МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
**«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»**
Кафедра математической кибернетики и компьютерных наук

РАЗРАБОТКА И ПРИМЕНЕНИЕ ИГРОВОГО ДВИЖКА

АВТОРЕФЕРАТ БАКАЛАВРСКОЙ РАБОТЫ

студента 4 курса 411 группы
направления 02.03.02 — Фундаментальная информатика и информационные технологии
факультета КНиИТ
Денисова Дмитрия Александровича

Научный руководитель
Зав. кафедрой _____ И. А. Батраева

Заведующий кафедрой
к. ф.-м. н. _____ С. В. Миронов

Саратов 2026

1 Постановка цели и задач

Работа состоит из введения, двух глав, заключения, списка литературы и трёх приложений. Во введении обосновывается актуальность работы, формулируются цели и задачи, стоящие перед автором. В частности описывается, что на сегодняшний день существует немалое количество игровых движков, обладающих различным функционалом и возможностями. Как и в случае с другими фреймворками, разработчик совершает выбор движка, основываясь на том, какие средства необходимы для реализации задуманной видеоигры, а также какие способности есть у него или команды. При этом упрощение использования известных игровых движков, происходящее с каждым их новым обновлением, а также возрастающий интерес к более простым фреймворкам на фоне успехов игр, сделанных с их помощью, объясняют тенденцию развития игровых движков как инструментов, доступных более широкому кругу людей для реализации своих творческих идей.

Автор ставит своей целью разработку игрового движка путём реализации системы рендеринга 3D-графики и системы логического взаимодействия игровых объектов, а также внедрения графического редактора для применения фреймворка в разработке обучающей видеоигры.

Актуальность описываемой работы заключается в разработке игрового движка, использующего представление игровых сцен в виде прямоугольного поля, наподобие шахматного. Поскольку имеющиеся на данный момент игровые движки либо требуют отдельной реализации подобного представления игровых сцен, что требует больше времени для разработки, либо не предоставляют достаточных инструментов для работы с 3D-графикой в подобных играх.

Для достижения поставленной цели в работе будут выполнены следующие задачи:

1. Изучение теоретической основы процесса рендеринга с помощью алгоритма растеризации;
2. Изучение теоретической основы систем сцен и игровых объектов игрового движка;
3. Расширение функционала систем сцен и игровых объектов для демонстрации их работы;
4. Разработка графического редактора для взаимодействия с системами игрового движка.

2 Теоретическое основание разработки

В следующей секции описываются существующие игровые движки, перечисляются их преимущества и недостатки, приводятся следующие движки: Unity3D, Godot, RPG Maker.

В работе указывается, что рассмотренные решения имеют как сильные, так и слабые стороны. Большинство игровых движков общего назначения, конечно, позволяют создавать игры жанров CRPG, JRPG, головоломок и прочих, но при этом требуют от разработчика тратить время и ресурсы на создание систем, ключевых для проектов подобных жанров. В свою очередь RPG Maker, его вариации и похожие фреймворки являются слишком специализированными для единственного жанра, из-за чего их функционал слишком ограничен. Поэтому целью данного проекта является разработка такого игрового движка, который позволит быстро разрабатывать игры, использующие в игровом процессе клеточное поле подобное шахматному, но который не будет ограничен только одним жанром.

В работе приведены основные инструменты, используемые для разработки, приводится их описание и обоснование использования. Автор использует язык программирования C++, графический API Vulkan, библиотеки pugiXML, tinyobjloader, GLFW, GLM, stb и DearImGui.

Автор приводит описание графического конвейера Vulkan – последовательности операций, которые переносят вершины и текстуры мешей в пиксели для отрисовки на экране. Описываются основные этапы конвейера: ассемблер входных данных, вершинный шейдер, шейдер тесселяции, геометрический шейдер, растеризация, фрагментный шейдер, смешивание цветов.

Далее приводится описание работы логических систем игрового движка: система представления игровых объектов, позволяющая создавать игровые объекты и изменять их игровую логику, и система сцен, отвечающая за создание и размещение игровых уровней, система представления игровых объектов, позволяющая создавать игровые объекты и изменять их игровую логику.

Также описывается роль графического редактора, как средство взаимодействия пользователя с вышеперечисленными компонентами игрового движка.

3 Описание разработки игрового движка и его применения

В работе приводится описание системы рендеринга 3D-графики. Система разделена на несколько подсистем, каждая из которых берёт на себя часть функционала по отрисовке финального изображения, за счёт чего добавление новых возможностей для вывода 3D-графики становится гораздо проще. На данный момент система состоит их четырёх подсистем:

— Система контроля рендеринга;
— Система основного рендеринга;
— Система рендеринга точечных источников света;
— Система контроля камеры.

Для того чтобы начать сам процесс рендеринга в Vulkan сначала необходимо провести ряд подготовительных действий, создав и инициализировав необходимые структуры, система контроля рендеринга выполняет именно эти функции, поскольку в Vulkan нет концепции «кадрового буфера по умолчанию», поэтому требуется инфраструктура, которая будет управлять буферами, осуществляющие рендеринг, до их отображения на экране. Эта инфраструктура называется цепочкой обмена и должна быть создана в Vulkan явно до того как начать сам процесс рендеринга. Цепочка обмена, по сути, представляет собой очередь изображений, ожидающих вывода на экран. То есть приложение получает такое изображение, отрисовывает его и возвращает в очередь. То, как именно работает очередь, и условия отображения изображения из очереди зависят от настройки цепочки обмена, но общая цель цепочки обмена — синхронизировать отображение изображений с частотой обновления экрана.

Система контроля рендеринга позволяет создать цепочку обмена, а также следить за её состоянием, поскольку в некоторых случаях может случиться такое, что цепочка обмена перестанет подходить для продолжения рендеринга. Одна из причин, по которой это может произойти, — изменение размера окна. Необходимо перехватывать эти события и пересоздавать цепочку обмена.

для создания цепочки обмена или её обновления используется метод recreateSwapChain, принадлежащий классу SmthRenderer. Метод beginFrame в начале каждого нового кадра проверяет состояние текущей цепочки, пытаясь получить из неё следующее изображение. Если попытка получения изображения оканчивается ошибкой, то метод либо вызывает метод recreateSwapChain, если ошибка была связана с несоответствием формата, как приведено в листинге 1.

```
VkCommandBuffer SmthRenderer::beginFrame() {
        VkResult result = smthSwapChain->acquireNextImage();
        //Если разрешение было изменено
    if (result == VK_ERROR_OUT_OF_DATE_KHR) {
            recreateSwapChain();
            return nullptr;
        }
        if (result != VK_SUCCESS) {
        throw std::runtime_error("Не удалось получить изображение");
        }
}
```

Листинг 1: Создание цепочки обмена

Далее описывается подсистема основного рендеринга – наиболее важная система, позволяющая отрисовывать загруженные модели и соответствующие им текстуры, для представления которых используются классы SmthModel, описывающий 3D-модели, необходимые для отрисовки, и SmthTexture, который описывает текстуры, накладываемые на 3D-модели. Графический конвейер же этой подсистемы не требует проведения смешения цветов и использует шейдеры, использующие для отрисовки объектов информацию, которая содержится в классе SmthModel.

Также приводится описание подсистемы управления камерой, которая передаёт в шейдеры матрицу проекции и view матрицу, определяющие тип проекции, положение и направление камеры, для чего также используется структура GlobalUBO, позволяющая передать данные о камере в шейдеры. Объект класса SmthCamera отвечает за обновление данных о камере каждый кадр работы программы, чтобы затем передать эти данные в структуру GlobalUBO.

Кроме того описывается работа системы рендеринга точечных источников света, позволяющей задавать игровому объекту компонент, превращающий его в точечный источник света. Для этого используется структура PointLight, поля которой передаются в шейдеры как часть структуры GlobalUBO. Эти шейдеры являются частью уже другого графического конвейера – конвейера точечных источников света, процесс отрисовки которых сильно отличается от отрисовки 3D-моделей, поскольку демонстрирующая сам источник модель является

полупрозрачным билбордом – фигурой, которая всегда остаётся повёрнутой к камере.

Далее приводится описание разработки системы сцен. Для представления сцен были разработаны классы Scene и Cell. Класс Cell представляет собой 3D-модель с компонентом TransformComponent, Этот класс необходим для создания вида самой сцены и размещения в ней игровых объектов по заданным координатам.

Также экземпляры класса Cell обладают своим номером и указателем на сцену. Основная задача этого класса – визуальная репрезентация сцены и сохранение координатов для размещения по ним игровых объектов.

Класс Scene содержит в себе вектор клеток cells, поля name, sizeX, sizeY, двумерный вектор sceneGraph и неупорядоченный словарь, содержащий используемые в сцене текстуры textures. Вектор cells содержит все клетки, присутствующие в сцене, которые содержат все игровые объекты сцены, sceneGraph – структура, описывающая какие клетки сцены являются смежными для упрощения перехода из клетки в клетку. Поля sizeX, sizeY содержат размер сцены, а поле name – название сцены. Класс также содержит методы addGameObject, getX, getY, find и статический метод serializeScene. Метод addGameObject позволяет добавить в сцену новый объект, getX и getY позволяют получить размеры сцены, а метод find найти в сцене объект по заданному номеру. Поле active необходимо, чтобы определить использует ли пользователь движок для редактирования сцен и разработки видеоигры или уже непосредственно играет в разработанную игру. Это необходимо для временного отключения игровой логики, чтобы игровые объекты не реагировали на пользовательский ввод до того, как начнётся процесс самой игры.

Помимо создания сцен необходимо обеспечить возможность перехода между ними, чтобы у пользователя была возможность назначать последовательность уровней или меню, для этого необходима система, которая позволит собрать все созданные сцены и упорядочить их по задумке пользователя, дав при этом возможность переходить между ними. Для этого в работе описывается класс SceneManager.

В этом классе есть вектор уникальных указателей на сцены scenes, метод addScene позволяет добавить в вектор новую сцену, deleteScene – удалить имеющуюся сцену, getScene позволяет получить указатель на сцену с указанным

именем, перегруженный оператор обращения по индексу – получить сцену по заданному индексу, метод nextScene позволяет перейти к следующей сцене в списке. Номер текущей сцены задаётся полем curScene, поле playing позволяет определить использует ли пользователь движок для разработки видеоигры или уже непосредственно играет в неё. В случае если пользователь начал играть, то для всех созданных сцен значение поля active устанавливается как истинное, позволяя сценам начать обработку логики игровых объектов, а значение curScene снова устанавливается в 0, чтобы игра начиналась с первого уровня или меню, заданного пользователем, эти задачи выполняет метод start, метод stop, наоборот, вновь останавливает обработку логики, позволяя пользователю вернуться к редактированию сцен.

В работе также приводится, что система представления игровых объектов имеет Has-a дизайн, это означает, что каждый игровой объект представляет собой набор компонентов, определяющих, на что он способен, например, игровой объект содержащий в себе объект класса SmthModel должен быть выведен на экран как 3D-модель, игровой объект, содержащий в себе объект класса PointLight должен излучать свет.

Для этого был создан класс GameObject, который содержит объект класса TransformComponent и указатели на объекты класса SmthModel, PointLightComponent и вектор указателей на прочие компоненты, унаследованные от класса Component, что приведено в листинге 2.

```
class GameObject {
public:
    static GameObject createGameObject()
    template <typename T>
        T* getComponent()
    void addComponent(Component* c)
    void update()
    TransformComponent transform{};
        std::shared_ptr<SmthModel> model{};
        std::unique_ptr<PointLightComponent> pointLight = nullptr;
        std::vector<Component*> components;
    int cellId;
    bool bound = true;
};
```

Листинг 2: Описание класса GameObject

Класс TransformComponent содержит все преобразования применённые для объекта, то есть перемещение, масштабирование и вращение. Каждый объект обязан обладать позицией в сцене, поэтому объект transform определён для каждого объекта и не может быть пустым, в отличие от остальных компонентов, так, например, источники света не обязаны иметь 3D-модель, поэтому для них указатель на объект класса SmthModel может быть пустым, как и все остальные компоненты.

Прочие компоненты унаследованы от класса Component, содержащего указатель на игровой объект, которому этот компонент принадлежит, и четыре виртуальных метода: initialize, update, serialize и deserialize, приведённых в листинге 3.

```
class Component {
public:
        Component() = default;
        Component(Component&& other) noexcept = default;
        virtual void update() {  }
        virtual void initialize() { };
        virtual void serialise(pugi::xml_node object)
        SmthGameObject* holder = nullptr;
        };
```

Листинг 3: Класс Component

Первый метод необходим для возможности инициализировать поля для нового компонента уже после добавления того к объекту, чтобы получить возможность обратиться к другим компонентам объекта (например, чтобы компонент, созданный пользователем, мог обратиться к компоненту TransformComponent и изменить положение объекта в пространстве при выполнении заданных пользователем действий). Метод update – наиболее важен, так как именно в нём может реализовываться логика, требующая вызова каждый новый кадр (например, проверка ввода пользователя). А методы serialize и deserialize необходимы для сохранения информации о компоненте в файл сцены и её восстановлении соответственно.

Компонент TransformComponent является обязательным компонентом, который есть у каждого игрового объекта, однако для задания поведения игровых объектов необходимо дать пользователю возможность создавать собственные компоненты и добавлять их к игровым объектам, для этого нужен некий контейнер, который позволит получать компоненты по какому-то идентификатору и добавлять их в вектор components игрового объекта. Для этого в работе был создан класс ComponentManager, описанный в листинге 4

```
class ComponentManager {
public:
    template <typename T>
    Component* createComponent();
    template <typename T>
    void registerComponent(std::string name);
    Component* createComponentByName(const std::string& name);
    using CreatorFunc = Component * (*)();
    std::unordered_map<std::string, CreatorFunc> allComponents;
    private:
    template <typename T>
    static Component* createComponentHelper();
};
```

Листинг 4: Описание класса ComponentManager

В классе поле allComponents представляет собой неупорядоченный словарь, где в качестве ключа находятся строки – имена компонентов, а в качестве значения – указатели на функции, возвращающие указатель на создаваемый компонент. С помощью метода registerComponent пользователь может зарегистрировать новый компонент, вставив в allComponents пару значений из названия нового компонента, записанного в строку и указателя на метод createComponent, который может создать новый компонент.

В работе описывается, что в качестве формата файлов используемых для сохранения был выбран xml, поскольку древовидная структура этого формата лучше всего подходит для описания подобной системы представления объектов.

Для сохранения сцены в класс Scene был добавлен статический метод serializeScene, который принимает на вход сцену и название файла для её сохранения. При вызове метода программа, используя средства библиотеки pugiXML создаёт xml-файл и записывает в него сначала размер сцены и её имя. Затем метод проходит по всем клеткам сцены, вызывая метод serialize для каждого игрового объекта в них, который записывает данные об объекте в соответствующий узел xml-документа, в частности все преобразования объекта, путь до файла, содержащий 3D-модель объекта и прочее.

Таким образом была получена возможность сохранять данные о сценах

в xml-файлах. Теперь необходима возможность восстанавливать их из указанных файлов. Для осуществления этой возможности также пригодится класс ComponentManager, который позволит восстановить созданные компоненты.

Используя методы из приведённого класса, можно реализовать статический метод deserialize для класса Scene, который позволит восстанавливать сцены из xml-файлов. Этот метод принимает на вход строку – путь до файла с сохранённой сценой, после чего, также используя средства библиотеки pugiXML считывает данные сначала о размере сцены, её имени и воссоздаёт её. Затем с помощью методов deserialize, реализованных у соответствующих компонентов, воссоздаёт игровые объекты и их компоненты в соответствующих клетках.

Далее в работе приводится описание разработки графического редактора движка, заключающаяся в создании четырёх основных меню, которые позволяют пользователю взаимодействовать с системами движка: меню сцен, которое позволяет открыть и редактировать текущую сцену, а также отображает все созданные на данный момент сцены, меню клетки, отображающееся при выборе соответствующей клетки через меню сцены, меню объекта, которое появляется при выборе соответствующего объекта и меню содержащее в себе доступные модели и текстуры. Также приводится пример использования, демонстрирующий возможность редактирования сцены и объектов в ней, а также возможность запуска разработанной пользователем видеоигры.

В конце работы приводится пример использования движка для разработки видеоигры со следующим игровым процессом: В одной из клеток сцены находится объект, являющийся источником, в другой объект-приёмник, в остальных клетках находятся провода, которые можно вращать, а также перемещать между клетками. Цель игрока – выстроить провода так, чтобы они соединяли объект-источник и объект-приёмник, после чего игрок переходит на следующий уровень.

Для выполнения этой задачи пользователю сначала потребуется разработать свои компоненты, которые могут выполнять описанную логику, пример реализации может включать в себя пять классов, которые наследуются от суперкласса Component и другого системного компонента MovableObject – компонента задающего передвижение игровых объектов по полю сцены. Это компоненты Source, Acceptor, WireStraight, WireBent и ObjectPicking первый компонент

будет относиться к тому объекту, который будет являться источником, второй соответственно относится к объекту-приёмнику, компоненты WireStraight и WireBent определяют провода, где первый определяет прямые провода, а второй провода с загибом. Последний же компонент будет присвоен объекту, позволяющий выбрать другой объект, который находится с ним в одной клетке. Приводится, что от пользователя требуется разработать логику работы указанных компонентов, для этого ему нужно описать методы update и initialize, а также задать поля классов, которые будут необходимы для работы компонента.

Затем после описания компонентов, необходимо зарегистрировать их, обратившись к методу объекта класса ComponentManager registerComponent.