DOI: 10.24412/2413-2527-2023-436-34-41

Разработка и практическое применение модели интеграции оператора импликации и перегруженного контейнера «Вектор» на языке программирования C++

А. А. Нельин, к.и.н. А. В. Забродин
Петербургский государственный университет путей сообщения Императора Александра I
Санкт-Петербург, Россия
puppi2016@mail.ru, teach-case@yandex.ru

Аннотация. **Рассматриваются теоретические и практические аспекты, необходимые для реализации новой интеграции из ключевых компонентов — оператора импликации и шаблонного контейнера «Вектор». Эти компоненты будут задействованы при разработке игры на графическом движке OpenGL с использованием дополнительной библиотеки GLUT и самостоятельно написанных шейдеров на C-подобном языке GLSL. Данная реализация может послужить источником вдохновения для программистов, побуждая их к активному развитию будущих идей. Практические разделы статьи включают тестовые примеры, демонстрирующие правильную работу моделей интеграции.**

Ключевые слова: **язык программирования C++, импликация, перегрузка операторов, вектор, выделение памяти, классы, интеграция, механика, OpenGL, шейдеры.**

Введение

C++ является одним из самых популярных языков программирования среди разработчиков. Этот язык унаследовал множество особенностей языка C [1]. Одним из ключевых аспектов в C++ является перегрузка операторов (+, −, * и т. д.), что упрощает взаимодействие с объектами, созданными пользовательскими классами. Хотя создание собственного оператора в C++ недоступно обычным пользователям в явном виде, перегрузка операторов может рассматриваться как неявная реализация новых операторов с целью облегчить взаимодействие с объектами.

Язык C++ предоставляет возможность не только перегружать операторы, но и использовать шаблоны для создания разнообразных контейнеров. Одним из самых устоявшихся и широко используемых контейнеров является вектор, представляющий собой динамический массив, гибко изменяющий свой размер при добавлении или удалении элементов. В отличие от обычных массивов, где программисту приходится управлять выделенной памятью вручную, используя оператор delete, в векторе этот процесс автоматизирован благодаря деструкторам контейнеров библиотеки STL. Это значительно упрощает работу программиста и обеспечивает безопасное управление памятью в приложениях на C++.

В языке программирования C++ представлены все необходимые средства для воплощения в жизнь любых идей. Путем сочетания перегрузки операторов и использования векторов в качестве отправной точки можно создать структуры данных с улучшенной производительностью и расширенной функциональностью.

В статье предлагается провести исследование с целью модернизации широко известной структуры данных — вектора. В рамках этого исследования будет рассмотрено создание нового оператора путем перегрузки нескольких других пользовательских операторов в правильном порядке. Далее, используя эти компоненты, предполагается разработать основные элементы игровой механики, которые будут реализованы на графическом движке OpenGL.

Основная цель статьи заключается в проверке возможности реализации предложенных идей в контексте языка программирования C++. Помимо этого, значительное внимание уделяется выяснению практических выгод, которые могут возникнуть в результате этих творческих усилий.

Новый пользовательский оператор

Количество пользовательских операторов в языке программирования C++ на данный момент насчитывает около трех десятков. При этом каждый обладает своим уровнем и группой приоритетности, символьным и словесным описанием, а также определенной группировкой значений при использовании. Во всех операторах группировка осуществляется слева направо [2, 3].

Основные операторы в языке C++:

– инкремент/декремент;

– функциональный (для определения функции);

– индексирование;

– логические (НЕ, И, ИЛИ, эквивалентность (равенство), импликация (следование)).

Приоритет операций в алгебре логики такой же, как и в математике: вначале выполняются операции в скобках, далее идет инвертирование операндов. Следом по приоритетности идет конъюнкция, далее дизъюнкция. В последнюю очередь идет импликация (логическое следование) и самой последней — эквивалентность. Все логические операторы, упомянутые выше, реализованы на языке C++ как пользовательские, за исключением импликации. На ней как раз стоит остановиться.

Основные свойства импликации:

1) x достаточное условие для y;

2) y необходимое условие для x;

3) результат операции ложный, если из 1 следует 0;

4) результат операции — истина, если выражение $x <= y$;

5) эквивалентна выражению ($!x \vee y$).

В языке программирования C++ операция следования выполняется как выражение в пункте 5. На первый взгляд, оно может показаться компактным. Однако что произойдет, если эту операцию нужно будет повторить десятки, сотни раз? В таком случае выражение станет чрезмерно громоздким, что ухудшит читабельность программного кода не только для людей, читающих его, но и для тех, кто его написал.

Создание нового оператора в языке программирования C++ представляет собой непростую задачу, так как напрямую этого сделать нельзя. Существуют два альтернативных способа для его реализации: через обыкновенную функцию и через перегрузку других операторов. Первый вариант довольно прост, но имеет недостаток в громоздкости при вызове функции каждый раз. Это может затруднить восприятие кода для читателя. Второй вариант предполагает создание оператора через перегрузку нескольких пользовательских операторов. Главное в этом случае — правильно определить «шаблон перегрузки»: параметры, которые он принимает, и возвращаемое значение, если таковое имеется. Возвращаемым значением должна быть ссылка на объект. А что будет происходить в теле оператора — решать уже нам: можно спокойно перегрузить оператор инкремента, при этом будет выполняться операция умножения.

Этапы реализации оператора следования:

1. Объявить шаблонную структуру для левой части оператора:

```
template<class T, class new_operator>
struct Left_side {
        T left;
};
```

В шаблоне указывается два абстрактных типа данных. Внутри структуры будет храниться единственное поле left. Внесем название оператора в конструкцию перечисления enum.

2. Перегрузить оператор «меньше» для левой части (до знака <) оператора следования:

```
template<class T>
Left_side<T, decltype(sledovanie)> operator<(const T& levaya,
decltype(sledovanie)){
        return {levaya};
}
```

Возвращаемым значением будет ранее созданная структура, в которую будет передаваться значение параметра *levaya*.

3. Перегрузить оператор «больше» — правую часть оператора импликации:

```
template<class T1, class T2>
bool operator>(Left_side<T1, decltype(sledovanie)> levaya, T2
right_part){
        T1& left_part = levaya.left;
        return (!left_part || right_part); // реализация оператора.
}
```

Для обеспечения корректной работы оператора следования необходимо применить перегруженные операторы < и >. Используем директиву define для создания макроса, который сокращает объем оператора до одного символа. Таким образом, оператор импликации готов к использованию.

Шаблонный контейнер «Вектор»

Контейнер является динамическим массивом, но, в отличие от статического, в него можно легко добавлять новые значения благодаря автоматическому увеличению пространства. Наглядным примером контейнера является всеми известная шаблонная структура — вектор.

Исследование [4] показывает, что шаблонные функции и классы устраняют необходимость в перегрузке функций для различных типов данных. Это достигается благодаря уникальному типу T в шаблонах, который можно заменить на любой тип данных языка C++ [5].

В объектно-ориентированном программировании рекомендуется называть функции, определенные внутри классов, методами, а данные — полями. Важно отметить разницу между классами и объектами. Документация [6] подробно описывает объекты как экземпляры класса, которые могут быть инициализированы как вручную путем прямого присвоения значений их полям, так и с использованием методов класса.

Основные методы шаблонного контейнера включают в себя:

- clear: очищает содержимое контейнера от всех элементов;
- push_back(новый_элемент): добавляет новый элемент в конец контейнера;
- pop_back(): удаляет последний элемент из контейнера;
- size(): возвращает количество элементов в контейнере, позволяя определить его размер;
- capacity(): возвращает текущую емкость контейнера, то есть количество элементов, которые могут быть сохранены в векторе без необходимости увеличения выделенной памяти.

Емкость должна быть больше или равна фактическому количеству элементов в векторе. Когда вектор заполняется, его емкость автоматически увеличивается для дополнительного хранения данных.

Для доступа к элементу вектора по индексу можно воспользоваться не только квадратными скобками, но и методом at, который позволяет найти элемент по указанному порядковому номеру.

Важность конструкторов и деструкторов в языке программирования C++ заключается в обеспечении корректного и безопасного управления ресурсами при создании и удалении объектов. Правильное использование этих элементов структуры классов является ключевым аспектом для обеспечения надежности и эффективности программного кода. Пример реализации конструктора и деструктора:

```
template <class T>
new_vector<T>::new_vector() {
   massive = new T[1];
   volume = 1;
   elements = 0;
}
template <class T>
new_vector<T>::~new_vector() {
  delete[] massive;
}
```

Конструктор по умолчанию выделяет память, устанавливает переменную объема в 1 и количество элементов в 0. Деструктор освобождает динамическую память, выделенную для массива вектора в процессе работы программы. Важно отметить, что new_vector<T> представляет собой пространство имен. Методы, используемые в этом пространстве имен (в данном случае, в классе), должны начинаться с указания имени пространства и разделяться символом «::».

Использование пространства имен (using namespace) является способом избежать конфликтов именований, позволяя локализовать идентификаторы. Как указано в [7], использование пространства имен является эффективным способом предотвратить конфликты идентификаторов.

Обращение к пространству имен ns-name :: member-name [8], представляет собой важный пример, используемый в данной работе.

Реализация основных методов работы с вектором включает в себя операции добавления, удаления, доступа к элементам и изменения размера. В рамках данной задачи предполагается создание функций, позволяющих добавлять элементы в вектор, удалять их, получать доступ к определенному элементу по индексу, а также изменять размер вектора в соответствии с требованиями приложения.

Для успешной реализации шаблонного контейнера нам предстоит выполнить следующие действия:

1. Реализовать метод добавления элемента в вектор. Чтобы не выйти за границы области памяти, выделенной для вектора, нужно каждый раз проверять, не заполнена ли она целиком. Если окажется, что места нет, предстоит выделить для массива памяти для хранения данных в 2 раза больше старого, то есть расширить контейнер вдвое. Но почему именно 2 раза?

Процесс расширения массива включает в себя выделение новой памяти с помощью оператора new (new тип[размер]) [9]. Этот процесс может быть довольно ресурсоемким, если выполнять его при каждой операции добавления новых элементов.

Из-за значительной нагрузки операция расширения динамического массива будет замедлять выполнение программы на порядки. Чтобы избежать этого и не занимать излишне много памяти, оптимальным решением является увеличение выделенного пространства в два раза при каждом последующем заполнении, по сравнению с предыдущим объемом.

Выделение памяти под вектор нового размера:

```
if (elements == volume) {
    T* new_massive = new T[2 * volume]; // процесс пересоздания массива
    for (int i = 0; i < volume; i++)
        new_massive[i] = massive[i]; // копирование по циклу значений в новый массив
    delete[] massive; // очистка памяти от старого массива.
    volume *= 2;
    massive = new_massive;
  }
```

После выделения памяти под массив происходит копирование элементов из старого массива в новый. Далее память очищается, объем хранилища увеличивается в 2 раза, а старому указателю присваивается новый на последний созданный массив. На место после крайнего элемента происходит добавление нового значения.

2. Реализовать метод, позволяющий получить конкретный элемент вектора на основе заданного индекса. Этот функционал необходим для обеспечения удобного доступа к элементам контейнера и последующей их обработки.

Для того чтобы написать метод получения элемента по индексу, нужно передавать параметр целочисленного индекса. Единственное условие, которое нужно проверять — выход индекса за границы массива: если индекс вышел за границы массива, ничего не возвращается.

3. Реализовать метод удаления последнего элемента из массива, обеспечивая гибкость управления содержимым. Этот метод необходим для операций, требующих динамического изменения размера массива.

Этот метод уменьшает количество элементов в векторе на один, не принимая на вход никаких параметров. Удаленный элемент будет заменен при добавлении нового элемента в будущем. Дополнительно, в классе реализован деструктор, который освободит память, выделенную для удаленного элемента. Удаленный элемент не отображается при выводе содержимого вектора на экран, так как его индекс превышает максимально допустимый. Метод возвращает текущий размер вектора, подсчитанный после удаления элемента.

Далее перегружаем операторы.

4. Перегрузить оператор равенства. Реализация оператора проверки на равенство векторов:

```
template <class T>
bool new_vector<T>::operator == (const new_vector<T>& new_v) {
  if (this->elements != new_v.elements)
    return false;
  for (size_t i = 0; i < this->elements; i++) {
    if (this->massive[i] != new_v.massive[i]
      return false;
  }
  return true;
}
```

В случае оператора равенства для векторов они могут быть неравными, если они различаются по размеру или хотя бы один из их элементов отличается при сравнении. Если хотя бы одно из этих условий не выполняется, оператор возвращает false.

5. Перегрузить оператор неравенства.

Операторы — это фактически функции, которые могут быть перегружены, поэтому они принимают определенные параметры. В данном случае можно вызвать ранее определенный оператор равенства рекурсивно и инвертировать его результат.

6. Перегрузить оператор индексации ([]), который позволяет обращаться к элементам контейнера (например, массива или вектора) по их индексам, что делает работу с данными структурами более удобной и эффективной. Перегрузка этого оператора позволяет определить специфическое поведение при обращении к элементам контейнера:

```
template <class T>
T& new_vector<T>::operator[](int index) {
        assert(index >= 0 && index < elements);
        return massive[index];
}
```

Сначала индекс проверяется с использованием функции assert для предотвращения ошибок, связанных с памятью. Если индекс находится в пределах массива, возвращается соответствующий элемент. В случае выхода за границы массива вызывается метод abort(), прекращающий выполнение программы и выводящий сообщение об ошибке.

7. Для удобства вывода содержимого вектора необходимо перегрузить оператор потокового вывода.

В стандартном векторе этот оператор не был перегружен, что создавало неудобство при выводе содержимого и требовало использования цикла for для этой цели в рамках рассматриваемой задачи. Реализация перегруженного оператора вывода:

```
template <class T9>
ostream& operator <<(ostream& vyvod, const new_vector<T9>&
new_v) {
  for (int i = 0; i < new_v.elements; i++)
    vyvod << new_v.massive[i] << " ";
  return vyvod;
}
```

Элементы массива выводятся с использованием цикла от 0 до последнего индекса массива, что типично для вывода содержимого обычного вектора. В конце оператор возвращает ссылку на поток для поддержки цепочки операций.

На этом реализация структуры new_vector с ее базовыми методами завершена.

ДВЕ НОВЫЕ ИНТЕГРАЦИИ – ЧАСТЬ ЧЕГО-ТО БОЛЬШЕГО?

Оператор импликации и перегруженный контейнер «Вектор» представляют собой мощные инструменты в мире программирования на C++. Первый используется для логических выражений, позволяя устанавливать условия и связи между данными. Второй — это гибкая структура данных, способная хранить и управлять коллекциями элементов, облегчая работу с данными и их обработку в программе. Их интеграция открывает двери для оптимизации и эффективного управления информацией в различных алгоритмах, например:

1. Упрощение программирования на C++. Контейнер «Вектор» помогает удобно хранить и управлять данными, упрощая создание новых структур данных или классов.

2. Оптимизация работы с данными. Оператор следования может использоваться для более эффективной обработки системных данных, улучшая алгоритмы работы с ними.

3. Разработка игровых механик. В C++ оператор импликации и «Вектор» подходят для создания интересных игровых механик и управления игровыми объектами.

При этом следует подчеркнуть, что оператор импликации («если *A*, то *B*») прекрасно используется в алгоритмах, обрабатывающих различные сценарии в играх. И его интеграция с контейнером «Вектор» позволяет создавать сложные игровые логики и управлять игровыми элементами.

Помимо этого, мы можем поднять планку. Вместо того, чтобы просто создавать игру на языке C++, мы можем воспользоваться мощью OpenGL и библиотеки Glut для разработки качественных графических сцен. А дополнительные возможности по созданию собственных шейдеров на языке GLSL придадут игре уникальный вид. Таким образом, C++ становится не только основой, но и ключом к созданию захватывающего игрового мира. Но перед тем, как углубиться в детали механики игры, рассмотрим основы работы с OpenGL, Glut и GLSL.

Однако стоит учитывать, что у нас еще есть реализованный контейнер «Вектор», который может хранить значения обработанных условий. Пункт 2, безусловно, подходит под практическое применение моделей интеграции, однако стоит ли останавливаться на одном алгоритме, если и он может быть только частью чего-то большего?

А что, если сделать собственную игру, которая будет реализована на базе языка C++? А что, если написать игру на движке OpenGL, взаимодействуя с одной из поддерживаемых библиотек — Glut? А что, если вдобавок ко всему этому не использовать стандартные шейдеры, заложенные программой, а реализовать их самому на C-подобном языке GLSL? Каков же будет результат? В таком случае использовать только язык программирования C++ не выйдет, однако он останется основой для написания программного кода к игре. Перед тем, как углубиться в идеи для механики игры, стоит немного поговорить про OpenGL, Glut и GLSL.

OpenGL (Open Graphics Library) — это современный кроссплатформенный API для работы с двухмерной и трехмерной графикой. Он функционирует по принципу клиент-сервер, где клиентом является пользователь, использующий движок OpenGL, а сервером выступает графическое ядро (GPU) и его драйвер [10].

Однако напрямую работать с чистым OpenGL не всегда удобно. Для более удобного и эффективного программирования следует воспользоваться дополнительными библиотеками:

1. glut.h (OpenGL Utility Toolkit) — библиотека для различных утилит, предназначенная для приложений на OpenGL. Она упрощает возможность создания графического окна, позволяет добавлять текст, определять функции отрисовки и даже устанавливать таймеры. Благодаря кроссплатформенной поддержке, код, написанный для одной операционной системы, будет работать и на других.

Пример использования библиотеки GLUT:

```
glutInitWindowSize(666, 666);
glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGB|GLUT_DEPTH);
glutInitWindowPosition(280, 80);
glutCreateWindow("");
glutDisplayFunc(displayMe);
glutReshapeFunc(reshape);
```

2. glew.h (OpenGL Extension Wrangler). Эта библиотека расширяет функциональность OpenGL для работы с шейдерами. Шейдеры — это специальные программы, которые выполняют графические вычисления на графическом процессоре (GPU). Они позволяют контролировать различные аспекты графики, такие как освещение, тени, отражения и преломления света.

При этом важно проверять статус функции инициализации библиотеки GLew:

```
if(glewInit() != GLEW_OK){
std::cout << "Error in glewInit\n";
return 1;
}
```

Для реализации предлагаемой модели интеграции нам понадобится GLSL (OpenGL Shading Language): GLSL — это язык программирования, который используется для написания шейдеров в OpenGL. GLSL напоминает язык программирования C и C++, что делает его относительно легким для изучения и использования программистами. Он предоставляет широкий набор функций и возможностей для создания сложных визуальных эффектов в компьютерной графике. Шейдеры пишутся на языке GLSL и компилируются на графическом процессоре в реальном времени во время выполнения программы OpenGL. В контексте OpenGL, GLSL позволяет программистам создавать сложные и красивые сцены, делая компьютерную графику более реалистичной и захватывающей.

При этом, в отличие от языка C++, в GLSL отсутствуют:

- строки;
- указатели;
- битовые поля;
- рекурсия;
- подключение каких-либо библиотек из-за отсутствия необходимости.

В GLSL есть векторные и матричные типы, которые позволяют работать с шейдерами:

- vec2, vec3, vec4 — двух-, трех- и четырехмерные векторы из компонент типа float;
- ivec2, ivec3, ivec4 — двух-, трех- и четырехмерные векторы из компонент типа int;
- uvec2, uvec3, uvec4 — двух-, трех- и четырехмерные векторы из компонент типа uint;
- bvec2, bvec3, bvec4 — двух-, трех- и четырехмерные векторы из компонент типа bool;
- mat2, mat3, mat4 — двух-, трех- и четырехмерные квадратные матрицы из компонент типа float;
- mat2x3, mat2x4, mat3x2, mat3x4, mat4x1, mat4x2, mat4x3 — матрицы из компонент типа float.

В статье рассматривается GLSL версии 3.30. Эта версия поддерживает входные и выходные типы данных, а также специальный спецификатор — uniform. Эти особенности позволяют более гибко управлять данными и параметрами, необходимыми для реализации шейдеров в контексте предлагаемой модели интеграции.

В контексте рассматриваемой версии GLSL входной тип (in) представляет собой данные, с которыми шейдер будет работать внутри программы, а выходной тип (out) возвращает полученное значение из шейдера в программу. Тип uniform задается пользователем в основной программе и передается шейдеру с помощью специальной функции getUniformLocation().

При этом существуют два основных способа загрузки шейдеров в программу:

– через передачу считанной строки из файла;

– через строку символов, написав код прямо в программе на C++.

Дополнительные сведения об основных типах шейдеров можно найти в [11, 12].

Примеры чтения шейдеров на языке C++:

1. Функция чтения файла в строку:

```
string txtFileRead(string filename){
   string startStr, returnStr;
   ifstream in;

   in.open(filename);
   if(in.is_open()){
      getline(in, startStr);
      while(in){
         returnStr += startStr + "\n";
         getline(in, startStr);
      }
   }
   in.close();

   return returnStr;
}
```

2. Передача кода шейдера на языке GLSL напрямую в строку символов:

```
const char* vertexShaderSource =
"#version 330 core\n"
"layout(location = 0) in vec3 position;\n"
"layout(location = 1) in vec3 Norm;\n"
"out float light;\n"
"uniform mat4 model;\n"
"uniform mat4 view;\n"
"uniform mat4 projection;\n"
"uniform vec3 lightPos;\n"
"void main(){\n"
"vec3 p = vec3(view*model* vec4(position, 1.0));\n"
"vec3 l = lightPos-p;\n"
"vec3 n = vec3(model * vec4(Norm, 1.0));\n"
"light = max(dot(n,l),0.0);\n"
"gl_Position = projection * view * model * vec4(position, 1.0);\n"
"}\n\0";
```

Для успешной компиляции шейдеров в программе на языке C++, обе методики требуют использования библиотеки GLew. После этого необходимо воспользоваться рядом функций:

– loadShader() — для загрузки шейдера;

– glGetShaderiv() — для проверки ошибок компиляции;

– glGetShaderInfoLog() — для получения информации об ошибках;

– glCreateProgram() — для создания программы, к которой будут присоединены шейдеры (glAttachShader());

– glLinkProgram() — для линковки программы;

– glGetProgramiv() и glGetProgramInfoLog() — для проверки ошибок линковки;

– glUseProgram() — отвечает за отрисовку данных, обработанных шейдерами.

Как упоминалось ранее, стандартные шейдеры в OpenGL предоставляют множество удобных функций для работы с графическими объектами, избавляя пользователя от необходимости работать с матрицами и векторами. Однако при подключении собственных шейдеров пользователь сам отвечает за правила взаимодействия с объектами. Это означает, что пользователю придется работать с матрицами (двумерными массивами) вручную!

Теперь, когда мы познакомились с основами OpenGL, рассмотрим, каким образом концепция будущей игры будет построена и на каких принципах она будет основана. Оператор импликации, о котором мы упоминали ранее, работает с условиями вида «если A, то B». В алгебре логики он представляется формулой $!A \,||\, B$ (не A или B). Допустим, оператор следования объединяет как минимум два действия в игре, произведенные пользователем. В этом случае вектор будет хранить значения, полученные в результате этих действий. После сохраненные значения будут проверяться в нашем шаблонном контейнере, и на основе этой проверки будет приниматься решение — true (истина) или false (ложь).

Конечная цель нашей игровой механики состоит в том, чтобы оператор импликации мог объединять различные действия игрока и принимать решения на основе этих действий.

Приведем простой пример: когда пользователь выполняет действия, например, подбирает ключ и затем пытается открыть дверь, значения этих действий сохраняются в нашем векторе. Далее происходит проверка сохраненных значений в шаблонном контейнере. Если оба условия выполнены (пользователь подобрал ключ и пытается открыть дверь), оператор импликации возвращает true (истина), указывая, что дверь должна открыться. В противном случае, если хотя бы одно из условий не выполнено, оператор импликации вернет false (ложь), и дверь останется закрытой. В ситуации с целой локацией дверей значения действий игрока будут храниться в векторе до тех пор, пока он не пройдет все двери без ошибок или хотя бы раз не произойдет просчет. В таких случаях значения шаблонного контейнера обнуляются, чтобы избежать неправильной работы игровой механики.

Таким образом, наш вектор хранит значения, полученные в результате различных действий игрока, и оператор импликации использует эти значения для принятия решений в игре, что делает механику более интересной и интерактивной для игрока.

Предложенная механика может быть весьма эффективна при создании игр-головоломок различной сложности. Эти логические игры могут быть самостоятельными проектами с увлекательным сюжетом или дополнительными элементами в более крупных играх. Особенно ценным является второй вариант, где головоломки становятся ключевым компонентом для развития сюжета и принятия игровых решений. В таких случаях игроки могут взаимодействовать с головоломками, чтобы открыть сюжетные ответвления и раскрывать новые аспекты игровой вселенной. Это подчеркивает важность данной механики в обогащении геймплея и создании увлекательного игрового опыта.

Общая диаграмма структуры игрового приложения представлена на рисунке 1.

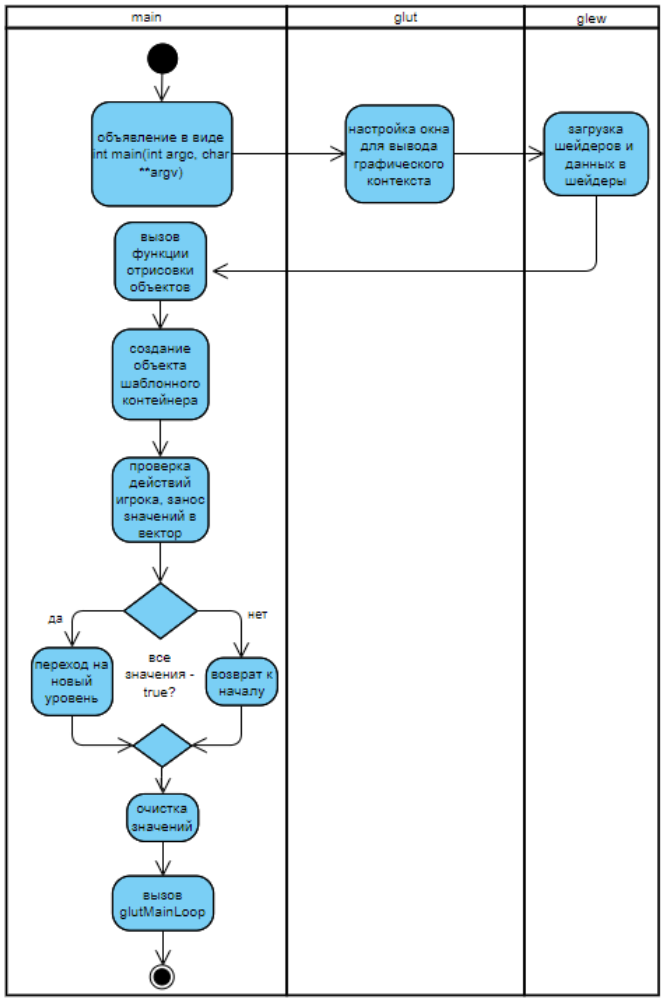

Рис. 1. Структура игрового приложения

Следует отметить, что OpenGL предоставляет возможность создавать как 2D, так и 3D приложения, что, в свою очередь, позволяет интегрировать головоломки как в двумерные, так и в трехмерные игры. Интересные головоломки могут усилить слабую механику игры, делая ее более увлекательной и уникальной. С другой стороны, сложные головоломки могут сделать простые игровые механики интересными и вызывающими. Помимо этого, можно экспериментировать с различными механиками, использующими векторы и операторы импликации. Но следует помнить, что разработка таких механик может быть сложной задачей, требующей внимательного проектирования.

Заключение

C++ — язык с бесконечными возможностями, позволяющий создавать уникальные модификации благодаря шаблонам, классам и перегрузке операторов. В данной статье представлена инновационная связка моделей интеграции: контейнера «Вектор» и оператора импликации. Эти компоненты стали основой для разработки механики игры на движке OpenGL, используя библиотеки GLUT и GLew, а также самописные шейдеры. Полученный опыт открывает новые перспективы для будущих проектов в области компьютерных игр и графики.

Представленная в статье механика игрового опыта предоставляет новаторские возможности для индустрии

развлечений, добавляя глубину и оригинальность игровым проектам. Однако, как и с любой новой идеей, данную модификацию нужно будет адаптировать под постоянно изменяющиеся технологии. В случае успешной реализации этой концепции планируется более подробное рассмотрение создания 2D или 3D игровых приложений с использованием предложенной на рисунке 1 структуры.

Представленные концепции могут послужить источником вдохновения для дальнейшего творчества. Все, что было рассмотрено в данной статье, является всего лишь крошечной частью богатства языка программирования C++ и современного кроссплатформенного API OpenGL. Возможности и потенциал этих технологий остаются практически безграничными, ожидая новых исследований и творческих решений.

Литература

1. Рейзлин, В. И. Язык C++ и программирование на нем: Учебное пособие. — 3-е изд., перераб. — Томск: Изд-во Томского политехнического университета, 2021. — 208 с.

2. Литвиненко, Н. А. Технология программирования на C++. Начальный курс. — Санкт-Петербург: БХВ-Петербург, 2005. — 288 с.

3. Введение в математическую логику. Нотации // Хекслет. URL: http://ru.hexlet.io/courses/logic/lessons/notation/theory_unit (дата обращения 08.12.2023).

4. Семенякин, В. С. Просто о шаблонах C++ // Хабр. — 2022. — 10 января. URL: http://habr.com/ru/articles/599801 (дата обращения 22.11.2023).

5. Абрамян, М. Э. Введение в стандартную библиотеку шаблонов C++: описание, примеры использования, учебные задачи: Учебник. — Ростов-на-Дону; Таганрог: Изд-во Южного федерального университета, 2017. — 177 с.

6. Классы в C++ // Язык программирования C++: Учебник. URL: http://cppstudio.com/post/439 (дата обращения 26.11.2023).

7. Шилдт, Г. C++: базовый курс. Третье издание = C++ from the Ground Up. Third Edition / Пер. с англ. и ред. Н. М. Лучко. — Москва: Издательский дом «Вильямс», 2010. — 624 с.

8. Namespaces // C++ Reference. — Обновлено 26.11.2023. URL: http://en.cppreference.com/w/cpp/language/namespace (дата обращения 29.11.2023).

9. Динамическое выделение памяти в C++ // Программирование. URL: http://prog-cpp.ru/cpp-newdelete (дата обращения 06.12.2023).

10. Васильев, С. А. OpenGL. Компьютерная графика: Учебное пособие. — Тамбов: Изд-во Тамб. гос. техн. ун-та, 2005. — 80 с.

11. Боресков, А. В. Программирование компьютерной графики. Современный OpenGL. — Москва: ДМК Пресс, 2019. — 372 с.

12. OpenGL ES 3.0. Руководство разработчика = OpenGL ES 3.0 Programming Guide. Second Edition / Д. Гинсбург, Б. Пурномо, Д. Шрейнер, А. Мунши; пер. с англ. А. В. Борескова. — Москва: ДМК Пресс, 2015. — 448 с.

DOI: 10.24412/2413-2527-2023-436-34-41

Development and Practical Application of the Integration Model of the Implication Operator and the Overloaded «Vector» Container in the C++ Programming Language

A. A. Nelin, PhD A. V. Zabrodin
Emperor Alexander I St. Petersburg State Transport University
Saint Petersburg, Russia
puppi2016@mail.ru, teach-case@yandex.ru

Abstract. **The article discusses the theoretical and practical aspects necessary for the implementation of a new integration of the key components — the implication operator and the template container «Vector». These components will serve in the development of the game on the OpenGL graphics engine using the additional GLUT library and independently written shaders in the GLSL language. This implementation can serve as a source of inspiration for programmers, encouraging them to actively develop future ideas. The practical sections of the article include test cases demonstrating the correct operation of integration models.**

Keywords: **C++ programming language, implication, operator overloading, vector, memory allocation, classes, integration, mechanics, OpenGL, shaders.**

REFERENCES

1. Reyzlin V. I. The C++ language and programming on it: Study guide [Yazyk C++ i programmirovanie na nem: Uchebnoe posobie]. Tomsk, Tomsk Polytechnic University, 2021, 208 p.

2. Litvinenko N. A. C++ programming technology. The initial course [Tekhnologiya programmirovaniya na C++. Nachalnyy kurs]. Saint Petersburg, BHV-Peterburg Publishing House, 2005, 288 p.

3. Introduction to Mathematical Logic. Notations [Vvedenie v matematicheskuyu logiku. Notatsii], *Hexlet [Khekslet]*. Available at: http://ru.hexlet.io/courses/logic/lessons/notation/theory_unit (accessed 08 Dec 2023).

4. Semenyakin V. S. Just About C++ Templates [Prosto o shablonakh C++], *Habr [Khabr]*. Published online at January 10, 2022. Available at: http://habr.com/ru/articles/599801 (accessed 22 Nov 2023).

5. Abramyan M. E. Introduction to the standard C++ template library: Description, examples of using, training tasks: Textbook [Vvedenie v standartnuyu biblioteku shablonov C++: opisanie, primery ispolzovaniya, uchebnye zadachi: Uchebnik]. Rostov-on-Don, Taganrog, Southern Federal University, 2017, 177 p.

6. Classes in the C++ [Klassy v C++], *C++ programming language: Textbook [Yazyk programmirovaniya C++: Uchebnik]*. Available at: http://cppstudio.com/post/439 (accessed 26 Nov 2023).

7. Schildt H. C++ from the Ground Up. Third Edition [C++: bazovyy kurs. Tretye izdanie]. Moscow, Williams Publishing House, 2010, 624 p.

8. Namespaces, *C++ Reference*. Last update at November 26, 2023. Available at: http://en.cppreference.com/w/cpp/language/namespace (accessed 29 Nov 2023).

9. Dynamic Memory Allocation in C++ [Dinamicheskoe vydelenie pamyati v C++], *Programming [Programmirovanie]*. Available at: http://prog-cpp.ru/cpp-newdelete (accessed 06 Dec 2023).

10. Vasilyev S. A. OpenGL. Computer graphics: Study guide [OpenGL. Kompyuternaya grafika: Uchebnoe posobie]. Tambov, Tambov State Technical University, 2005, 80 p.

11. Boreskov A. V. Computer graphics programming. Modern OpenGL [Programmirovanie kompyuternoy grafiki. Sovremennyy OpenGL]. Moscow, DMK Press Publishing House, 2019, 372 p.

12. Ginsburg D., Purnomo B., Shreiner D., Munshi A. OpenGL ES 3.0 Programming Guide. Second Edition [OpenGL ES 3.0. Rukovodstvo razrabotchika]. Moscow, DMK Press Publishing House, 2015, 448 p.