые места в программном коде для их дальнейшего исправления. Так же тесты выполняются каждый раз при внесении изменения в программе для проверки на корректность работы того, что ранее работало и проходило тестирование.

3.1 Реализация программы

Реализация программы начинается с подготовки проекта. Проект является модульным, поэтому необходимо заранее заложить в его структуру модульность. При проектировании проекта в качестве системы сборки проекта была выбрана программа CMake.Модульность в CMake достигается путём включения в проект других проектов в качестве динамически или статически подключаемой библиотеки. Наиболее эффективным и удобным способом является подключение динамических библиотек, так как для их замены не нужно вносить изменения в другие модули программы, при условии отсутствия изменений в интерфейсе модуля. Разрабатываемое программное обеспечение состоит из четырёх модулей:

— программа – главный модуль;
— редактор – дочерний модуль;
— компилятор – дочерний модуль;
— визуализатор – дочерний модуль.

Структура проекта всех модулей одинакова и будет представлена в следующем виде:

— заголовок – описание названия проекта и минимальных требований;
— инициализация – установка всех флагов и объявление используемых файлов;
— поиск зависимостей;
— установка типа проекта;
— дополнительные параметры.

Основываясь на этом можно воссоздать базовую структуру проекта. Каждый дочерний проект описывается в новых каталогах, которые для удобства являются дочерними каталогами основного проекта. Каждый каталог, включая родительский, будут содержать файлы CMakeLists.txt, в которых будет описана структура проекта в описанном ранее порядке. Содержание файлов CMake проектов, на примере главного проекта, представлен в листинге 3.1.

Листинг 3.1 – Проект CMake

```
1:    cmake_minimum_required(VERSION 3.14.0 FATAL_ERROR)

2:    project(MultilingualShaderDesigner LANGUAGES CXX)

3:    set(CMAKE_INCLUDE_CURRENT_DIR ON)

4:    find_package(Qt5 REQUIRED COMPONENTS
5:          Core
6:          Widgets
7:          Gui
8:    )
9:    set(CMAKE_AUTOMOC ON)
10:   set(CMAKE_AUTORCC ON)
11:   set(CMAKE_AUTOUIC ON)
12:   set(project_ui)
13:   set(CMAKE_CXX_STANDARD 17)
14:   set(CMAKE_CXX_STANDARD_REQUIRED ON)
15:   set(CMAKE_CXX_EXTENSIONS OFF)
16:   set(CMAKE_LIBRARY_OUTPUT_DIRECTORY ${PROJECT_BINARY_DIR})
17:
18:   set(project_sources
19:   )
20:
21:   set( project_resources
22:   )
23:
24:   set( project_ui
25:   )
26:
27:   add_executable(${PROJECT_NAME}
28:         ${project_sources}
29:         ${project_sources_moc}
30:         ${project_headers_wrapped}
31:         ${project_resources}
32:         ${project_ui}
33:   )
34:
35:   add_subdirectory(editor)
36:   add_subdirectory(visualizers)
37:   add_subdirectory(compilers)
38:
39:   target_link_libraries(${PROJECT_NAME}
40:         PUBLIC
41:               editor
42:               visualizers
43:               compilers
44:   )
45:
46:   set_target_properties(${PROJECT_NAME}
47:         PROPERTIES
48:         ARCHIVE_OUTPUT_DIRECTORY ${CMAKE_LIBRARY_OUTPUT_DIRECTORY}/lib
49:         LIBRARY_OUTPUT_DIRECTORY ${CMAKE_LIBRARY_OUTPUT_DIRECTORY}/lib
50:         RUNTIME_OUTPUT_DIRECTORY ${CMAKE_LIBRARY_OUTPUT_DIRECTORY}/bin
51:   )
```

Каждый модуль разрабатываемого программного обеспечения будет разрабатываться в изоляции друг от друга для удобства и увеличения быстродействия. Первым будет реализован

модуль редактирования, с помощью которого будет реализовываться все идеи конечного пользователя.

Компонент редактирования, спроектированный в главе 2, состоит из четырёх основных компонентов:

— контроллер редактора – главный компонент;

— графический редактор – дочерний компонент;

— контроллер переменных – дочерний компонент;

— магазин доступных объектов – дочерний компонент.

Контроллер редактора является главным компонентом модуля редактирования, так как с помощью него осуществляется управление его дочерними компонентами. Контроллер редактора, при создании, должен создать все дочерние компоненты, описанные выше, внедрив и распределив все зависимости, необходимые для корректной работы, и связать их, если это требуется. Кроме того, контроллер должен реализовать свой внешний интерфейс, с помощью которого будет происходить общение с модулем.

Главным пользовательским инструментов в модуле редактирования является графический редактор. Графический редактор представляет из себя окно, внутри которого происходит создание, перемещение, соединение и удаление графических компонентов, которые являются узлами логики в разрабатываемой шейдерной программе. Решить поставленную задачу можно двумя способами: реализация функционала с нуля или использование готового решение. Для ускорения разработки было принято решение выбрать второй вариант, внеся в него все необходимые изменения для получения необходимого результата. Отличным кандидатом является библиотека Qt Node Editor[11], оригинальный исходный код которой можно посмотреть в репозитории её автора. При работе программы появляется необходимость использовать узлы с переменной моделью – структура узла может изменятся во время работы программы в зависимости от текущей ситуации. Для достижения поставленной задачи был внесён ряд изменений в основных классах библиотеки. Такой решение принято в основном из-за отсутствия возможности масштабировать существующую систему для получения желаемого результата из-за некоторых ограничений самой библиотеки.

Первое и немаловажное изменение – внедрение понятия отсутствия типа. Имея возможность сообщить системе, что у места подключения не задан конкретный тип, можно добиться возможности выбирать, соединения каких типов может принимать данный порт узла. Вторым изменение, на котором основано первое, является возможность опроса сообщить пользовательской модели о попытке установки соединения, имеющего определённый тип данных, в один из доступных портов. Система отправляет модели информацию о типе данных

порта, из которого устанавливается соединение, и номер порта, к которому производится подключение. В качестве ответа модель должна вернуть истину, если она готова принять этот тип данных, или лож, если она не готова принимать указанный тип данных в указанном порту. Третье изменение касается возможности модели меняться во время работы программы. Для этого в модель был внедрён сигнал, с которым, в момент создания, связывается узел. При изменении модель сообщает узлу о смене состояния, после чего узел обновляет все данные о модели и перестраивается. Одним из важных этапов обновления модели является процесс уничтожения существующих подключений в случае, если они больше не соответствуют модели данных. Четвёртое изменение касается работу соединений. В класс соединения был внедрён метод тестирования валидности подключения, который, в случае невалидности, сообщает системе о своей невалидности. Система, получив сообщение о невалидности соединения, удаляет это соединение.

На основе исходных и изменённых компонентов реализуется набор типов данных и узлов, работающих с ними. Каждый компонент, в последствии отображаемый на сцене, должен переопределить класс QtNodes::NodeDataModel, который представляет из себя модель данных в паттерне проектирования Model-View-Controller[10].

Одной из функциональных возможностей графического редактора на этапе проектирования является возможность порождения новых узлов, путём их переноса из внешних компонентов в текущий. Для этого был реализован класс EditorGraphicsScene, который реализует возможность принимать данные, переносимые из внешних компонентов. В Qt уже заложена возможность создавать, переносить, и отправлять данные из одного элемента управления в другой. Данная функциональная возможность называется Drug-and-drop[7], а её реализация в классе EditorGraphicsScene представлена в листинге 3.2.

Листинг 3.2 – Реализация Drug-and-drop в классе EditorGraphicsScene

```
1:    void
2:    EditorGraphicsScene::
3:    dragEnterEvent(QGraphicsSceneDragDropEvent* event)
4:    {
5:        if ( event->mimeData()->hasFormat("ShaderNodes/Node") ) {
6:            event->accept();
7:        }
8:        else {
9:            event->ignore();
10:       }
11:   }
12:
13:   void
14:   EditorGraphicsScene::
15:   dragMoveEvent(QGraphicsSceneDragDropEvent* event)
16:   {
17:       if ( event->mimeData()->hasFormat("ShaderNodes/Node") ) {
18:           event->accept();
```

```
19:         }
20:         else {
21:             event->ignore();
22:         }
23:     }
24:
25:     void
26:     EditorGraphicsScene::
27:     dropEvent(QGraphicsSceneDragDropEvent* event)
28:     {
29:         if ( event->mimeData()->hasFormat("ShaderNodes/Node") ) {
30:             auto data = event->mimeData()->data("ShaderNodes/Node");
31:             QDataStream stream( &data, QIODevice::ReadOnly );
32:             QString dataModelName;
33:             stream >> dataModelName;
34:             auto registeredModelsFactories = registry()
35:                       .registeredModelCreators();
36:             auto& node = createNode(
37:                       registeredModelsFactories[dataModelName]());
38:             node.nodeGraphicsObject().setPos(event->scenePos());
39:             nodePlaced(node);
40:             event->accept();
41:         }
42:         else {
43:             event->ignore();
44:         }
45:     }
```

Так же был реализован класс DefaultDataModelRegistry, который автоматически регистрирует запрограммированный набор моделей узлов, которые могут быть установлены в графический редактор. Чтобы дать возможность сторонним компонентам получать только определённый набор зарегистрированных моделей, класс реализует два метода регистрации моделей: основной и дополнительный. Контроллер сначала регистрирует основной набор моделей, после чего ожидает, пока все классы воспользуются установленными данными, после чего регистрирует оставшуюся часть моделей узлов.

В результате получился функционирующий графический редактор, визуальное представление которого представлено на рисунке 3.1.

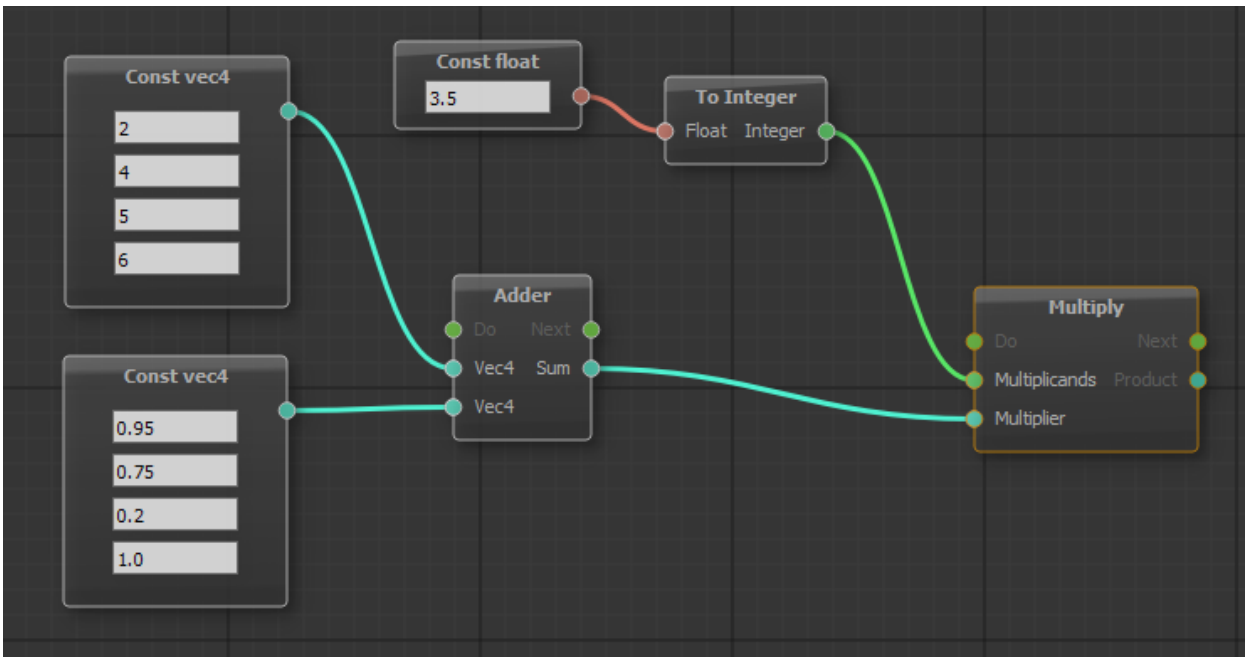

Рисунок 3.1 – Графический редактор

Следующий компонент модуля редактирования – магазин доступных объектов. Данный компонент предназначен для отображения списка доступных пользователю узлов в виде нарисованных узлов. В функции объекта входит перенос узлов из компонента на сцену графического редактора. Для реализации такой возможности необходимо создать новый виджет - NodeWidget, который переопределит метод обработки события клика мыши. Вместо обработки события по умолчанию будет создаваться объект переносимый объект, который затем можно будет перенести на графический редактор, который обработает событие переноса и создаст нужный узел. Так же пользователю необходимо знать, что за виджет ему представлен. Для этого необходимо переопределить метод обработки события рисования. Новый метод будет рисовать изображение узла в центре виджета. Библиотека Qt Node Editor предоставляет класс, который, на основе информации об узле, рисует изображение узла. Этот класс будет использован для рисования изображений на виджете. Сохранение полученного изображения позволит сэкономить вычислительные ресурсы за счёт потребления памяти устройства. Реализация класса NodeWidget представлена в листинге 3.3.

Листинг 3.3 – Реализация класса NodeWidget

```
1:     NodeWidget::NodeWidget( Node& node
2:                           , DataModelRegistry& registry
3:                           , QWidget* parent)
4:         : QWidget(parent)
5:     {
6:         // Определение геометрических параметров
7:         auto geometry = node.nodeGeometry();
8:         geometry.recalculateSize(false);
9:         auto pointDiameter = node.nodeDataModel()
```

```
10:                     ->nodeStyle().ConnectionPointDiameter;
11:       auto twoPointsDiameter = pointDiameter * 2;
12:       auto fourPointsDiameter = pointDiameter * 4;
13:       // Вычисление размеров
14:       QSize size( geometry.width()+fourPointsDiameter
15:                 , geometry.height()+fourPointsDiameter);
16:       // Создание изображения
17:       _nodePixmap = std::make_shared<QPixmap>(size);
18:       _nodePixmap->fill(Qt::transparent);
19:       setFixedSize(size);
20:       // Рисование изображения в буфер
21:       QPainter nodePainter;
22:       nodePainter.begin(_nodePixmap.get());
23:       nodePainter.translate(twoPointsDiameter, twoPointsDiameter);
24:       nodePainter.setRenderHint(QPainter::RenderHint::Antialiasing);
25:       QtNodes::NodePainter::paint(&nodePainter, node, registry, false);
26:       nodePainter.end();
27:       // Установка имени узла, используемого в Drug and drop
28:       _nodeName = node.nodeDataModel()->name();
29:   }
30:   void NodeWidget::paintEvent(QPaintEvent*)
31:   {
32:       // Рисование картинки на виджите
33:       QPainter painter;
34:       painter.begin(this);
35:       painter.setBackgroundMode(Qt::BGMode::TransparentMode);
36:       painter.drawPixmap( _nodePixmap->rect()
37:                         , *_nodePixmap
38:                         , _nodePixmap->rect() );
39:       painter.end();
40:   }
41:   void NodeWidget::mousePressEvent(QMouseEvent*)
42:   {
43:       // Создание перемещаемого объекта
44:       auto drag = new QDrag(this);
45:       // Создание хранилища переносимых данных
46:       auto mimeData = new QMimeData();
47:
48:       // Подготовка данных
49:       QByteArray data;
50:       QDataStream stream( &data, QIODevice::WriteOnly );
51:       stream << _nodeName;
52:
53:       // Запись данных
54:       mimeData->setData("ShaderNodes/Node", data);
55:       drag->setMimeData(mimeData);
56:       drag->setPixmap(_nodePixmap->copy());
57:
58:       // Начало переноса данных
59:       drag->exec();
60:   }
```

Виджет, хранящий набор всех узлов для их вставки в редактор представляет из себя виджет с панелью для скроллинга содержимого и набором дочерних виджетов внутри. На рисунке 3.2 представлен внешний вид магазина узлов.

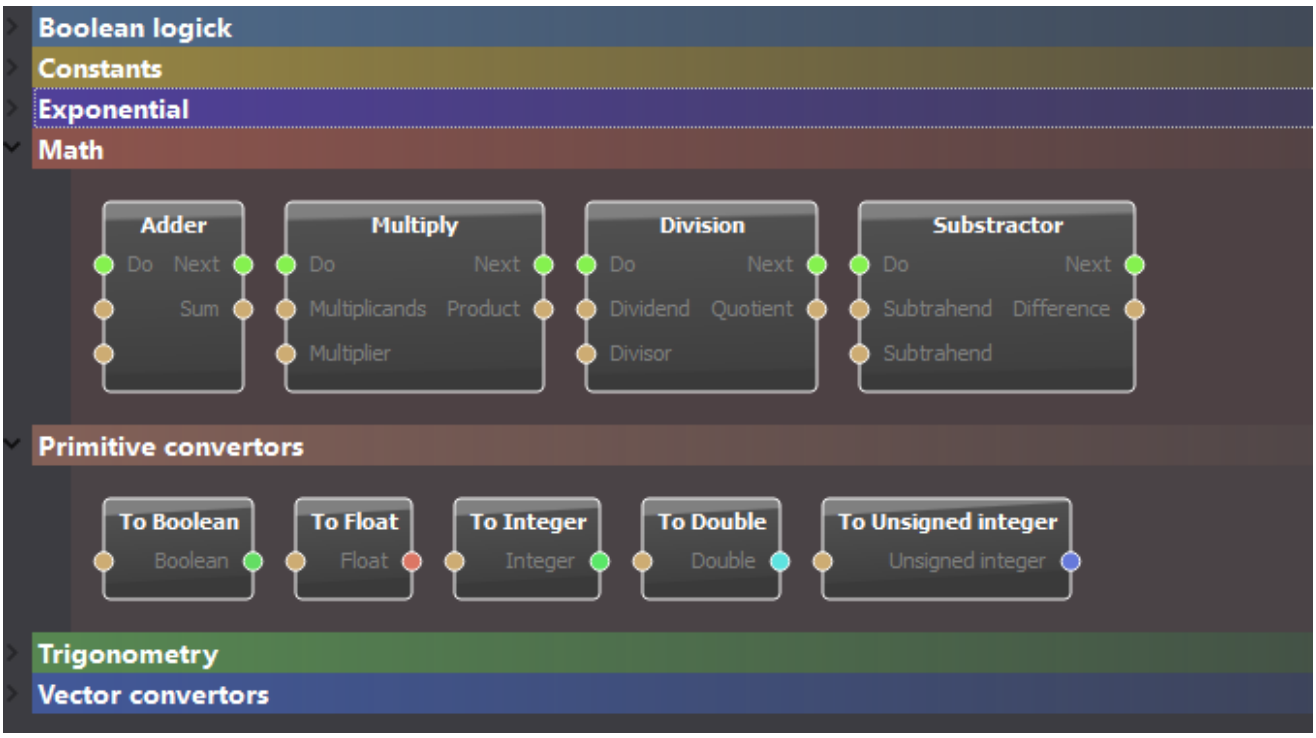

Рисунок 3.2 – Магазин узлов

Последним дочерним компонентом графического редактора является контроллер переменных. Контроллер переменных отвечает за хранение состояния всех пользовательских переменных, а также за управление узлами на графическом редакторе. Дополнительно к контроллеру предоставляется виджет для создания, удаления, отображения и редактирования переменных.

Переменная представляет из себя набор данных, состоящий из имени переменной и её значения. Для реализации переменных был введён класс VariableDataModel, хранящий имя переменной, а также шаблонный класс VariableDataModelTemplate, который предоставляет дополнительное шаблонное поле для хранения значения переменной. С помощью шаблонного класса VariableDataModelTemplate можно воссоздать желаемый тип данных. Для порождения переменных реализован класс VariableDataModelsFactory – порождающий класс-фабрика, отвечающий за создание объектов переменных. Для создания переменной необходимо сообщить конструирующей функции желаемый тип данных переменной. Так же фабрика предоставляет метод, возвращающий список типов данных, поддерживаемых в данный момент.

Для редактирования переменных реализован класс VariablesEditorWidget, который расширяет класс виджета и предоставляет элементы управления для изменения имени переменной и её типа. Так же редактор переменных содержит контейнер для редакторов значений переменных. Для редактирования значений переменных реализован базовый абстрактный класс VariableValueEditor, который предоставляет интерфейс для управления

данными. Базовыми методами являются методы установки и получения объектов класса VariableDataModelTemplate. Все классы-наследники обязаны реализовать метод установки значения по умолчанию. Для получения редакторов значений переменных реализован класс-фабрика VariableValueEditorsFactory, который создаёт нужный объект редактора, в зависимости от переданного ему типа данных переменной.

Для создания новых, удаления старых и отображения списка существующих переменных был реализован класс-виджет VariablesListWidget. Данный класс выполняет следующие функции:

— предоставляет элементы управления для создания и удаления переменных;

— автоматически генерирует новые имена для новых переменных;

— отображает существующие переменные;

— позволяет выбирать переменные в списке;

— позволяет создавать узлы переменных на сцене;

— поддерживает актуальность списка существующих переменных.

Данный класс имеет два режима работы: с возможностью создания и удаления переменных и без неё. Это необходимо для определения типа списка – динамический или статический.

VariablesControllerWidget – класс-виджет, отвечающий за агрегацию всех виджетов, отвечающих за управление переменными, в одном месте. Так же этот класс создаёт связи между контроллером переменных и виджетами управления этими переменными. Внешний вид данного виджета представлен на рисунке 3.3.

Класс VariablesController реализует контроллер переменных. Данный класс предоставляет методы для добавления, переименования и удаления переменных, изменения типа данных переменных, а также сохранение и восстановления списка переменных и их значений. Для хранения информации о переменных класс VariablesController кэширует имена переменных, типы переменных, а также информацию о узлах, добавленных на сцену.

Для управления узлами переменных класс VariablesController напрямую работает с представлением и сценой графического редактора. Данный класс сам создаёт узлы и связывает их с моделями, после чего сохраняет информацию о них у себя в кэше. Это необходимо для двух случаев:

— изменение имени переменной – все узлы переменной будут обновлены;

— удаление переменной – все узлы переменной будут удалены.

На листинге 3.4 представлен исходный код класса VariablesController, выполняем при переименовании и удалении переменных.

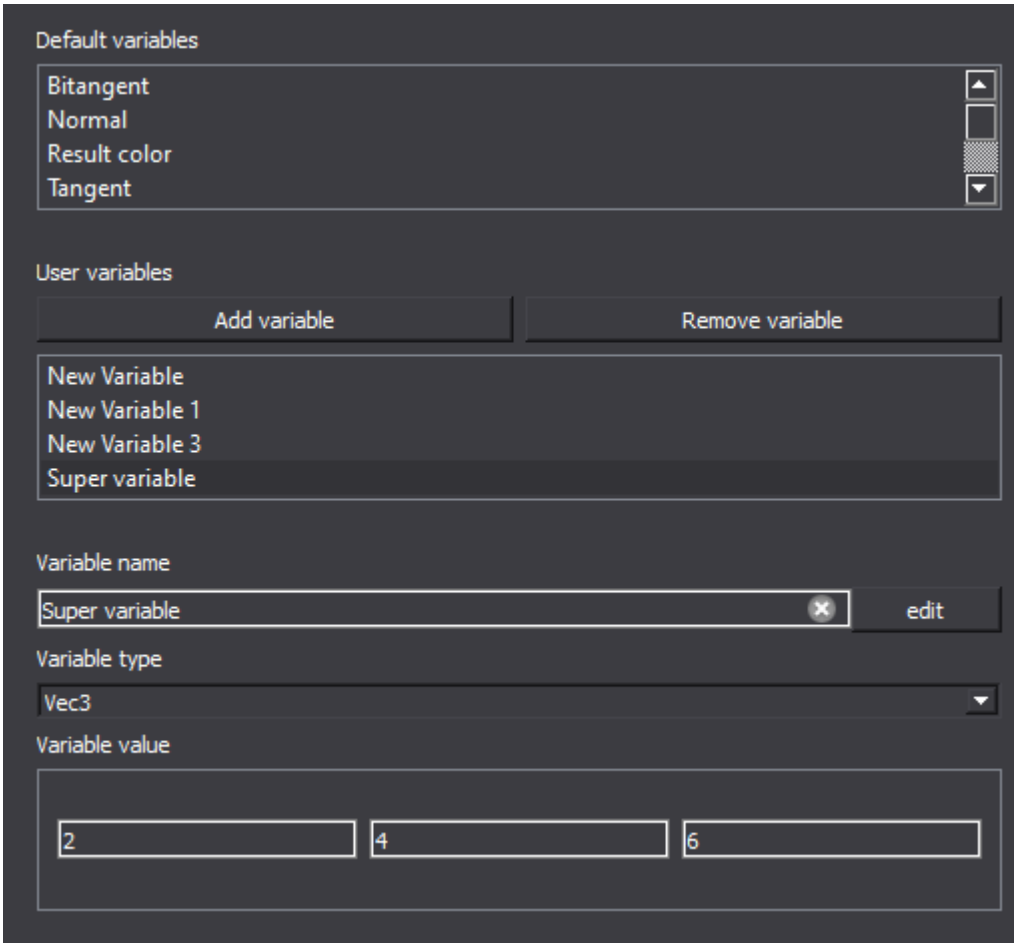

Рисунок 3.3 – Внешний вид виджета VariablesControllerWidget

Листинг 3.4 – Методы переименования и удаления переменных в классе VariablesController

```
1:    void
2:    VariablesController::
3:    renameVariable( const QString& oldName
4:                  , const QString& newName)
5:    {
6:        // Дублирование данных для нового имени
7:        _variablesTypes[newName] = _variablesTypes[oldName];
8:        _variablesNodesIds[newName] = _variablesNodesIds[oldName];
9:
10:       // Обход всех зарегестрированных узлов и установка нового имени
11:       for ( auto id : _variablesNodesIds[newName]) {
12:           _nodesIdAssociate[id] = newName;
13:           static_cast<VariableNode*>
14:                   (_nodesDataModels[id])->setVariableName(newName);
15:       }
16:
17:       // Удаление старых записей
18:       _variablesTypes.remove(oldName);
19:       _variablesNodesIds.remove(oldName);
20:   }
21:
22:   void
23:   VariablesController::
24:   removeVariable( const QString& name )
25:   {
26:       // Удаление информации о типе переменной
```

```
27:         _variablesTypes.remove(name);
28:
29:         // Обход и удаление всех зарегестрированных узлов
30:         auto ids = _variablesNodesIds[name];
31:         for ( auto id : ids) {
32:             auto& node = *(_scene->nodes().at(id));
33:             _scene->removeNode(node);
34:         }
35:
36:         // Удаление информации об Id узла
37:         _variablesNodesIds.remove(name);
38:     }
```

Модуль компилятора является вторым самым важным модулем. Его работа заключается в обработке полученных данных и преобразовании их в исходный код шейдерных программ. Для достижения этой цели в модуль компилятора входят четыре класса:

— CompilerController – основной класс модуля, через которое ведётся управления всеми процессами, связанными с компиляцией данных;

— CompilerWrapper – управляет процессом компиляции. Обеспечивает обмен информацией между потоками приложения и потоком компиляции;

— ICompiler – интерфейс класса-компилятора. Этот интерфейс должен быть реализован всеми классами, которые будут выполнять компиляцию.

— DefaultGlslCompiler – базовая реализация интерфейса ICompiler, встроенная в модуль компиляции.

Класс CompilerController реализует интерфейс обмена информации между внутренними компонентами модуля и для обмена информацией между модулями. Для обмена информацией с объектом класса CompilerController доступны следующие методы:

— compile – сообщает контроллеру, чтобы он скомпилировал исходный код с помощью компилятора с заданным именем;

— stopCompile – сообщает контроллеру, чтобы он остановил процесс компиляции. Компиляторы могут не поддерживать данную возможность.

— compileComplete – сообщает пользователю объекта класса, что компиляция была успешно завершена;

— compileChangeState – сообщает пользователю объекта класса, что текущее состояние процесса компиляции изменилось;

— compileError – сообщает пользователю объекта класса, что компиляция завершилась ошибкой.

При построении объекта класса CompilerController создаётся объект потока QThread и объект класса CompilerWrapper, который перемещается в новый поток. После переноса объекта

класса CompilerWrapper в другой поток, контроллер настраивает связи между собой и обёрткой компилятора для организации общения.

Класс CompilerWrapper представляет из себя обёртку для компилятора. Пользователи объектов класса обязаны передать обёртке адрес компилятора и адрес исходных данных для последующей компиляции. При передаче адреса компилятора обёртка устанавливает связь с этим компилятором для последующей работы с ним. В основном класс CompilerWrapper реализует тот же интерфейс, что и CompilerController, но есть ещё один метод-сигнал, который используется непосредственно во время процесса компиляции: nextStep. Данный сигнал используется для создания цикла событий компиляции внутри потока компиляции. Объект класса CompilerWrapper особым образом связывается сам с собой через сигнал nextStep и внутренний слот onNextStep посредством очереди событий. Для достижения цикличности с возможностью безопасно прервать рабочий поток, CompilerWrapper сообщает компилятору, что он должен выполнить следующий шаг в процессе компиляции, после чего отправляет сам себе отправляет сообщение в очередь событий. Таким образом процесс компиляции выполняется пошагово в порядке очереди, благодаря чему в любой момент потоку можно послать сигнал о завершении компиляции, который будет обработан и приведён в исполнение. Код метода onNextStep, с помощью которого реализуется цикл компиляции, представлен в листинге 3.5.

Листинг 3.5 – Реализация метода onNextStep

```
1:    void
2:    CompilerWrapper::
3:    onNextStep()
4:    {
5:        // Выполнение шага компиляции
6:        _compiler->nextStep();
7:
8:        // Сообщение о намерение выполнить ещё один шаг компиляции
9:        if ( _compiling ) {
10:           emit nextStep();
11:       }
12:   }
```

Класс ICompiler является абстрактным классом-интерфейсом компилятора, который должен быть реализован в классах-потомках для реализации возможности общаться с компилятором через специальный интерфейс. Код класса ICompiler представлен в листинге 3.6.

Листинг 3.6 – Класс-интерфейс ICompiler

```
1:    class ICompiler : public QObject
2:    {
3:        Q_OBJECT
4:
5:    public:
6:        virtual ~ICompiler() {}
7:
```

```
8:          virtual void
9:          compile(std::shared_ptr<QJsonObject> sources) = 0;
10:
11:     public slots:
12:         virtual void
13:         nextStep() = 0;
14:
15:     signals:
16:         void
17:         compileComplete(const QString& fragmentShaderText);
18:
19:         void
20:         compileChangeState( int state
21:                           , const QString& stateText );
22:
23:         void
24:         compileError(const QString& what);
25:
26:     };
```

Класс DefaultGlslCompiler является базовой реализацией интерфейса ICompiler и входит в состав модуля компилирования. Данный класс превращает исходный код разработанной шейдерной логики в исходный код фрагментного шейдера на языке программирования OpenGL Shading Language[9]. Самым сложным этапом является этап взвешивания узлов. Для его реализации необходимо воспользоваться схемой алгоритма взвешивания узлов, описанной в главе 2. Код самого сложного этапа, связанного с взвешиванием узлов, приведён в листинге 3.7.

Листинг 3.7 – Реализация алгоритма взвешивания узлов

```
1:      auto calcWeight = [&](QUuid uuid) {
2:          auto recursiveCalcWeight = [&](QUuid uuid, auto subCalc) -> int {
3:              auto nodeWeight = _nodesWeights.find(uuid);
4:              if ( nodeWeight != _nodesWeights.end() ) {
5:                  return *nodeWeight;
6:              }
7:
8:              const auto& nodeDependecis = _dependencis[uuid];
9:              if ( nodeDependecis.size() == 0 ) {
10:                 _nodesWeights[uuid] = 0;
11:                 return 0;
12:             }
13:
14:             int weight = nodeDependecis.size();
15:             for ( auto& id : nodeDependecis ) {
16:                 weight += subCalc(id, subCalc);
17:             }
18:             _nodesWeights[uuid] = weight;
19:             return weight;
20:         };
21:
22:         recursiveCalcWeight(uuid, recursiveCalcWeight);
23:     };
24:
25:     for ( auto id : _models.keys() ) {
26:         calcWeight(id);
27:     }
```

Следующим модулем, который будет реализован, является модуль графических сцен. Каждая графическая сцена должна реализовывать интерфейс сцены. Каждая сцена должна представлять из себя виджет, в котором происходит рендеринг итогового изображения с последующим выводом на экран. Сама же сцена будет управляться виджетом-контейнером, который будет отвечать за создание виджета целевой сцены, а также за управление этой сценой в рамках интерфейса. Первым виджетом сцены будет виджет, который для рендеринга изображения будет использовать OpenGL API. Одно из решений, которое в него заложено – использование динамической последовательности вызова функций. Это достигается за счёт построения вектора из функторов, которые будут вызываться по порядку при каждом рендеринге изображения. Набор функций выбирается в зависимости от конфигурации, которая будет установлена в данном виджите. На рисунке 3.4 представлено изображение разработанного графического элемента визуализации сцены.

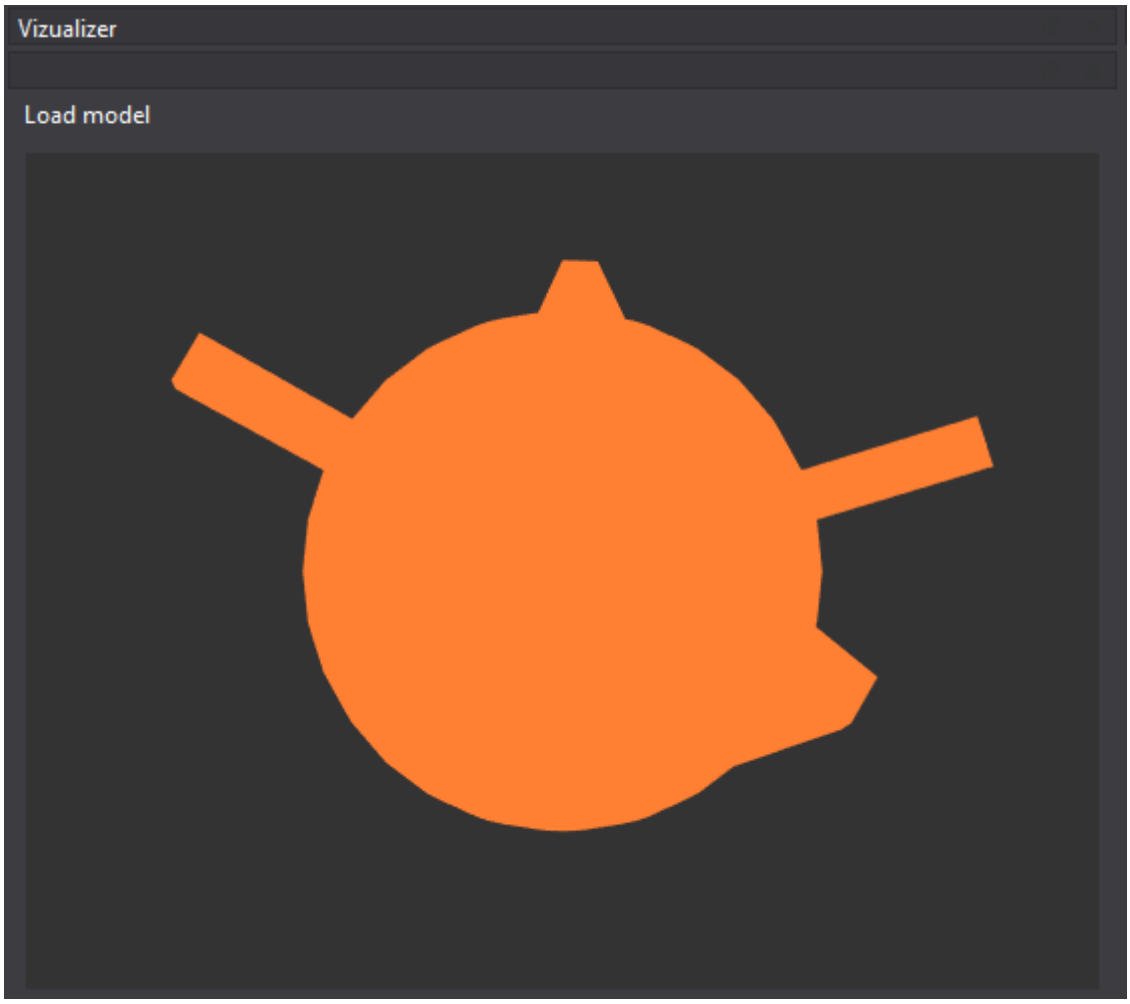

Рисунок 3.4 – Окно визуализации сцены

3.2 Разработка программной документации

Для данного дипломного проекта была разработана следующая документация, приведенная в приложениях:

— Методика проведения испытаний (см. приложение Б);

— описание программы (см. приложение В);

— руководство пользователя (см. приложение Г).

Описание программы представляет собой документ, предназначенный для указания области применения программы, функционального состава, круга решаемых задач и технических и программных средств, необходимых для нормального функционирования. Также должна быть приведена логическая структура, описан способ начала работы. Описание программной документации составляется в соответствии с ГОСТ 19.402-2000 [2].

Руководство оператора разрабатывается для описания работы с приложением. Руководство выполняется по ГОСТ 19.505-79 ЕСПД [3]. При разработке руководства оператора приводятся следующие описания:

— подробное описание использования программы;

— описания возможных сообщений при работе с программой с подробным и понятным описанием их значения.

Программа и методика испытаний используется для реализации полноценного тестирования разработанного программного продукта, описывает совокупность функций, подлежащих проверке на корректность выполнения.

Данный документ позволяет провести тестирование программы, не участвуя в разработке самого продукта, и на основании полученных результатов сделать вывод о работоспособности и необходимости проведения доработки перед началом внедрения и эксплуатации. Программа и методика испытаний оформляется в соответствии ГОСТ 19.301-2000 [1].

3.3 Тестирование программы

Для потдверждения того, что программа работает правильно, её необходимо протестировать. Тестирование программы подтвердит, что программный продукт удовлетворяет требованиям, указанным в техническом задании.

Тестирование будет разделено на 3 этапа, в которых будут тестироваться модули:

— модуль редактирования;

— модуль компилирования;

— модуль визуализации.

План испытаний и параметры технических средств, для проведения тестирования, приведены в документе «Программа и методика испытаний», представленном в приложении Г. На основании этого документа было проведено тестирование.

Результаты тестирования модуля редактирования представлены в таблице 3.1.

Таблица 3.1 – Результаты тестирования модуля редактирования

Тестовый случай	**№ тестов**	**№ пройденых**	**№ не пройденных**	**% выполненных**
Случай 1	5	5	0	100
Случай 2	5	5	0	100
Случай 3	5	5	0	100
Случай 4	5	5	0	100
Случай 5	5	5	0	100
Случай 6	5	5	0	100
Итог	**30**	**30**	**0**	**100**

Результаты тестирования модуля компилирования представлены в таблице 3.2.

Таблица 3.2 – Результаты тестирования модуля компилирования

Тестовый случай	**№ тестов**	**№ пройденых**	**№ не пройденных**	**% выполненных**
Случай 1	10	10	0	100
Случай 2	10	10	0	100
Случай 3	10	10	0	100
Случай 4	10	10	0	100
Итого	**40**	**40**	**0**	**100**

Результаты тестирования модуля визуализации представлены в таблице 3.3.

Таблица 3.3 – Результаты тестирования модуля визуализации

Тестовый случай	**№ тестов**	**№ пройденых**	**№ не пройденных**	**% выполненных**
Случай 1	5	5	0	100
Случай 2	5	5	0	100
Случай 3	5	5	0	100
Случай 4	5	5	0	100
Итого	**20**	**20**	**0**	**100**

На основе результатов рестирования можно составить сводный отчёт с результатами тестирования. Отчёт о результатах тестирование представлен в таблице 3.4.

Таблица 3.4 – Сводный отчёт с результатами тестирования всех модулей

Тестовый набор	**№ тестов**	**№ пройденых**	**№ не пройденных**	**% выполненных**
Модуль редактирования	30	30	0	100
Случай 2	40	40	0	100
Случай 3	20	20	0	100
Итого	**90**	**90**	**0**	**100**

В результате тестирования некорректного поведения выявлено не было. Разработанное программное обеспечение прошло все тесты, продемонстрировало стабильную и устойчивую работоспособность. Результат данного тестирования гарантирует пользователю, что процесс его взаимодействия с программным обеспечением не приведёт к неожиданным результатам, вызванным сбоями или ошибками в разработанном программном обеспечение.

4 ЭКОНОМИЧЕСКАЯ ЧАСТЬ

4.1 Обоснование необходимости выведения продукта на рынок

Цель экономического раздела дипломного проекта – рассчитать затраты на разработку программного обеспечения и определить экономическую эффективность от его внедрения.

Графическое приложение для разработки шейдерных программ с использованием визуального программирования позиционируется как универсальный инструмент для разработки шейдерных программ, используя для этого один из популярных подходов к визуальному программированию. Наличие множества целевых платформ, разработка шейдеров для которых является индивидуальной задачей, препятствует быстрой разработке аналогичных решений для альтернативных платформ. Разрабатываемое графическое приложение для разработки шейдерных программ с использованием визуального программирования призвано решить эту проблему, предоставив инструмент для разработки логики программы в интерактивном графическом режиме, а также возможность экспорта готового решения на целевые платформы.

Целевой аудиторией готового программного обеспечения являются специалисты, тем или иным образом связанные с компьютерной графикой, учащиеся, студенты и другие. Для профессионалов графическое приложение для разработки шейдерных программ с использованием визуального программирования предоставит мощный и гибкий инструмент для разработки шейдерных программ, с последующей их реализации как в небольших, так и в крупномасштабных проектах. Кроме того, разрабатываемое программное обеспечение позволит быстро перенести наработки на другие целевые платформы без необходимости вносить изменения. Для учащихся школ, университетов и других учреждений образования данное программное обеспечение позволит в графическом интерактивном режиме развивать логическое и креативное мышления, а также экспериментально подтверждать свои знания математики, линейной алгебры, физики и другое.

Вывод разрабатываемого программного обеспечения на рынок позволит потребителям повысить производительность труда при коммерческой и не коммерческой разработке, непосредственно связанной с компьютерной графикой. Увеличение эффективности разработки экономически выгодно для потребителя. Благодаря своей уникальности, эффективности и своему удобству разрабатываемое программное обеспечение способно занять и укрепить своё место на рынке.

4.2 Структура (этапы) работ по созданию программного обеспечения

Жизненный цикл разработки целевого программного обеспечения может быть разбит на следующие этапы:

— анализ исходных данных – сбор информации о предмете и объекте исследования;

— постановка задач – постановка задач разработки программного обеспечения и составление технического задания;

— проектирование – разработка архитектуры программного обеспечения;

— реализация – программирование логики работы программного обеспечения и реализация встраиваемых ресурсов (иконки, изображения, файлы, прочее);

— тестирование – тестирование готового программного обеспечения;

— документирование – оформление руководства пользователя.

Суммарное время, выделенное на разработку программного обеспечения – 84 дня.

Анализ исходных данных является первым этапом разработки целевого программного обеспечения. Во время этого этапа собираются данные об объекте и предмете исследования, которые затем будут использованы при принятие архитектурных и дизайнерских решений. На анализ исходных данных выделено 3 дня (3.57% общего времени).

На этапе «постановка задач» выполняется агрегирование полученных ранее данных в группы, на основе которых выполняется постановка задач проектирования, выбора инструментов для разработки и подготовка рабочей среды к работе. На данный этап выделено 4 дня (4.76% общего времени).

Этап «проектирование» является одним из самых продолжительных этапов, так как на этом этапе разрабатывается дизайн и архитектура программного обеспечения, способные выполнять поставленные задачи. Архитектурные решения, принятые на данном этапе, не могут быть изменены в будущем, поэтому необходимо уделить достаточное количество времени на разработку, проверку и доработку архитектуры разрабатываемого программного обеспечения. На данный этап выделяется 21 день (25% общего времени).

На этапе «тестирование» выполняется проверка реализованного функционала на наличие ошибок в его работе, которые затем оперативно исправляются. В результате должно быть получено исправно работающее программное обеспечение, выполняющее поставленные задачи. На данный этап выделено 2 дня (2.38% общего времени).

Этап «разработка» является самым продолжительным этапом. На данном этапе реализуются архитектура и дизайн разрабатываемого программного обеспечения. Кроме реализации одновременно выполняется оптимизация готового функционала с целью повышения производительности: выбираются оптимальные или оптимизируются

существующие алгоритмы и структуры данных. На данный этап выделено 49 дней (58.33% общего времени).

На этапе «документирование» выполняется подготовка руководства использования разработанного программного обеспечения, в котором максимально детально описываются способы взаимодействия, ограничения, важные нюансы и прочее. Результатом проделанной работы является полностью структурированный и понятный конечному пользователю документ с описание возможных взаимодействий с разрабатываемым программным обеспечением. На данный этап выделено 5 дней (5.95% общего времени).

Диаграмма распределения времени работы при разработке программного обеспечения представлена на рисунке 4.1.

Рисунок 4.1 – Диаграмма распределения времени работы при разработке программного обеспечения

Диаграмма на рисунке 4.1 показывает, что больше половины жизненного цикла разрабатываемого программного обеспечения уходит на реализацию этого программного обеспечения, в то время как проектирование занимает приблизительно четверть от общего времени жизненного цикла. Этапы, идущие перед проектированием и после разработки, занимают шестую часть общего времени разработки разрабатываемого программного обеспечения.

4.3 Составление сметы затрат на разработку программного обеспечения

Стоимостная оценка ПО предполагает составление сметы затрат, которая в денежном выражении включает следующие статьи расходов:

— материалы и комплектующие (М);

— электроэнергия (Э);

— основная заработная плата разработчиков (Зо);

— дополнительная заработная плата разработчиков (Зд);

— отчисления на социальные нужды (Осн);

— амортизация основных средств и нематериальных активов (А);

— расходы на спецоборудование (Рс);

— прочие прямые расходы (Пз).

Расходы по статье «Материалы и комплектующие» (М) отражают расходы на магнитные носители, бумагу, красящие ленты и другие материалы, необходимые для разработки программного обеспечения. На статью «материалы» входят затраты на материалы и принадлежности, необходимые для проведения научно-исследовательской работы. Затраты определяются по действующим отпускным ценам. Стоимость материалов представлена в таблице 4.1

Таблица 4.1 – Стоимость материалов

Наименование	Количество	Стоимость	Суммарная стоимость
Пачка бумаги формата А4 (500 листов)	1	9	9
Комплект чернил для принтера (4 цвета, 75мл)	1	34	34
Компакт-диск CD RW	1	1.5	1.5

Общая стоимость всех материалов (М), представленных в таблице 4.1, составляет 44.5 белорусских рублей

Затраты на электроэнергию находятся исходя из продолжительности периода разработки программного обеспечения, количества кВт/ч, затраченных на проектирование программного обеспечения и тарифа за 1 кВт/ч.

Базовый тариф для прочих потребителей с 01.01.2019 г. (при соответствии курса белорусского рубля к доллару США 2.159:1) составляет 0,31990 руб. за 1 кВт/ч [5].

При изменении курса доллара США тариф индексируется по формуле:

$$T_н = T_б \times (0.24 + \frac{0.76 \times К_н}{К_б}), \quad (4.1)$$

где Тн – тариф на электрическую энергию, подлежащий применению на день оформления платежных документов и день оплаты потребителем за потребленную электрическую энергию;

Тб – тариф на электрическую энергию, установленный декларацией;

Кн – значение курса белорусского рубля по отношению к доллару США, установленного Национальным банком РБ, на день оформления платежных документов и день оплаты за потребленную электрическую энергию;

Кб – значение курса белорусского рубля по отношению к доллару США, установленного Национальным банком РБ, на дату установления тарифа на электрическую энергию.

Рассчитав по формуле (4.1), получен следующий результат:

$$T_н = 0.31990 \times (0.24 + \frac{0.76 \times 2.096}{2.159}) = 0.3128 \text{ (рублей).}$$

Затраты на электроэнергию определяются по формуле:

$$Э = К_э \times T_н \times t_{ПС}, \quad (4.2)$$

где Кэ – стоимость 1 кВт/ч. Базовый тариф для прочих юридических лиц и индивидуальных предпринимателей с 01.01.2019 г. составляет 0.32 рублей за 1 кВт/ч.;

tпс – период разработки программы, месяцев. Определяется в соответствие с общим временем на разработку программного обеспечения из пункта 4.2 и составляет 3 месяца.

По формуле (4.2), затраты на электроэнергию составляют:

$$Э = 0.3128 \times 45 \times 3 = 42.228 \text{ (рублей).}$$

Основная заработная плата исполнителей работ рассчитывается по формуле:

$$З_о = ЗП_{ср} \times n \times t_{мес} \quad (4.3)$$

где ЗПср – средняя заработная плата работников РБ в сфере информационных технологий и в области информационного обслуживания за месяц, равная 4231.7 рублей по данным Национального статистического комитета РБ;

n – количество исполнителей, занятых разработкой конкретного программного обеспечения;

tмес – период времени, затраченный на разработку программного обеспечения.

У данного проекта один исполнитель, а время выполнения данного проекта указано в пункте 4.2, тогда основная заработная плата исполнителя проекта, рассчитанная по формуле (4.3), будет следующей:

$$З_о = 4321.7 \times 1 \times 3 = 12965.1 \text{ (рублей).}$$

Дополнительная заработная плата исполнителей работ определяется по нормативу в процентах к основной заработной плате:

$$З_д = З_о \times \frac{Н_д}{100\%} \quad (4.4)$$

где Нд – норматив дополнительной заработной платы (принимается в размере 15%).

Дополнительная заработная плата исполнителей проекта, рассчитанная по формуле (4.4), следующая:

$$З_д = 12965.1 \times \frac{15\%}{100\%} = 1944.77 \text{ (рублей).}$$

К затратам на социальные нужды относят отчисления в фонд социальной защиты населения (Нсз – 34 %) и отчисления на обязательное страхование от несчастных случаев (Нстр – 0,3 %).

Отчисления на социальные нужды определяются в соответствии с действующими законодательными актами по нормативу в процентном отношении к фонду основной и дополнительной зарплаты исполнителей, определенной по нормативу, установленному в целом по организации:

$$З_{сз} = \frac{(З_д + З_о) \times (Н_{сз} + Н_{стр})}{100\%} \quad (4.5)$$

Рассчитанные по формуле (4.5), затраты на социальные нужды составляют:

$$З_{сз} = \frac{(1944.77 + 12965.1) \times (34\% + 0.3\%)}{100\%} = 5114.09 \text{ (рублей).}$$

Затраты по статье «Амортизация основных средств и нематериальных активов», рассчитываются одним из нелинейных методов начисления амортизации.

Амортизация начисляется на все основные средства и нематериальные активы, находящиеся на балансе предприятия и отраслей промышленности, независимо от характера их участия в производственном процессе. При разработке программного обеспечения использовался персональный компьютер, общая стоимость которого, с учётом программного обеспечения, составляет 1400 рублей.

Норма амортизации – это установленный размер амортизационных отчислений на полное восстановление, выраженное в %. Норма амортизации устанавливается на основе экономически целесообразного срока службы и должна обеспечить возмещение износа основных средств к моменту возможного их морального и физического износа и создать экономическую основу для замены.

Норма амортизации для нелинейного способа начисления вычисляется по формуле:

$$H_a = \frac{100\%}{T_н}, \quad (4.6)$$

где На – норма амортизации;

Тн – нормативный срок службы.

Нормативные сроки службы машин и оборудования составляют 5 лет, следовательно, норма амортизации, рассчитанная по формуле (4.6) равна:

$$T_н = \frac{100\%}{5} = 20\%.$$

Амортизационные отчисления(А) за период разработки в три месяца, рассчитанные по формуле (4.6) равны:

$$A = \frac{3}{12} \times 1400 \times 0.2 = 70 \text{ (рублей).}$$

Статья «Прочие прямые расходы» (Пз) на конкретное программное обеспечение включает затраты: на оплату услуг связи, Интернета, транспортные расходы, канцтовары, приобретение и подготовку специальной научно-технической информации и специальной литературы. Для разработки данного программного обеспечения необходимо:

— интернет соединение на период разработки – 110.7(рублей);

— оплата транспортных услуг – 24.09(рублей).

Общая стоимость прочих расходов составляет 134.79(рублей).

Общая сумма расходов по смете (плановая себестоимость) (С) на ПО рассчитывается по формуле:

$$C = M + Э + З_о + З_д + О_{сн} + A + P_c + П_з \quad (4.7)$$

Результат расчётов общей суммы расходов по формуле (4.7) приведён в таблице 4.2.

Таблица 4.2 – Расчет плановой себестоимости программного обеспечения

Статья затрат	Затраты, рублей
Материалы и комплектующие (М)	44.5
Электроэнергия (Э)	42.228
Основная заработная плата разработчиков (Зо)	12965.1
Дополнительная заработная плата разработчиков (Зд)	1944.77
Отчисления на социальные нужды (Осн)	5114.09
Амортизация основных средств и нематериальных активов (А)	70
Расходы на спецоборудование (Рс)	0

Прочие прямые расходы (Пз)	134.79
Общая сумма расходов по смете (плановая себестоимость С)	20315.478

Визуальное представление таблицы 4.2 представлено в виде диаграммы на рисунке 4.2.

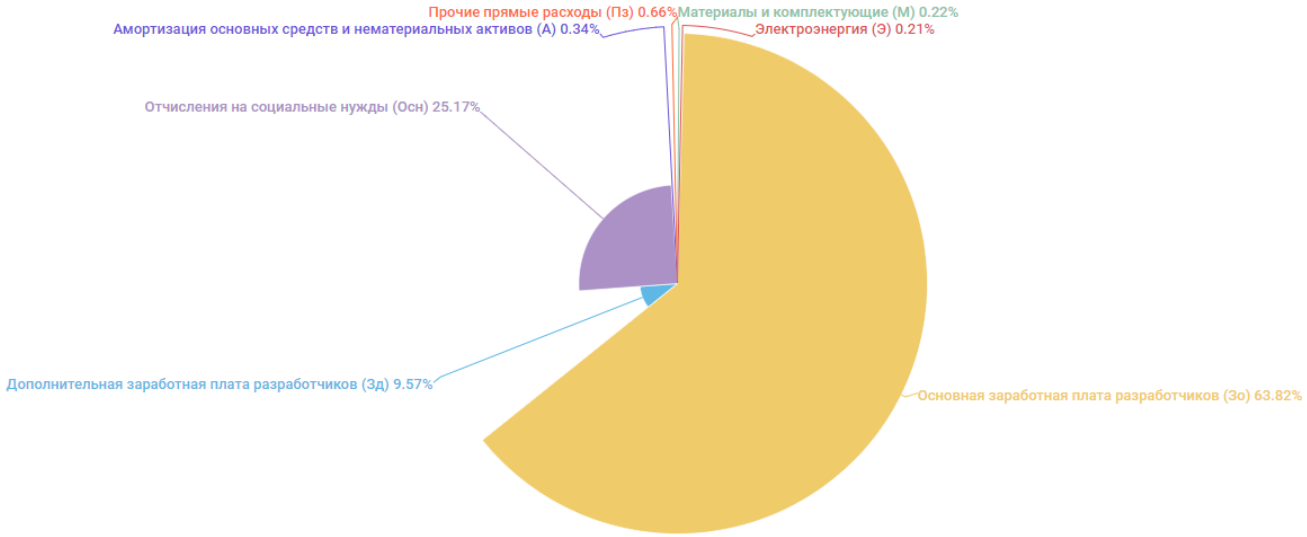

Рисунок 4.2 - Структура затрат на разработку программного обеспечения

4.4 Расчет экономического эффекта разработчика и пользователя (заказчика) программного обеспечения

4.4.1 Экономический эффект у разработчика программного обеспечения

Отпускная цена продукции формируется исходя из плановой себестоимости производства продукции, всех видов установленных налогов и прибыли, а также качества, потребительских свойств продукции и конъюнктуры рынка.

С учетом действующих в республике нормативных документов отпускная цена на продукцию предприятия рассчитывается по формуле:

$$ОЦ = С + П, \quad (4.8)$$

где ОЦ – отпускная цена изготовителя, рублей;

С – плановая себестоимость, рублей;

П – прибыль, рублей.

Прибыль закладывается в цену по нормативу рентабельности (устанавливается самостоятельно), расчет производится по следующей формуле:

$$П = R \times С, \quad (4.9)$$

где R – норматив рентабельности (например, если рентабельность 20 %, то при расчете переводим в коэффициент и получаем 0,2);

С – плановая себестоимость, руб.

Отпускная цена изготовителя с налогом на добавленную стоимость:

$$НДС = (С + П) \times \frac{ставкаНДС(\%)}{100\%}, \quad (4.10)$$

где С – плановая себестоимость, рублей;

П – прибыль, рублей;

Ставка НДС = 20%.

Прогнозируемая отпускная цена на программное обеспечение с НДС:

$$ОЦсНДС = С + П + НДС. \quad (4.11)$$

Прибыль, рассчитанная по формуле (4.9) составляет:

$$П = R \times С = 0.2 \times 20315.478 = 4063.0956 \text{ (рублей)}.$$

Оценочная стоимость с учётом НДС, рассчитанная по формуле (4.11) равна:

$$ОЦсНДС = 20315.478 + 4063.0956 + (20315.478 + 4063.0956) \times 0.2 = 29254.2883 \text{ (рублей)}.$$

Таким образом, разработчик программного обеспечения может продать заказчику программное обеспечение по рассчитанной цене, что покроет затраты и обеспечит прибыль за разработку проекта.

4.4.2 Экономический эффект от использования программного обеспечения у пользователя (заказчика)

При разработке программного обеспечения много внимания уделяется времени разработки программного обеспечения, т.к. от времени напрямую зависят качество и объём разработки. Разработанное программное обеспечение ускоряет разработку шейдерных программ, а также гарантирует минимальное время на перенос логики с одной платформы на другую. Таким образом, уменьшив время на разработку и минимизировав время переноса с логики с одной платформы на другую платформу, уменьшается время и стоимость разработки программного обеспечения, использующего данную логику.

4.5 Вывод по экономической части

Разработанное программное обеспечение является экономически выгодным решением, так как с помощью данного программного обеспечения достигается ускорение процессов разработки целевых продуктов, что повышает прибыль потребителя.

Аналогичные решения не способны предоставить функционал, заложенный в разработанное программное обеспечение, что является стимулирующим фактором при принятии решения о приобретении данного программного обеспечения.

ЗАКЛЮЧЕНИЕ

В результате реализации дипломного проекта «Графическое приложение для разработки шейдерных программ с использованием визуального программирования» была достигнута цель дипломного проекта – была реализована программа, функциональные и графические особенности которой позволят упростить и ускорить процесс разработки шейдеров для разных API.

Для разработки программного продукта был выбран язык программирования C++ совместно с Qt Framework для разработки графической части программного продукта, а также для использования дополнительных возможностей Qt.

В процессе разработки были поставлены и решены следующие задачи:

— спроектирована модульная система;

— каждый функциональный модуль является независимой единицей;

— реализован механизм построения логики шейдерных прогрмм;

— реализован механизм управления пользовательскими переменными;

— реализован удобный способ добавления новых узлов в графический редактор;

— реализован компилятор, способного представить представить разработанную логику шейдера в виде исходного текста шейдерной программы;

— реализован графический интерфейс для визуализации

Разработанная программа может использоваться для быстрой и удобной разработки шейдерных программ, а благодаря своей архитектуре, она легко масштабируема, что позволит ей существовать на рынке достаточно долгое время.

СПИСОК ИСПОЛЬЗОВАНЫХ ИСТОЧНИКОВ

1 ГОСТ 19.301-2000 ЕСПД. Программа и методика испытаний. Требования к содержанию, оформлению и контролю качества. – Взамен ГОСТ 19.301-79; введ. 2001-09-01 – М.: Издательство стандартов, 2001. – 16 с.

2 ГОСТ 19.402-2000 ЕСПД. Описание программы. Требования к содержанию, оформлению и контролю качества. – Взамен ГОСТ 19.402-78; введ. 2001-09-01 – М.: Издательство стандартов, 2001. – 20 с.

3 ГОСТ 19.505-79 ЕСПД. Руководство оператора. Требования к содержанию и оформлению. – Введ. 1980-01-01, с изм. №1 – Минск: Межгос. совет по стандартизации, метрологии и сертификации, 1987. – 4 с.

4 Графический конвейер [Электронный ресурс] // SavePearlHarbor, ещё одна копия хабора. – Электронные данные. – Режим доступа: https://savepearlharbor.com/?p=164065. – Дата доступа: 26.03.2019.

5 Действующие тарифы на электрическую энергию для юридических лиц и индивидуальных предпринимателей в Республике Беларусь [Электронный ресурс] // Министерство энергетики Республики Беларусь. – Электронные данные. – Режим доступа: http://minenergo.gov.by/wp-content/uploads/tarif-elektro180219.pdf. – Дата доступа: 27.05.2019.

6 Растеризация [Электронный ресурс] // Википедия, свободная энциклопедия. – Электронные данные. – Режим доступа: https://ru.wikipedia.org/wiki/Растеризация. – Дата доступа: 26.03.2019.

7 Шейдеры [Электронный ресурс] // Википедия, свободная энциклопедия. – Электронные данные. – Режим доступа: https://ru.wikipedia.org/wiki/Шейдер. – Дата доступа: 26.03.2019.

8 Drug and drop [Электронный ресурс] // Qt Documentation. – Электронные данные. – Режим доступа: https://doc.qt.io/qt-5/dnd.html. – Дата доступа: 31.03.2019.

9 OpenGL Shading Language [Электронный ресурс] // Википедия, свободная энциклопедия. – Электронные данные. – Режим доступа: https://ru.wikipedia.org/wiki/OpenGL_Shading_Language. – Дата доступа: 01.05.2019

10 Model-View-Controller [Электронный ресурс] // Википедия, свободная энциклопедия. – Электронные данные. – Режим доступа: https://ru.wikipedia.org/wiki/Model-View-Controller. – Дата доступа: 30.03.2019.

11 NodeEditor [Электронный ресурс] // Github. – Электронные данные. – Режим доступа: https://github.com/paceholder/nodeeditor. Дата доступа: 18.04.2019.