Вступление

Добро пожаловать, чтобы создать свой собственный движок 2D-игры и создавать отличные веб-игры. Поскольку вы взяли в руки эту книгу, вас, вероятно, интересуют подробности игрового движка и создания ваших собственных игр, в которые можно играть через Интернет. Эта книга научит вас, как создать движок 2D-игры, охватывая соответствующие технические концепции, демонстрируя примеры реализаций и показывая, как организовать большое количество исходного кода и файлов ресурсов для поддержки разработки игр. В этой книге также обсуждается, как каждая охваченная техническая тематическая область соотносится с элементами игрового дизайна, чтобы вы могли создавать, играйте, анализируйте и узнавайте о разработке 2D игровых движков и игр.
Примеры реализаций в этой книге основаны на HTML5, JavaScript и WebGL2, которые являются технологиями, находящимися в свободном доступе и поддерживаемыми практически всеми веб-браузерами. После прочтения этой книги в разработанный вами игровой движок и связанные с ним игры можно будет играть через веб-браузер из любой точки Интернета.

В этой книге представлены актуальные концепции из области разработки программного обеспечения, компьютерной графики,математики, физики, разработки игр и геймдизайна — все в контексте создание 2D игрового движка. Презентации тесно интегрированы с анализом и разработкой исходного кода; большую часть книги вы потратите на создание игровых концептуальных проектов, демонстрирующих функциональность компонентов игрового движка.
Опираясь на исходный код, представленный ранее, книга поведет вас в путешествие, в ходе которого вы освоите основные концепции, лежащие в основе движка 2D-игр, одновременно приобретая практический опыт разработки простых, но работающих 2D-игр.Начиная с главы 4, раздел “Конструктивные соображения” включен в конце каждая глава должна соотносить рассмотренные технические концепции с элементами игрового дизайна. К концу книги вы будете знакомы с концепциями и техническими деталями 2D игровых движков, почувствуете себя компетентным в реализации функциональности в 2D игровом движке для поддержки часто встречающихся требований к 2D играм и сможете рассматривать технические темы игрового движка в контексте элементов игрового дизайна для создания веселых и увлекательных игры.

Новое во втором издании

Ключевые дополнения ко второму изданию включают обновление языка JavaScript и WebGL API, а также специальные главы с существенными подробностями о физике и компонентах систем частиц.Все примеры на протяжении всей книги доработаны с учетом новейших возможностей языка JavaScript. Хотя некоторые обновления являются обычными, например,замена синтаксиса цепочки прототипов, новейший синтаксис позволяет значительно улучшить общее представление и читаемость кода. Новая и гораздо более чистая асинхронная поддержка позволила создать совершенно новую архитектуру загрузки ресурсов с единой синхронизацией точка для всего двигателя (глава 4). Контекст WebGL обновляется для подключения к WebGL 2.0. Специальные главы позволяют более детально и постепенно знакомиться со сложной физикой и компонентами систем частиц. Подробные математические выводы включаются там, где это уместно.

Кому следует прочитать Эту книгу

Эта книга предназначена для программистов, знакомых с базовыми концепциями объектно-ориентированного программирования и обладающих базовыми или промежуточными знаниями объектно-ориентированного языка программирования, такого как Java или C#. Например, если вы студент
, прослушавший несколько вводных курсов программирования, опытный разработчик,новичок в программировании игр и графики, или энтузиаст-самоучка,вы сможете без особых проблем следовать концепциям и коду, представленным в этой книге. Если вы новичок в программировании в целом, рекомендуется, чтобы вы сначала стали
освоитесь с языком программирования JavaScript и концепциями объектно-ориентированного программирования, прежде чем приступать к содержанию, представленному в этой книге.

Допущения

У вас должен быть опыт программирования на объектно-ориентированном языке программирования, таком как Java или C#. Знания и опыт работы с JavaScript были бы плюсом, но не являются необходимыми. Примеры в этой книге были созданы в предположении, что вы понимаете инкапсуляцию данных и наследование. Кроме того, вы должны быть знакомы с базовыми структурами данных, такими как связанные списки и словари, и чувствовать себя комфортно, работая с основами алгебры и геометрии, особенно с линейными уравнениями и системами координат.

Кому не следует читать Эту книгу

Эта книга не предназначена для того, чтобы научить читателей программированию, и в ней не делается попытки объяснить сложные детали HTML5, JavaScript или WebGL2. Если у вас нет предыдущего опыта разработки программного обеспечения на объектно-ориентированном языке программирования, вам, вероятно, будет трудно следовать примерам из этой книги.

С другой стороны, если у вас есть обширный опыт разработки игровых движков на других платформах, содержание этой книги будет слишком простым; эта книга предназначена для разработчиков, не имеющих опыта разработки 2D игровых движков. Тем не менее, вы все равно можете почерпнуть несколько полезных советов о 2D-игровом движке и разработке 2D-игр для платформ, описанных в этой книге.

Организация этой книги

Эта книга учит, как разработать игровой движок, описывая базовую инфраструктуру, графическую систему, поведение игровых объектов, манипуляции с камерой и пример создания игры на основе движка.

В главах 2-4 описывается базовая инфраструктура игрового движка. Глава 2 устанавливает начальную инфраструктуру путем разделения системы исходного кода на папки и файлы, которые содержат следующее: логику ядра, специфичную для JavaScript, Шейдерные программы, специфичные для WebGL2 GLSL, и содержимое веб–страницы, специфичное для HTML5. Этот организация позволяет постоянно расширять функциональность движка при сохранении системных изменений локализованного исходного кода. Например, при внесении улучшений в поведение игровых объектов необходимо изменять только файлы исходного кода JavaScript. В главе 3 создается платформа рисования, позволяющая инкапсулировать и скрывать особенности рисования WebGL2 от остальной части движка. Эта система рисования позволяет разрабатывать поведение игровых объектов, не отвлекаясь на то, как они нарисованы. В главе 4 представлены и интегрированы основные функциональные компоненты игрового движка, включая игровой цикл, клавиатуру ввод, эффективная загрузка ресурсов и игрового уровня, а также поддержка звука.

В главах 5-7 представлены основные функциональные возможности игрового движка: система рисования,поведение и взаимодействия, а также манипуляции с камерой. Глава 5 посвящена работе с отображением текстур, включая листы спрайтов, анимацию с помощью листов спрайтов и рисование растровых шрифтов. В главе 6 предлагаются абстракции для игровых объектов и их поведения,

включая обнаружение столкновений с точностью до пикселя. В главе 7 подробно описываются манипуляции и взаимодействия с камерой, включая программирование с несколькими камеры и поддержка ввода с помощью мыши.

Главы 8–11 поднимают представленную функциональность на более продвинутый уровень. Глава 8 посвящена моделированию эффектов 3D-освещения в 2D-игровых сценах. В главе 9 обсуждается моделирование поведения на физической основе. В главе 10 представлены основы систем частиц, которые подходят для моделирования взрывов. Глава 11 рассматриваются более продвинутые функциональные возможности камеры, включая бесконечную прокрутку с помощью плиток и параллакса. Глава 12 подводит итог книге, знакомя вас с дизайном полноценной игры, основанной на разработанном вами игровом движке.

Примеры кода

Каждая глава этой книги содержит примеры, которые позволяют вам интерактивно экспериментировать с новыми материалами и изучать их. Вы можете получить доступ к исходному коду для всех проектов, включая связанные ресурсы (изображения, аудиоклипы или шрифты), нажав кнопку Загрузить
Кнопка с исходным кодом, расположенная на www.apress.com/9781484273760 . Вы должны увидеть структуру папок, организованную по номерам глав. Внутри каждой папки находятся подпапки, содержащие проекты Visual Studio Code (VS Code), соответствующие разделам этой книги .

Глава 1

Представляем 2D-игру Разработка движка с использованием JavaScript

Видеоигры - это сложные, интерактивные, мультимедийные программные системы. Они должны в режиме реального времени обрабатывать вводимые игроком данные, имитировать взаимодействие полуавтономных объектов
и генерировать высококачественную графику и аудиовыходы, пытаясь при этом заинтересовать игроков. Попытки создать видеоигру могут быстро стать непосильными из-за необходимости хорошо разбираться в разработке программного обеспечения, а также в том, как создавать привлекательный
игровой опыт. Первую проблему можно решить с помощью библиотеки программного обеспечения или игры движок, содержащий согласованный набор утилит и объектов, разработанных специально для разработки видеоигр. Цель вовлечения игрока обычно достигается за счет тщательного проектирования игрового процесса и тонкой настройки на протяжении всего процесса разработки видеоигр. Эта книга посвящена проектированию и разработке игрового движка; основное внимание в ней будет уделено реализации и скрытию рутинных операций движка при одновременной поддержке множества сложных симуляций. С помощью проектов, описанных в этой книге, вы создадите практический
игровой движок для разработки видеоигр, доступных через Интернет. Игровой движок избавляет разработчиков игр от необходимости выполнять простые рутинные задачи, такие как расшифровка нажатий определенных клавиш на клавиатуре, разработка сложных алгоритмов для обычных операций, таких как имитация теней в 2D-мире, и понимание нюансов в реализациях, таких как обеспечение допусков точности физического моделирования. Коммерческие и хорошо зарекомендовавшие себя игровые движки, такие как Unity, Unreal
Engine и Panda3D представляют свои системы через графический пользовательский интерфейс (GUI).Дружественный графический интерфейс не только упрощает некоторые утомительные процессы игрового дизайна например, создание и размещение объектов на уровне, но, что более важно, это гарантирует, что эти игровые движки будут доступны креативным дизайнерам с различным опытом работы, которые могут найти специфику разработки программного обеспечения отвлекающей.

Эта книга посвящена основной функциональности игрового движка, независимого от графический интерфейс. В то время как комплексная система графического

интерфейса может улучшить работу конечного пользователя,
требования к реализации также могут отвлекать и усложнять основы игрового движка. Например, проблемы, касающиеся применения совместимых типов данных в системе пользовательского интерфейса, такие как ограничение объектов из определенного класса назначаться в качестве приемников теней, важны для дизайна графического интерфейса, но не имеют отношения к основной функциональности игрового движка.

Эта книга рассматривает разработку игрового движка с двух важных аспектов: программируемость и ремонтопригодность. Как библиотека программного обеспечения, интерфейс игрового движка должен способствовать программированию разработчиками игр с помощью хорошо абстрагированных служебных методов и объектов, которые скрывают простые рутинные задачи и поддерживают сложные, но распространенные операции. Как программная система, кодовая база игрового движка должна поддерживать ремонтопригодность с помощью хорошо спроектированной инфраструктуры и хорошо организованных систем исходного кода, которые обеспечивают повторное использование кода, постоянное обслуживание системы, улучшение и расширение.

В этой главе описывается технология внедрения и организация этой книги. Обсуждение проведет вас через этапы загрузки, установки и настройки среды разработки, поможет вам создать ваше первое приложение HTML5 и использует этот первый опыт разработки приложений, чтобы объяснить наилучший подход к чтению и изучению этой книги.

Технологии

Цель создания игрового движка, который позволяет играм быть доступными по всему миру Всемирная паутина поддерживается свободно доступными технологиями. JavaScript поддерживается практически всеми веб-браузерами, поскольку интерпретатор установлен почти на каждом персональном компьютере в мире. Как язык программирования, JavaScript динамически типизируется, поддерживает наследование и функционирует как первоклассные объекты,
и его легко изучать в хорошо зарекомендовавших себя сообществах пользователей и разработчиков. Благодаря стратегическому выбору этой технологии видеоигры, разработанные на основе JavaScript, могут быть
доступен любому пользователю через Интернет через соответствующие веб-браузеры. Таким образом, JavaScript является одним из лучших языков программирования для разработки видеоигр для широких масс. 6

В то время как JavaScript служит отличным инструментом для реализации игровой логики и алгоритмов, дополнительные технологии в виде программных библиотек или интерфейсов прикладного программирования (API) необходимы для поддержки

требований пользовательского ввода и вывода мультимедиа. С целью создания игр, доступных через Интернет через веб-браузеры, HTML5 и WebGL предоставляют идеальные дополнительные API ввода и вывода.

HTML5 предназначен для структурирования и представления контента в Интернете. Он включает в себя подробные модели обработки и связанные с ними API для обработки пользовательского ввода и мультимедийных выходных данных. Эти API являются родными для JavaScript и идеально подходят для реализации браузерных видеоигр. В то время как HTML5 предлагает базовую масштабируемую API векторной графики (SVG), он не поддерживает сложность, требуемую видеоиграми для таких эффектов, как освещение в реальном времени, взрывы или тени. Веб-графика Библиотека (WebGL) - это JavaScript API, разработанный специально для генерации 2D и 3D компьютерная графика через веб-браузеры. С его поддержкой затенения OpenGL Язык (GLSL) и возможность доступа к графическому процессору (GPU) на клиентских компьютерах, WebGL обладает способностью создавать очень сложные графические эффекты в режиме реального времени и идеально подходит в качестве графического API для браузерных видеоигр.

Эта книга о концепциях и разработке игрового движка, в котором JavaScript, HTML5 и WebGL являются просто инструментами для реализации. Обсуждение в этой книге сосредоточено на применении технологий для реализации требуемых реализаций и не пытается охватить детали технологий. Например, в игровом движке наследование реализовано с помощью функциональности класса JavaScript, которая основана на цепочке прототипов объектов; однако достоинства языков сценариев на основе прототипов не обсуждаются. Функциональные возможности звукового сигнала движка и фоновой музыки основаны на интерфейсе AudioContext HTML5, и все же диапазон его возможностей не описан. Объекты игрового движка отрисовываются на основе текстурных карт WebGL, в то время как возможности текстурной подсистемы WebGL не представлены. Специфика технологий отвлекла бы от обсуждения игрового движка. Ключевыми результатами изучения книги являются концепции и стратегии внедрения игрового движка, а не детали какой-либо из технологий. Таким образом, прочитав эту книгу, вы сможете построить аналогичную игровой движок, основанный на любом сопоставимом наборе технологий, таких как C# и MonoGame, Java и JOGL, C++ и Direct3D и так далее. Если вы хотите узнать больше о JavaScript, HTML5 или WebGL или освежить их в памяти, пожалуйста, обратитесь к ссылкам в разделе “Технологии”. раздел в конце этой главы.

Настройка вашей среды разработки

Игровой движок, который вы собираетесь создать, будет доступен через веб-браузеры, которые могут быть запущены в любой операционной системе (OS). Среда

разработки, которую вы собираетесь настроить, также не зависит от операционной системы. Для простоты следующие инструкции основаны на операционной системе Windows 10. Вы должны быть в состоянии воспроизвести аналогичную среду с незначительными изменениями в среде на базе Unix, такой как macOS или Ubuntu.

IDE: Все проекты в этой книге основаны на VS Code IDE. Вы можете скачать и установить программу с https://code .visualstudio.com /.

Среда выполнения: Вы будете выполнять свои игровые проекты ввеб-браузере Google Chrome. Вы можете скачать и установить этот браузер с www.google.com/chrome/browser /.

matrix math library: Это библиотека, которая реализует основополагающие математические операции. Вы можете скачать эту библиотеку с http://glMatrix.net/. Вы интегрируете эту библиотеку в свой игровой движок в главе 3, поэтому там будет предоставлена более подробная информация.

Обратите внимание, что нет никаких конкретных системных требований для поддержки языка программирования JavaScript, HTML5 или WebGL. Все эти технологии встроены в среду выполнения веб-браузера.

Примечание: как уже упоминалось, мы выбрали среду разработки на основе vS Code, потому что сочли ее наиболее удобной. существует множество других альтернатив, которые также бесплатны, включая, но не ограничиваясь ими, NetBeans, IntelliJ Idea, eclipse и Sublime.

Загрузка и установка средства проверки синтаксиса JavaScript

Мы обнаружили, что ESLint является эффективным инструментом для обнаружения потенциальных ошибок в исходном коде JavaScript. Вы можете интегрировать ESLint в

VS Code, выполнив следующие шаги:

- Перейти к https://marketplace.visualstudio.com/ `items?itemName=dbaeumer.vscode-eslint` и нажмите установить.
- Вам будет предложено открыть VS Code, и, возможно, потребуется снова нажать кнопку установить в приложении.

Ниже приведены некоторые полезные рекомендации по работе с клиентами:

- Инструкции по работе с ESLint см. https://eslint.org/docs/user-guide/
- Для получения подробной информации о том, как работает ESLint, см. https://eslint.org/docs/developer-guide/.

Загрузка и установка Live Server

Для запуска вашего игрового движка требуется расширение Live Server для VS Code. Он запускает веб-сервер локально на вашем компьютере с помощью VS Code для размещения разработанных игр. Подобно ESLint, вы можете установить Live Server, выполнив следующие действия:

- Перейти к https://marketplace.visualstudio.com/ предметы?название предмета= ритвик дей.Работающий сервер и нажмите установить.
- Вам будет предложено открыть VS Code, и, возможно, потребуется снова нажать кнопку установить в приложении.

Работа в среде разработки VS Code

С IDE VS Code легко работать, и для проектов, описанных в этой книге, требуется только редактор. Соответствующие файлы исходного кода, организованные в родительской папке, интерпретируются VS Cod как проект. Чтобы открыть проект, выберите Файл >> Открыть папку и перейдите и выберите родительскую папку, содержащую файлы исходного кода проекта. Как только проект открыт, вам необходимо ознакомиться с основными окнами VS Code, как показано в Рисунок 1-1.

- Окно проводника: В этом окне отображаются файлы исходного кода проекта. Если вы случайно закроете это окно, вы можете вызвать его, выбрав Просмотр >> Проводник.
- Окно редактора: В этом окне отображается исходный код вашего проекта, который вы можете редактировать. Вы можете выбрать файл исходного кода для работы, щелкнув один раз по соответствующему имени файла в окне Проводника.
- Окно вывода: Это окно не используется в наших проектах; не стесняйтесь закрыть его, нажав на значок “x” в правом верхнем углу окна.

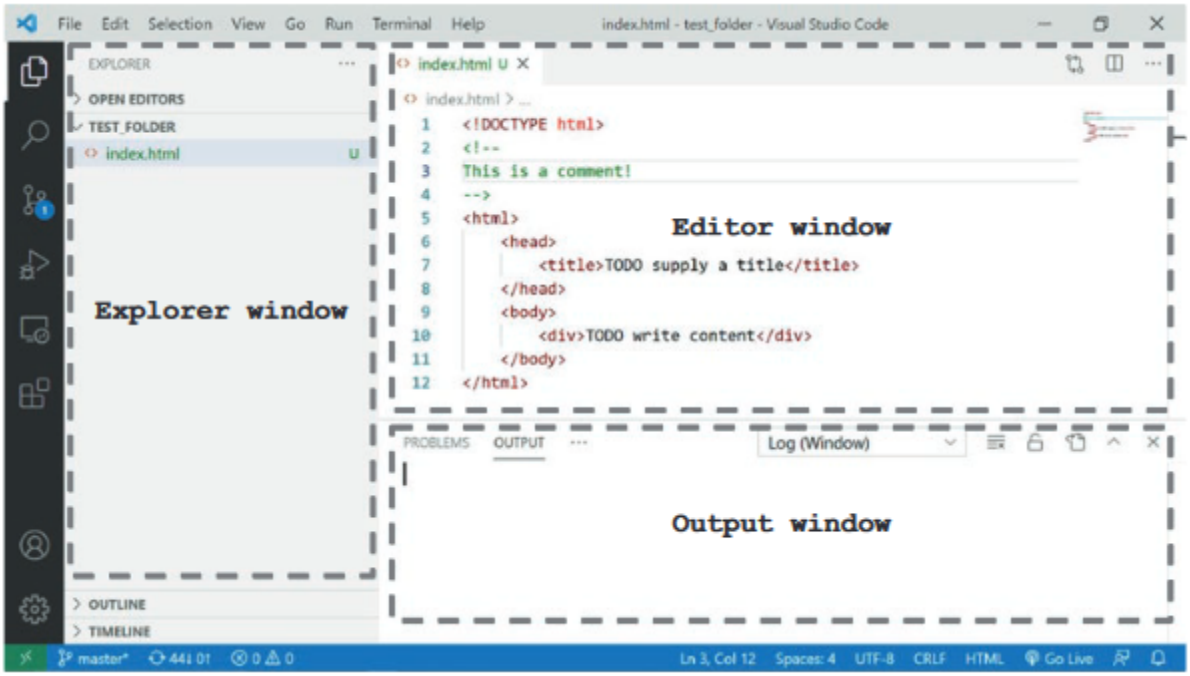

Рисунок 1–1: VS Code IDE

Создание проекта HTML5 в VS Code

Теперь вы готовы создать свой первый проект на HTML5:

• Используя проводник файлов, создайте каталог в том месте, где вы хотели бы сохранить свои проекты. Этот каталог будет содержать все файлы исходного кода, относящиеся к вашим проектам. В VS Code выберите Файл >> Открыть >> Папку и перейдите в созданный вами каталог.

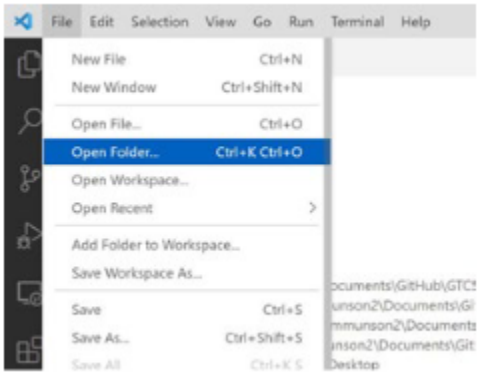

Рисунок 1–2. Открытие папки проекта

- VS Code откроет папку проекта. Ваша IDE должна выглядеть аналогично рисунку 1–3; обратите внимание, что окно проводника пусто, когда ваша папка проекта пуста.

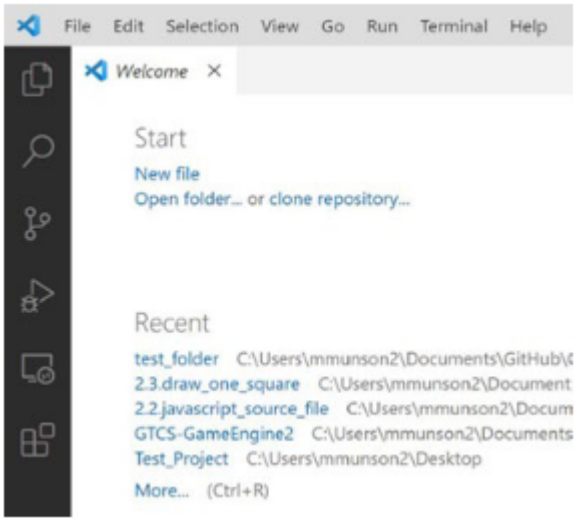

Рисунок 1–3. Пустой проект VS Code

- Теперь вы можете создать свой первый HTML-файл, index.html . Выберите Файл >> Новый файл и назовите его index.html . Это будет использоваться в качестве домашней или целевой страницы при запуске вашего приложения.

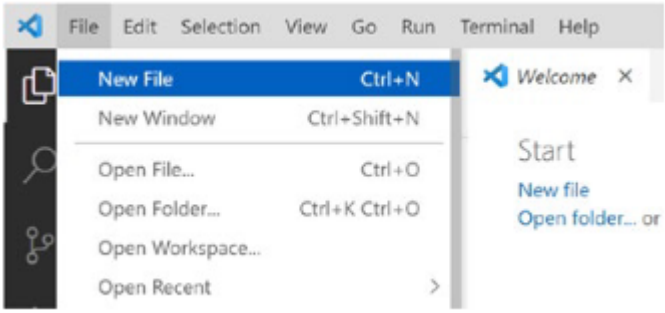

Рисунок 1–4. Создание файла index.html

- В окне редактора введите следующий текст в свой index.html:


```
1  <!DOCTYPE html>
2  <!--This is a comment!-->
3  <html>
4    <head>
5      <title>TODO supply a title</title>
6    </head>
7    <body>
8      <div>TODO write content</div>
9    </body>
10 </html>
```

В первой строке файл объявляется HTML-файлом. Блок, который следует внутри

теги <!-- и --> - это блок комментариев. Дополнительные теги **<html></html>** содержат весь HTML-код. В этом случае шаблон определяет разделы head и body. Заголовок задает заголовок веб-страницы, а в теле будет размещено всё содержимое веб-страницы.

Как показано на рисунке 1-5, вы можете запустить этот проект, нажав кнопку "Перейти к работе" в правом нижнем углу вашего VS-кода или нажав **Alt+L Alt+O**. Есть шанс что сразу после того, как вы впервые ввели предыдущий HTML-код, кнопка "Перейти в режим реального времени" может не появиться. В этом случае просто щелкните правой кнопкой мыши index.html файл в окне проводника и нажмите пункт меню "Открыть с помощью **Live Server**", чтобы запустить веб-страницу. После первого запуска в правой нижней части IDE появится кнопка "Перейти в режим реального времени", как показано на рисунке 1-5.

Рисунок **1–5**.Нажмите кнопку Go Live что бы запустить проект.

Примечание: для запуска проекта необходимо index.html файл этого проекта должен быть открыт в редакторе при нажатии кнопки "go live" или при вводе клавиш alt+l alt+o. Это станет важным в последующих главах, когда появятся другие Файлы исходного кода JavaScript в проекте.

На рисунке 1-6 показан пример того, как выглядит проект по умолчанию, когда вы его запускаете. Обратите внимание, что после запуска проекта кнопка "Перейти в режим реального времени" обновляет свою метку, чтобы показать "Порт:**5500**". Вы можете нажать эту кнопку еще раз, чтобы отключить IDE от веб-страницы и снова увидеть надпись "Go Live". Нажатие на кнопку еще раз приведет к повторному запуску проекта.

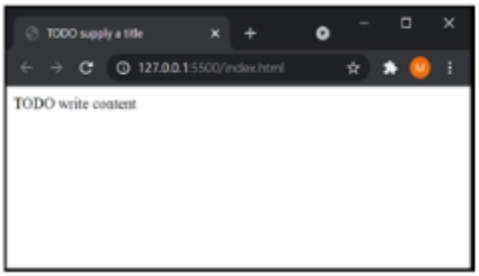

Рисунок 1–6. Запуск простого проекта HTML5

Чтобы остановить программу, просто закройте веб-страницу. Вы успешно запустили свой первый проект на HTML5. Благодаря разработке этого очень простого проекта вы ознакомились со средой IDE.

Примечание: Для отладки мы рекомендуем использовать инструменты разработчика Chrome. доступ к этим инструментам можно получить, набрав **Ctrl+Shift +I** (или клавишу F12) в окне браузера во время запуска вашего проекта. чтобы узнать больше об этих инструментах, пожалуйста, обратитесь к https://developer.chrome.com/docs/devtools /.

Как пользоваться этой книгой

Эта книга проведет вас через разработку игрового движка путем создания проектов, подобных тому, с которым вы только что столкнулись. Каждая глава посвящена важному компоненту типичного игрового движка, а разделы в каждой главе описывают важные концепции и проекты реализации, которые создают соответствующий компонент. На протяжении всего текста проект из каждого раздела основывается на результатах предшествующих ему проектов. Хотя это несколько затрудняет ознакомление с книгой, это даст вам практический опыт и четкое понимание того, как различные концепции взаимосвязаны. Кроме того, вместо того, чтобы постоянно работать с новыми и минималистичными проектами, вы приобретаете опыт создания более крупных и интересных проектов, одновременно интегрируя новые функциональные возможности в свой расширяющийся игровой движок.

Проекты начинаются с демонстрации простых концепций, таких как рисование простого квадрата, но быстро развиваются в представление более сложных концепций, таких как работа с пользовательскими системами координат и реализация обнаружения столкновений с точностью до пикселя. Первоначально, как вы уже испытали при создании первого приложения HTML5, вы будете ознакомлены с подробными шагами и полным списком исходного кода. По мере того, как вы знакомитесь со средой разработки и технологиями, руководства и списки исходного кода, сопровождающие каждый проект, будут меняться, чтобы

акцентировать внимание на важных деталях реализации. В конечном счете, по мере увеличения сложности проектов, обсуждение будет сосредоточено только на жизненно важных и релевантных вопросах, в то время как простые изменения исходного кода упоминаться не будут.

Окончательная кодовая база, которую вы будете постепенно разрабатывать на протяжении книги, представляет собой законченный и практичный игровой движок; это отличная платформа, на которой вы можете начать создавать свои собственные 2D-игры. Это именно то, что делает последняя глава книги
, ведущая вас от концептуализации к дизайну и реализации казуальной 2D-игры.

Есть несколько способов, которыми вы можете воспользоваться вместе с этой книгой. Самый очевидный -ввести код в свой проект, следуя каждому шагу в книге. С точки зрения обучения, это наиболее эффективный способ усвоения представленной информации; однако мы понимаем, что он может быть не самым реалистичным из-за объема кода или для отладки этого подхода может потребоваться. В качестве альтернативы, мы рекомендуем вам запустить и
изучить исходный код завершенного проекта, когда вы начинаете новый раздел. Это позволяет вам просмотреть проект текущего раздела, дает вам четкое представление о конечной цели и позволяет увидеть, чего пытается достичь проект. Вы также можете счесть завершенный код проекта полезным, когда у вас возникнут проблемы при самостоятельном создании кода, потому что во время сложных ситуаций отладки вы можете сравнить свой код с кодом
завершенного проекта.

Примечание: мы нашли программу WinMerge (http://winmerge.org/), чтобы быть отличным инструментом для сравнения файлов и папок с исходным кодом. пользователи Mac могут воспользоваться утилитой Filemerge для аналогичной цели.

Наконец, после завершения проекта мы рекомендуем вам сравнить поведение вашей реализации с предоставленной завершенной реализацией. Поступая таким образом, вы можете наблюдать, ведет ли ваш код себя так, как ожидалось.

Как вы создаете отличную видеоигру?

Хотя основное внимание в этой книге уделяется проектированию и внедрению

игрового движка, важно оценить, как различные компоненты могут способствовать созданию веселой и увлекательной видеоигры. Начиная с главы 4, “Соображения по игровому дизайну” раздел включен в конце каждой главы, чтобы связать функциональность компонента движка с элементами игрового дизайна. В этом разделе представлены рамки для этих обсуждений.

Это сложный вопрос, и нет точной формулы для создания видеоигры, которая людям понравится играть, точно так же, как нет точной формулы для создания фильма, который люди будут любить смотреть. Мы все видели высокобюджетные фильмы, которые выглядят великолепно и демонстрируют лучшие актерские, писательские и режиссерские таланты, взрывающиеся в прокате, и мы все видели высокобюджетные игры от крупных студий, которые не смогли поразить воображение игроков. Точно так же фильмы неизвестных режиссеров могут привлечь внимание всего мира, а игры от небольших неизвестных студий могут штурмом захватить рынок.

Хотя не существует четких инструкций по созданию отличной игры, ряд элементов работайте вместе в гармонии, чтобы создать конечный опыт, превосходящий сумму его частей, и все геймдизайнеры должны успешно решать каждую из них, чтобы создать что-то, во что стоит поиграть. Эти элементы включают в себя следующее:

- Технический дизайн: он включает в себя весь игровой код и игровую платформу и, как правило, не предоставляется игрокам напрямую; скорее, он формирует основу для всех аспектов игрового процесса. Эта книга в первую очередь сосредоточена на вопросах, связанных с техническим дизайном игр, включая конкретные задачи, такие как строки кода, необходимые для рисования элементов на экране, и другие архитектурные соображения, такие как определение стратегии того, как и когда загружать ресурсы в память. Технические проблемы дизайна влияют на игровой опыт многими способами (например, количество раз, когда игрок испытывает задержки “загрузки” во время игры или количество кадров в секунду, отображаемых в игре), но технический дизайн обычно невидим для игроков, поскольку он работает под так называемым презентационным слоем или всей аудиовизуальной и / или тактильной обратной связью, с которой игрок сталкивается во время игры.

- Игровая механика : Игровая механика – это абстрактное описание того, что можно назвать основой игры для данного игрового процесса. Типы игровой механики включают головоломки,испытания на ловкость, такие как прыжки или прицеливание, временные события, боевые столкновения и тому подобное. Игровая механика – это фреймворк; конкретные головоломки, встречи и игровые взаимодействия являются реализациями фреймворка.

Стратегия в реальном времени (RTS) игра может включать в себя механику сбора ресурсов, например, где механика может быть описана как "Игроки должны собирать определенные типы ресурсов и комбинировать их для создания юнитов, которые они могут использовать в бою". Конкретная реализация этой механики (как игроки находят и извлекают ресурсы в игре, как они транспортируют их из одного места в другое и правила объединения ресурсов для создания юнитов) является аспектом проектирования систем, дизайн уровней и модель взаимодействия / игровой цикл (описано далее в этом разделе).

• Проектирование систем: внутренние правила и логические взаимосвязи, которые обеспечивают структурированный вызов основной игровой механике, относятся к системному дизайну игры. Используя предыдущий пример RTS, игра может потребовать, чтобы игроки собрали определенное количество металлической руды и объединили ее с определенным количеством древесины, чтобы создать игровой объект; конкретные правила, определяющие, сколько каждого ресурса требуется для изготовления объектов, и уникальный процесс создания объектов (например, объекты могут быть созданы только в определенных структурах на базе игрока и появляются через x минут после того, как игрок запускает процесс) являются аспектами проектирования систем. Казуальные игры могут имеют базовые конструкции систем. Простая игра-головоломка, такая как Pull the Pin от Popcore Games, например, - это игра с небольшим количеством систем и низкой сложностью, в то время как игры крупных жанров, такие как RTS, могут иметь глубоко сложные и взаимосвязанные системные проекты, созданные и сбалансированные целыми командами дизайнеров. Дизайн игровых систем часто является тем местом, где существует самая скрытая сложность игрового дизайна; когда дизайнеры проходят через упражнение по определению всех переменных, которые способствуют реализации игровой механики, легко потеряться в море сложностей и зависимостей баланса.Системы, которые кажутся игрокам довольно простыми, могут потребовать, чтобы множество компонентов работали вместе и идеально сбалансированы друг с другом, и недооценка сложности системы, возможно, является одной из самых больших ошибок, с которыми сталкиваются начинающие (и опытные!) разработчики игр.

Пока вы не поймете , во что ввязываетесь, всегда предполагайте, что создаваемые вами системы окажутся значительно сложнее, чем вы ожидаете.

• Дизайн уровней: Дизайн уровней игры отражает специфические способы сочетания каждого из восьми других элементов в контексте отдельных "фрагментов" игрового процесса, где игроки должны выполнить определенный набор задач, прежде чем перейти к следующему разделу (в некоторых играх может быть

только один уровень, в то время как в других их будет десятки). Конструкции уровней в рамках одной игры все могут быть вариациями одной и той же базовой механики и системного дизайна (такие игры, как Tetris и Bejeweled, являются примерами игр со многими уровнями, все из которых сосредоточены на одной и той же механике), в то время как другие игры будут сочетать механику и дизайн систем для разнообразия уровней. В большинстве игр используется одна основная механика и подход к проектированию систем, охватывающий всю игру, и будут добавлены незначительные вариации между уровнями, чтобы сохранить ощущение свежести (изменение окружения, изменение сложности, добавление ограничений по времени, увеличение сложности и тому подобное), хотя иногда игры вводят новые уровни, которые основаны на совершенно разных механиках и системы , позволяющие удивлять игроков и удерживать их интерес. Отличный дизайн уровней в играх – это баланс между созданием "фрагментов" игры, которые демонстрируют механика и системы разрабатываются и достаточно изменяются между этими фрагментами, чтобы поддерживать интерес игроков
по мере прохождения игры (но не настолько сильно меняются между фрагментами, чтобы игровой процесс казался бессвязным).

• Модель взаимодействия: Модель взаимодействия – это комбинация клавиш, кнопок, джойстиков контроллера, сенсорных жестов и так далее, используемых для взаимодействия с игрой для выполнения задач, а также графические пользовательские интерфейсы, которые поддерживают эти взаимодействия в игровом мире. Некоторые теоретики игр выделяют дизайн пользовательского интерфейса игры в отдельную категорию (пользовательский интерфейс игры включает в себя такие элементы, как дизайн меню, описи товаров, информационные табло [HUD]), но модель взаимодействия тесно связана с дизайном пользовательского интерфейса, и хорошей практикой является рассмотрение этих два элемента как неразделимые.В случае игры RTS, упоминавшейся ранее, модель взаимодействия включает действия, необходимые для выбора объектов в игре, перемещения этих объектов, открытия меню и управления запасами, сохранения прогресса, инициирования боя и постановки задач сборки в очередь. Модель взаимодействия полностью независима от механики и дизайна систем и касается только физических действий, которые игрок должен предпринять, чтобы инициировать поведение (например, щелкнуть кнопкой мыши, нажать клавишу, переместить джойстик, колесо прокрутки); пользовательский интерфейс – это аудиовизуальная или тактильная обратная связь, подключенная к этим действиям (экранные кнопки, меню, статусы, звуковые сигналы, вибрации и тому подобное).

• Обстановка игры: Вы находитесь на чужой планете? В мире фантазий? В

абстрактной среде? Игровая обстановка является важнейшей частью игрового процесса и в партнерстве с аудиовизуальным дизайном превращает то, что в противном случае было бы несвязанным набором базовых взаимодействий , в увлекательный опыт с учетом контекста. Настройки игры не обязательно должны быть сложными, чтобы быть эффективными; неизменно популярная игра-головоломка Tetris имеет довольно простой сеттинг без реальной повествовательной оболочки, но сочетание абстрактного сеттинга, аудиовизуального оформления и дизайна уровней уникально хорошо подобран и вносит значительный вклад в миллионы часов, которые игроки вкладывают в этот опыт год за годом.

• Визуальный дизайн: Видеоигры существуют в основном в визуальной среде, поэтому неудивительно, что компании часто тратят на визуальный дизайн своих игр столько же или даже больше, сколько на техническое выполнение кода. Большие игры представляют собой совокупность тысяч визуальных ресурсов, включая окружение, персонажей, объекты, анимацию и кинематографию; даже небольшие казуальные игры обычно поставляются с сотнями или тысячами отдельных визуальных элементов. Каждый объект, с которым игрок взаимодействует в игре, должен быть уникальным активом, и если этот актив включает в себя более сложную анимацию, чем просто перемещение его
при перемещении из одного места на экране в другое или изменении масштаба или непрозрачности объект, скорее всего, потребуется анимировать художнику.
Игровая графика не обязательно должна быть фотореалистичной или стилистически продуманной , чтобы быть визуально превосходной или эффективно представлять обстановку (многие игры намеренно используют упрощенный визуальный стиль), но в лучших играх художественное направление и визуальный стиль являются ключевыми для игрока впечатлений, а визуальный выбор будет преднамеренным и хорошо подобранным к игровому сеттингу и механике.

• Звуковое оформление: сюда входят музыка и звуковые эффекты, окружающие фоновые звуки и все звуки, связанные с действиями игрока (выбор / использование / замена элемента, открытие инвентаря, вызов меню и тому подобное). Дизайн 18 звука работает рука об руку с визуальным дизайном, передавая и усиливая игровую обстановку, и многие начинающие дизайнеры значительно
недооценивают влияние звука на погружение игроков в игровые миры. Представьте себе "Звездные войны", например, без музыки, звукового эффекта светового меча, дыхания Дарта Вейдера или характерных звуковых сигналов R2D2; звуковые эффекты и музыкальное сопровождение столь же фундаментальны для воспринимайте как визуальные эффекты.

• Мета-игра: Мета-игра сосредоточена на том, как индивидуальные цели

объединяются, чтобы продвинуть игроков в игровом процессе (часто с помощью подсчета очков, последовательного открытия отдельных уровней, прохождения повествования и тому подобного). Во многих современных играх мета-игра представляет собой повествовательную дугу или историю; игроки часто не получают "оценку" как таковую, а скорее раскрывают линейную или полулинейную историю по мере прохождения игровых уровней, продвигаясь вперед, чтобы завершить историю. Другие игры (особенно социальные и соревновательные игры) предполагают "прокачку игроков прокачивать" своих персонажей, что может произойти в результате прохождения повествовательного процесса, охватывающего всю игру, или просто погружения в игровой мир и выполнения индивидуальных заданий, которые дают персонажам очки опыта. Другие игры, конечно, по-прежнему сосредоточены на том, чтобы набирать очки или выигрывать раунды у других игроков.

Магия видеоигр обычно возникает из взаимодействия между этими девятью элементами, и самые успешные игры тонко балансируют каждый из них как часть единого видения, чтобы обеспечить гармоничный опыт; этот баланс всегда будет уникальным для каждого отдельного проекта и встречается в играх, начиная от Animal Crossing от Nintendo и заканчивая Red Dead Redemption 2 от Rockstar. Основная игровая механика во многих успешных играх часто представляет собой вариацию на одну или несколько довольно простых, распространенных тем (например, Pull the Pin - это игра, полностью основанная на извлечении виртуальных кеглей из контейнера для освобождения цветные шары), но визуальный дизайн, повествовательный контекст, звуковые эффекты, взаимодействия и система прогрессии работают вместе с игровой механикой, создавая уникальный опыт , который значительно более увлекателен, чем сумма его отдельных частей, заставляя игроков хотеть возвращаться к нему снова и снова. Отличные игры варьируются от простых до сложных, но все они отличаются элегантным балансом вспомогательных элементов дизайна.

Рекомендации

Примеры в этой книге созданы исходя из предположения, что вы понимаете инкапсуляцию данных, наследование и базовые структуры данных, такие как связанные списки и словари, и вам удобно работать с основами алгебры и геометрии, особенно с линейными уравнениями и системами координат. Многие примеры в этой книге применяют и реализуют концепции компьютерной графики и линейной алгебры. Эти концепции требуют гораздо более углубленного изучения. Заинтересованные читатели могут узнать больше об этих темах из других книг.

- Компьютерная графика:
  - Маршнер и Ширли. Основы компьютерной графики, 4-е издание. CRC Press, 2016.
  - Ангел и Шрайнер. Интерактивная компьютерная графика: Сверху вниз Подход с помощью WebGL, 7-е издание. Pearson Education, 2014.
- Линейная алгебра:
  - Сун и Смит. Базовая математика для разработки игр с Unity 3D: Руководство для начинающих по математическим основам. Апрель 2019 года.
  - Джонсон, Рисс и Арнольд. Введение в линейную алгебру, 5 -е издание. Эддисон-Уэсли, 2002.
  - Антон и Роррес. Элементарная линейная алгебра: Приложения Версия, 11-е издание. Уайли, 2013.

Технологии

В следующем списке приведены ссылки для получения дополнительной информации о технологиях, используемых
в этой книге:

- **JavaScript**: `www.w3schools.com/js`
- **HTML5:** `www.w3schools.com/html/html5_intro.asp`
- **WebGL:** `www.khronos.org/webgl`
- **OpenGL:** `www.opengl.org`
- **Visual Studio Code:** `https://code.visualstudio.com/`
- **Chrome:** `www.google.com/chrome`
- **glMatrix:** `http://glMatrix.net`
- **ESLint:** `www.eslint.org`

Глава 2
Работа с HTML5 и WebGL

После завершения этой главы вы сможете

- Создайте новый файл исходного кода JavaScript для вашего простого игрового движка
- Нарисуйте простой квадрат постоянного цвета с помощью WebGL
- Определите модули и классы JavaScript для инкапсуляции и реализации основных функций игрового движка.
- Оцените важность абстракции и организации структуры вашего исходного кода

для поддержки роста сложности.

Вступление

Рисование - одна из наиболее важных функций, общих для всех видеоигр. Игровой движок должен предлагать гибкий и удобный для программиста интерфейс к своей системе рисования. Таким образом, при создании игры дизайнеры и разработчики могут сосредоточиться на важных аспектах самой игры, таких как механика, логика и эстетика.

WebGL - это современный графический интерфейс прикладного программирования на JavaScript (API) разработан для приложений на базе веб–браузера, который обеспечивает качество и эффективность благодаря прямому доступу к графическому оборудованию. По этим причинам WebGL служит отличной базой для поддержки рисования в игровом движке, особенно для видеоигр, предназначенных для воспроизведения через Интернет.

В этой главе рассматриваются основы рисования с помощью WebGL, разрабатываются абстракции для инкапсуляции несущественных деталей для облегчения программирования и создается базовая инфраструктура для организации сложной системы исходного кода для поддержки будущего расширения.

Примечание. Игровой движок, который вы разработаете в этой книге, основан на последней версии спецификации WebGL: версии 2.0. Для краткости термин WebGL будет использоваться для обозначения этого API.

Холст для рисования

Чтобы рисовать, вы должны сначала определить и выделить область внутри веб-страницы. Вы можете легко достичь этого, используя HTMLCanvasElement для определения области для рисования WebGL. Элемент canvas - это контейнер для рисования, к которому вы можете получить доступ и которым можно манипулировать с помощью JavaScript.

Проект HTML5 Canvas

Этот проект демонстрирует, как создать и очистить элемент canvas на веб -странице. На рисунке 2-1 показан пример запуска этого проекта, который определен в папке chapter2/2.1.html5_canvas.

Рисунок 2-1.Запуск проекта HTML5 Canvas

Цели проекта заключаются в следующем:

- Чтобы узнать, как настроить элемент HTML canvas
- Чтобы узнать, как извлечь элемент canvas из HTML-документа для использования в JavaScript
- Чтобы узнать, как создать ссылочный контекст для WebGL из извлеченного элемента canvas и управлять canvas с помощью контекста WebGL.

Создание и очистка HTML-холста

В этом первом проекте вы создадите пустой холст HTML5 и очистите холст до определенного цвета с помощью WebGL:

1. Создайте новый проект, создав новую папку с именем **html5_ canvas** в выбранном вами каталоге и скопируйте и вставьте **index.html** файл, который вы создали в предыдущем проекте в главе 1.

Примечание. С этого момента, когда вас попросят создать новый проект, вы должны следовать процессу, описанному ранее. То есть создайте новую папку с именем проекта и скопируйте /вставьте файлы предыдущего проекта. Таким образом, ваши новые проекты могут дополнять ваши старые, сохраняя при этом первоначальную функциональность.

---

2. Откройте **index.html** файл в редакторе, открыв **html5_canvas** папку, развернув ее при необходимости и нажав на **index.html** файл, как показано на рисунке 2-2.

Рисунок 2-2.Редактирование index.html файл в вашем проекте

3. Создайте HTML-холст для рисования, добавив следующие строки в **index.html** файл внутри элемента **body**:

```
<canvas id="GLCanvas" width="640" height="480">
<!-- Your browser does not support the HTML5 canvas. -->
</canvas>
```

Код определяет элемент canvas с именем GLCanvas с указанными атрибутами width и height. Как вы узнаете позже, вы получите ссылку на GLCanvas для рисования в этой области. Текст внутри элемента будет отображаться, если ваш браузер не поддерживает рисование с помощью WebGL.

Примечание. Строки между тегами **<body>** и **</body>** обозначаются как **“внутри элемента body”**. В остальной части этой книги **“внутри элемента AnyTag”** будет использоваться для обозначения любой строки между началом (**<AnyTag>**) и концом (**</AnyTag>**) элемента.

---

4. Создайте элемент **script** для включения программного кода JavaScript, еще раз внутри элемента **body**:

```
<script type="text/javascript">
  // JavaScript code goes here.
</script>
```

Это позаботится о HTML-части этого проекта. Теперь вы напишете JavaScript-код для остальной части примера:

5. Извлеките ссылку на GLCanvas в коде JavaScript, добавив следующую строку в элемент script:

```
"use strict";
let canvas = document.getElementById("GLCanvas");
```

Примечание ключевое слово **let** в JavaScript определяет переменные.

Первая строка, **“use strict”**, представляет собой директиву JavaScript, указывающую, что код должен выполняться в “строгом режиме”, где использование необъявленных переменных является ошибкой времени выполнения. Вторая строка создает новую

переменную с именем canvas и ссылается на переменную в области рисования Canvas.

Примечание Все имена локальных переменных начинаются со строчной буквы, как в canvas

---

6. Извлеките и привяжите ссылку на контекст WebGL к области рисования, добавив следующий код:

```
let gl = canvas.getContext("webgl2") ||
         canvas.getContext("experimental-webgl2");
```

Как указывает код, полученная ссылка на контекст WebGL версии 2 хранится в локальной переменной с именем gl. Из этой переменной у вас есть доступ ко всем функциональным возможностям WebGL 2.0. Еще раз, в остальной части этой книги термин WebGL будет использоваться для обозначения WebGL версии 2.0 API.

7. Очистите область рисования на холсте до вашего любимого цвета с помощью WebGL, добавив следующее:

```
if (gl !== null) {
  gl.clearColor(0.0, 0.8, 0.0, 1.0);
  gl.clear(gl.COLOR_BUFFER_BIT);
}
```

Этот код проверяет правильность извлечения контекста WebGL, устанавливает чистый цвет и очищает область рисования. Обратите внимание, что цвет очистки задан в формате RGBA со значениями с плавающей запятой в диапазоне от 0.0 до 1.0. Четвертое число в формате RGBA - это альфа-канал. Вы узнаете больше об альфа-канале в последующих главах. На данный момент всегда присваивайте 1.0 альфа-каналу. Указанный цвет, (0.0,0.8, 0.0, 1.0), имеет нулевые значения для красного и синего каналов и 0,8, или 80 процентов, интенсивность на зеленом канале. По этой причине область холста очищается до светло-зеленого цвета.

8. Добавьте в документ простую команду записи для идентификации холста, вставив следующую строку:

```
document.write("<br><b>The above is WebGL draw area!</b>");
```

Вы можете обратиться к окончательному исходному коду в **index.html** файл в проекте главы **2/2.1.html5_canvas**. Запустите проект, и вы должны увидеть светло-зеленую область в окне вашего браузера, как показано на рисунке 2-1. Это область рисования на холсте размером 640×480, которую вы определили. Вы можете попробовать изменить очищенный цвет на белый, установив RGBA **gl.clearColor()** равным 1, или на черный, установив цвет равным 0 и оставив альфа-значение 1. Обратите внимание, что если вы установите альфа-канал равным 0, цвет холста исчезнет. Это связано с тем, что значение 0 в альфа-канале представляет полную прозрачность, и, таким образом, вы будете “видеть насквозь” холст и наблюдать за цветом фона веб-страницы.

Вы также можете попробовать изменить разрешение холста, изменив значения 640 × 480 на любое число, которое вам нравится. Обратите внимание, что эти два числа относятся к количеству пикселей и, следовательно, всегда должны быть целыми числами.

Разделение HTML и JavaScript

В предыдущем проекте вы создали элемент HTML canvas и очистили область, определенную canvas, с помощью WebGL. Обратите внимание, что вся функциональность сгруппирована в index.html файл. По мере увеличения сложности проекта такая кластеризация функциональных возможностей может быстро стать неуправляемой и негативно сказаться на программируемости вашей системы. По этой причине на протяжении всего процесса разработки в этой книге, после представления концепции, усилия будут направлены на разделение связанного исходного кода на либо четко определенные файлы исходного кода, либо классы в объектно-ориентированном стиле программирования. Чтобы начать этот процесс, исходный код HTML и JavaScript из предыдущего проекта будет разделен на разные файлы исходного кода.

Проект исходного файла JavaScript

Этот проект демонстрирует, как логически разделить исходный код на соответствующие файлы. Вы можете достичь этого, создав отдельный файл исходного кода JavaScript с именем core.js который реализует соответствующую функциональность в **index.html** файл. Веб-страница загрузит исходный код JavaScript в соответствии с инструкциями кода в **index.html** файл. Как показано на рисунке 2-3, этот проект при запуске выглядит идентично предыдущему проекту. Исходный код этого проекта находится в папке **chapter 2/2.2.javascript_source_file**.

Рисунок 2-3 Запуск проекта исходного файла JavaScript

Цели проекта заключаются в следующем:

- Чтобы узнать, как разделить исходный код на разные файлы
- Организовать ваш код в логическую структуру

Отдельный файл исходного кода JavaScript

В этом разделе подробно описывается, как создать и отредактировать новый файл исходного кода JavaScript. Вам следует ознакомиться с этим процессом, поскольку на протяжении всей этой книги вы будете создавать множество файлов исходного кода.

1. Создайте новый проект HTML5 под названием **javascript_source_file**. Напомним, что новый проект создается путем создания папки с соответствующим именем, копирования файлов из предыдущего проекта и редактирования элемента **<Title>** в **index.html** чтобы отразить новый проект.
2. Создайте новую папку с именем **src** внутри папки проекта, щелкнув значок новой папки при наведении курсора мыши на папку проекта, как показано на рисунке 2-4. Эта папка будет содержать весь ваш исходный код.

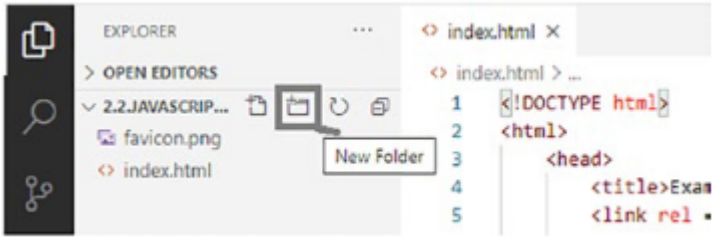

Рисунок 2-4 Создание новой папки с исходным кодом

3. Создайте новый файл исходного кода в папке **src**, щелкнув правой кнопкой мыши папку **src**, как показано на рисунке 2-5. Назовите новый исходный файл **core.js .**

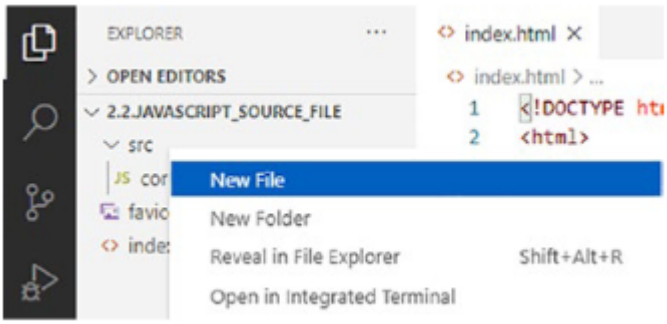

Рисунок 2-5 Добавление нового файла исходного кода JavaScript

---

Примечание. В VS Code вы можете создавать/копировать/переименовывать папки и файлы с помощью контекстных меню в окне проводника.

---

4. Откройте новый **core.js** исходный файл для редактирования.
5. Определите переменную для ссылки на контекст WebGL и добавьте функцию, которая позволяет вам получить доступ к переменной:

```
"use strict";
let mGL = null;
function getGL() { return mGL; }
```

Примечание Переменные, доступные по всему файлу или модулю, имеют имена, начинающиеся со строчной буквы “**m**”, как в **mGL**.

---

6. Определите функцию **initWebGL()** для извлечения **GLCanvas**, передав соответствующий идентификатор canvas в качестве параметра, привяжите область рисования к контексту WebGL, сохраните результаты в определенной переменной **mGL** и очистите область рисования:

```
function initWebGL(htmlCanvasID) {
  let canvas = document.getElementById(htmlCanvasID);
  mGL = canvas.getContext("webgl2") ||
        canvas.getContext("experimental-webgl2");
  if (mGL === null) {
    document.write("<br><b>WebGL 2 is not supported!</b>");
    return;
  }
  mGL.clearColor(0.0, 0.8, 0.0, 1.0);
}
```

Обратите внимание, что эта функция похожа на исходный код JavaScript, который вы ввели в предыдущем проекте. Это связано с тем, что в данном случае все, что вы делаете по-другому, это отделяете исходный код JavaScript от HTML-кода.

Примечание Все имена функций начинаются со строчной буквы, как в **initWebGL().**

---

7. Определите функцию **ClearCanvas()** для вызова контекста WebGL для очистки области рисования canvas:

```
function clearCanvas() {
  mGL.clear(mGL.COLOR_BUFFER_BIT);
}
```

8. Определите функцию для выполнения инициализации и очистки области canvas после того, как веб-браузер завершит загрузку index.html файл:

```
window.onload = function() {
  initWebGL("GLCanvas");
  clearCanvas();
}
```

Загрузите и запустите исходный код JavaScript из index.html

Со всеми функциональными возможностями JavaScript, определенными в core.js файл, теперь вам нужно загрузить этот файл для работы на вашей веб-странице через index.html файл:

1. Откройте index.html файл для редактирования.

2. Создайте HTML-холст GLCanvas, как в предыдущем проекте.
3. Загрузите core.js исходный код, включив следующий код внутри элемента **head**:

```
<script type="module" src="js/core.js"></script>
```

С помощью этого кода, **core.js** файл будет загружен как часть **index.html** определенная веб-страница. Напомним, что вы определили функцию для **window.onload**, и эта функция будет вызвана при загрузке index.html завершено.

Вы можете обратиться к окончательному исходному коду в **core.js** и **index.html** файлы в папке проекта **chapter 2/2.2.javascript_source_file**. Хотя результаты этого проекта идентичны результатам предыдущего проекта, организация вашего кода позволит вам расширять, отлаживать и понимать игровой движок по мере добавления новых функциональных возможностей.

Примечание Напомним, что для запуска проекта вы нажимаете кнопку **“Перейти к работе”** в правом нижнем углу окна VS Code или вводите клавиши **Alt+L Alt +O**, в то время как соответствующий index.html файл открывается в окне редактора. В этом случае проект не запустится, если вы нажмете кнопку **“Перейти в режим реального времени”**, в то время как **core.js** файл открывается в окне редактора.

Наблюдения

Изучите свой index.html файл внимательно и сравните его содержимое с тем же файлом из предыдущего проекта. Вы заметите, что index.html файл из предыдущего проекта содержит два типа информации (HTML и JavaScript-код) и что тот же файл из этого проекта содержит только первый, при этом весь JavaScript-код извлекается в **core.js** . Такое четкое разделение информации позволяет легко понять исходный код и улучшает поддержку более сложных систем. С этого момента весь исходный код JavaScript будет добавлен в отдельные файлы исходного кода.

Элементарное рисование с помощью WebGL

Как правило, рисование включает в себя геометрические данные и инструкции по обработке этих данных. В случае WebGL инструкции по обработке данных указаны на языке затенения OpenGL (GLSL) и называются шейдерами. Чтобы рисовать с помощью WebGL, программисты должны определить геометрические данные и шейдеры GLSL в процессоре и загрузить оба в аппаратное обеспечение для рисования или графический процессор (GPU). Этот процесс включает в себя значительное количество вызовов функций WebGL. В этом разделе подробно представлены этапы рисования WebGL.

Важно сосредоточиться на изучении этих основных шагов и не отвлекаться на менее важные нюансы настройки WebGL, чтобы вы могли продолжать изучать общие концепции, используемые при создании вашего игрового движка.

В следующем проекте вы узнаете о рисовании с помощью WebGL, сосредоточившись на самых элементарных операциях. Это включает в себя загрузку простой геометрии квадрата из центрального процессора в графический процессор, создание шейдера постоянного цвета и основные инструкции по рисованию простого квадрата с двумя треугольниками.

Проект "Нарисуй один квадрат"

Этот проект проведет вас через шаги, необходимые для рисования одного квадрата на холсте. На рисунке 2-6 показан пример запуска этого проекта, который определен в главе 2/2.3.папка **draw_one_square.**

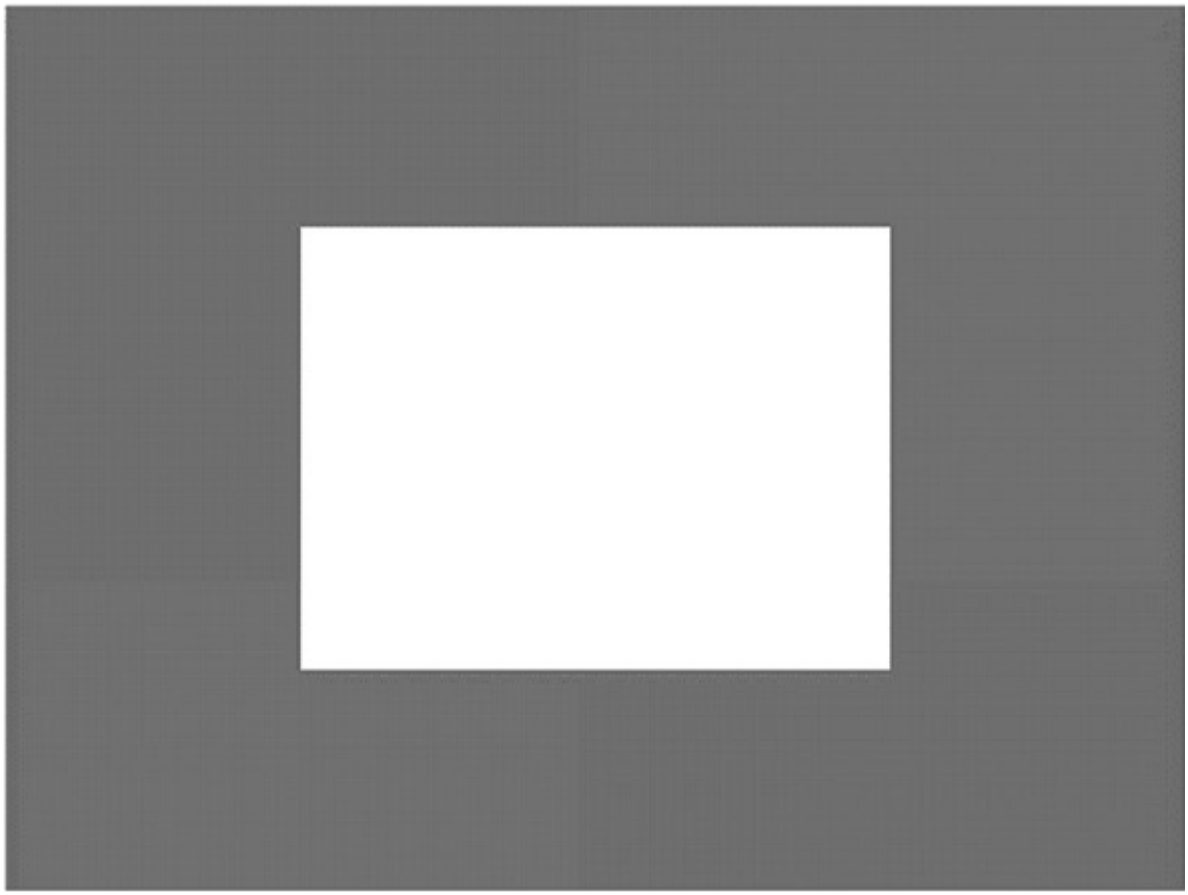

Рисунок 2-6 Запуск проекта "Нарисуй один квадрат"

Цели проекта заключаются в следующем:

- Чтобы понять, как загружать геометрические данные в графический процессор
- Чтобы узнать о простых шейдерах GLSL для рисования с помощью WebGL
- Чтобы узнать, как компилировать и загружать шейдеры на графический процессор
- Чтобы понять шаги, необходимые для рисования с помощью WebGL
- Продемонстрировать реализацию одноэлементного модуля JavaScript, основанного на простых файлах исходного кода

Настройка и загрузка данных примитивной геометрии

Чтобы эффективно рисовать с помощью WebGL, данные, связанные с рисуемой геометрией, такие как положение вершин квадрата, должны храниться в аппаратном обеспечении графического процессора. На следующих шагах вы создадите непрерывный буфер в графическом процессоре, загрузите положения вершин единичного квадрата в буфер и сохраните ссылку на буфер графического процессора в переменной. Исходя из предыдущего проекта, соответствующий код JavaScript будет сохранен в новом файле исходного кода, **vertex_buffer.js** .

Примечание Единичный квадрат - это квадрат размером 1×1 с центром в начале координат.

---

1. Создайте новый исходный файл JavaScript в папке **src** и назовите его **vertex_buffer.js** .
2. Импортируйте все экспортированные функциональные возможности из **core.js** файл как ядро с инструкцией импорта JavaScript:

```
"use strict";
import * as core from "./core.js";
```

Примечание. С помощью инструкций импорта JavaScript и, которые вскоре появятся, экспорта функции и функциональные возможности, определенные в файле, могут быть удобно инкапсулированы и доступны к ним. В этом случае функциональность, экспортированная из **core.js** импортируется в **vertex_buffer.js** и доступен через идентификатор модуля **core**. Например, как вы увидите, в этом проекте, core.js определяет и экспортирует функцию **getGL()**. С помощью данного оператора import к этой функции можно получить доступ как **core.getGL()** в **vertex_buffer.js** файл.

---

3. Объявите переменную **glvertexbuffer** для сохранения ссылки на местоположение буфера WebGL. Не забудьте определить функцию для доступа к этой переменной.

```
let mGLVertexBuffer = null;
function get() { return mGLVertexBuffer; }
```

4. Определите переменную **mVerticesOfSquare** вершин квадрата и инициализируйте ее вершинами единичного квадрата:

```
let mVerticesOfSquare = [
  0.5, 0.5, 0.0,
 -0.5, 0.5, 0.0,
  0.5, -0.5, 0.0,
 -0.5, -0.5, 0.0
];
```

В показанном коде каждая строка из трех чисел представляет собой положение вершины в координатах x, y и z. Обратите внимание, что измерение z установлено равным 0.0, потому что вы создаете движок 2D-игры. Также обратите внимание, что 0.5 используется для того, чтобы мы определили квадрат в двумерном пространстве, стороны которого равны 1 и центрированы в начале координат или единичный квадрат.

5. Определите функцию **init()** для выделения буфера в графическом процессоре через контекст **gl** и загрузите вершины в выделенный буфер в графическом процессоре:

```
function init() {
  let gl = core.getGL();
  // Step A: Create a buffer on the gl context for our vertex positions
  mGLVertexBuffer = gl.createBuffer();
  // Step B: Activate vertexBuffer
  gl.bindBuffer(gl.ARRAY_BUFFER, mGLVertexBuffer);
  // Step C: Loads mVerticesOfSquare into the vertexBuffer
  gl.bufferData(gl.ARRAY_BUFFER,
  new Float32Array(mVerticesOfSquare), gl.STATIC_DRAW);
}
```

Этот код сначала получает доступ к контексту рисования WebGL через функцию **core.getGL()**. После чего шаг **A** создает буфер на графическом процессоре для хранения положений вершин квадрата и сохраняет ссылку на буфер графического процессора в переменной **glvertexbuffer**. Шаг **B** активирует вновь созданный буфер, а шаг **C** загружает положение вершины квадрата в активированный буфер на графическом процессоре. Ключевое слово **STATIC_DRAW** сообщает аппаратному обеспечению рисования, что содержимое этого буфера не будет изменено.

Совет. Помните, что переменная **mGL**, доступ к которой осуществляется через функцию **getGL()**, определена в **core.js** файл и инициализируется функцией **initWebGL()**. Вы определите инструкцию экспорта в **core.js** файл, чтобы предоставить доступ к этой функции на следующих этапах.

---

6. Предоставьте доступ к функциям **init()** и **get()** остальной части вашего движка, экспортировав их с помощью следующего кода:

```
export {init, get}
```

Теперь, когда функциональность загрузки позиций вершин определена, вы готовы определить и загрузить шейдеры GLSL.

Настройка шейдеров GLSL

Термин **шейдер** относится к программам или набору инструкций, которые выполняются на графическом процессоре. В контексте игрового движка шейдеры всегда должны определяться парами, состоящими из вершинного шейдера и соответствующего фрагментарного шейдера. Графический процессор выполнит вершинный шейдер один раз для каждой примитивной вершины и фрагментный шейдер один раз для каждого пикселя, покрытого примитивом. Например, вы можете определить квадрат с четырьмя вершинами и отобразить этот квадрат так, чтобы он покрывал область размером 100 × 100 пикселей. Чтобы нарисовать этот квадрат, WebGL вызовет вершинный шейдер 4 раза (по одному разу для каждой вершины) и выполнит фрагментный шейдер 10000 раз (по одному разу для каждого из 100×100 пикселей)!

В случае WebGL шейдеры вершин и фрагментов реализованы на языке затенения OpenGL (GLSL). **GLSL** - это язык с синтаксисом, который похож на язык программирования C и разработан специально для обработки и отображения графических примитивов. Вы изучите **GLSL** в достаточном количестве, чтобы при необходимости поддерживать рисование для игрового движка.

На следующих шагах вы загрузите в память графического процессора исходный код как

для вершинных, так и для фрагментных шейдеров, скомпилируете и свяжете их в единую шейдерную программу, и загрузите связанную программу в память графического процессора для рисования. В этом проекте исходный код шейдера определен в **index.html** файл, в то время как загрузка, компиляция и компоновка шейдеров определены в **shader_support.js** исходный файл.

Примечание Контекст WebGL можно рассматривать как абстракцию аппаратного обеспечения графического процессора. Чтобы облегчить читаемость, два термина WebGL и GPU иногда используются взаимозаменяемо.

Определение шейдеров, вершин и фрагментов

GLSL шейдеры - это просто программы, состоящие из инструкций GLSL:

1. Определите вершинный шейдер, открыв **index.html** файл, и внутри элемента **head** добавьте следующий код:

```
<script  type="x-shader/x-vertex" id="VertexShader">
   // this is the vertex shader attribute vec3 aVertexPosition;
   // Expects one vertex position
   // naming convention, attributes always begin with "a"

   attribute vec3 aVertexPosition;
   void main(void) {
     // Convert the vec3 into vec4 for scan conversion and
     // assign to gl_Position to pass vertex to the fragment shader

     gl_Position = vec4(aVertexPosition,1.0);
   }

   // End of vertex shader
</script>
```

Примечание Переменные атрибута шейдера имеют имена, начинающиеся со строчной буквы **“a”**, как в **aVertexPosition.**

Тип элемента **script** установлен в **x-shader/x-vertex**, поскольку это общее соглашение для шейдеров. Как вы увидите, поле **id** со значением **Vertex Shader** позволяет вам идентифицировать и загрузить этот вершинный шейдер в память.

Ключевое слово атрибута **GLSL** идентифицирует данные для каждой вершины, которые будут переданы вершинному шейдеру в графическом процессоре. В этом случае атрибут **aVertexPosition** имеет тип данных **vec3** или массив из трех чисел с плавающей запятой. Как вы увидите на последующих шагах, **aVertexPosition** будет установлен для привязки к позициям вершин единичного квадрата.

gl_Position - это встроенная переменная GLSL, в частности массив из четырех чисел с плавающей запятой, который должен содержать положение вершины. В этом случае четвертая позиция массива всегда будет равна 1.0. Код показывает шейдер, преобразующий **aVertexPosition** в **vec4** и передающий информацию в WebGL.

2. Определите фрагментный шейдер в **index.html** путем добавления следующего кода в элемент **head**:

```
<script type="x-shader/x-fragment" id="FragmentShader">
  // this is the fragment (or pixel) shader
  void main(void) {
    // for every pixel called (within the square) sets
    // constant color white with alpha-channel value of 1.0
    gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);
  }
  // End of fragment/pixel shader
</script>
```

Обратите внимание на различные поля типа и идентификатора. Напомним, что фрагментный шейдер вызывается один раз на пиксель. Переменная **gl_FragColor** - это встроенная переменная, которая определяет цвет пикселя. В этом случае возвращается цвет (1,1,1,1) или белый. Это означает, что все покрытые пиксели будут затенены до постоянного белого цвета.

С шейдерами вершин и фрагментов, определенными в **index.html** файл, теперь вы готовы реализовать функциональность для компиляции, связывания и загрузки результирующей шейдерной программы на графический процессор.

Скомпилируйте, свяжите и загрузите шейдеры вершин и фрагментов

Чтобы сохранить исходный код в логически разделенных исходных файлах, вы создадите функциональность поддержки шейдеров в новом файле исходного кода, shader_support.js .

1. Создайте новый файл JavaScript, **shader_support.js** .
2. Импортируйте функциональность из **core.js** и **vertex_buffer.js** файлы:

```
"use strict"; // Variables must be declared before used!
import * as core from "./core.js"; // access as core module
import * as vertexBuffer from "./vertex_buffer.js"; //vertexBuffer module
```

3. Определите две переменные, **mCompiledShader** и **mVertexPositionRef**, для ссылки на программу шейдера и атрибут **vertex position** в графическом процессоре:

```
let mCompiledShader = null;
let mVertexPositionRef = null;
```

4. Создайте функцию для загрузки и компиляции шейдера, определенного вами в **index.html**:

```
function loadAndCompileShader(id, shaderType) {
  let shaderSource = null, compiledShader = null;
  // Step A: Get the shader source from index.html
  let shaderText = document.getElementById(id);
  shaderSource = shaderText.firstChild.textContent;
  let gl = core.getGL();
  // Step B: Create shader based on type: vertex or fragment
  compiledShader = gl.createShader(shaderType);
  // Step C: Compile the created shader
  gl.shaderSource(compiledShader, shaderSource);
  gl.compileShader(compiledShader);
  // Step D: check for errors and return results (null if error)
  // The log info is how shader compilation errors are displayed.
  // This is useful for debugging the shaders.
  if (!gl.getShaderParameter(compiledShader, gl.COMPILE_STATUS)) {
    throw new Error("A shader compiling error occurred: " +
    gl.getShaderInfoLog(compiledShader));
  }
  return compiledShader;
}
```

Шаг **A** кода находит исходный код шейдера в **index.html** файл, используя поле **id**, указанное вами при определении шейдеров, будь то вершинный шейдер или фрагментный шейдер. Шаг **B** создает указанный шейдер (либо вершину, либо фрагмент) в графическом процессоре. Шаг **C** определяет исходный код и компилирует шейдер. Наконец, шаг **D** проверяет и возвращает ссылку на скомпилированный шейдер, выдавая ошибку, если компиляция шейдера не удалась.

5. Теперь вы готовы создать, скомпилировать и связать шейдерную программу, определив функцию **init()**:

```
function init(vertexShaderID, fragmentShaderID) {
  let gl = core.getGL();
  // Step A: load and compile vertex and fragment shaders
  let vertexShader = loadAndCompileShader(vertexShaderID,
  gl.VERTEX_SHADER);
  let fragmentShader = loadAndCompileShader(fragmentShaderID,
  gl.FRAGMENT_SHADER);
  // Step B: Create and link the shaders into a program.
  mCompiledShader = gl.createProgram();
  gl.attachShader(mCompiledShader, vertexShader);
  gl.attachShader(mCompiledShader, fragmentShader);
  gl.linkProgram(mCompiledShader);
  // Step C: check for error
  if (!gl.getProgramParameter(mCompiledShader, gl.LINK_STATUS)) {
    throw new Error("Error linking shader");
    return null;
  }
  // Step D: Gets reference to aVertexPosition attribute in the shader
  mVertexPositionRef = gl.getAttribLocation(mCompiledShader,
  "aVertexPosition");
}
```

Шаг **A** загружает и компилирует код шейдера, который вы определили в **index.html** путем вызова функции **loadAndCompileShader()** с соответствующими параметрами. Шаг **B** присоединяет скомпилированные шейдеры и связывает два шейдера в программу. Ссылка на эту программу хранится в переменной **mCompiledShader**. После проверки ошибки на шаге **C**, шаг **D** находит и сохраняет ссылку на атрибут **aVertexPosition**, определенный в вашем вершинном шейдере.

6. Определите функцию, позволяющую активировать шейдер, чтобы его можно было использовать для рисования квадрата:

```
function activate() {
  // Step A: access to the webgl context
  let gl = core.getGL();
  // Step B: identify the compiled shader to use
  gl.useProgram(mCompiledShader);
  // Step C: bind vertex buffer to attribute defined in vertex shader
  gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer.get());
  gl.vertexAttribPointer(this.mVertexPositionRef,
  3, // each element is a 3-float (x,y.z)
  gl.FLOAT, // data type is FLOAT
  false, // if the content is normalized vectors
  0, // number of bytes to skip in between elements
  0); // offsets to the first element
  gl.enableVertexAttribArray(this.mVertexPositionRef);
}
```

В показанном коде шаг **A** устанавливает переменную **gl** в контекст WebGL через основной модуль. Шаг **B** загружает скомпилированную шейдерную программу в память графического процессора, в то время как шаг **C** привязывает буфер вершин, созданный в **vertex_buffer.js** к

атрибуту **aVertexPosition**, определенному в вершинном шейдере. Функция **gl.vertexAttribPointer()** фиксирует тот факт, что буфер вершин был загружен вершинами единичного квадрата, состоящими из трех значений с плавающей запятой для каждой позиции вершины.

7. Наконец, предоставьте доступ к функциям **init()** и **activate()** для остальной части игрового движка, экспортировав их с помощью инструкции **export**

```
export { init, activate }
```

Примечание Функция **loadAndCompileShader()** исключена из инструкции **export**. Эта функция больше нигде не нужна и, таким образом, следуя хорошей практике разработки по сокрытию деталей локальной реализации, должна оставаться закрытой для этого файла

Теперь определена функциональность загрузки и компиляции шейдеров. Теперь вы можете использовать и активировать эти функции для рисования с помощью WebGL.

Настройка рисования с помощью WebGL

Определив данные вершин и функциональность шейдера, теперь вы можете выполнить следующие шаги для рисования с помощью WebGL. Напомним из предыдущего проекта, что код инициализации и рисования определен в **core.js** файл. Теперь откройте этот файл для редактирования.

1. Импортируйте определенную функциональность из **vertex_buffer.js** и **shader_support.js** файлы:

```
import * as vertexBuffer from "./vertex_buffer.js";
import * as simpleShader from "./shader_support.js";
```

2. Добавте код в функцию **initWebGL()**, чтобы включить инициализацию буфера вершин и программы шейдеров:

```
function initWebGL(htmlCanvasID) {
  let canvas = document.getElementById(htmlCanvasID);
  mGL = canvas.getContext("webgl") ||
        canvas.getContext("experimental-webgl2");
  if (mGL === null) {
     document.write("<br><b>WebGL 2 is not supported!</b>");
     return;
  }
  mGL.clearColor(0.0, 0.8, 0.0, 1.0);

  //1. initialize buffer with vertex positions for the unit square
  vertexBuffer.init(); // function defined in the vertex_buffer.js
  //2. now load and compile the vertex and fragment shaders
  simpleShader.init("VertexShader", "FragmentShader");
  //the two shaders are defined in the index.html file
  //init() function is defined in shader_support.js file
}
```

Как показано в коде, после успешного получения ссылки на контекст WebGL и установки прозрачного цвета вам следует сначала вызвать функцию **init()**, определенную в **vertex_buffer.js** чтобы инициализировать буфер вершин графического процессора вершинами единичного квадрата, а затем вызвать функцию **init()**, определенную в **shader_support.js** чтобы загрузить и скомпилировать шейдеры вершин и фрагментов.

3. Добавьте функцию **DrawSquare()** для рисования определенного квадрата:

```
function drawSquare() {
// Step A: Activate the shader
  simpleShader.activate();
// Step B. draw with the above settings
  mGL.drawArrays(mGL.TRIANGLE_STRIP, 0, 4);
}
```

Этот код показывает шаги по рисованию с помощью WebGL. Шаг **A** активирует программу шейдера для использования. Шаг **B** выдает команду **WebGL draw**. В этом случае вы даете команду нарисовать четыре вершины в виде двух соединенных треугольников, образующих квадрат.

4. Теперь вам просто нужно изменить функцию **window.onload**, чтобы вызвать недавно определенную функцию **drawSquare():**

```
window.onload = function() {
  initWebGL("GLCanvas"); // Binds mGL context to WebGL functionality
  clearCanvas(); // Clears the GL area
  drawSquare(); //Draws one square
}
```

5. Наконец, предоставьте доступ к контексту WebGL остальной части движка, экспортировав функцию **getGL()**. Помните, что эта функция импортирована и была вызвана для доступа к контексту WebGL в обоих **vertex_buffer.js** и

simple_shader.js

Напомним, что функция, которая ограничена **window.onload**, будет вызвана после того как **indexl.html** будет загружен веб-браузером. По этой причине WebGL будет инициализирован, холст станет светло-зеленым и будет нарисован белый квадрат. Вы можете обратиться к исходному коду в главе 2/2.3.проект **draw_one_square** для всей описанной системы.

Наблюдения

Запустите проект, и вы увидите белый прямоугольник на зеленом холсте. Что случилось с площадью? Помните, что положение вершины вашего квадрата 1 ×1 было определено в точках (±0,5, ±0,5). Теперь обратите внимание на результат проекта: белый прямоугольник расположен в середине зеленого холста, занимая ровно половину ширины и высоты холста. Как оказалось, WebGL рисует вершины в диапазоне ±1.0 на всей определенной области рисования. В этом случае ±1,0 в измерении x отображается на 640 пикселей, в то время как ±1,0 в размер y сопоставляется с 480 пикселями (размер созданного холста равен 640×480). Квадрат 1x1 рисуется на площади 640x480 или области с соотношением сторон 4:3. С тех пор как соотношение сторон квадрата 1:1 не соответствует соотношению сторон области отображения 4:3, квадрат отображается в виде прямоугольника 4:3. Эта проблема будет решена позже в следующей главе.

Вы можете попробовать отредактировать фрагментный шейдер в **index.html** изменив цвет, установленный в функции **gl_FragColor**, чтобы изменить цвет белого квадрата. Обратите внимание, что значение меньше 1 в альфа-канале не приводит к тому, что белый квадрат становится прозрачным. Прозрачность нарисованных примитивов будет обсуждаться в последующих главах.

Наконец, обратите внимание, что этот проект определяет три отдельных файла и скрывает информацию с помощью инструкций импорта/экспорта JavaScript. Функциональность, определенная в этих файлах с соответствующими инструкциями импорта и экспорта, называется модулями JavaScript. Модуль можно рассматривать как глобальный одноэлементный объект и он отлично подходит для сокрытия деталей реализации. Функция **loadAndCompileShader()** в модуле **shader_support** служит отличным примером этой концепции. Однако модули не очень хорошо подходят для поддержки абстракции и специализации. В следующих разделах вы начнете работать с классами JavaScript для дальнейшей инкапсуляции частей этого примера, чтобы сформировать основу фреймворка игрового движка.

Абстракция с помощью классов JavaScript

Предыдущий проект разложил чертеж квадрата на логические модули и реализовал модули в виде файлов, содержащих глобальные функции. В программной инженерии этот процесс называется функциональной декомпозицией, а реализация - процедурным программированием. Процедурное программирование часто приводит к хорошо структурированным и простым для понимания решениям. Вот почему функциональная декомпозиция и процедурное программирование часто используются для создания прототипов концепций или изучения новых методов.

Этот проект дополняет проект **Draw One Square** объектно-ориентированным анализом и программированием для внедрения абстракции данных. По мере появления дополнительных концепций и роста сложности игрового движка надлежащая абстракция данных

поддерживает простой дизайн, специализацию поведения и повторное использование кода посредством наследования.

Проект объектов JavaScript

Этот проект демонстрирует, как абстрагировать глобальные функции из проекта **Draw One Square** в классы и объекты JavaScript. Эта объектно-ориентированная абстракция приведет к созданию фреймворка, который обеспечивает управляемость и расширяемость для последующих проектов. Как показано на рисунке 2-7, при запуске этот проект отображает белый прямоугольник на зеленоватом холсте, идентичный прямоугольнику из проекта **Draw One Square.** Исходный код этого проекта можно найти в папке глава **2/2.4.javascript_objects**

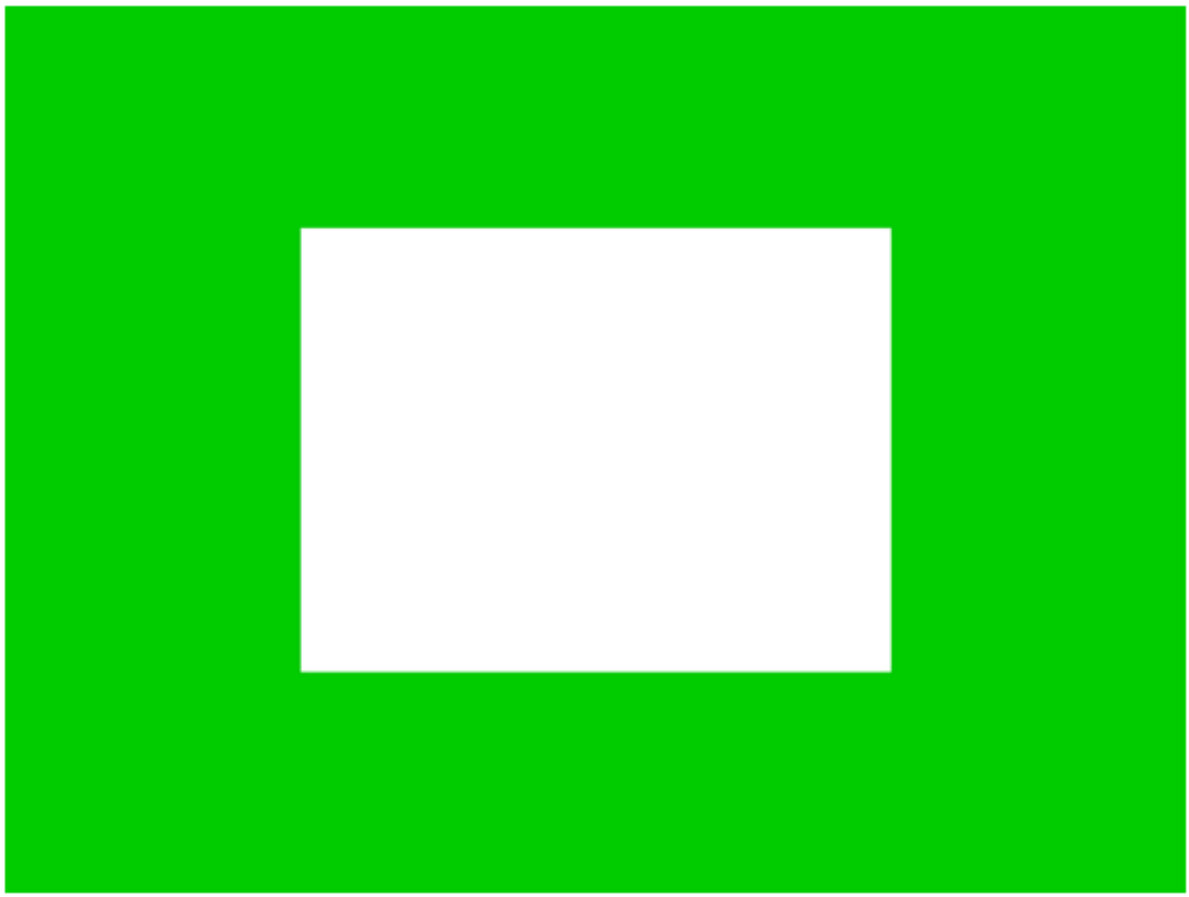

Рисунок 2-7 Запуск проекта JavaScript Objects

Цели проекта заключаются в следующем:

- Отделить код для игрового движка от кода для игровой логики
- Понять, как создавать абстракции с помощью классов и объектов JavaScript

Шаги по созданию этого проекта следующие:

1. Создайте отдельные папки для организации исходного кода игрового движка и логики игры.
2. Определите класс JavaScript для абстрагирования **simple_shader** и работы с экземпляром этого класса.
3. Определите класс JavaScript для реализации рисования одного квадрата, что на данный момент является логикой вашей простой игры.

Организация исходного кода

Создайте новый проект HTML5 с помощью VS Code, создав новую папку и добавив папку с

исходным кодом с именем src. В **src** создайте **engine** и **my_game** в качестве вложенных папок, как показано на **рисунке 2-8**.

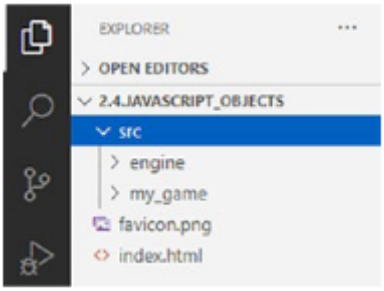

Рисунок 2-8 Создаем **engine** и **my_game** в папке **src**
Папка **src/engine** будет содержать весь исходный код игрового движка, а папка **src/my_game** будет содержать исходный код логики вашей игры. Важно тщательно организовать исходный код, поскольку сложность системы и количество файлов будут быстро увеличиваться по мере внедрения новых концепций. Хорошо организованная структура исходного кода облегчает понимание и расширение.

Совет: Исходный код в папке **my_game** реализует игру, полагаясь на функциональность, предоставляемую игровым движком, определенным в папке **engine**. По этой причине в этой книге исходный код в папке **my_game** часто упоминается как клиент игрового движка.

Абстрагирование игрового движка

Готовый игровой движок включал бы в себя множество автономных подсистем для выполнения различных обязанностей. Например, вы можете быть знакомы или слышали о подсистеме геометрии для управления геометриями, которые будут нарисованы, подсистеме управления ресурсами для управления изображениями и аудиоклипами, подсистеме физики для управления взаимодействиями объектов и так далее. В большинстве случаев игровой движок включал бы по одному уникальному экземпляру каждой из этих подсистем, то есть по одному экземпляру подсистемы геометрии, подсистемы управления ресурсами, подсистемы физики и так далее.

Эти подсистемы будут рассмотрены в последующих главах этой книги. В этом разделе основное внимание уделяется созданию механизма и организации для реализации этой функциональности, подобной единому экземпляру или **синглтону**, на основе модуля JavaScript, с которым вы работали в предыдущем проекте.

Примечание:Все имена переменных модуля и экземпляра начинаются с **“m”** и сопровождаются заглавной буквой, как в переменной. Хотя JavaScript и не применяется принудительно, вы никогда не должны получать доступ к модулю или переменной экземпляра извне **module / class**. Например, вы никогда не должны получать доступ к **core.mGL** напрямую; вместо этого вызовите функцию **core.getGL()** для доступа к переменной.

Класс шейдеров

Хотя код в **shader_support.js** файл из предыдущего проекта должным образом реализует требуемую функциональность, переменные и функции плохо поддаются специализации поведения и повторному использованию кода. Например, в случаях, когда требуются различные типы шейдеров, может быть сложно модифицировать реализацию, добиваясь при

этом поведения и повторного использования кода. Этот раздел следует принципам объектно-ориентированного проектирования и определяет класс **SimpleShader** для абстрагирования поведения и скрытия внутренних представлений шейдеров. Помимо возможности создавать несколько экземпляров **SimpleShader** объекта, базовая функциональность остается в основном неизменной.

Примечание: Идентификаторы модулей начинаются со строчной буквы, например, **core** или **vertex Buffer**. Названия классов начинаются с верхнего регистра, например, **SimpleShader** или **MyGame**.

---

1. Создайте новый исходный файл в папке **src/engine** и назовите файл **simple_shader.js** для реализации класса **SimpleShader**.
2. Импортируйте модули **core** и **vertex_buffer**:

```
import * as core from "./core.js";
import * as vertexBuffer from "./vertex_buffer.js";
```

3. Объявите SimpleShader как класс JavaScript:

```
class SimpleShader {
... implementation to follow ...
}
```

4. Определите конструктор в классе **SimpleShader** для загрузки, компиляции и связывания шейдеров вершин и фрагментов в программу и для создания ссылки на атрибут **aVertexPosition** в вершинном шейдере для загрузки квадратных позиций вершин из буфера вершин WebGL для рисования:

```
class SimpleShader {
  constructor(vertexShaderID, fragmentShaderID) {
    // instance variables
    // Convention: all instance variables: mVariables
    this.mCompiledShader = null; // ref to compiled shader in webgl
    this.mVertexPositionRef = null; // ref to VertexPosition in shader
    let gl = core.getGL();
    // Step A: load and compile vertex and fragment shaders
    this.mVertexShader = loadAndCompileShader(vertexShaderID,
    gl.VERTEX_SHADER);
    this.mFragmentShader = loadAndCompileShader(fragmentShaderID,
    gl.FRAGMENT_SHADER);
    // Step B: Create and link the shaders into a program.
    this.mCompiledShader = gl.createProgram();
    gl.attachShader(this.mCompiledShader, this.mVertexShader);
    gl.attachShader(this.mCompiledShader, this.mFragmentShader);
    gl.linkProgram(this.mCompiledShader);
    // Step C: check for error
    if (!gl.getProgramParameter(this.mCompiledShader, gl.LINK_STATUS)) {
    throw new Error("Error linking shader");
    return null;
    }
    // Step D: reference to aVertexPosition attribute in the shaders
    this.mVertexPositionRef = gl.getAttribLocation(
    this.mCompiledShader, "aVertexPosition");
  }
}
```

Обратите внимание, что этот конструктор, по сути, такой же, как функция **init()** в **shader_support.js** модуль из предыдущего проекта.

Примечание: Ключевое слово JavaScript **constructor** определяет конструктор класса.

---

5. Добавьте метод в класс **SimpleShader**, чтобы активировать шейдер для рисования. Еще раз, аналогично вашей функции **activate()** в **shader_support.js** из предыдущего проекта.

```
activate() {
  let gl = core.getGL();
  gl.useProgram(this.mCompiledShader);
  // bind vertex buffer
  gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer.get());
  gl.vertexAttribPointer(this.mVertexPositionRef,
  3, // each element is a 3-float (x,y.z)
  gl.FLOAT, // data type is FLOAT
  false, // if the content is normalized vectors
  0, // number of bytes to skip in between elements
  0); // offsets to the first element
  gl.enableVertexAttribArray(this.mVertexPositionRef);
}
```

6. Добавьте закрытый метод, к которому нельзя получить доступ извне **simple_shader.js** файл, создав функцию вне класса **SimpleShader** для выполнения фактической загрузки и компиляции функциональности:

```
function loadAndCompileShader(id, shaderType) {
  let shaderSource = null, compiledShader = null;
  let gl = core.getGL();
  // Step A: Get the shader source from index.html
  let shaderText = document.getElementById(id);
  shaderSource = shaderText.firstChild.textContent;
  // Step B: Create shader based on type: vertex or fragment
  compiledShader = gl.createShader(shaderType);
  // Step C: Compile the created shader
  gl.shaderSource(compiledShader, shaderSource);
  gl.compileShader(compiledShader);
  // Step D: check for errors and return results (null if error)
  // The log info is how shader compilation errors are displayed
  // This is useful for debugging the shaders.
  if (!gl.getShaderParameter(compiledShader, gl.COMPILE_STATUS)) {
    throw new Error("A shader compiling error occurred: " +
    gl.getShaderInfoLog(compiledShader));
  }
  return compiledShader;
}
```

Обратите внимание, что эта функция идентична той, которую вы создали в **shader_support.js**

Примечание: Префикс **JavaScript #**, определяющий закрытые члены, в этой книге не используется, поскольку отсутствие видимости из подклассов усложняет специализацию поведения при наследовании.

---

7. Наконец, добавьте экспорт для класса **SimpleShader** таким образом, чтобы к нему можно было получить доступ и создать экземпляр за пределами этого файла:

```
export default SimpleShader;
```

Примечание: Ключевое слово **default** означает, что имя **SimpleShader** не может быть изменено с помощью инструкций **import**

Ядро игрового движка: core.js

Ядро содержит общую функциональность, разделяемую всем игровым движком. Это может включать одноразовую инициализацию **WebGL** (или GPU), общих ресурсов, служебных функций и так далее.

1. Создайте копию вашего **core.js** в новой папке **js/engine.**
2. Определите функцию для создания нового экземпляра **SimpleShader** объекта:

```
// The shader
let mShader = null;
function createShader() {
  mShader = new SimpleShader(
  "VertexShader", // IDs of the script tag in the index.html
  "FragmentShader"); //
}
```

3. Измените функцию **initWebGL()**, чтобы сосредоточиться только на инициализации WebGL следующим образом:

```
// initialize the WebGL
function initWebGL(htmlCanvasID) {
  let canvas = document.getElementById(htmlCanvasID);
  // Get standard or experimental webgl and binds to the Canvas area
  // store the results to the instance variable mGL
  mGL = canvas.getContext("webgl") ||
  canvas.getContext("experimental-webgl");
    if (mGL === null) {
    document.write("<br><b>WebGL 2 is not supported!</b>");
  return;
  }
}
```

4. Создайте функцию **init()** для выполнения инициализации всей системы движка, которая включает в себя инициализацию WebGL и буфера вершин и создание экземпляра **SimpleShader**:

```
function init(htmlCanvasID) {
  initWebGL(htmlCanvasID); // setup mGL
  vertexBuffer.init(); // setup mGLVertexBuffer
  createShader(); // create the shader
}
```

5. Измените функцию **clearCanvas**, чтобы параметризовать цвет, который должен быть очищен, на:

```
function clearCanvas() {
  mGL.clearColor(color[0], color[1], color[2], color[3]);
  mGL.clear(mGL.COLOR_BUFFER_BIT); // clear to the color set
}
```

6. Экспортируйте соответствующие функции для доступа к остальной части игрового движка

```
export {getGL, init, clearCanvas, drawSquare}
```

7. Наконец, удалите функцию **window.onload**, поскольку поведение реальной игры должно определяться клиентом игрового движка или, в данном случае, классом **MyGame**.

Папка **src/engine** теперь содержит базовый исходный код для всего игрового движка. Благодаря этим структурным изменениям в вашем исходном коде игровой движок теперь может функционировать как простая библиотека, предоставляющая функциональность для создания игр или простой интерфейс прикладного программирования (**API**). На данный момент ваш игровой движок состоит из трех файлов, которые поддерживают инициализацию WebGL и рисование единичного квадрата, модуля **core**, модуля **vertex_buffer** и класса **SimpleShader**. Новые исходные файлы и функциональные возможности будут по-прежнему добавляться в эту папку во всех остальных проектах. В конечном счете, эта папка будет содержать полный и сложный игровой движок. Однако базовая структура, подобная библиотеке, определенная здесь, сохранится.

Исходный код клиента

Папка **src/my_game** будет содержать фактический исходный код игры. Как уже упоминалось, код в этой папке будет называться клиентом игрового движка. На данный момент исходный код в папке **my_game** будет сосредоточен на рисовании простого квадрата с использованием функциональности определенного вами движка simple game engine.

1. Создайте новый исходный файл в папке **src/my_game** или папке клиента и назовите файл **my_game.js**
2. Импортируйте основной модуль следующим образом:

```
import * as engine from "../engine/core.js";
```

3. Определите **MyGame** как класс JavaScript и добавьте конструктор для инициализации игрового движка, очистки canvas и рисования квадрата:

```
class MyGame {
  constructor(htmlCanvasID) {
    // Step A: Initialize the game engine
    engine.init(htmlCanvasID);
    // Step B: Clear the canvas
    engine.clearCanvas([0, 0.8, 0, 1]);
    // Step C: Draw the square
    engine.drawSquare();
  }
}
```

4. Привяжите создание нового экземпляра(*объекта*) **MyGame** к функции **window.onload:**

```
window.onload = function() {
  new MyGame('GLCanvas');
}
```

5. Наконец, измените **index.html** чтобы загрузить игровой клиент, а не движок **core.js** внутри элемента **head:**

```
<script type="module" src="./js/my_game/my_game.js"></script>
```

Наблюдения

Хотя вы выполняете те же задачи, что и в предыдущем проекте, с помощью этого проекта вы создали инфраструктуру, которая поддерживает последующие модификации и расширения вашего игрового движка. Вы организовали свой исходный код в отдельные логические папки, организовали модули, подобные **singleton**, для реализации основных функциональных возможностей движка и приобрели опыт абстрагирования класса **SimpleShader**, который будет поддерживать будущий дизайн и повторное использование кода. Теперь, когда движок состоит из четко определенных модулей и объектов с понятными интерфейсными методами, вы можете сосредоточиться на изучении новых концепции, абстрагирование концепций и интеграция нового исходного кода реализации в ваш движок.

Отделение GLSL от HTML

До сих пор в ваших проектах код шейдера GLSL встроен в исходный код HTML **index.html** . Такая организация означает, что новые шейдеры должны быть добавлены путем редактирования **index.html** файл. Логически шейдеры GLSL должны быть организованы отдельно от исходных файлов HTML; с точки зрения логистики, постоянное добавление к index.html приведет к созданию загроможденного и неуправляемого файла, с которым станет трудно работать. По этим причинам шейдеры GLSL должны храниться в отдельных исходных файлах

Проект исходного файла шейдера

Этот проект демонстрирует, как разделить шейдеры GLSL на отдельные файлы. Как показано на **рисунке 2-9**, при запуске этого проекта на зеленоватом холсте отображается белый прямоугольник, идентичный предыдущим проектам. Исходный код этого проекта определен в папке главы **2/2.5.shader_source_files.**

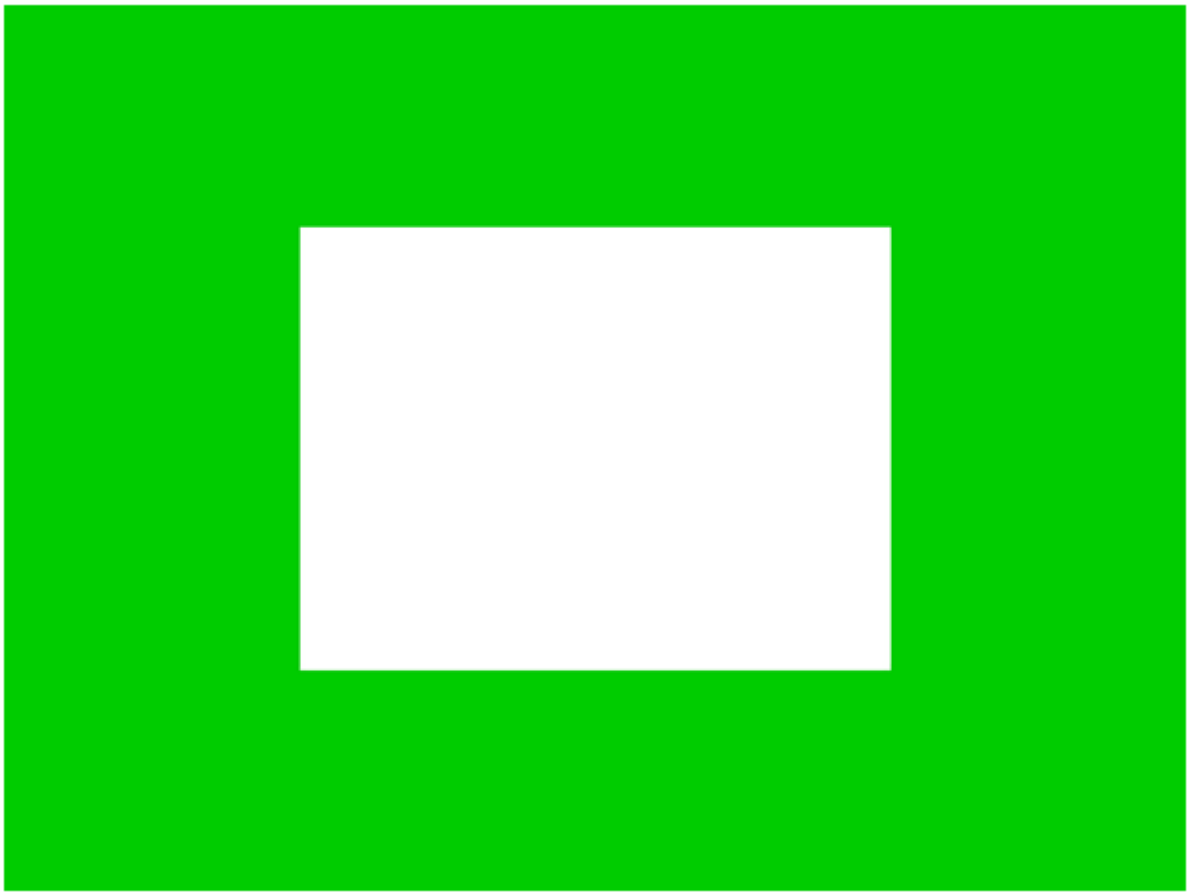

Рисунок 2-9 Запуск проекта исходного файла шейдера
Цели проекта заключаются в следующем:

- Чтобы отделить шейдеры GLSL от исходного кода HTML
- Чтобы продемонстрировать, как загружать файлы исходного кода шейдера во время выполнения

Загрузка шейдеров в SimpleShader

Вместо загрузки GLSL-шейдеров как части HTML-документа, **loadAndCompileShader()** в **SimpleShader** можно изменить для загрузки GLSL-шейдеров в виде отдельных файлов:

1. Продолжите работу с предыдущим проектом, откройте **simple_shader.js** файл и отредактируйте функцию **loadAndCompileShader()**, чтобы получить путь к файлу вместо HTML-идентификатора:

```
function loadAndCompileShader(filePath, shaderType)
```

2. В функции **loadAndCompileShader()** замените код извлечения HTML-элемента на шаге **A** следующим **XMLHttpRequest** для загрузки файла:

Обратите внимание, что загрузка файла будет происходить синхронно, когда веб-браузер фактически остановится и будет ждать завершения функции **xmlReq.open()**, чтобы вернуться с содержимым открытого файла. Если файл должен отсутствовать, операция открытия завершится неудачей, и текст ответа будет равен **null**.

Синхронизированная **“остановка и ожидание”** завершения функции **xmlReq.open()** неэффективна и может привести к медленной загрузке веб-страницы. Этот недостаток будет

устранен в главе 4, когда вы узнаете об асинхронной загрузке игровых ресурсов.

Примечание: Объекту **XMLHttpRequest()** требуется работающий веб-сервер для выполнения **HTTP-запроса get**. Это означает, что вы сможете протестировать этот проект из VS Code с установленным расширением **"Go Live"**. однако, если на вашем компьютере не запущен веб-сервер, вы не сможете запустить этот проект, дважды щелкнув **index.html** файл напрямую. Это связано с тем, что нет сервера для выполнения **HTTP get запросов**, и загрузка шейдера GLSL завершится неудачей.

С помощью этой модификации конструктор **SimpleShader** теперь можно модифицировать для получения и пересылки путей к файлам функции loadAndCompileShader() вместо идентификаторов HTML-элементов.

Извлечение шейдеров в их собственные файлы

Следующие шаги извлекают исходный код шейдеров вершин и фрагментов из **index.html** файл и создайте отдельные файлы для их хранения:

1. Создайте новую папку, которая будет содержать все файлы исходного кода GLSL shader в папке **src**(**js**), и назовите ее **glsl_shaders**, как показано на рисунке **2–10**

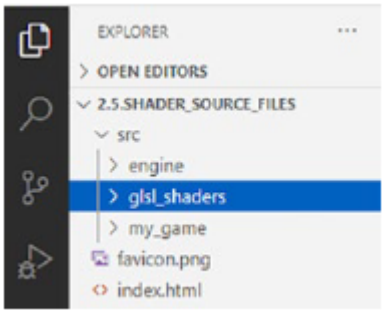

Рисунок 2-10 Создание папки **glsl_shaders**

2. Создайте два новых текстовых файла в папке **glsl_shaders** и назовите их **simple_vs.glsl** и **white_fs.gls**l для простого вершинного шейдера и белого фрагментного шейдера

Примечание:Все файлы исходного кода шейдеров GLSL будут заканчиваться расширением .**glsl** .**vs** в именах файлов шейдеров означает, что файл содержит вершинный шейдер, в то время как **fs** означает фрагментный шейдер.

---

3. Создайте исходный код вершинного шейдера GLSL, отредактировав **simple_vs.glsl** и вставив код вершинного шейдера в **index.html** файл из предыдущего проекта:

```
attribute vec3 aVertexPosition; // Vertex shader expects one position
  void main(void) {
  // Convert the vec3 into vec4 for scan conversion and
  // assign to gl_Position to pass the vertex to the fragment shader
  gl_Position = vec4(aVertexPosition, 1.0);
}
```

4. Создайте исходный код шейдера фрагмента GLSL, отредактировав **white_fs.glsl** и вставив код шейдера фрагмента в **index.html** файл из предыдущего проекта:

```
precision mediump float; // precision for float computation
  void main(void) {
  // for every pixel called (within the square) sets
  // constant color white with alpha-channel value of 1.0
  gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);
}
```

Очистка HTML-кода

Поскольку шейдеры вершин и фрагментов хранятся в отдельных файлах, теперь можно очистить **index.html** файл такой, чтобы он содержал только HTML-код:

1. Удалите весь код шейдера GLSL из **index.html** , так что этот файл становится следующим:

```
<!DOCTYPE html>
<html charset="utf-8">
  <head>
   <title>GameProject</title>
   <meta name="description" content="Simple Starter Description">
   <meta name="viewport" content="width=device-width, initial-scale=1.0, minimum-scale=1.0,
   maximum-scale=1.0, user-scalable=no">
   <link rel="icon" href="favicon.png">
   <script type="module" src="./js/my_game/my_game.js"></script>
  </head>
  <body>
     <canvas id="GLCanvas" width="640" height="480">
        Your browser does not support the HTML5 canvas.
     </canvas>
  </body>
</html>
```

Обратите внимание, что **index.html** больше не содержит никакого кода шейдера GLSL и только одну ссылку на код JavaScript. С этой организацией, **index.html** файл можно правильно рассматривать как представляющий веб-страницу, где вам не нужно редактировать этот файл для изменения шейдеров с этого момента.

2. Измените функцию **createShader()** в **core.js** чтобы загрузить файлы шейдеров вместо идентификаторов HTML-элементов:

```
function createShader() {
  mShader = new SimpleShader(
    "js/glsl_shaders/simple_vs.glsl", // Path to VertexShader
    "js/glsl_shaders/white_fs.glsl"); // Path to FragmentShader
  }
```

Организация исходного кода

Разделение логических компонентов в исходном коде движка перешло в следующее состояние:

- **index.html** : Это файл, содержащий HTML-код, который определяет холст на веб-странице игры и загружает исходный код для вашей игры.
- **js/glsl_shaders**: Это папка, содержащая все файлы исходного кода GLSL-шейдеров, которые рисуют элементы вашей игры.
- **js/engine**: Это папка, которая содержит все файлы исходного кода для вашего игрового движка.
- **js/my_game**: Это клиентская папка, которая содержит исходный код для самой игры.

Изменение шейдера и управление цветом

Поскольку шейдеры GLSL хранятся в отдельных файлах исходного кода, теперь можно редактировать или заменять шейдеры, внося относительно незначительные изменения в остальной исходный код. Следующий проект демонстрирует это удобство, заменяя ограничительный постоянный фрагментный шейдер белого цвета **white_fs.glsl** на шейдер, который можно параметризовать для рисования любым цветом.

Проект параметризованного фрагментного шейдера

Этот проект заменяет **white_fs.glsl** на **simple_fs.glsl**, который поддерживает рисование любым цветом. На рисунке **2–11** показан результат запуска проекта параметризованного фрагментного шейдера; обратите внимание, что красный квадрат заменяет белый квадрат из предыдущих проектов. Исходный код для этого проекта определен в папке **chapter2/2.6.parameterized_fragment_shader**.

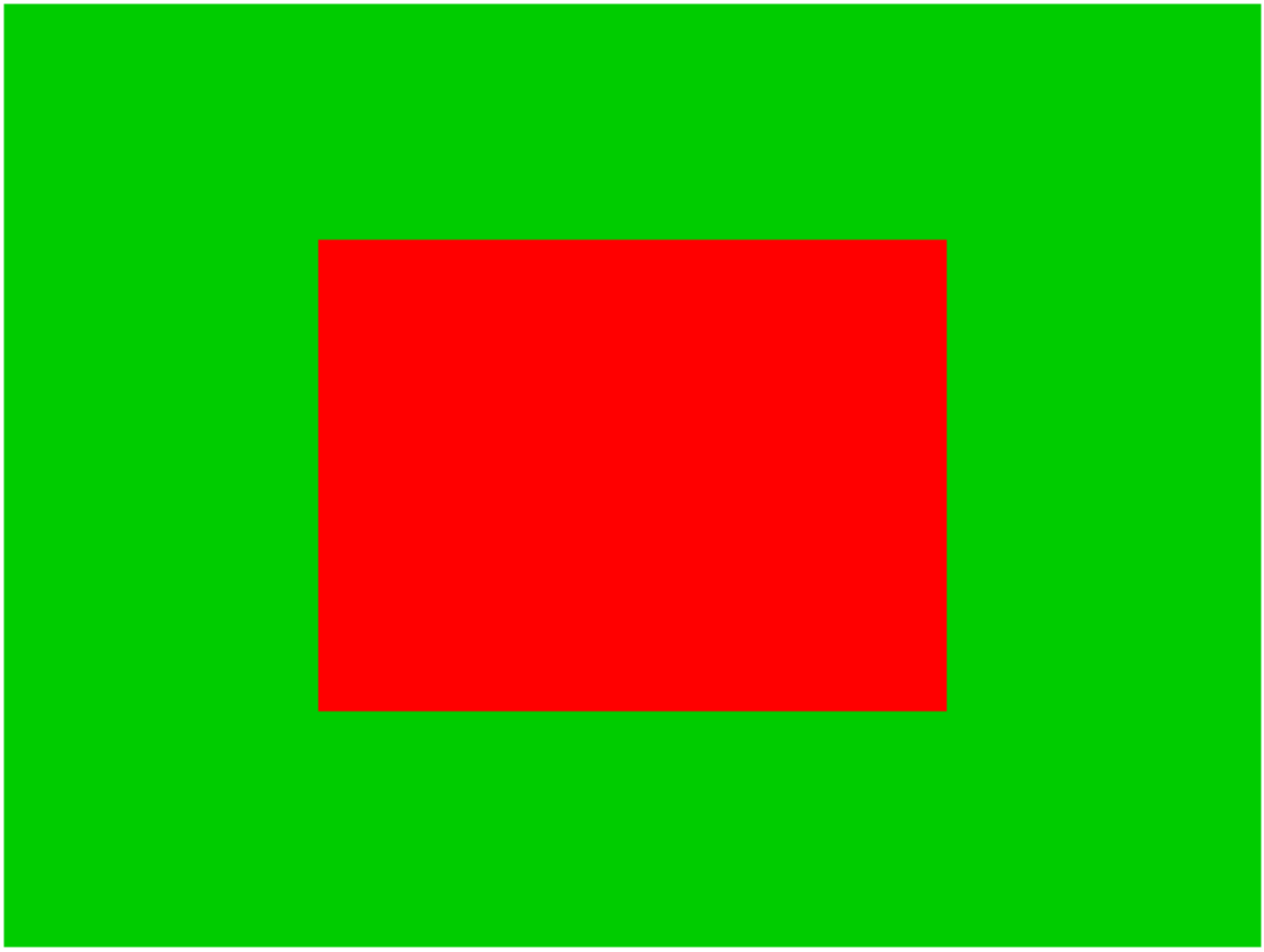

Рисунок 2–11. Запуск проекта параметризованного фрагментного шейдера

Цели проекта заключаются в следующем:

- Получить опыт создания шейдера GLSL в структуре исходного кода
- Чтобы узнать о переменной **uniform** и определить фрагментный шейдер с параметром **color**

Определение шейдера фрагмента simple_fs.glsl

Необходимо создать новый фрагментный шейдер, поддерживающий изменение цвета пикселя для каждой операции рисования. Этого можно достичь, создав новый шейдер фрагмента GLSL в папке **js/glsl_shaders** и назвав его **simple_fs.glsl**. Отредактируйте этот файл, чтобы добавить следующее:

```
precision mediump float; // precision for float computation
// Color of pixel
uniform vec4 uPixelColor;
void main(void) {
  // for every pixel called sets to the user specified color
  gl_FragColor = uPixelColor;
}
```

Напомним, что ключевое слово атрибута GLSL идентифицирует данные, которые изменяются для каждой позиции вершины. В этом случае ключевое слово **uniform** обозначает, что переменная является постоянной для всех вершин. Переменная **pixel color** может быть установлена с помощью JavaScript для управления конечным цветом пикселя. Ключевые слова **precision mediump** определяют плавающие значения точности для вычислений.

Примечание: Точность с плавающей запятой обменивает точность вычислений на производительность. Пожалуйста, следуйте ссылкам в главе 1 для получения дополнительной информации о WebGL.

Дополните класс SimpleShader для поддержки параметра Color

Класс **SimpleShader** теперь можно изменить, чтобы получить доступ к новой переменной **uPixelColor** :

1. Редактировать **simple_shader.js** и добавьте новую переменную экземпляра для ссылки на **uPixelColor** в конструкторе:

```
this.mPixelColorRef = null; // pixelColor uniform in fragment shader
```

2 Добавьте код в конец конструктора, чтобы создать ссылку на **uPixelColor**:

```
// Step E: Gets uniform variable uPixelColor in fragment shader
this.mPixelColorRef = gl.getUniformLocation(
this.mCompiledShader, "uPixelColor");
```

3. Измените активацию шейдера, чтобы разрешить настройку цвета пикселя с помощью функции **uniform4fv**()(добавте эту строчку кода в метод **activate**() класса **SimpleShader**):

```
    activate(pixelColor) {
// load uniforms
gl.uniform4fv(this.mPixelColorRef, pixelColor);
```

Функция **gl.uniform4fv()** копирует четыре значения с плавающей запятой из массива **pixelColor** в местоположение WebGL, на которое ссылается **mPixelColorRef** или **uPixelColor** в шейдере фрагмента **simple_fs.glsl**

Рисование с помощью нового шейдера

Чтобы протестировать **simple_fs.glsl**, измените **core.js** модуль для создания **SimpleShader** с новым **simple_fs** и использования параметризованного цвета при рисовании с помощью нового шейдера:

```
// The shader
function createShader() {
  mShader = new SimpleShader(
    "http://gameproject/js/glsl_shader/simple_vs.glsl", // Path to VertexShader
    "http://gameproject/js/glsl_shader/simple_fs.glsl"); // Path to FragmentShader
                                  --------------
}
function drawSquare(color) {
                    -----
// Step A: Activate the shader
  mShader.activate(color);
                    -----
// Step B: Draw with currently activated geometry and shader
  mGL.drawArrays(mGL.TRIANGLE_STRIP, 0, 4);
}
```

Наконец, отредактируйте конструктор класса **MyGame**, чтобы включить цвет при рисовании квадрата, в данном случае красный:

```
// Step C: Draw the square in red
engine.drawSquare([1, 0, 0, 1]);
```

Обратите внимание, что значение цвета, массив из четырех значений с плавающей точкой, теперь требуется для нового шейдера **simple_fs.glsl** (вместо **white_fs**) и что важно передавать цвет рисунка при активации шейдера. С новым **simple_fs** теперь вы можете поэкспериментировать с рисованием квадратов любым

желаемым цветом.

Как вы уже поняли в этом проекте, структура исходного кода поддерживает простые и локализованные изменения при расширении или модификации игрового движка. В этом случае вносятся изменения только в **simple_shader.js** файл и незначительные изменения в **core.js** и **my_game.js** которые были необходимы. Это демонстрирует преимущества правильной инкапсуляции и организации исходного кода.

Резюме

На данный момент игровой движок прост и поддерживает только инициализацию WebGL и рисование одного цветного квадрата. Однако благодаря проектам, описанным в этой главе, вы приобрели опыт работы с методами, необходимыми для создания отличной основы для игрового движка. Вы также структурировали исходный код для поддержки дальнейшей сложности с ограниченными изменениями существующей кодовой базы, и теперь вы готовы к дальнейшей инкапсуляции функциональности игрового движка для обеспечения дополнительных функций. Следующая глава будет посвящена созданию надлежащей структуры в игровом движке для поддержки более гибких и настраиваемых чертежей.

Глава 3

Рисование объектов в окружающем мире

После завершения этой главы вы сможете

- Создавайте и рисуйте несколько прямоугольных объектов
- Управляйте положением, размером, поворотом и цветом созданных прямоугольных объектов
- Определите систему координат для рисования из
- Определите целевую подобласть на холсте для рисования
- Работа с абстрактными представлениями визуализируемых объектов, операторов преобразования и камер

Вступление

В идеале движок видеоигр должен обеспечивать надлежащие абстракции для поддержки проектирования и сборки игр в значимых контекстах. Например, при проектировании футбольного в игре вместо одного квадрата с фиксированным

диапазоном рисования ± 1,0 игровой движок должен предоставлять надлежащие утилиты для поддержки дизайна в контексте игроков, бегущих по футбольному полю. Эта высокоуровневая абстракция требует инкапсуляции базовых операций со скрытием данных и значимых функций для настройки и получения желаемых результатов.

Хотя эта книга посвящена созданию абстракций для игрового движка, в этой главе основное внимание уделяется созданию фундаментальных абстракций для поддержки рисования. Основываясь на примере футбольного матча, поддержка рисования в эффективном игровом движке, скорее всего, включите возможность легко создавать футболистов, контролировать их размер и ориентацию, а также позволять перемещать их и рисовать на футбольном поле. Кроме того, для поддержки надлежащего представления игровой движок должен позволять отображать на холсте определенные субрегионы, чтобы в разных субрегионах можно было отображать отдельный статус игры, например футбольное поле в одном субрегионе и статистику игроков и результаты в другом субрегионе.

В этой главе определяются надлежащие объекты абстракции для основных операций рисования, знакомит с операторами, основанными на фундаментальной математике для управления рисованием, рассматривает инструменты WebGL для настройки canvas для поддержки рисования объектов, определяет классы JavaScript для реализации этих концепций и интегрирует эти реализации в игровой движок, сохраняя организованную структуру исходного кода.

Инкапсулирующий чертеж

Хотя способность рисовать является одной из наиболее фундаментальных функций игрового движка, детали того, как реализуются рисунки, как правило, отвлекают от программирования игрового процесса. Например, важно создавать, контролировать расположение футболистов и рисовать их в футбольном матче. Однако раскрытие деталей того, как на самом деле определяется каждый игрок (набором вершин, образующих треугольники), может быстро перегружать и усложнять процесс разработки игры. Таким образом, для игрового движка важно предоставлять четко определенный абстрактный интерфейс для операций рисования.

При хорошо организованной структуре исходного кода возможно постепенно и систематически увеличивать сложность игрового движка путем внедрения новых концепций с локализованными изменениями в соответствующих папках. Первая задача состоит в том, чтобы расширить движок для поддержки инкапсуляции рисования таким образом, чтобы стало возможным манипулировать операциями рисования как логической сущностью или как объектом, который может быть визуализирован.

Примечание В контексте компьютерной графики и видеоигр слово "рендеринг"

относится к процессу изменения цвета пикселей, соответствующего абстрактному представлению. Например, в предыдущей главе вы узнали, как отобразить квадрат.

Проект визуализируемых объектов

В этом проекте вводится класс **Renderable** для инкапсуляции операции рисования. В течение следующих нескольких проектов вы узнаете больше вспомогательных концепций, позволяющих усовершенствовать реализацию класса **Renderable** таким образом, чтобы можно было создавать несколько экземпляров и управлять ими. На рисунке **3–1** показаны результаты выполнения проекта **Renderable Objects**. Исходный код этого проекта определен в папке **chapter3/3.1.renderable_objects**.

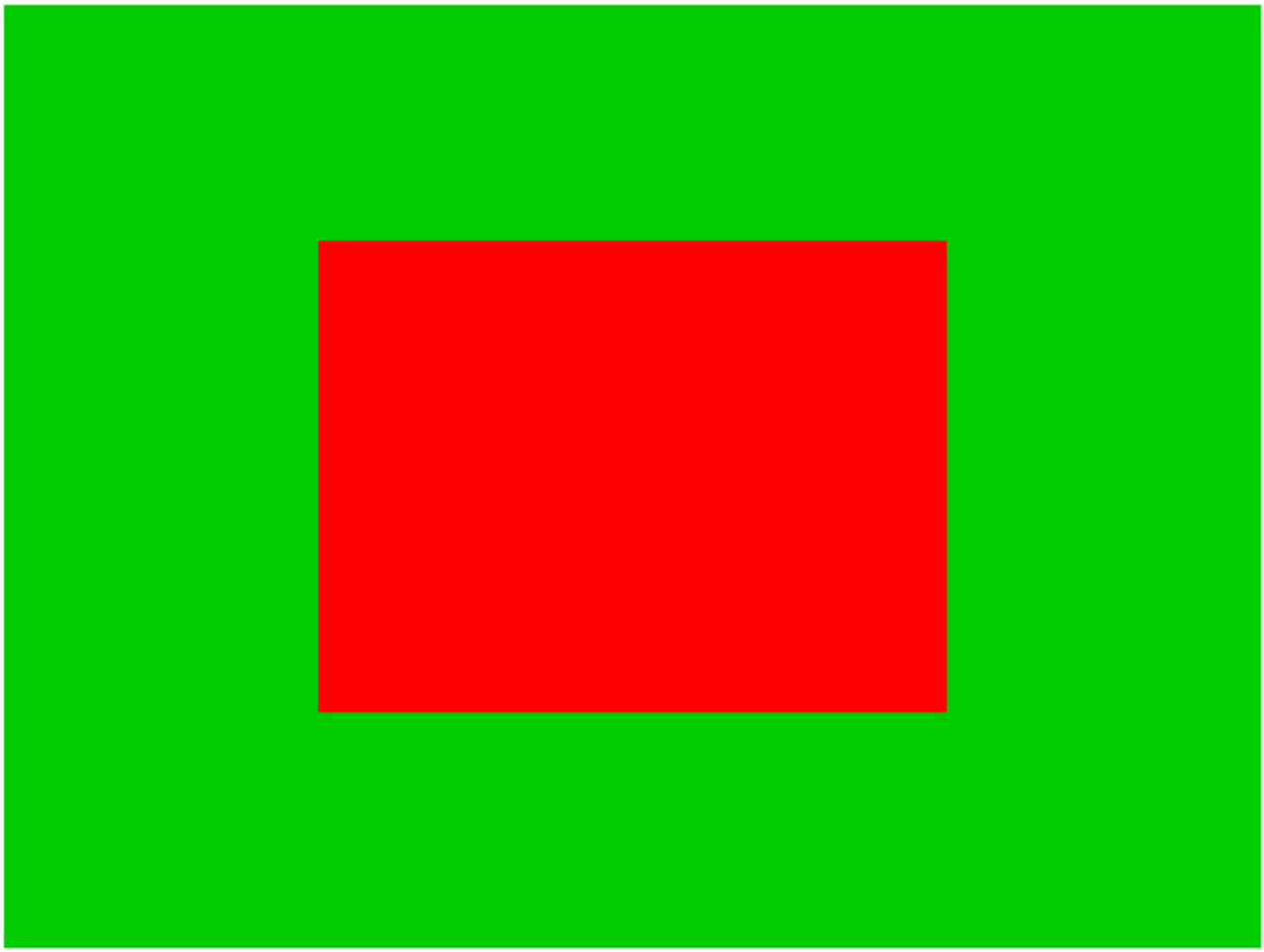

Рисунок **3–1**. Запуск проекта визуализируемых объектов

Цели проекта заключаются в следующем:

- Реорганизовать структуру исходного кода в ожидании увеличения функциональности
- Для поддержки совместного использования внутренних ресурсов игрового

движка

- Внедрить систематизированный интерфейс для разработчика игры через **index.js** файл
- Начать процесс создания класса для инкапсуляции операций рисования, сначала абстрагировав соответствующую функциональность рисования
- Продемонстрировать способность создавать несколько визуализируемых объектов

Реорганизация структуры исходного кода

Прежде чем внедрять дополнительные функциональные возможности в игровой движок, важно признать некоторые недостатки организации исходного кода движка по сравнению с предыдущим проектом. В частности, примите к сведению следующее:

1. **core.js** файл исходного кода содержит интерфейс WebGL, инициализацию движка и функции рисования. Они должны быть модульными, чтобы поддерживать ожидаемое увеличение сложности системы.
2. Должна быть определена система, поддерживающая совместное использование внутренних ресурсов игрового движка. Например, **SimpleShader** отвечает за взаимодействие игрового движка с шейдером GLSL, скомпилированным из файлов исходного кода **simple_vs.glsl** и **simple_fs.glsl**. Поскольку существует только одна копия скомпилированного шейдера, должен быть только один экземпляр объекта **SimpleShader**. Игровой движок должен способствовать этому, позволяя удобно создавать объект и совместно использовать его.
3. Как вы уже испытали, оператор экспорта JavaScript может быть отличным инструментом для сокрытия подробных реализаций. Однако также верно и то, что определение того, какие классы или модули импортировать выбор из нескольких файлов может привести к путанице и перегрузке в большой и сложной системе, такой как игровой движок, который вы собираетесь разрабатывать. Должен быть предоставлен простой в работе и систематизированный интерфейс, чтобы разработчик игры, пользователи игрового движка, могли быть изолированы от этих деталей.

В следующем разделе исходный код игрового движка будет реорганизован для решения этих проблем.

Определите модуль, специфичный

для WebGL

Первым шагом в реорганизации исходного кода является распознавание и изоляция функциональности, которая является внутренней и не должна быть доступна клиентам игрового движка:

1. В вашем проекте, в папке **js**/**engine**, создайте новую папку и назовите ее **core**. С этого момента эта папка будет содержать всю функциональность, которая является внутренней для игрового движка и не будет экспортироваться разработчикам игры.
2. Вы можете вырезать и вставить **vertex_buffer.js** файл исходного кода из предыдущего проекта в папку **js**/**engine**/**core**.
   Детали примитивных вершин являются внутренними для игрового движка и не должны быть видны или доступны клиентам игрового движка.
3. Создайте новый файл исходного кода в папке **js**/**engine**/**core**, назовите его **gl.js** , и определите методы инициализации и доступа WebGL.

```
"use strict"
let mCanvas = null;
let mGL = null;
function get() { return mGL; }
function init(htmlCanvasID) {
  mCanvas = document.getElementById(htmlCanvasID);
  if (mCanvas == null){
    throw new Error("Engine init [" +
    htmlCanvasID + "] HTML element id not found");
    // Get standard or experimental webgl and binds to the Canvas area
    // store the results to the instance variable mGL
  }
  mGL = mCanvas.getContext("webgl2") ||
  mCanvas.getContext("experimental-webgl2");

  if (mGL === null) {
    document.write("<br><b>WebGL 2 is not supported!</b>");
    return;
  }
}
export {init, get}
```

Обратите внимание, что функция **init**() идентична функции **initWebGL**() в **core.js** из предыдущего проекта. В отличие от предыдущего **core.js** файл исходного кода, **gl.js** файл содержит только функциональность, специфичную для WebGL.

Определите систему для

совместного использования внутренних ресурсов шейдера

Поскольку только одна копия шейдера GLSL создается и компилируется из файлов исходного кода **simple_vs.glsl** и **simple_fs.glsl**, в игровом движке требуется только одна копия объекта **SimpleShader** для взаимодействия со скомпилированным шейдером. Теперь вы создадите простую систему совместного использования ресурсов для поддержки будущих дополнений различных типов шейдеров.

Создайте новый файл исходного кода в папке **js/engine/core**, назовите его **shader_resources.js** , и определите методы создания и доступа к **SimpleShader**.

Примечание Напомним из предыдущей главы, что класс **SimpleShader** определен в **simple_shader.js** файл, который находится в папке **js/engine**. Не забудьте скопировать все соответствующие файлы исходного кода из предыдущего проекта.

```
"use strict";
import SimpleShader from "../simple_shader.js";
// Simple Shader
let kSimpleVS = "js/glsl_shaders/simple_vs.glsl"; // to VertexShader
let kSimpleFS = "js/glsl_shaders/simple_fs.glsl"; // to FragmentShader
let mConstColorShader = null;

function createShaders() {
  mConstColorShader = new SimpleShader(kSimpleVS, kSimpleFS);
}
function init() {
  createShaders();
}
function getConstColorShader() { return mConstColorShader; }
export {init, getConstColorShader}
```

Пимечание Переменные, ссылающиеся на постоянные значения, имеют имена, начинающиеся со строчной буквы “**k**”, как в **kSimpleVS**.

Поскольку модуль **shader_resources** расположен в папке **js/engine/core**, определенные шейдеры являются общими внутри и не могут быть доступны из клиентов игрового движка.

Определите файл доступа для

разработчика игры

Вы определите файл доступа к движку, **index.js** , для реализации основных функций игрового движка и для выполнения той же цели, что и заголовочный файл C++, инструкция **import** в Java или инструкция **using** в **C#**, где к функциональности можно легко получить доступ без глубоких знаний движка структура исходного кода. То есть, импортируя **index.js** , клиент может получить доступ ко всем компонентам и функциональности движка для создания своей игры.

1. Создайте **index.js** файл в папке **js/engine**; **import from gl.js** , **vertex_buffer.js** , и **shader_resources.js** ; и определите функцию **init()** для инициализации игрового движка путем вызова соответствующих функций **init()** трех импортированных модулей:

```
// local to this file only
import * as glSys from "./core/gl.js";
import * as vertexBuffer from "./core/vertex_buffer.js";
import * as shaderResources from "./core/shader_resources.js";

// general engine utilities
function init(htmlCanvasID) {
  glSys.init(htmlCanvasID);
  vertexBuffer.init();
  shaderResources.init();
}
```

2. Определите функцию **clearCanvas()**, чтобы очистить холст для рисования:

```
function clearCanvas(color) {
  let gl = glSys.get();
  gl.clearColor(color[0], color[1], color[2], color[3]);
  gl.clear(gl.COLOR_BUFFER_BIT); // clear to the color set
}
```

3. Теперь, чтобы должным образом предоставить **Renderable** символ клиентам игрового движка, убедитесь, что импортирован таким образом, чтобы класс можно было правильно экспортировать. Класс **Renderable** будет подробно представлен в следующем разделе.

```
// general utilities
import Renderable from "./renderable.js";
```

4. Наконец, не забудьте экспортировать соответствующие символы и функциональность для клиентов игрового движка:

```
export default {
  // Util classes
  Renderable,
  // functions
  init, clearCanvas
}
```

При надлежащем обслуживании и обновлении этого **index.js** файл, клиенты вашего игрового движка, разработчики игр, могут просто импортировать из **index.js** файл, чтобы получить доступ ко всей функциональности игрового движка без каких-либо знаний о структуре исходного кода. Наконец, обратите внимание, что внутренняя функциональность **glSys**, **vertexBuffer** и **shaderResources**, определенная в папке **js/engine/core**, не экспортируется **index.js** и следовательно, недоступны разработчикам игр.

Класс **Renderable**

Наконец, вы готовы определить класс **Renderable** для инкапсуляции процесса рисования:

1. Определите класс визуализации в игровом движке, создав новый файл исходного кода в папке **js/engine**, и назовите файл **renderable.js** .
2. Откройте **renderable.js** , импорт из **gl.js** и **shader_resources.js** и определите класс **Renderable** с конструктором для инициализации ссылки на шейдер и переменную экземпляра color. Обратите внимание, что шейдер является ссылкой на общий экземпляр **Simple Shader**, определенный в **shader_resources**.

```
import * as glSys from "./core/gl.js";
import * as shaderResources from "./core/shader_resources.js";

class Renderable {
  constructor() {
    this.mShader = shaderResources.getConstColorShader();
    this.mColor = [1, 1, 1, 1]; // color of pixel
  }
  ... implementation to follow ...
}
```

3 Определите метод **draw()** класса **Renderable**

```
draw() {
  let gl = glSys.get();
  this.mShader.activate(this.mColor);
  gl.drawArrays(gl.TRIANGLE_STRIP, 0, 4);
}
```

Обратите внимание, что важно активировать соответствующий шейдер GLSL в графическом процессоре, вызвав функцию **activate**() перед отправкой вершин с помощью функции **gl**.**drawArrays**()

4. Определите функции получения и установки для переменной **color**:

```
setColor(color) {this.mColor = color; }
getColor() { return this.mColor; }
```

5. Экспортируйте отображаемый символ по умолчанию, чтобы убедиться, что этот идентификатор не может быть переименован:

```
export default Renderable;
```

Благодаря этому простому примеру теперь можно создавать и отрисовывать несколько экземпляров **Renderable** объектов разными цветами.

Тестирование визуализируемого объекта

Чтобы протестировать **Renderable** объекты в **MyGame**, белые и красные экземпляры создаются и отрисовываются следующим образом:

```
// import from engine/index.js for all engine symbols
import engine from "../engine/index.js";
class MyGame {
  constructor(htmlCanvasID) {
  // Step A: Initialize the webGL Context
  engine.init(htmlCanvasID);
  // Step B: Create the Renderable objects:
  this.mWhiteSq = new engine.Renderable();
  this.mWhiteSq.setColor([1, 1, 1, 1]);
  this.mRedSq = new engine.Renderable();
  this.mRedSq.setColor([1, 0, 0, 1]);
  // Step C: Draw!
  engine.clearCanvas([0, 0.8, 0, 1]); // Clear the canvas
  // Step C1: Draw Renderable objects with the white shader
  this.mWhiteSq.draw();
  // Step C2: Draw Renderable objects with the red shader
  this.mRedSq.draw();
  }
}
```

Обратите внимание, что инструкция **import** изменена для импорта из папки **engine** файла **index.js** . Кроме того, конструктор **MyGame** изменен, чтобы включить следующие шаги:

1. Шаг **A** инициализирует движок.
2. Шаг **B** создает два экземпляра **Renderable** и соответствующим образом задает цвета объектов.
3. Шаг **C** очищает холст; шаги **C1** и **C2** просто вызывают соответствующие функции **draw()** для белого и красного квадратов. Хотя оба квадрата нарисованы, на данный момент вы можете видеть только последний из нарисованных квадратов на холсте. Пожалуйста, обратитесь к следующему обсуждению для получения подробной информации.

Наблюдения

Запустите проект, и вы заметите, что видна только красная площадь! Что происходит , так это то, что оба квадрата рисуются в одном и том же месте. Будучи одинакового размера, два квадрата просто идеально перекрываются. Поскольку красный квадрат рисуется последним, он перезаписывает все пиксели белого квадрата. Вы можете убедиться в этом, закомментировав рисунок красного квадрата (закомментируйте строку **mRedSq.draw()**) и повторно запустив проект. Интересное наблюдение заключается в том, что объекты, которые появляются спереди, рисуются последними (красный квадрат).
Вы воспользуетесь этим наблюдением гораздо позже, когда будете работать с

прозрачностью.

Это простое наблюдение приводит к вашей следующей задаче — сделать видимыми несколько экземпляров **Renderable** одновременно. Каждый экземпляр визуализируемого объекта должен поддерживать возможность рисования в разных местах, с разными размерами и ориентацией , чтобы они не перекрывали друг друга.

Преобразование **Renderable** объектов

Требуется механизм для управления положением, размером и ориентацией **Renderable** объекта. В следующих нескольких проектах вы узнаете о том, как матричные преобразования можно использовать для перевода или перемещения положения объекта, масштабирования размера объекта и изменения ориентации или поворота объекта на холсте. Эти операции являются наиболее интуитивно понятными для манипуляций с объектами. Однако перед недрением матриц преобразования требуется быстрый обзор операций и возможностей матриц.

Матрицы как операторы преобразования

Прежде чем мы начнем, важно признать, что матрицы и преобразования являются общими тематическими областями в математике. Нижеследующее обсуждение не является попыткой всесторонне охватить эти темы. Вместо этого основное внимание уделяется небольшому набору соответствующих концепций и операторов с точки зрения того, что требуется игровому движку. Таким образом, основное внимание уделяется тому, как использовать операторы, а не теории. Если вас интересуют особенности матриц и то, как они соотносятся с компьютерной графикой, пожалуйста, обратитесь к обсуждению в главе 1, где вы можете узнать подробнее об этих темах читайте в соответствующих книгах по линейной алгебре и компьютерной графике.

Матрица – это двумерный массив чисел из $\mathbf{m}$ строк и $\mathbf{n}$ столбцов. Для целей этого игрового движка вы будете работать исключительно с матрицами $\mathbf{4 \times 4}$. В то время как движок $2D$-игры мог бы обойтись матрицами $\mathbf{3 \times 3}$, матрица $\mathbf{4 \times 4}$ используется для поддержки функций, которые будут представлены в последующих главах. Среди множества мощных приложений матрицы $\mathbf{4 \times 4}$ могут быть построены как операторы преобразования для положений вершин. Наиболее важные и интуитивно понятными из этих операторов являются операторы перевода, масштабирования, поворота и идентификации.

• Оператор перевода **T(tx,ty)**, как показано на рисунке **3–2**, переводит или перемещает заданное положение вершины из **(x, y)** в **(x+tx, y+ty)**.Обратите внимание, что **T(0,0)** не изменяет значение заданного положения вершины и является удобным начальным значением для накопления операций перевода.

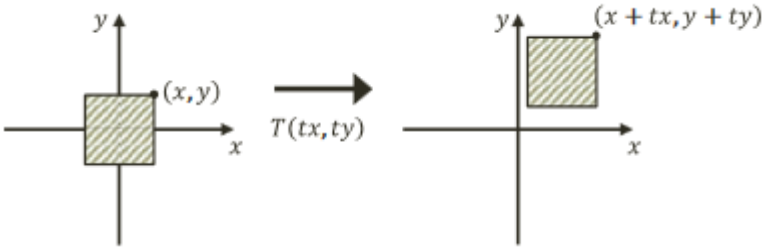

Рисунок **3–2**. Перевод квадрата на **T(tx, ty)**

• Оператор масштабирования **S(sx, sy)**, как показано на рисунке **3–3**, масштабирует или изменяет размер заданного положения вершины с **(x, y)** до **(x×sx, y×sy)**. Обратите внимание, что **S(1, 1)** не изменяет значение заданного положения вершины и является удобным начальным значением для накопления операций масштабирования.

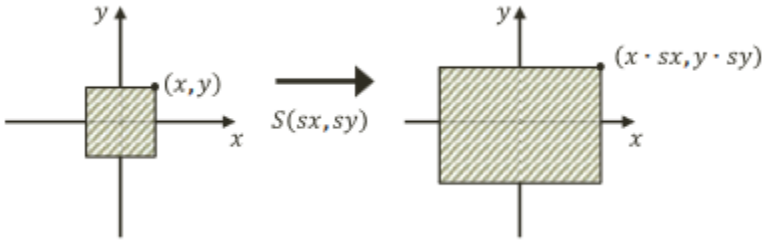

Рисунок **3–3**. Масштабирование квадрата на **S(sx, sy)**

• Оператор поворота **R(θ)**, как показано на рис. **3–4**, поворачивает заданное положение вершины относительно начала координат.

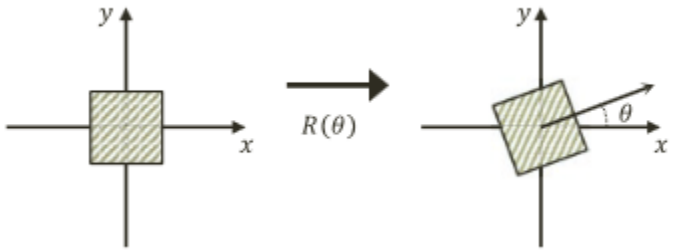

Рисунок **3–4**. Поворот квадрата на **R(θ)**

В случае поворота **R(0)** не изменяет значение данной вершины и является удобным начальным значением для накопления операций поворота. Значения для θ обычно выражаются в радианах (а не в градусах).

• Оператор идентификации **I** не влияет на заданное положение вершины.

Этот оператор в основном используется для инициализации.

В качестве примера идентификационная матрица 4×4 выглядит следующим образом:

$$I = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Математически оператор преобразования матрицы воздействует на вершину посредством умножения матрицы на вектор. Чтобы поддержать эту операцию, положение вершины $\mathbf{p} = (\mathbf{x}, \mathbf{y}, \mathbf{z})$ должно быть представлено в виде вектора $\mathbf{4x1}$ следующим образом:

$$p = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Примечание компонент $\mathbf{z}$ – это третье измерение, или информация о глубине, положения вершины. В большинстве случаев вам следует оставить $\mathbf{z}$–компонент равным $\mathbf{0}$.

Например, если позиция $\mathbf{p'}$ является результатом оператора преобразования $\mathbf{T}$, работающего с позицией вершины $\mathbf{p}$, математически $\mathbf{p'}$ будет вычисляться следующим образом:

$$p' = T \times p = Tp$$

Объединение матричных операторов

Несколько матричных операторов могут быть объединены в один оператор, сохраняя при этом те же характеристики преобразования, что и исходные операторы. Например,вы можете захотеть применить операторы масштабирования($\mathbf{S}$), за которыми следует оператор поворота ($\mathbf{R}$) и, наконец, оператор перевода($\mathbf{T}$), к заданному положению вершины или вычислить $\mathbf{p'}$ с помощью следующего:

$$p' = TRSp$$

В качестве альтернативы вы можете вычислить новый оператор $\mathbf{M}$, объединив все

операторы преобразования следующим образом:

$$M = TRS$$

А затем оператор **M** воздействует на положение вершины **p** следующим образом, чтобы получить идентичные результаты:

$$p' = Mp$$

Оператор M – это удобный и эффективный способ записи и повторного применения результатов нескольких операторов.

Наконец, обратите внимание, что при работе с операторами преобразования важен порядок выполнения операций. Например, операция масштабирования, за которой следует операция перевода, в общем случае отличается от перевода, за которым следует масштабирование или, в общем:

$$ST \neq TS$$

Библиотека glMatrix

Детали матричных операторов и операций, мягко говоря, нетривиальны. Разработка полной библиотеки матриц отнимает много времени и не является целью этой книги. К счастью, существует множество хорошо разработанных и хорошо документированных библиотек матриц, доступных в общественном достоянии. Библиотека **glMatrix** является одним из таких примеров. Чтобы интегрировать эту библиотеку в структуру вашего исходного кода, выполните следующие действия:

1. Создайте новую папку в папке **js** и назовите новую папку **lib**
2. Перейдите к **http://glMatrix.net** , как показано на рисунке **3–5**, и загрузите, распакуйте и сохраните полученный **glMatrix.js** исходный файл в новую папку **lib**.

Рисунок **3–5**. Загрузка библиотеки **glMatrix**

Все проекты в этой книге основаны на **glMatrix** версии 2.2.2.

3. В качестве библиотеки, которая должна быть доступна как игровому движку, так и разработчику клиентской игры, вы загрузите исходный

файл в основной **index.html** добавив следующее перед загрузкой **my_game.js**:

```
<!-- external library -->
<script type="text/javascript" src="js/lib/gl-matrix.js"></script>

<!-- our game -->
<script type="module" src="./js/my_game/my_game.js"></script>
```

Проект преобразования матрицы

Этот проект знакомит и демонстрирует, как использовать матрицы преобразования в качестве операторов для управления положением, размером и ориентацией **Renderable** объектов, нарисованных на холсте. Таким образом, **Renderable** объект теперь можно отрисовать в любом месте, с любым размером и любой ориентацией. На рисунке **3–6** показаны результаты выполнения проекта преобразования матрицы. Исходный код этого проекта определен в папке главы **3/3.2.matrix_transform**.

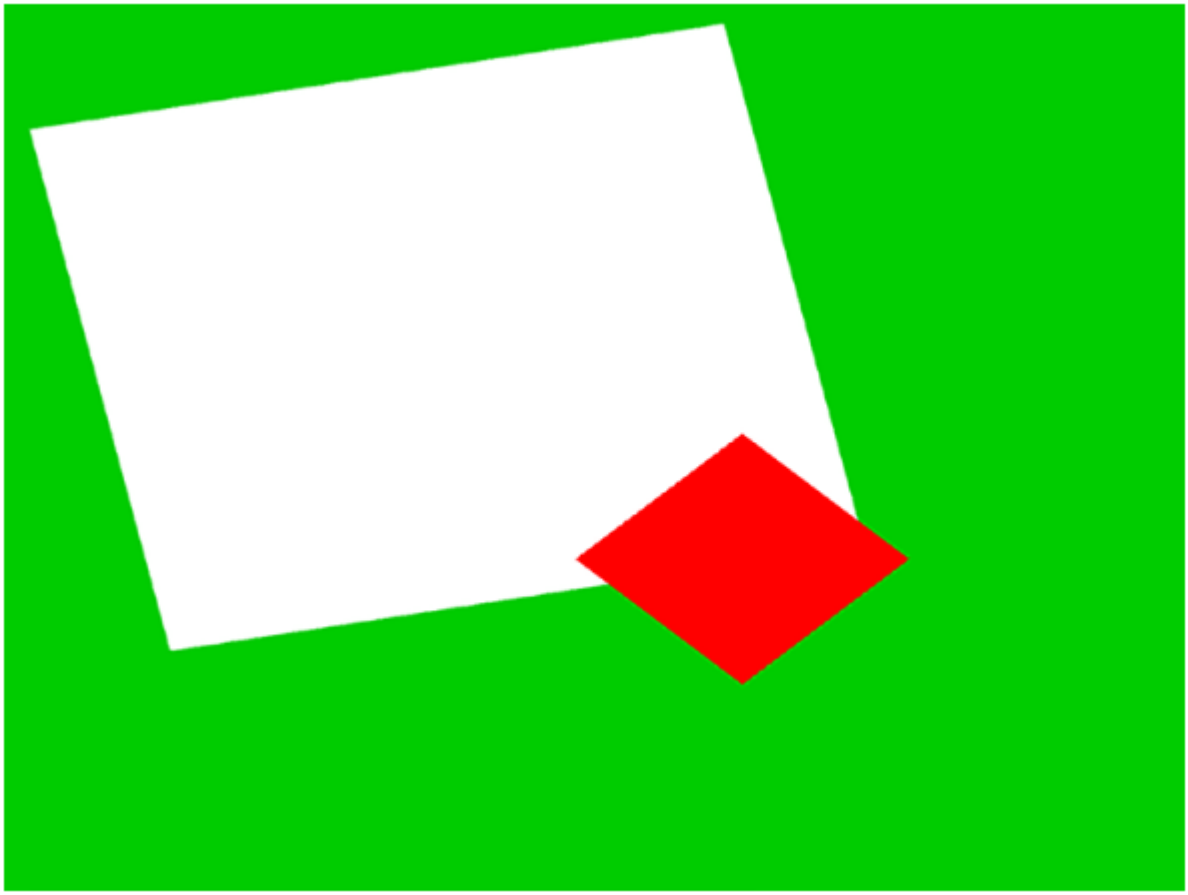

Рисунок **3–6**. Запуск проекта преобразования матрицы

Цели проекта заключаются в следующем:

- Ввести матрицы преобразования в качестве операторов для построения **Renderable**
- Чтобы понять, как работать с операторами преобразования для манипулирования отображаемым

Изменение Vertex Shader для поддержки преобразований

Как обсуждалось, операторы матричного преобразования работают с вершинами геометрий. Вершинный шейдер – это место, куда передаются все вершины из контекста WebGL, и это наиболее удобное место для применения операций преобразования.

Вы продолжите работу с предыдущим проектом для поддержки оператора преобразования в вершинном шейдере:

1. Отредактируйте **simple_vs.glsl**, чтобы объявить однородную матрицу **4×4**:

Примечание Напомним из обсуждения в главе 2, что файлы **glsl** содержат инструкции **Opengl shading language** (**glsl**), которые будут загружены в графический процессор и выполнены им. Вы можете узнать больше о **glsl**, обратившись к ссылкам на **webgl** и **Opengl**, приведенным в конце главы 1.

Напомним, что ключевое слово **uniform** в шейдере GLSL объявляет переменную со значениями, которые не изменяются для всех вершин внутри этого шейдера. В этом случае переменная **uModelXformMatrix** является оператором преобразования для всех вершин.

Примечание унифицированные имена переменных **glsl** всегда начинаются со строчной буквы "**u**", как в **uModelXformMatrix**.

---

2. В функции **main**() примените **uModelXformMatrix** к текущей позиции вершины, на которую ссылается:

```
gl_Position = uModelXformMatrix * vec4(aVertexPosition, 1.0);
```

Обратите внимание, что операция непосредственно вытекает из обсуждения операторов преобразования матрицы. Причина преобразования **aVertexPosition** в **vec4** заключается в поддержке умножения матрицы на вектор.

С помощью этой простой модификации положения вершин единичного квадрата будут обрабатываться оператором **uModelXformMatrix**, и, таким образом, квадрат может быть нарисован в разных местах. Теперь задача состоит в том, чтобы настроить **SimpleShader** для загрузки соответствующего оператора преобразования в **uModelXformMatrix**.

Измените SimpleShader для загрузки оператора преобразования

Выполните следующие действия:

1. Отредактируйте **simple_shader.js** и добавьте переменную экземпляра для хранения ссылки на матрицу **uModelXformMatrix** в вершинном шейдере:

```
this.mModelMatrixRef = null;
```

2. В конце конструктора **SimpleShader** на шаге **E**, после установки ссылки на **uPixelColor**, добавьте следующий код для инициализации этой ссылки:

```
// Step E: Gets a reference to uniform variables in fragment shader
this.mPixelColorRef  = gl.getUniformLocation(
                       this.mCompiledShader, "uPixelColor");
this.mModelMatrixRef = gl.getUniformLocation(
                       this.mCompiledShader, "uModelXformMatrix");
```

3. Измените функцию **activate()**, чтобы получить второй параметр, и загрузите значение в **uModelXformMatrix** через **mModelMatrixRef**

```
activate(pixelColor, trsMatrix) {
  let gl = glSys.get();
  gl.useProgram(this.mCompiledShader);
  ... identical to previous code ...
  // load uniforms
  gl.uniform4fv(this.mPixelColorRef, pixelColor);
  gl.uniformMatrix4fv(this.mModelMatrixRef, false, trsMatrix);
}
```

Функция **gl.uniformMatrix4fv()** копирует значения из **trsMatrix** в местоположение вершинного шейдера, указанное **this.mModelMatrixRef** или оператор **uModelX from Matrix** в вершинном шейдере. Имя переменной, **trsMatrix**, означает, что это должен быть матричный оператор, содержащий объединенный результат преобразования (**T**), поворота (**R**) и масштабирования (**S**) или **TRS**.

Измените Renderable класс, чтобы

задать оператор преобразования

Редактировать **renderable.js** чтобы изменить функцию **draw()** для получения и пересылки оператора преобразования в функцию **mShader.activate()** для загрузки в шейдер GLSL:

```
draw(trsMatrix) {
  let gl = glSys.get();
  this.mShader.activate(this.mColor, trsMatrix);
  gl.drawArrays(gl.TRIANGLE_STRIP, 0, 4);
}
```

Таким образом, когда вершины единичного квадрата обрабатываются вершинным шейдером, **uModelXformMatrix** будет содержать соответствующий оператор для преобразования вершин и, таким образом, рисования квадрата в желаемом местоположении, размере и повороте.

Тестирование преобразований

Теперь, когда игровой движок поддерживает преобразование, вам нужно изменить клиентский код , чтобы рисовать с его помощью:

1. Редактировать **my_game.js** ; после шага **C**, вместо активации и рисования двух квадратов, замените шаги **C1** и **C2**, чтобы создать новый оператор преобразования идентификатора, **trsMatrix**:

```
// create a new identify transform operator
let trsMatrix = mat4.create();
```

2. Вычислите объединение матриц с помощью одного оператора преобразования, который реализует перевод (T), поворот (R) и масштабирование (S) или TRS:

В книге ошибка при обращении к библиотеки, нужно дописывать объект **glMatrix**

```
glMatrix.mat4
```

```
glMatrix.vec3
```

```
// Step D: compute the white square transform
mat4.translate(trsMatrix, trsMatrix, vec3.fromValues(-0.25, 0.25, 0.0));
mat4.rotateZ(trsMatrix, trsMatrix, 0.2); // rotation is in radian
mat4.scale(trsMatrix, trsMatrix, vec3.fromValues(1.2, 1.2, 1.0));
// Step E: draw the white square with the computed transform
this.mWhiteSq.draw(trsMatrix);
```

Шаг D объединяет T (–0.25, 0.25), перемещаясь влево и вверх; с R (0.2), поворачиваясь по часовой стрелке на 0.2 радиана; и S (1.2, 1.2), увеличивая размер в

1.2 раза. Порядок объединения сначала применяет оператор масштабирования, за которым следует поворот, причем перевод является последней операцией, или **trsMatrix=TRS**. На шаге **E Renderable** объект рисуется с помощью оператора **trsMatrix** или белого прямоугольника 1.2=1.2, слегка повернутого и расположенного несколько слева вверху от центра.

3. Наконец, шаг **F** определяет оператор **trsMatrix**, который позволяет нарисовать квадрат размером **0,4 × 0,4**, повернутый на 45 градусов и расположенный немного в правом нижнем углу от центра холста, а шаг **G** рисует красный квадрат:

```
// Step F: compute the red square transform
mat4.identity(trsMatrix); // restart
mat4.translate(trsMatrix, trsMatrix, vec3.fromValues(0.25, -0.25, 0.0));
mat4.rotateZ(trsMatrix, trsMatrix, -0.785); // about -45-degrees
mat4.scale(trsMatrix, trsMatrix, vec3.fromValues(0.4, 0.4, 1.0));
// Step G: draw the red square with the computed transform
this.mRedSq.draw(trsMatrix);
```

Наблюдения

Запустите проект, и вы должны увидеть соответствующие белые и красные прямоугольники, нарисованные на холсте. Вы можете получить некоторое представление об операторах, изменив значения; например, переместите и масштабируйте квадраты в разные места с разными размерами. Вы можете попробовать изменить порядок объединения, переместив соответствующую строку кода; например, переместите **mat4.scale()** перед **mat4.translate()**. Вы заметите, что, в целом, преобразованные результаты не соответствуют вашей интуиции. В этой книге вы всегда будет применять операторы преобразования в фиксированном порядке TRS. Такое упорядочение операторов преобразования соответствует типичной человеческой интуиции. Порядку выполнения операций TRS следуют большинство, если не все, графических API и приложений, поддерживающих операции преобразования.

Теперь, когда вы понимаете, как работать с операторами преобразования матрицы, пришло время абстрагировать их и скрыть их детали.

Инкапсуляция оператора преобразования

В предыдущем проекте операторы преобразования вычислялись непосредственно на основе матриц. Хотя результаты были важны, вычисление связано с отвлекающими деталями и повторяющимся кодом. Этот проект поможет вам

следовать рекомендациям по кодированию, чтобы инкапсулировать операторы преобразования, скрыв подробные вычисления с помощью класса. Таким образом, вы можете сохранить модульность и доступность игрового движка, поддерживая дальнейшее расширение при сохранении программируемости.

Проект "Преобразование объектов"

Этот проект определяет класс **Transform**, чтобы обеспечить логический интерфейс для манипулирования и сокрытия деталей работы с операторами преобразования матрицы. Рисунок 3-7 показывает результат выполнения проекта преобразования матрицы. Обратите внимание, что выходные данные этого проекта идентичны результатам предыдущего проекта. Исходный код этого проекта определен в папке главы **3/3.3.transform_objects**

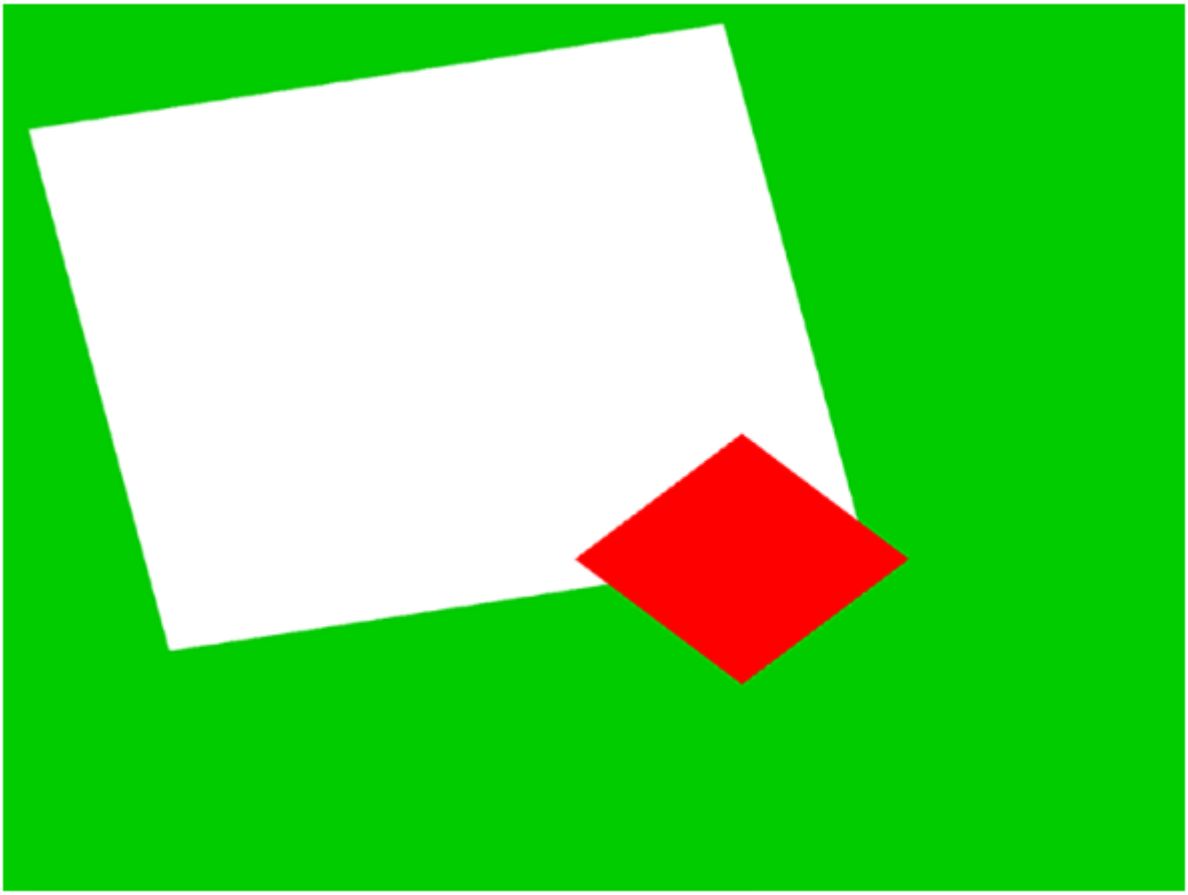

Рисунок **3-7**. Запуск проекта "Преобразование объектов"

Целями проекта являются следующие:

- Создать класс **Transform** для инкапсуляции функциональности преобразования матрицы
- Интегрировать класс **Transform** в игровой движок
- Продемонстрировать, как работать с объектами класса **Transform**

Класс Transform

Продолжайте работать с предыдущим проектом:

1. Определите класс **Transform** в игровом движке, создав новый файл исходного кода в папке **js/engine**, и назовите файл **transform.js**.
2. Определите конструктор для инициализации переменных экземпляра, соответствующих операторам: **mPosition** для перемещения, **mScale** для масштабирования, и **mRotationInRad** для вращения.

```
class Transform {
  constructor() {
    this.mPosition = vec2.fromValues(0, 0); // translation
    this.mScale = vec2.fromValues(1, 1); // width (x), height (y)
    this.mRotationInRad = 0.0; // in radians!
  }
  ... implementation to follow ...
}
```

3. Добавьте средства получения(**getters**) и настройки(**setters**) для значений каждого оператора:

```
// Position getters and setters
setPosition(xPos, yPos) { this.setXPos(xPos); this.setYPos(yPos); }
getPosition() { return this.mPosition; }
// ... additional get and set functions for position not shown
// Size setters and getters
setSize(width, height) {
  this.setWidth(width);
  this.setHeight(height);
}
getSize() { return this.mScale; }
// ... additional get and set functions for size not shown
// Rotation getters and setters
setRotationInRad(rotationInRadians) {
  this.mRotationInRad = rotationInRadians;
  while (this.mRotationInRad > (2 * Math.PI)) {
    this.mRotationInRad -= (2 * Math.PI);
  }
}
setRotationInDegree(rotationInDegree) {
  this.setRotationInRad(rotationInDegree * Math.PI / 180.0);
}
// ... additional get and set functions for rotation not shown
```

Законченое решение

```
// Position getters and setters
setPosition(xPos, yPos) { this.setXPos(xPos); this.setYPos(yPos); }
getPosition() { return this.mPosition; }
setXPos(xPos){this.mPosition[0] = xPos}
setYPos(yPos){this.mPosition[1] = yPos}
getXPos(){return this.mPosition[0]}
getYPos(){return this.mPosition[1]}

// Size setters and getters
setSize(width, height) {
  this.setWidth(width);
  this.setHeight(height);
}
getSize() { return this.mScale; }
setWidth(width){this.mScale[0] = width}
setHeight(height){this.mScale[1] = height}
getWidth(){return this.mScale[0]}
getHeight(){return this.mScale[1]}
// Rotation getters and setters

setRotationInRad(rotationInRadians) {
  this.mRotationInRad = rotationInRadians;
  while (this.mRotationInRad > (2 * Math.PI)) {
    this.mRotationInRad -= (2 * Math.PI);
  }
}

setRotationInDegree(rotationInDegree) {
  this.setRotationInRad(rotationInDegree * Math.PI / 180.0);
}
getRotationInRad(){
  return this.mRotationInRad
}
```

4 Определите функцию **getTRSMatrix()** для вычисления и возврата объединенного оператора преобразования, TRS

```
getTRSMatrix() {
  // Creates a blank identity matrix
  let matrix = mat4.create();
  // Step A: compute translation, for now z is always at 0.0
  mat4.translate(matrix, matrix,
  vec3.fromValues(this.getXPos(), this.getYPos(), 0.0));
  // Step B: concatenate with rotation.
  mat4.rotateZ(matrix, matrix, this.getRotationInRad());
  // Step C: concatenate with scaling
  mat4.scale(matrix, matrix,
  vec3.fromValues(this.getWidth(), this.getHeight(), 1.0));
  return matrix;
}
```

Этот код аналогичен шагам **D** и **F** в **my_game.js** из предыдущего проекта. Объединенный оператор TRS сначала выполняет масштабирование, затем поворот и, наконец перевод.

5.Наконец, не забудьте экспортировать недавно определенный класс **Transform**:

```
export default Transform;
```

Преобразуемый визуализируемый класс

Благодаря интеграции класса **Transform** отображаемый объект теперь может иметь положение, размер (масштаб) и ориентацию (поворот). Эта интеграция может быть легко осуществлена с помощью следующих шагов:

1.Отредактируйте **renderable.js** и добавьте новую переменную экземпляра для ссылки на объект **Transform** в конструкторе:

```
this.mXform = new Transform(); // transform operator for the object
```

2.Определите средство доступа для оператора преобразования:

```
getXform() { return this.mXform; }
```

3.Измените функцию draw(), чтобы передать оператор **trsMatrix** объекта **mXform** для активации шейдера перед рисованием единичного квадрата:

```
draw(trsMatrix) {
  let gl = glSys.get();
  this.mShader.activate(this.mColor,this.mXform.getTRSMatrix());
  gl.drawArrays(gl.TRIANGLE_STRIP, 0, 4);
}
```

С помощью этой простой модификации **Renderable** объекты будут отрисовываться с характеристиками, определяемыми значениями его собственных операторов преобразования.

Измените файл доступа к движку для экспорта Transform

Важно поддерживать файл доступа к движку, index.js , обновленный таким образом, чтобы разработчик игры мог получить доступ к недавно определенному классу **Transform**.:

1. Отредактируйте **index.js** ; импортировав из вновь определенного файла **transform.js** :

```
// general utilities
import Transform from "./transform.js";
import Renderable from "./renderable.js";
```

2. Экспортируйте **Transform** для доступа клиента:

```
export default {
  // Util classes
  Transform,Renderable,
  // functions
  init, clearCanvas
}
```

Измените чертеж для поддержки объекта Transform

Чтобы протестировать **Transform** и **Renderable** классы, конструктор **MyGame** можно модифицировать, чтобы соответствующим образом задать операторы преобразования в каждом из **Renderable** объектов:

```
// Step D: sets the white Renderable object's transform
this.mWhiteSq.getXform().setPosition(-0.25, 0.25);
this.mWhiteSq.getXform().setRotationInRad(0.2); // In Radians
this.mWhiteSq.getXform().setSize(1.2, 1.2);
// // Step E: draws the white square (transform behavior in the object)
this.mWhiteSq.draw();
// Step F: sets the red square transform
this.mRedSq.getXform().setXPos(0.25); // alternative to setPosition
this.mRedSq.getXform().setYPos(-0.25);// setX/Y separately
this.mRedSq.getXform().setRotationInDegree(45); // this is in Degree
this.mRedSq.getXform().setWidth(0.4); // alternative to setSize
this.mRedSq.getXform().setHeight(0.4);// set width/height separately
// Step G: draw the red square (transform in the object)
this.mRedSq.draw();
```

Запустите проект, чтобы получить результаты, идентичные результатам предыдущего проекта. Теперь вы можете создавать и рисовать визуализируемый объект в любом месте холста, и оператор преобразования теперь правильно инкапсулирован.

Преобразование камеры и видовые экраны

При проектировании и построении видеоигры геймдизайнеры и программисты должны уметь сосредоточиться на внутренней логике и презентации. Чтобы облегчить эти аспекты, важно, чтобы дизайнеры и программисты могли формулировать решения в удобном измерении и пространстве.

Например, продолжая идею футбольного матча, рассмотрим задачу создания футбольного поля. Насколько велико это поле? Что такое единица измерения? В целом, при создании игрового мира часто бывает проще разработать решение, обратившись к реальному миру. В реальном мире футбольные поля имеют длину около 100 метров. Однако в игровом или графическом мире единицы измерения являются произвольными. Итак, простым решением может быть создание поля размером 100 единиц в метрах и координатного пространства, где начало координат расположено в центре футбольного поля. Таким образом, противоположные стороны полей можно просто определить по знаку значения x, и розыгрыш игрока в местоположении $(0, 1)$ будет означать розыгрыш игрока в 1 метре справа от центра футбольного поля.

Контрастным примером может служить создание настольной игры, похожей на шахматы. Это может быть удобнее проектировать решение на основе безразмерной сетки $n \times n$ с началом координат, расположенным в левом нижнем углу доски. В этом сценарии рисование фигуры в местоположении $(0, 1)$ означало бы рисование фигуры

в местоположении на одну ячейку или единицу вправо от нижнего левого угла доски. Как будет обсуждаться, возможность определения конкретных систем координат часто достигается путем вычисления и работы с матрицей, представляющей вид с камеры.

Во всех случаях, чтобы обеспечить надлежащую презентацию игры, важно обеспечить программатор для управления нанесением содержимого в любое место на холсте. Например, вы можете захотеть отрисовать футбольное поле и игроков в одном субрегионе и нарисовать мини-карту в другом субрегионе. Эти выровненные по оси прямоугольные области рисования или подобласти холста называются видовыми экранами.

В этом разделе вы узнаете о системах координат и о том, как использовать матричное преобразование в качестве инструмента для определения области рисования, соответствующей фиксированному диапазону рисования ±1 в WebGL.

Системы координат и преобразования

Двумерная система координат однозначно идентифицирует каждое положение на двумерной плоскости. Все проекты в этой книге выполняются в декартовой системе координат, где положения определяются в соответствии с перпендикулярными расстояниями от контрольной точки, известной как начало координат(**origin**), как показано на рис. 3-8. Перпендикулярные направления для измерения расстояний известны как главные оси(**major axes**). В двумерном пространстве это знакомые оси x и y.

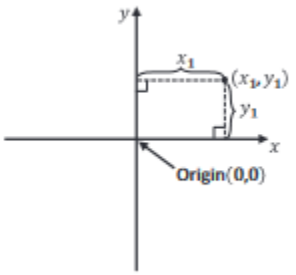

Рисунок **3-8**. Работа с двумерной декартовой системой координат

Моделирование и Нормализованные системы координат устройств

До сих пор в этой книге у вас был опыт работы с двумя различными системами координат. Первая – это система координат, которая определяет вершины для квадрата 1×1 в буфере вершин. Это называется системой координат моделирования, которая определяет пространство модели. Пространство модели уникально для каждого геометрического объекта, как и в случае с единичным квадратом. Пространство модели определено для описания геометрии отдельной модели. Вторая система координат, с которой вы работали, – это та, к которой обращается WebGL, где диапазоны осей x / y ограничены ±1.0. Это известно как Нормализованное устройство Система координат (**NDC**). Как вы уже испытали, WebGL всегда рисует в пространстве **NDC** и что содержимое в диапазоне ±1,0 покрывает все пиксели на холсте.

Преобразование моделирования, обычно определяемое оператором преобразования матрицы, – это операция, которая преобразует геометрию из пространства модели в другое координатное пространство, удобное для рисования. В предыдущем проекте переменная **uModelXformMatrix** в **simple_vs.glsl** является преобразованием моделирования. Как показано на рис. 3-9, в этом случае преобразование моделирования преобразовало единичный квадрат в NDC WebGL пространство. Крайняя правая стрелка, помеченная меткой фиксированного отображения на рис. 3-9, которая указывает от WebGL NDC к координатам Canvas, означает, что WebGL всегда отображает все содержимое пространства NDC на canvas.

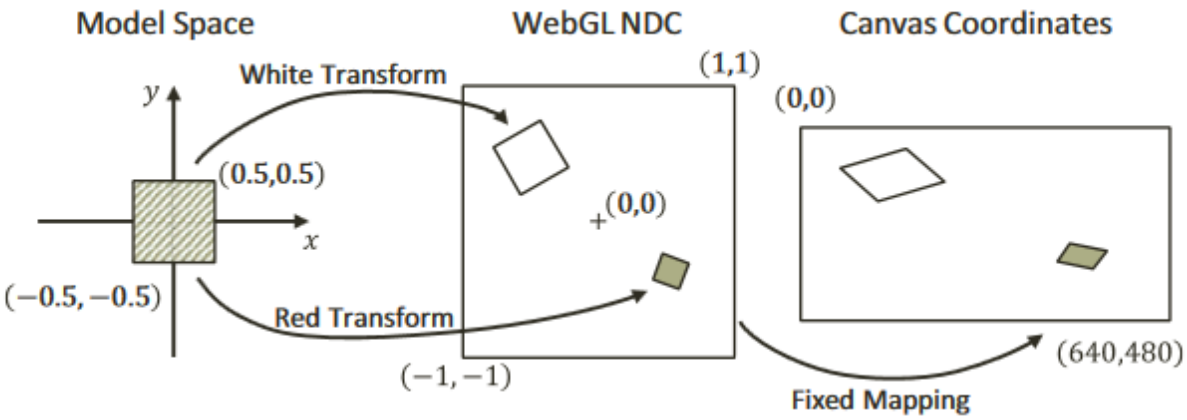

Рисунок **3–9**. Преобразование квадрата из модели в пространство NDC

Мировая система координат

Хотя с помощью преобразования моделирования можно рисовать в любом месте, непропорциональное масштабирование, при котором квадраты рисуются в виде прямоугольников, по-прежнему является проблемой. Кроме того, пространство NDC с фиксированными значениями -1.0 и 1.0 не является удобным пространством координат для разработки игр. Мировая система координат (**WC**) описывает удобное мировое пространство, которое решает эти проблемы. Для удобства и

удобочитаемости в остальной части этой книги WC также будет использоваться для обозначения мирового пространства, которое определяется определенной Мировой Координатой Системой.

Как показано на рис. 3-10, при использовании WC вместо фиксированного пространства NDC преобразования моделирования могут преобразовывать модели в удобную систему координат, пригодную для разработки игр. В примере с футбольным матчем измерением мирового пространства может быть размер футбольного поля. Как и в любой декартовой системе координат, система WC определяется исходным положением, а также его шириной и высотой. Исходным положением может быть либо левый нижний угол, либо центр туалета.

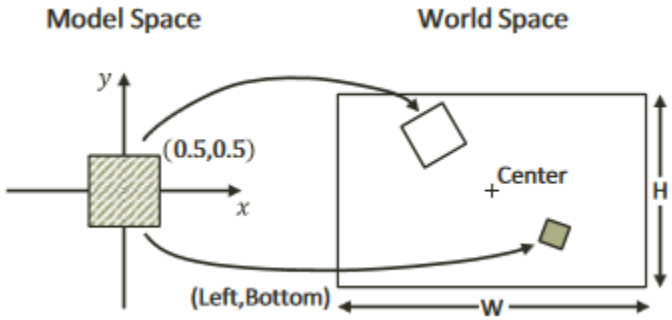

Рисунок **3-10**. Работа с мировой системой координат (WC)

WC - это удобная система координат для проектирования игр. Однако это не то пространство, к которому обращается WebGL. По этой причине важно перейти от **WC** к **NDC**. В этой книге это преобразование называется преобразованием камеры. Чтобы выполнить это преобразование, вам нужно будет создать оператор для выравнивания центра **WC** по центру **NDC** (начало координат), а затем масштабировать размер WC WxH в соответствии с шириной и высотой NDC. Обратите внимание, что пространство NDC имеет постоянный диапазон от -1 до +1 и, следовательно, фиксированный размер 2x2. Таким образом, преобразование камеры - это просто перевод, за которым следует операция масштабирования:

$$M = S\left(\frac{2}{W}, \frac{2}{H}\right) T\left(-center.x, -center.y\right)$$

В этом случае (**center**.**x**, **center**.**y**) и **WxH** являются центром и размерами системы **WC**.

Окно просмотра(**Viewport**)

Видовой экран(Viewport) - это область, на которую нужно обратить внимание. Как вы уже поняли, по умолчанию WebGL определяет весь холст в качестве видового экрана для рисования. Удобно, что WebGL предоставляет функцию для переопределения этого поведения по умолчанию:

```
gl.viewport(
  x, // x position of bottom-left corner of the area to be drawn
  y, // y position of bottom-left corner of the area to be drawn
  width, // width of the area to be drawn
  height // height of the area to be drawn
);
```

Функция **gl.viewport()** определяет видовой экран(**Viewport**) для всех последующих чертежей. На **рис. 3-11** показано преобразование камеры и рисование с помощью видового экрана.

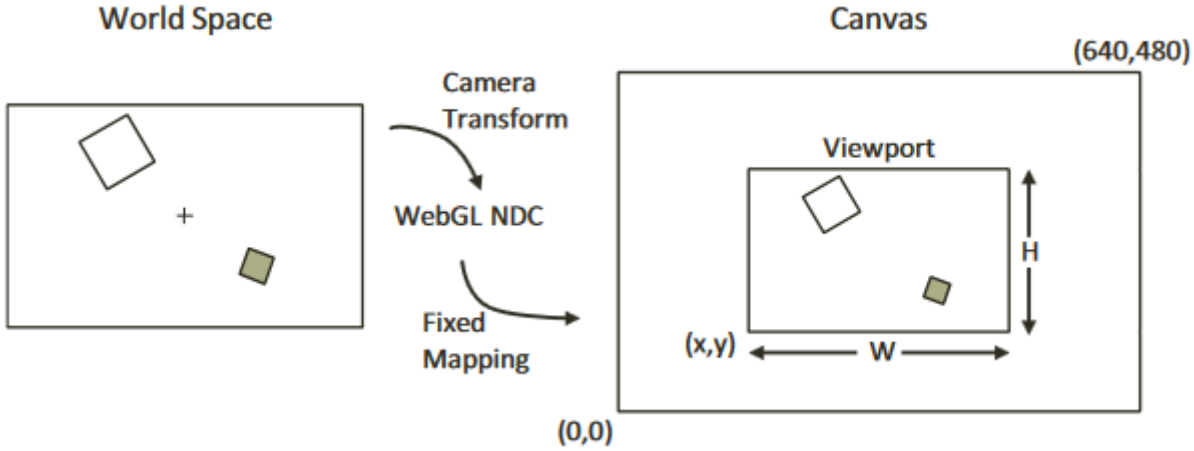

Рисунок 3-11. Работа с видовым экраном WebGL

Преобразование камеры и проект видового экрана

Этот проект демонстрирует, как использовать преобразование камеры для рисования из любого желаемого местоположения с координатами в любую область холста или видового экрана. На рис. 3-12 показаны результаты выполнения проекта преобразования камеры и видового экрана. Исходный код этого проекта определен в папке главы **3/3.4.camera_transform_and_viewport**

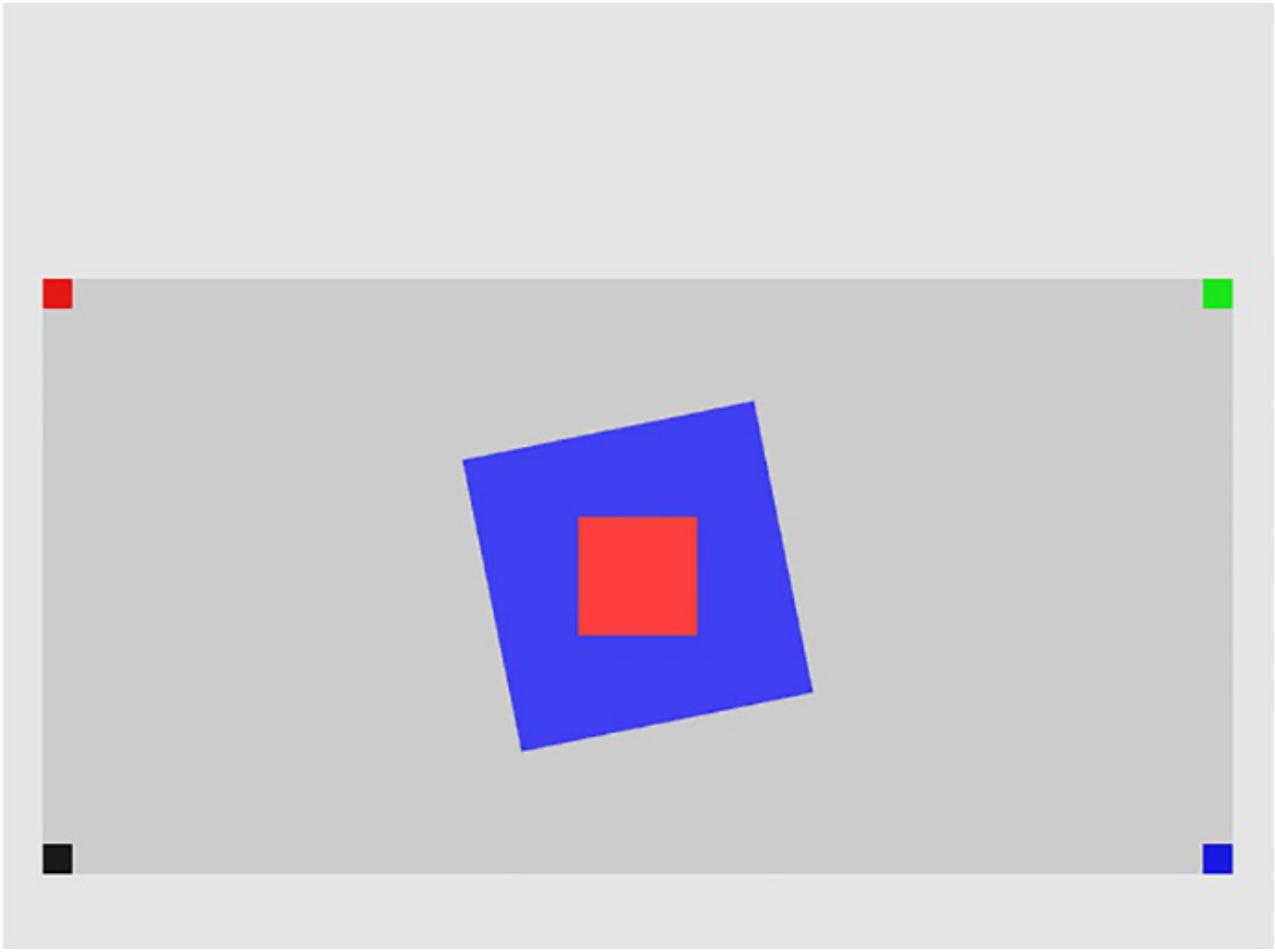

Рисунок 3-12. Запуск проекта преобразования камеры и видового экрана

Целями проекта являются следующие:

- Понимать различные системы координат
- Получить опыт работы с видовым экраном WebGL для определения и рисования различных подобластей на холсте
- Понимания преобразования камеры
- Начать отрисовку в определенной пользователем мировой системе координат

Теперь вы готовы модифицировать игровой движок для поддержки преобразования камеры, чтобы определить свой собственный WC и соответствующий видовой экран для рисования. Первым шагом является изменение шейдеров для поддержки нового оператора преобразования.

Измените вершинный шейдер для поддержки преобразования камеры

Для добавления поддержки преобразования камеры требуются относительно незначительные изменения:

1. Отредактируйте **simple_vs.glsl**, чтобы добавить новый оператор однородной матрицы для представления преобразования камеры:

```
uniform mat4 uCameraXformMatrix;
```

2.Обязательно примените оператор к позициям вершин в программе **vertex shader:**

```
uniform mat4 uCameraXformMatrix;
attribute vec3 aVertexPosition;
uniform mat4 uModelXformMatrix;
void main(void) {
  // gl_Position = vec4(aVertexPosition, 1.0);
  gl_Position = uCameraXformMatrix * uModelXformMatrix * vec4(aVertexPosition, 1.0);
}
```

Напомним, что порядок выполнения матричных операций важен. В этом случае **uModelXformMatrix** сначала преобразует положения вершин из пространства модели в WC, а затем **uCameraXformMatrix** преобразует из WC в NDC. Порядок матриц **uModelXformMatrix** и **uCameraXformMatrix** не может быть изменен.

Измените SimpleShader для поддержки преобразования камеры

Объект **SimpleShader** должен быть изменен, чтобы получить доступ к матрице преобразования камеры и передать ее вершинному шейдеру:

1.Редактировать **simple_shader.js** и в **конструкторе** добавьте переменную экземпляра для хранения ссылки на оператор преобразования камеры в **simple_vs.glsl**

```
this.mCameraMatrixRef = null;
```

2.В конце **SimpleShader** извлеките ссылку на оператор преобразования камеры, **uCameraXformMatrix**, после извлечения ссылок для **uModelXformMatrix** и **uPixelColor:**

```
// Step E: Gets reference to uniform variables in fragment shader
this.mPixelColorRef = gl.getUniformLocation(this.mCompiledShader, "uPixelColor");
this.mModelMatrixRef = gl.getUniformLocation(this.mCompiledShader, "uModelXformMatrix");
this.mCameraMatrixRef = gl.getUniformLocation(this.mCompiledShader, "uCameraXformMatrix");
```

3.Измените функцию **activate()**, чтобы получить матрицу преобразования камеры и передать ее шейдеру:

```
activate(pixelColor,trsMatrix,cameraMatrix) {
  let gl = core.get();
  gl.useProgram(this.mCompiledShader);
  // bind vertex buffer
  gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer.get());
  gl.vertexAttribPointer(this.mVertexPositionRef,
  3, // each element is a 3-float (x,y.z)
  gl.FLOAT, // data type is FLOAT
  false, // if the content is normalized vectors
  0, // number of bytes to skip in between elements
  0); // offsets to the first element
  gl.enableVertexAttribArray(this.mVertexPositionRef);

  // load uniforms
  gl.uniform4fv(this.mPixelColorRef, pixelColor);
  gl.uniformMatrix4fv(this.mModelMatrixRef, false, trsMatrix);
  gl.uniformMatrix4fv(this.mCameraMatrixRef, false, cameraMatrix);
```

Как вы видели ранее, функция **gl.uniformMatrix4fv()** копирует содержимое **cameraMatrix** оператору **uCameraXformMatrix**.

Измените Renderable для поддержки преобразования камеры

Напомним, что шейдеры активируются в функции **draw()** класса **Renderable**; как таковой, **Renderable** также должен быть изменен для получения и передачи **cameraMatrix** для активации шейдера:

```
draw(cameraMatrix) {
  let gl = glSys.get();
  this.mShader.activate(this.mColor,this.mXform.getTRSMatrix(),cameraMatrix);
  gl.drawArrays(gl.TRIANGLE_STRIP, 0, 4);
}
```

Теперь можно настроить WC для рисования и определить подобласть на холсте для рисования.

Спроектируйте сцену

Как показано на **рис. 3-13**, для целей тестирования будет определено мировое пространство (WC) с центром в точке (20, 60) и размером 20×10. В центре WC будут нарисованы два повернутых квадрата, синий квадрат 5x5 и красный квадрат 2×2. Чтобы проверить границы координат, в каждом из углов WC будет нарисован квадрат размером 1×1 с определенным цветом.

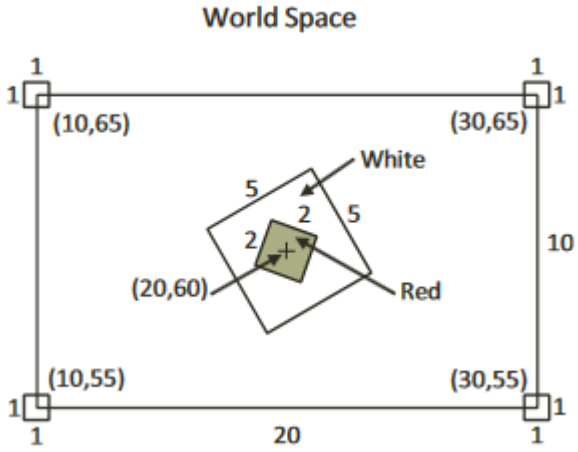

Рисунок 3-13. Проектирование WC для поддержки рисования

Как показано на рис. 3-14, WC будет отображен в окне просмотра с нижним левым углом, расположенным в (20, 40), и размером 600×300 пикселей. Важно отметить, что для того, чтобы квадраты отображались пропорционально, соотношение сторон ширины к высоте WC должен совпадать с WC видового экрана. В этом случае WC имеет соотношение сторон 20:10, и это соотношение 2:1 соответствует соотношению 600:300 в окне просмотра.

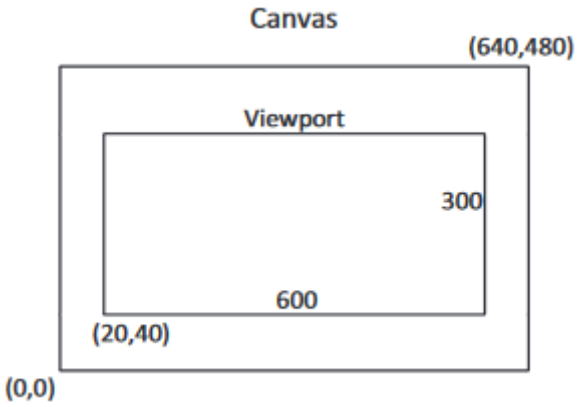

Рисунок 3-14. Вывод WC на видовой экран

Обратите внимание, что детали WC, расположенные по центру в точках (20, 60) и имеющие размер 20x10, и окно просмотра в левом нижнем углу в точках (20, 40) и размер 600x300, выбраны довольно случайным образом. Это просто разумные значения, которые могут продемонстрировать правильность реализации.

Внедрить дизайн

Класс **MyGame** будет изменен для реализации дизайна:

1. Редактировать **my_game.js** . В конструкторе выполните шаг **A** для инициализации игрового движка и шаг **B** для создания шести визуализируемых объектов (два для рисования в центре и по четыре в каждом углу экрана) с соответствующими цветами.

```
constructor(htmlCanvasID) {
    // Step A: Initialize the webGL Context
    engine.init(htmlCanvasID);
    // Step B: Create the Renderable objects:
    this.mBlueSq = new engine.Renderable();
    this.mBlueSq.setColor([0.25, 0.25, 0.95, 1]);
    this.mRedSq = new engine.Renderable();
    this.mRedSq.setColor([1, 0.25, 0.25, 1]);
    this.mTLSq = new engine.Renderable();
    this.mTLSq.setColor([0.9, 0.1, 0.1, 1]);
    this.mTRSq = new engine.Renderable();
    this.mTRSq.setColor([0.1, 0.9, 0.1, 1]);
    this.mBRSq = new engine.Renderable();
    this.mBRSq.setColor([0.1, 0.1, 0.9, 1]);
    this.mBLSq = new engine.Renderable();
    this.mBLSq.setColor([0.1, 0.1, 0.1, 1]);
}
```

2.Шаги C и D очистите весь холст, настройте видовой экран и измените его цвет на другой. Этот код находится в нутри конструктора(**constructor**) **MyGame**:

```
// Step C: Clear the entire canvas first
engine.clearCanvas([0.9, 0.9, 0.9, 1]);

// get access to the gl connection to the GPU
let gl = glSys.get();
// Step D: Setting up Viewport
// Step D1: Set up the viewport: area on canvas to be drawn
gl.viewport(
20, // x position of bottom-left corner of the area to be drawn
40, // y position of bottom-left corner of the area to be drawn
600, // width of the area to be drawn
300); // height of the area to be drawn
// Step D2: set up the corresponding scissor area to limit clear area
gl.scissor(
20, // x position of bottom-left corner of the area to be drawn
40, // y position of bottom-left corner of the area to be drawn
600, // width of the area to be drawn
300); // height of the area to be drawn
// Step D3: enable scissor area, clear and then disable the scissor area
gl.enable(gl.SCISSOR_TEST);
engine.clearCanvas([0.8, 0.8, 0.8, 1.0]); // clear the scissor area
gl.disable(gl.SCISSOR_TEST);
```

Шаг D1 определяет область просмотра, а шаг D2 определяет соответствующую область срезания. Зона действия отсечения проверяет и ограничивает площадь, подлежащую очистке. Поскольку тестирование, связанное с **gl.scissor()**, требует больших вычислительных затрат, оно отключается сразу после использования.

3.Шаг E определяет WC с преобразованием камеры путемобъединения соответствующих операторов масштабирования и преобразования:

```
// Step E: Set up camera transform matrix
// assume camera position and dimension
let cameraCenter = vec2.fromValues(20, 60);
let wcSize = vec2.fromValues(20, 10);
let cameraMatrix = mat4.create();
// Step E1: after translation, scale to: -1 to 1: a 2x2 square at origin
mat4.scale(cameraMatrix, mat4.create(),
vec3.fromValues(2.0/wcSize[0], 2.0/wcSize[1], 1.0));
// Step E2: first to perform is to translate camera center to origin
mat4.translate(cameraMatrix, cameraMatrix,
vec3.fromValues(-cameraCenter[0], -cameraCenter[1], 0));

window.onload = function() {
  new MyGame('GLCanvas');
```

Шаг E1 определяет оператор масштабирования S(2/W, 2/H) для масштабирования WC WxH до NDC размер 2x2, и шаг E2 определяет оператор перевода, T(-center.x, -center.y), для выравнивания WC с центром NDC. Обратите внимание, что порядок объединения сначала реализует преобразование, за которым следует оператор масштабирования. Это именно преобразование камеры, описанное ранее, которое определяет WC следующим образом.

a. Center: (20,60)
b. Top-left corner: (10, 65)
c. Top-right corner: (30, 65)
d. Bottom-right corner: (30, 55)
e. Bottom-left corner: (10, 55)

Напомним, что порядок умножения важен и что порядок операций масштабирования и преобразования нельзя менять местами.

4. Установите слегка повернутый синий квадрат размером 5x5 в центре WC и нарисуйте с помощью оператора преобразования камеры **cameraMatrix**:

```
// Step F: Draw the blue square
// Center Blue, slightly rotated square
this.mBlueSq.getXform().setPosition(20, 60);
this.mBlueSq.getXform().setRotationInRad(0.2); // In Radians
this.mBlueSq.getXform().setSize(5, 5);
this.mBlueSq.draw(cameraMatrix);
```

5. Теперь нарисуйте остальные пять квадратов, сначала 2x2 в центре и по одному в углу WC:

```
// Step G: Draw the center and the corner squares
// center red square
this.mRedSq.getXform().setPosition(20, 60);
this.mRedSq.getXform().setSize(2, 2);
this.mRedSq.draw(cameraMatrix);
// top left
this.mTLSq.getXform().setPosition(10, 65);
this.mTLSq.draw(cameraMatrix);
// top right
this.mTRSq.getXform().setPosition(30, 65);
this.mTRSq.draw(cameraMatrix);
// bottom right
this.mBRSq.getXform().setPosition(30, 55);
this.mBRSq.draw(cameraMatrix);
// bottom left
this.mBLSq.getXform().setPosition(10, 55);
this.mBLSq.draw(cameraMatrix);
```

Запустите этот проект и обратите внимание на различные цвета в четырех углах: верхний левый (mTLSq) красным, верхний правый (mTRSq) - зеленым, нижний правый (mBRSq) - синим и нижний левый (mBLSq) темно-серого цвета. Измените расположение угловых квадратов, чтобы убедиться, что центральные положения этих квадратов расположены в пределах WC, и, таким образом, на самом деле видна только одна четверть квадратов. Например, установите для mBlSq значение (12, 57), чтобы увидеть, что темно-серый квадрат на самом деле в четыре раза больше. Это наблюдение подтверждает, что области квадраты за пределами области просмотра/"ножницы" обрезаются WebGL.

Несмотря на отсутствие должной абстракции, теперь можно определить любую удобную систему WC и любые прямоугольные подобласти холста для рисования. С моделированием и Благодаря преобразованиям камеры игровой программист теперь может разработать игровое решение, основанное на семантических потребностях игры, и игнорировать нерелевантный диапазон рисования WebGL NDC. Однако код в классе MyGame сложен и может отвлекать. Как вы уже видели, важным следующим шагом является определение абстракции, чтобы скрыть детали Вычисление матрицы преобразования камеры.

Камера

Преобразование камеры позволяет определить местоположение WC. В физическом мире это аналогично фотосъемке с помощью фотоаппарата. Центр видоискателя вашей камеры - это центр WC, а ширина и высота мира, видимого через видоискатель, - это размеры WC. По этой аналогии процесс фотографирования эквивалентен вычислению рисунка каждого объекта в WC. Наконец, окно просмотра описывает местоположение для отображения вычисленного изображения.

Проект объектов камеры

Этот проект демонстрирует, как абстрагировать преобразование камеры и видовой экран, чтобы скрыть детали вычисления матрицы и конфигурации WebGL. На рис. 3-15 показаны результаты запуска проекта **CameraObjects**; обратите внимание, что результаты этого проекта идентичны результатам предыдущего проекта. Исходный код этого проекта определен в папке **chapter 3/3.5.camera_objects**.

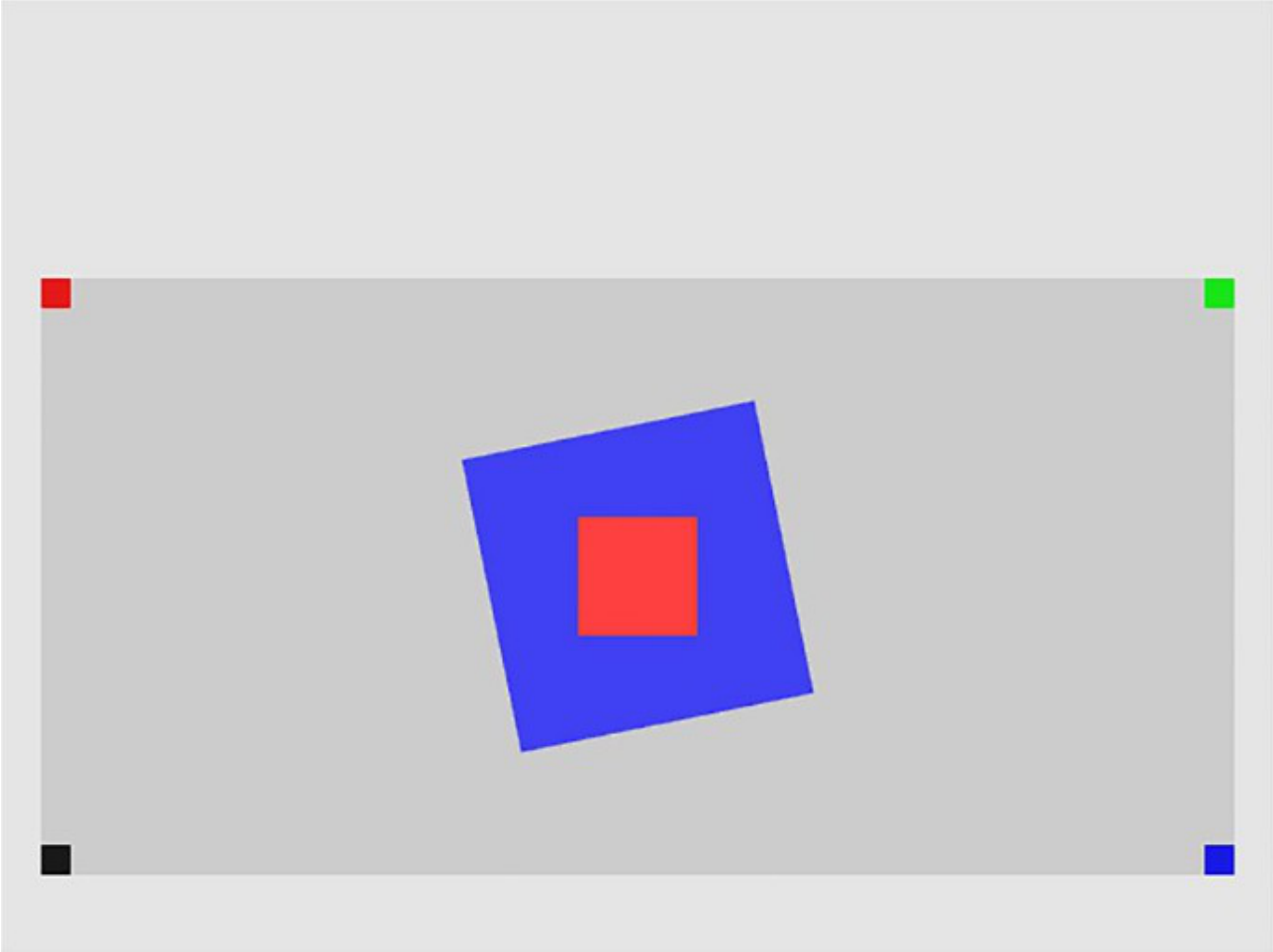

Рисунок 3-15. Запуск проекта "Объекты камеры"

Целями проекта являются следующие:

- Определить класс **Camera** для инкапсуляции определения WC и функциональности видового экрана
- Интегрировать класс **Camera** в игровой движок
- Продемонстрировать, как работать с объектом камеры

Класс Camera

Класс **Camera** должен инкапсулировать функциональность, определенную операторами масштабирования и трансляции в конструкторе **MyGame** из предыдущего примера. Чистый и многоразовый дизайн класса должен быть дополнен соответствующими функциями получения и настройки.

1. Определите класс камеры в игровом движке, создав новый исходный файл в папке **js/engine**, и назовите файл **camera.js** .
2. Добавьте конструктор для камеры

```
class Camera {
  constructor(wcCenter, wcWidth, viewportArray) {
    // WC and viewport position and size
    this.mWCCenter = wcCenter;
    this.mWCWidth = wcWidth;
    this.mViewport = viewportArray; // [x, y, width, height]
    // Camera transform operator
    this.mCameraMatrix = mat4.create();
    // background color
    this.mBGColor = [0.8, 0.8, 0.8, 1]; // RGB and Alpha
    }
... implementation to follow ...
}
```

Камера определяет центр и ширину экрана, область просмотра, оператора преобразования камеры и цвет фона. Примите к сведению следующее:

a. mWCCenter - это **vec2 (vec2 определен в библиотеке glMatrix)**.
Это массив с плавающей точкой, состоящий из двух элементов. Первый элемент, позиция индекса **0**, из **vec2** - это **x**, а второй элемент, позиция индекса **1**, - это позиция **y**.

b. Четыре элемента **viewportArray** - это позиции **x** и **y** в нижнем левом углу, а также ширина и высота видового экрана в таком порядке. Это компактное представление области просмотра сохраняет сводит количество переменных экземпляра к минимуму и помогает сохранить управляемость классом камеры.

c. mWCWidth - это ширина WC. Чтобы гарантировать соответствие соотношения сторон между WC и видовым экраном, высота WC всегда вычисляется исходя из соотношения сторон видового экрана и **mWCWidth**.

d. mBgColor - это массив из четырех значений с плавающей точкой, представляющих красный, зеленый, синий и альфа-компоненты цвета.

3. Вне определения класса камеры определите перечислимые индексы для доступа к **viewportArray**:

```
const eViewport = Object.freeze({
  eOrgX: 0,
  eOrgY: 1,
  eWidth: 2,
  eHeight: 3
});
```

Обратите внимание, что имена перечисленных элементов начинаются со строчной буквы **“e”**, как в **eViewport** и **eOrgX** .

---

4. Определите функцию для вычисления высоты WC на основе соотношения сторон видового экрана:

```
getWCHeight() {
  // viewportH/viewportW
  let ratio = this.mViewport[eViewport.eHeight] / this.mViewport[eViewport.eWidth];
  return this.getWCWidth() * ratio;
  }
```

5. Добавьте **getters** и **setters** для переменных экземпляра

```
setWCCenter(xPos, yPos) {
  this.mWCCenter[0] = xPos;
  this.mWCCenter[1] = yPos;
}
getWCCenter() { return this.mWCCenter; }
setWCWidth(width) { this.mWCWidth = width; }
setViewport(viewportArray) { this.mViewport = viewportArray; }
getViewport() { return this.mViewport; }
setBackgroundColor(newColor) { this.mBGColor = newColor; }
getBackgroundColor() { return this.mBGColor; }
```

6. Создайте функцию для установки видового экрана и вычисления оператора преобразования камеры для этой камеры

```
// Initializes the camera to begin drawing
setViewAndCameraMatrix() {
let gl = glSys.get();
// Step A: Configure the viewport

// Step B: compute the Camera Matrix

}
```

Обратите внимание, что эта функция называется **setViewAndCameraMatrix**(), поскольку она настраивает WebGL для рисования на желаемом видовом экране и настройки оператора преобразования камеры. Ниже объясняются детали шагов **A** и **B**.

7. Код для настройки видового экрана на шаге **A** выглядит следующим образом

```
setViewAndCameraMatrix() {
  let gl = glSys.get();
  // Step A: Configure the viewport
  // Step A1: Set up the viewport: area on canvas to be drawn
  gl.viewport(this.mViewport[0], // x of bottom-left of area to be drawn
  this.mViewport[1], // y of bottom-left of area to be drawn
  this.mViewport[2], // width of the area to be drawn
  this.mViewport[3]); // height of the area to be drawn
  // Step A2: set up the corresponding scissor area to limit the clear area
  gl.scissor(this.mViewport[0], // x of bottom-left of area to be drawn
  this.mViewport[1], // y of bottom-left of area to be drawn
  this.mViewport[2], // width of the area to be drawn
  this.mViewport[3]);// height of the area to be drawn
  // Step A3: set the color to be clear
  gl.clearColor(this.mBGColor[0], this.mBGColor[1],
  this.mBGColor[2], this.mBGColor[3]);
  // set the color to be cleared
  // Step A4: enable scissor area, clear and then disable the scissor area
  gl.enable(gl.SCISSOR_TEST);
  gl.clear(gl.COLOR_BUFFER_BIT);
  gl.disable(gl.SCISSOR_TEST);
  // Step B: compute the Camera Matrix

}
```

Обратите внимание на сходство этих шагов с кодом настройки видового экрана в **MyGame** из предыдущего примера. Единственное отличие заключается в правильных ссылках на переменные экземпляра с помощью **this**.

8. Код для настройки оператора преобразования камеры на шаге **B** следующий

```
// Step B: compute the Camera Matrix
let center = this.getWCCenter();
// Step B1: after translation, scale to -1 to 1: 2x2 square at origin
glMatrix.mat4.scale(this.mCameraMatrix, mat4.create(),
glMatrix.vec3.fromValues(2.0 / this.getWCWidth(),
2.0 / this.getWCHeight(), 1.0));
// Step B2: first translate camera center to the origin
glMatrix.mat4.translate(this.mCameraMatrix, this.mCameraMatrix,
glMatrix.vec3.fromValues(-center[0], -center[1], 0));
```

Еще раз, этот код похож на конструктор **MyGame** из предыдущего примера.

9. Определите функцию для доступа к вычисленной матрице камеры:

```
getCameraMatrix() { return this.mCameraMatrix; }
```

10. Наконец, не забудьте экспортировать недавно определенный класс **Camera**

```
export default Camera.
```

Измените **Renderable** для поддержки класса **Camera**

Функция **draw()** класса **Renderable** должна быть изменена для получения вновь определенной камеры, чтобы получить доступ к вычисленной матрице камеры:

```
draw(camera) {
  let gl = glSys.get();
  this.mShader.activate(this.mColor,this.mXform.getTRSMatrix(),camera.getCameraMatrix());
  gl.drawArrays(gl.TRIANGLE_STRIP, 0, 4);
}
```

Измените файл доступа к движку для экспорта камеры

Важно поддерживать файл доступа к движку, **index.js** обновлен таким образом, чтобы разработчик игры мог получить доступ к недавно определенному классу **Camera**.:

1. Редактировать **index.js** ; импорт из вновь определенного **camera.js** файл:

```
// general utilities
import Camera from "./camera.js";
import Transform from "./transform.js";
import Renderable from "./renderable.js";
```

2.Экспорт **Camera** для доступа клиента:

```
export default {
  // Util classes
  Camera,Transform,Renderable,
  // functions
  init, clearCanvas
}
```

Протестируйте камеру

Правильно определив класс **Camera**, протестируйте его с **my_game.js** это просто:

1. Редактировать **my_game.js** ; после инициализации игрового движка на шаге **A**, создайте экземпляр объекта **Camera** с настройками, которые определяют **WC** и видовой экран из предыдущего проекта на шаге **B**:

```
// Step A1: Setup the camera
this.mCamera = new engine.Camera(
  vec2.fromValues(20, 60), // center of the WC
  20, // width of WC
  [20, 40, 600, 300] // viewport:orgX, orgY, W, H
);
```

2. Продолжите создание шести объектов, доступных для визуализации, и очистку холста на этапах **C** и **D**:

3. Теперь вызовите функцию **setViewCameraMatrix()** объекта **Camera**, чтобы настроить видовой экран **WebGL** и вычислить матрицу камеры на шаге **E**, и нарисуйте все объекты **Renderable** с помощью объекта **Camera** на шагах **F** и **G**.

```
// Step E: Set up camera transform matrix
// assume camera position and dimension
let cameraCenter = glMatrix.vec2.fromValues(20, 60);
let wcSize = glMatrix.vec2.fromValues(20, 10);
let cameraMatrix = glMatrix.mat4.create();
// Step E1: after translation, scale to: -1 to 1: a 2x2 square at origin
glMatrix.mat4.scale(cameraMatrix, glMatrix.mat4.create(),
glMatrix.vec3.fromValues(2.0/wcSize[0], 2.0/wcSize[1], 1.0));
// Step E2: first to perform is to translate camera center to origin
glMatrix.mat4.translate(cameraMatrix, cameraMatrix,
glMatrix.vec3.fromValues(-cameraCenter[0], -cameraCenter[1], 0));
 // Step E: Starts the drawing by activating the camera
this.mCamera.setViewAndCameraMatrix()
// Step F: Draw the blue square
// Center Blue, slightly rotated square
this.mBlueSq.getXform().setPosition(20, 60);
this.mBlueSq.getXform().setRotationInRad(0.2); // In Radians
this.mBlueSq.getXform().setSize(5, 5);
this.mBlueSq.draw(this.mCamera);

// Step G: Draw the center and the corner squares
// center red square
this.mRedSq.getXform().setPosition(20, 60);
this.mRedSq.getXform().setSize(2, 2);
this.mRedSq.draw(this.mCamera);
// top left
this.mTLSq.getXform().setPosition(10, 65);
this.mTLSq.draw(this.mCamera);
// top right
this.mTRSq.getXform().setPosition(30, 65);
this.mTRSq.draw(this.mCamera);
// bottom right
this.mBRSq.getXform().setPosition(30, 55);
this.mBRSq.draw(this.mCamera);
// bottom left
this.mBLSq.getXform().setPosition(10, 55);
this.mBLSq.draw(this.mCamera);
```

Объект **mCamera** передается функции **draw()** объектов **Renderable** таким образом, что оператор матрицы преобразования камеры может быть извлечен и использован для активации шейдера

Резюме

В этой главе вы узнали, как создать систему, которая может поддерживать рисование многих объектов. Система состоит из трех частей: объекты, сведения о каждом объекте и отображение объектов на холсте браузера. Объекты инкапсулируются средством **Renderable**, которое использует **Transform** для захвата их деталей —

положения, размера и поворота. Особенности отображения объектов определяются **Camera**, где объекты в определенных местах могут отображаться в желаемых подобластях на холсте.

Вы также узнали, что все объекты рисуются относительно мирового пространства или WC, удобной системы координат. WC определяется для композиций сцен, основанных на преобразованиях координат. Наконец, преобразование камеры используется для выбора того, какая часть WC на самом деле должна отображаться на холсте в браузере. Этого можно достичь, определив область, доступную для просмотра камерой, и используя функциональность видового экрана, предоставляемую WebGL.

При создании системы рисования структура исходного кода игрового движка была изменена последовательно преобразован в абстрагированные и инкапсулированные компоненты. Таким образом, структура исходного кода продолжает поддерживать дальнейшее расширение, включая дополнительную функциональность, которая будет обсуждаться в следующей главе.

Глава 4

Реализация общих компонентов видеоигр

После завершения этой главы вы сможете

- Управляйте положением, размером и вращением **Renderable** объектов для создания сложных движений и анимаций
- Получать ввод с клавиатуры от проигрывателя для управления и анимации **Renderable** объектов
- Работа с асинхронной загрузкой и выгрузкой внешних ресурсов
- Определите, загрузите и выполните простой игровой уровень из файла сцены
- Меняйте уровни игры, загружая новую сцену
- Работа со звуковыми клипами для фоновой музыки и звуковых сигналов

Вступление

В предыдущих главах была создана структура игрового движка для поддержки базовых операций рисования. Рисование – это первый шаг к созданию вашего

игрового движка, поскольку оно позволяет вам наблюдать за результатом, продолжая расширять функциональность игрового движка. В этой главе будут рассмотрены и добавлены в игровой движок два важных механизма – интерактивность и ресурсная поддержка. Интерактивность позволяет движку получать и интерпретировать вводимые игроком данные, в то время как ресурсная поддержка относится к функциональности работа с внешними файлами, такими как файлы исходного кода GLSL-шейдера, аудиоклипы и изображения.

Эта глава начинается с ознакомления игрового цикла, важнейшим компонентом , который создает ощущение взаимодействия в реальном времени и непосредственности почти во всех видеоиграх. Основанный на **game loop foundation**, ввод с клавиатуры игроком будет поддерживаться за счет интеграции соответствующей функциональности HTML5. Инфраструктура управления ресурсами будет создана с нуля для поддержки эффективной загрузки, хранения, извлечения и использования внешних файлов. Функциональность для работы с внешними текстовыми файлами (например, файлы исходного кода GLSL-шейдера) и аудиоклипы будут интегрированы с соответствующими примерами проектов. Кроме того, будет разработана архитектура игровых сцен для поддержки возможности работы с несколькими сценами и переходами между сценами, включая сцены, определенные во внешних файлах сцен. К концу этой главы ваш игровой движок будет поддерживать взаимодействие с игроком с помощью клавиатуры, будет иметь возможность обеспечивать звуковую обратную связь и сможет переходить между различными уровнями игры, включая загрузку уровня из внешнего файла.

Игровой цикл

Одной из самых основных операций любой видеоигры является поддержка, казалось бы, мгновенного взаимодействия между вводимыми игроками данными и графическими игровыми элементами. В действительности эти взаимодействия реализованы в виде непрерывного цикла, который принимает и обрабатывает входные данные игрока, обновляет состояние игры и рендерит игру.
Этот постоянно выполняющийся цикл называется **game loop**.

Чтобы передать должное ощущение мгновенности, каждый цикл игрового цикла должен быть завершен за время реакции обычного человека. Это часто называют реальным временем, это промежуток времени, который слишком короток для того, чтобы люди могли обнаружить его визуально. Как правило, реального времени можно достичь, когда игровой цикл выполняется со скоростью более 40-60 циклов в секунду. Поскольку в каждом цикле игрового цикла обычно выполняется одна операция рисования, скорость этого цикла также называется частотой кадров в секунду (FPS) или **frame rate**. 60 кадров в секунду – хороший показатель производительности. Это означает, что ваш игровой движок должен получать данные от игрока, обновлять

игровой мир, а затем отрисовывать игровой мир – и все это в течение $1/60$ доли секунды!

Сам игровой цикл, включая детали реализации, является наиболее фундаментальной структурой управления для игры. Поскольку основной целью является поддержание производительности в режиме реального времени, детали работы игрового цикла не имеют отношения к остальной части игрового движка. По этой причине реализация игрового цикла должна быть плотно инкапсулирована в ядро игрового движка с его подробными операциями, скрытыми от других игровых элементов.

Типичные реализации игрового цикла

Игровой цикл(**game loop**) – это механизм, с помощью которого непрерывно выполняются логика и рисование. Простой игровой цикл состоит из рисования всех объектов, обработки входных данных игрока и обновления состояния этих объектов, как показано в следующем псевдокоде:

```
initialize();
while(game running) {
  draw();
  input();
  update();
}
```

Как уже говорилось, для поддержания ощущения интерактивности в реальном времени требуется 60 кадров в секунду. Когда сложность игры возрастает, одна из проблем, которая может возникнуть, заключается в том, что иногда выполнение одного цикла может занимать больше $1/60$ секунды, в результате чего игра запускается с пониженной частотой кадров. Когда это произойдет, вся игра, по-видимому, замедлится. Распространенным решением является установление приоритета одних операций над другими. То есть движок может быть сконструирован таким образом, чтобы зациклить игровой цикл на выполнении операций, которые движок считает более важным пропускать другие. Поскольку для того, чтобы игра функционировала так, как задумано, требуются правильные входные данные и обновления , при необходимости часто пропускается операция розыгрыша. Это называется пропуском кадра, и следующий псевдокод иллюстрирует одну из таких реализаций

```
elapsedTime = now;
previousLoop = now;
while(game running) {
  elapsedTime += now - previousLoop;
  previousLoop = now;
  draw();
  input();
  while( elapsedTime >= UPDATE_TIME_RATE ) {
    update();
    elapsedTime -= UPDATE_TIME_RATE;
  }
}
```

В предыдущем списке псевдокодов **UPDATE_TIME_RATE** – это требуемая частота обновления в реальном времени. Когда время, прошедшее между циклами игрового цикла, превышает значение **UPDATE_TIME_RATE**, функция **update()** будет вызываться до тех пор, пока не догонит время. Это означает, что операция **draw()** по существу пропускается, когда игровой цикл выполняется слишком медленно. Когда это произойдет, будет казаться, что вся игра запускается медленно, с запаздывающими ответами на ввод в игровой процесс и пропущенными кадрами. Однако игровая логика будет продолжать функционировать корректно.

Обратите внимание, что цикл **while**, который включает в себя вызов функции **update()**, имитирует фиксированный временной шаг обновления **UPDATE_TIME_RATE**. Это обновление с фиксированным временным шагом обеспечивает простую реализацию для поддержания детерминированного состояния игры. Это важный компонент, позволяющий убедиться, что ваш игровой движок функционирует должным образом, независимо от того, работает он оптимально или медленно.

Чтобы гарантировать, что внимание сосредоточено исключительно на понимании операций рисования и обновления основного игрового цикла, вводимые данные будут игнорироваться до следующего проекта.

Проект Game Loop

Этот проект демонстрирует, как включить игровой цикл в ваш игровой движок и поддерживать анимацию в реальном времени путем рисования и обновления **Renderable** объектов. Вы можете увидеть пример выполнения этого проекта на рисунке 4-1. Исходный код этого проекта определен в папке **chapter 4/4.1.game_loop**

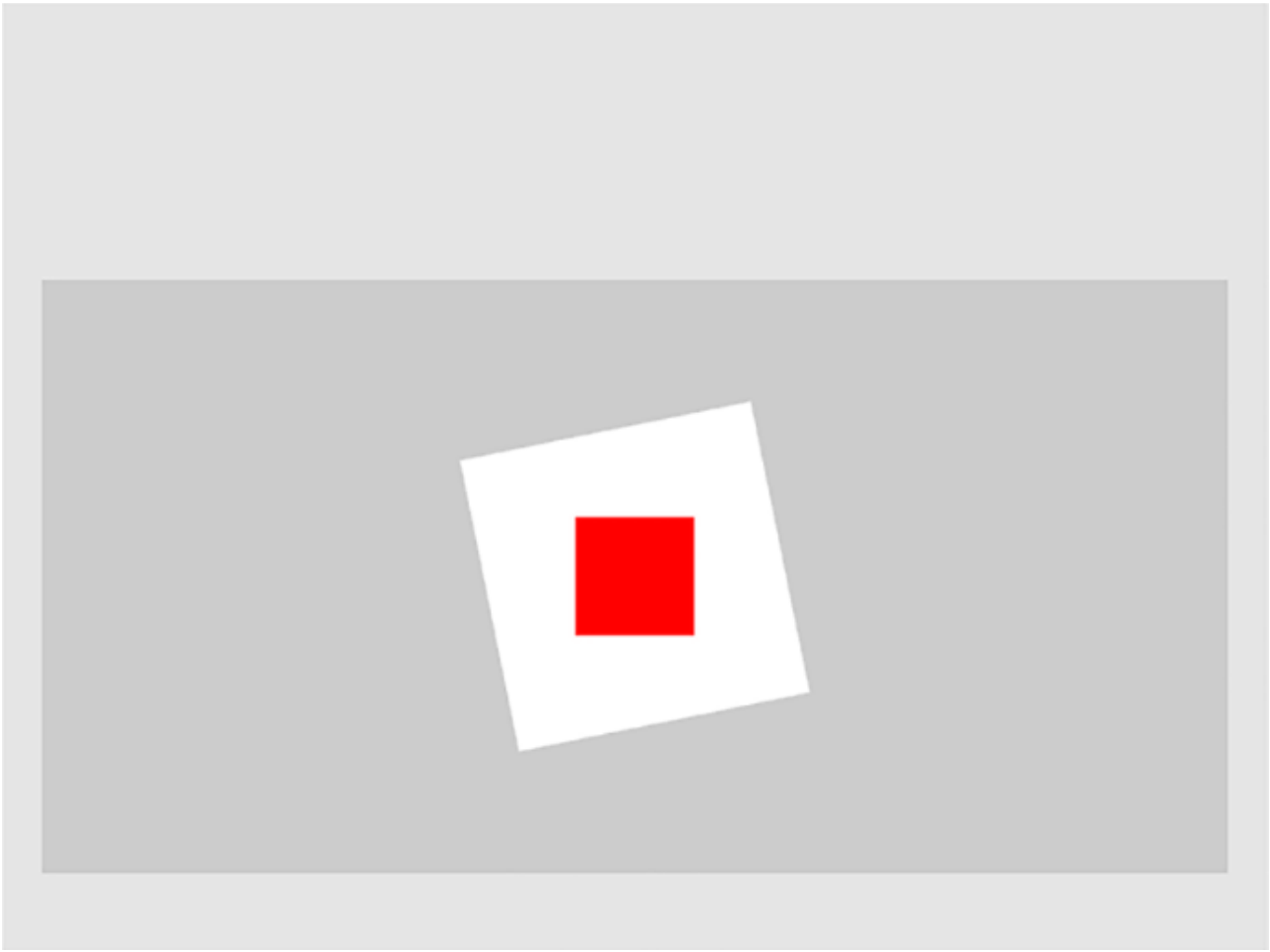

Рисунок 4–1. Запуск проекта Game Loop

Целями проекта являются следующие:

- Понимать внутренние операции игрового цикла
- Для реализации и инкапсуляции операций игрового цикла
- Получить опыт работы с непрерывным рисованием и обновлением для создания анимации

Реализовать компонент Game Loop

Компонент **game loop** является основой функциональности игрового движка и следовательно должен располагаться аналогично компоненту **vertex_buffer** в виде файла, определенного в папке **js**/**engine**/**core**:

1. Создайте новый файл для модуля **loop** в папке **js**/**engine**/**core** и назовите файл **loop.js** .

2. Определите следующие переменные экземпляра для отслеживания частоты кадров, времени обработки в миллисекундах на кадр, текущего состояния выполнения игрового цикла и ссылки на текущую сцену следующим образом:

```
"use strict"
const kUPS = 60; // Updates per second
const kMPF = 1000 / kUPS; // Milliseconds per update.
// Variables for timing gameloop.
let mPrevTime;
let mLagTime;
// The current loop state (running or should stop)
let mLoopRunning = false;
let mCurrentScene = null;
let mFrameID = -1;
```

Обратите внимание, что **kUPS** – это количество обновлений в секунду, аналогичное обсуждаемому **FPS**, и оно установлено на 60 или 60 обновлений в секунду. Время, доступное для каждого обновления, составляет всего 1/60 секунды. Поскольку в секунде содержится 1000 миллисекунд, доступное время для каждого обновления в миллисекундах равно 1000 * (1/60), или **kMPF**.

Примечание: Когда игра работает оптимально, отрисовка кадров и обновления поддерживаются с одинаковой скоростью; **FPS** и **kUPS** можно рассматривать как взаимозаменяемые понятия. однако, когда возникает задержка, цикл пропускает рисование фрейма и устанавливает приоритеты обновлений. В этом случае **FPS** уменьшится, в то время как **kUPS** будет сохранен.

3.Добавьте функцию для запуска основного цикла следующим образом:

```
function loopOnce() {
  if (mLoopRunning) {
    // Step A: set up for next call to LoopOnce
    mFrameID = requestAnimationFrame(loopOnce);
    // Step B: now let's draw
    // draw() MUST be called before update()
    // as update() may stop the loop!
    mCurrentScene.draw();

    // Step C: compute time elapsed since last loopOnce was executed
    let currentTime = performance.now();
    let elapsedTime = currentTime - mPrevTime;
    mPrevTime = currentTime;
    mLagTime += elapsedTime;
    // Step D: update the game the appropriate number of times.
    // Update only every kMPF (1/60 of a second)
    // If lag larger then update frames, update until caught up.
    while ((mLagTime >= kMPF) && mLoopRunning) {
      mCurrentScene.update();
      mLagTime -= kMPF;
    }
  }
}
```

Примечание: **performance.now()** – это функция **Javascript**, которая возвращает временную метку в миллисекундах.

Обратите внимание на сходство между рассмотренным ранее псевдокодом и шагами **B**, **C** и **D** функции **loopOnce()**, то есть рисованием сцены или игры на шаге **B**, вычислением времени, прошедшего с момента последнего обновления на шаге **C**, и установлением приоритета обновления, если движок отстает.

Основное отличие заключается в том, что сам внешний цикл **while** реализован на основе вызова функции **HTML5 requestAnimationFrame()** на шаге **A**. Функция **requestAnimationFrame()** будет выполнять с приблизительной частотой 60 раз в секунду. во-вторых, вызовите указатель на функцию, который передается в качестве ее параметра. В этом случае функция **loopOnce()** будет вызываться непрерывно примерно 60 раз в секунду. Обратите внимание, что каждый вызов функции **requestAnimationFrame()** приведет ровно к одному выполнению соответствующей функции **loopOnce()** и, таким образом, отрисуется только один раз. Однако, если система работает с задержкой, в течение этого одного кадра может произойти несколько обновлений.

Примечание Функция **requestAnimationFrame()** – это утилита **html5**, предоставляемая браузером, в котором размещена ваша игра. точное поведение этой функции зависит от реализации браузера.

mLoopRunning в цикле **while** на шаге **D** на данный момент является избыточной проверкой. Это условие станет важным в последующих разделах при **update()** можно вызвать **stop()**, чтобы остановить цикл (например, для перехода на уровень или окончания игры).

---

4. Объявите функцию, чтобы запустить игровой цикл. Эта функция инициализирует игру или сцену, переменные времени кадра и флаг выполнения цикла перед вызовом первого **requestAnimationFrame()** с функцией **loopOnce** в качестве параметра для начала игрового цикла.

```
function start(scene) {
  if (mLoopRunning) {
    throw new Error("loop already running")
  }
  mCurrentScene = scene;
  mCurrentScene.init();
  mPrevTime = performance.now();
  mLagTime = 0.0;
  mLoopRunning = true;
  mFrameID = requestAnimationFrame(loopOnce);
}
```

5. Объявите функцию, чтобы остановить игровой цикл. Эта функция просто останавливает цикл, устанавливая **mLoopRunning** в значение **false**, и отменяет последний запрошенный кадр анимации.

```
function stop() {
  mLoopRunning = false;
  // make sure no more animation frames
  cancelAnimationFrame(mFrameID);
}
```

6. Наконец, не забудьте экспортировать желаемую функциональность в остальную часть игрового движка, в данном случае только функции запуска и остановки:

```
export {start, stop}
```

Работа с игровым циклом

Чтобы протестировать реализацию игрового цикла, ваш игровой класс теперь должен реализовать функции **draw**(), **update**() и **init**(). Это связано с тем, что для координации начала и непрерывной работы вашей игры эти функции вызываются из ядра игрового цикла — функция **init**() вызывается из **loop.start**(), в то время как функции **draw**() и **update**() вызываются из **loop.loopOnce**().

1. Отредактируйте ваш **my_game.js** файл для предоставления доступа к циклу путем импорта из модуля. Предоставление разработчику игры доступа к модулю **game loop** является временной мерой и будет исправлено в последующих разделах.

```
// Accessing engine internal is not ideal,
// this must be resolved! (later)
import * as loop from "../engine/core/loop.js";
```

2.Замените конструктор **MyGame** следующим:

```
constructor() {
    // variables for the squares
    this.mWhiteSq = null; // these are the Renderable objects
    this.mRedSq = null;
    // The camera to view the scene
    this.mCamera = null;
}
```

3.Добавьте функцию инициализации для настройки камеры и двух **Renderable** объектов:

```
init() {
    // Step A: set up the cameras
    this.mCamera = new engine.Camera(
    vec2.fromValues(20, 60), // position of the camera
    20, // width of camera
    [20, 40, 600, 300] // viewport (orgX, orgY, width, height)
    );

    this.mCamera.setBackgroundColor([0.8, 0.8, 0.8, 1]);
    // sets the background to gray
    // Step B: Create the Renderable objects:
    this.mWhiteSq = new engine.Renderable();
    this.mWhiteSq.setColor([1, 1, 1, 1]);
    this.mRedSq = new engine.Renderable();
    this.mRedSq.setColor([1, 0, 0, 1]);
    // Step C: Init the white Renderable: centered, 5x5, rotated
    this.mWhiteSq.getXform().setPosition(20, 60);
    this.mWhiteSq.getXform().setRotationInRad(0.2); // In Radians
    this.mWhiteSq.getXform().setSize(5, 5);
    // Step D: Initialize the red Renderable object: centered 2x2
    this.mRedSq.getXform().setPosition(20, 60);
    this.mRedSq.getXform().setSize(2, 2);
}
```

4.Нарисуйте сцену, как и раньше, очистив холст, настроив камеру и нарисовав каждый квадрат

```
draw() {
    // Step A: clear the canvas
    engine.clearCanvas([0.9, 0.9, 0.9, 1.0]); // clear to light gray
    // Step B: Activate the drawing Camera
    this.mCamera.setViewAndCameraMatrix();
    // Step C: Activate the white shader to draw
    this.mWhiteSq.draw(this.mCamera);
    // Step D: Activate the red shader to draw
    this.mRedSq.draw(this.mCamera);
}
```

5. Добавьте функцию **update()** для анимации движущегося белого квадрата и пульсирующего красного квадрата:

```
update() {
  // Simple game: move the white square and pulse the red
  let whiteXform = this.mWhiteSq.getXform();
  let deltaX = 0.05;
  // Step A: Rotate the white square
  if (whiteXform.getXPos() > 30) // the right-bound of the window
  whiteXform.setPosition(10, 60);
  whiteXform.incXPosBy(deltaX);
  whiteXform.incRotationByDegree(1);
  // Step B: pulse the red square
  let redXform = this.mRedSq.getXform();
  if (redXform.getWidth() > 5)
  redXform.setSize(2, 2);
  redXform.incSizeBy(0.05);
}
```

Напомним, что функция **update()** вызывается примерно 60 раз в секунду, и каждый раз происходит следующее:

- Шаг A для белого квадрата: Увеличьте поворот на **1** градус, увеличьте положение **x** на **0,05** и сбросьте значение до **10**, если результирующее положение **x** больше **30**.
- Шаг **B** для красного квадрата: Увеличьте размер на **0,05** и сбросьте его на **2**, если результирующий размер больше **5**.
- Поскольку предыдущие операции выполняются непрерывно примерно 60 раз в секунду вы можете ожидать увидеть следующее:
  - **a**. Белый квадрат, вращающийся при движении вправо и при достижении правой границы оборачивающийся вокруг левой границы
  - **b**. Красный квадрат, увеличивающийся в размерах и уменьшающийся до размера 2, когда размер достигает 5, таким образом, кажется пульсирующим

6. Запустите игровой цикл из функции **window.onload**. Обратите внимание, что ссылка на экземпляр **MyGame** передается в цикл.

```
window.onload = function () {
  engine.init("GLCanvas");
  let myGame = new MyGame();
  // new begins the game
  loop.start(myGame);
}
```

Теперь вы можете запустить проект, чтобы наблюдать движущийся вправо, вращающийся белый квадрат и пульсирующий красный квадрат. Вы можете управлять скоростью перемещения, вращения и пульсации, изменяя соответствующие значения функций **incXPosBy**(), **incRotationByDegree**() и **incSizeBy**(). В этих случаях значения положения, поворота и размера изменяются на постоянную величину за фиксированный интервал времени. По сути, параметрами этих функций являются скорость изменения , **incXPosBy**(**0,05**) – это скорость перемещения вправо 0,05 единиц за 1/60 секунды или 3 единицы в секунду. В этом проекте ширина мира составляет 20 единиц, а поскольку белый квадрат перемещается со скоростью 3 единицы в секунду, вы можете убедиться, что белому квадрату требуется чуть больше 6 секунд , чтобы переместиться от левой границы к правой.

Обратите внимание, что в ядре модуля **loop** функция **requestAnimationFrame**() вполне может вызывать функцию **loopOnce**() несколько раз в течение одного интервала **kMPF**. Когда это произойдет, функция **draw**() будет вызываться многократно без каких-либо вызовов функции **update**(). Таким образом, игровой цикл может завершиться до отрисовки одного и того же игрового состояния несколько раз. Пожалуйста, обратитесь к следующим ссылкам для обсуждения вспомогательных экстраполяций в функции **draw**(), чтобы воспользоваться преимуществами эффективных игровых циклов:

- **http://gameprogrammingpatterns.com/game-loop.html#play-catch-up**

- **http://gafferongames.com/game-physics/fix-your-timestep/**

Чтобы четко описать каждый компонент игрового движка и проиллюстрировать, как эти компоненты взаимодействуют, эта книга не поддерживает экстраполяцию функции **draw()**

Ввод с клавиатуры

Очевидно, что надлежащая поддержка для получения информации от игрока важна для интерактивных видеоигр. Для типичного персонального вычислительного устройства, такого как ПК или Mac, двумя распространенными устройствами ввода являются клавиатура и мышь. В то время как ввод с клавиатуры принимается в виде потока символов, ввод с помощью мыши упакован с информацией о местоположении и связан с видами с камеры. По этой причине ввод с клавиатуры более прост в поддержке на данном этапе разработки движка. В этом разделе будут представлены и интегрируйте поддержку клавиатуры в свой игровой движок. Ввод с помощью мыши будет рассмотрен в проекте ввода с помощью мыши в главе 7, после того как будет рассмотрена поддержка нескольких камер в одной игре.

Проект поддержки клавиатуры

В этом проекте рассматривается поддержка ввода с клавиатуры и интегрируется функциональность в игровой движок. Положение, поворот и размер игровых объектов в этом проекте находятся под вашим контролем. Вы можете увидеть пример выполнения этого проекта на рисунке 4-2. Исходный код этого проекта определен в папке **chapter 4/4.2.keyboard_support**.

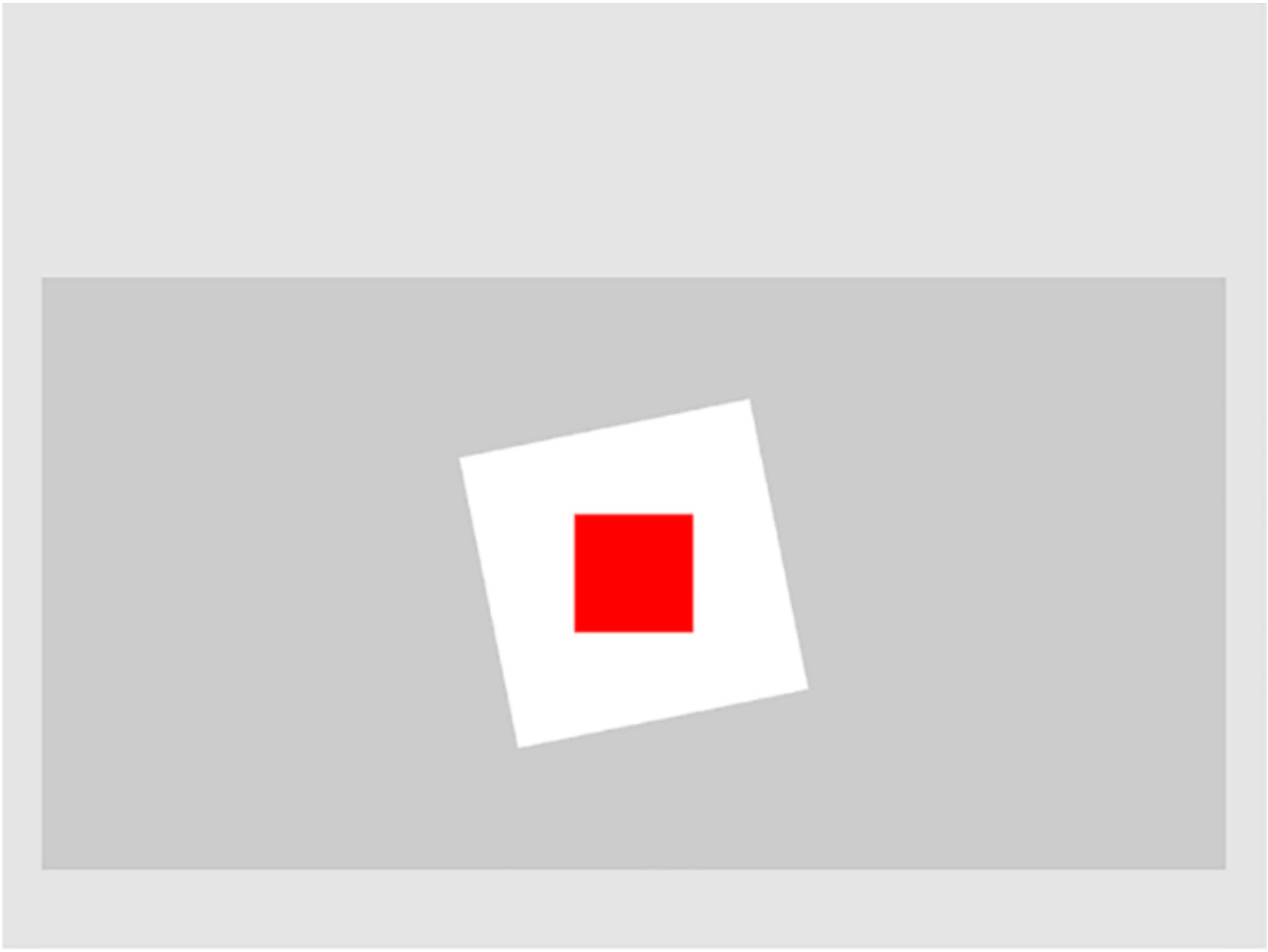

Рисунок 4-2. Запуск проекта поддержки клавиатуры

Элементы управления проектом следующие:

- **Клавиша со стрелкой вправо**: перемещает белый квадрат вправо и переносит его в левую часть игрового окна.
- **Клавиша со стрелкой вверх**: поворачивает белый квадрат
- **Клавиша со стрелкой вниз**: увеличивает размер красного квадрата, а затем сбрасывает его на пороговое значение.

Целями проекта являются следующие:

- Реализовать компонент движка для приема ввода с клавиатуры
- Чтобы понять разницу между состоянием клавиши (если клавиша отпущена или нажата) и событием клавиши (когда состояние клавиши меняется)
- Чтобы понять, как интегрировать входной компонент в игровой цикл

Добавьте компонент Input в движок

Напомним, что компонент **loop** является частью ядра игрового движка и не должен быть доступен разработчику клиентской игры. Напротив, четко определенный модуль ввода должен помогать разработчику клиентской игры запрашивать состояния клавиатуры, не отвлекаясь на какие-либо детали. По этой причине модуль ввода будет определен в папке **js/engine**.

1. Создайте новый файл в папке **js/engine** и назовите его **input.js** .
2. Определите объект **keys** для контроля управления игры:

```
"use strict"
  // Key code constants
  const keys = {
  // arrows
  Left: 37,
  Up: 38,
  Right: 39,
  Down: 40,
  // space bar
  Space: 32,
  // numbers
  Zero: 48,
  One: 49,
  Two: 50,
  Three: 51,
  Four: 52,
  Five : 53,
  Six : 54,
  Seven : 55,
  Eight : 56,
  Nine : 57,
  // Alphabets
  A : 65,
  D : 68,
  E : 69,
  F : 70,
  G : 71,
  I : 73,
  J : 74,
  K : 75,
  L : 76,
  Q : 81,
  R : 82,
  S : 83,
  W : 87,
  LastKeyCode: 222
}
```

Коды клавиш - это уникальные цифры, представляющие каждый символ клавиатуры. Обратите внимание, что существует до 222 уникальных ключей. В списке в словаре определено только небольшое подмножество ключей, тех, которые имеют отношение к данному проекту.

Примечание Ключевые коды для алфавитов являются непрерывными, начиная с 65 для **a** и заканчивая **90** для **Z**. Вы можете смело добавлять любые символы для своего собственного игрового движка. полный список кодов ключей приведен в разделе:
`www.cambiaresearch.com/articles/15/javascript-char-codes-key-codes`

3.Создайте переменные экземпляра массива для отслеживания состояний каждого ключа:

Все три массива определяют состояние каждого ключа как логическое значение. **mKeyPreviousState** записывает состояния ключей из предыдущего цикла обновления, а **mIsKeyPressed** записывает текущее состояние ключей. Записи кода клавиш в этих двух массивах имеют значение **true** при нажатии соответствующих клавиш клавиатуры и **false** в противном случае. Массив **mIsKeyClicked** фиксирует события нажатия клавиш. Записи кода клавиши в этом массиве являются истинными только тогда, когда соответствующая клавиша клавиатуры переходит от отпускания к нажатию в течение двух последовательных циклов обновления.

Важно отметить, что **KeyPress** - это состояние клавиши, в то время как **KeyClicked** — это событие. Например, если игрок нажимает клавишу **A** в течение одной секунды, прежде чем отпустить ее, то длительность всего этого второго **KeyPress** для **A** равна **true**, в то время как **KeyClicked** для **A** верно только один раз — цикл обновления сразу после нажатия клавиши.

4.Определите функции для фиксации фактических изменений состояния клавиатуры

```
// Event handler functions
function onKeyDown(event) {
  mIsKeyPressed[event.keyCode] = true;
}
function onKeyUp(event) {
  mIsKeyPressed[event.keyCode] = false;
}
```

При вызове этих функций код клавиши из параметра используется для записи соответствующих изменений состояния клавиатуры. Ожидается, что **вызывающий** эти функции передаст соответствующий код ключа в аргументе

5.Добавьте функцию для инициализации всех ключевых состояний и зарегистрируйте обработчики ключевых событий в браузере. Функция **window.addEventListener()** регистрирует обработчики событий **onKeyUp/Down()** в браузере таким образом, что соответствующие функции будут вызываться, когда игрок нажимает или отпускает клавиши на клавиатуре

```
function init() {
  let i;
  for (i = 0; i < keys.LastKeyCode; i++) {
    mIsKeyPressed[i] = false;
    mKeyPreviousState[i] = false;
    mIsKeyClicked[i] = false;
  }

  // register handlers
  window.addEventListener('keyup', onKeyUp);
  window.addEventListener('keydown', onKeyDown);
}
```

6. Добавьте функцию **update()** для получения событий нажатия клавиши. Функция **update()** использует **mIsKeyPressed** и **mKeyPreviousState**, чтобы определить, произошло ли событие нажатия клавиши

```
function update() {
  let i;
  for (i = 0; i < keys.LastKeyCode; i++) {
    mIsKeyClicked[i] = (!mKeyPreviousState[i]) && mIsKeyPressed[i];
    mKeyPreviousState[i] = mIsKeyPressed[i];
  }
}
```

7. Добавьте общедоступные функции для запросов к текущим состояниям клавишь для поддержки разработчика клиентской игры

```
// Function for GameEngine programmer to test if a key is pressed down
function isKeyPressed(keyCode) {
  return mIsKeyPressed[keyCode];
}
function isKeyClicked(keyCode) {
  return mIsKeyClicked[keyCode];
}
```

8. Наконец, экспортируйте общедоступные функции и ключевые константы:

```
export {keys, init,
  update,
  isKeyClicked,
  isKeyPressed
}
```

Измените движок для поддержки ввода с клавиатуры

Чтобы должным образом поддерживать ввод, перед началом игрового цикла движок должен инициализировать массивы предыдущих состояний **mIsKeyPressed**, **mIsKeyClicked** и **mKeyPreviousState** . Чтобы должным образом фиксировать действия игрока во время игрового процесса из ядра игрового цикла, эти массивы должны быть соответствующим образом обновлены.

1. Инициализация входного состояния: Изменить **index.js** путем импорта **input.js** модуль, добавляющий инициализацию ввода в функцию движка **init()** и добавляющий модуль ввода в экспортируемый список, чтобы разрешить доступ от разработчика клиентской игры.

```
import * as input from "./input.js"

// general utilities
import Camera from "./camera.js";
import Transform from "./transform.js";
import Renderable from "./renderable.js";

// general engine utilities
function init(htmlCanvasID) {
  glSys.init(htmlCanvasID);
  vertexBuffer.init();
  shaderResources.init();
  input.init();
}
```

```
export default {
  // input support
  input,
  // Util classes
  Camera,Transform,Renderable,
  // functions
  init, clearCanvas
}
```

2. Чтобы точно фиксировать изменения состояния клавиатуры, компонент ввода должен быть интегрирован с ядром игрового цикла. Включите функцию **update()** в основной игровой цикл, добавив следующие строки в **loop.js** . Обратите внимание, что остальная часть кода идентична.

```
import * as input from "../input.js"
"use strict"
const kUPS = 60; // Updates per second
const kMPF = 1000 / kUPS; // Milliseconds per update.
// Variables for timing gameloop.
let mPrevTime;
let mLagTime;
// The current loop state (running or should stop)
let mLoopRunning = false;
let mCurrentScene = null;
let mFrameID = -1;

function loopOnce() {
  if (mLoopRunning) {
    // Step A: set up for next call to LoopOnce
    mFrameID = requestAnimationFrame(loopOnce);
    // Step B: now let's draw
    // draw() MUST be called before update()
    // as update() may stop the loop!
    mCurrentScene.draw();

    // Step C: compute time elapsed since last loopOnce was executed
    let currentTime = performance.now();
    let elapsedTime = currentTime - mPrevTime;
    mPrevTime = currentTime;
    mLagTime += elapsedTime;
    // Step D: update the game the appropriate number of times.
    // Update only every kMPF (1/60 of a second)
    // If lag larger then update frames, update until caught up.
    while ((mLagTime >= kMPF) && mLoopRunning) {
      input.update()
      mCurrentScene.update();
      mLagTime -= kMPF;
    }
  }
}
```

Тестовый ввод с клавиатуры

Вы можете протестировать функциональность ввода, изменив **Renderable** объекты в классе **MyGame**. Замените код в функции **MyGame update()** следующим:

```
update() {
  // Simple game: move the white square and pulse the red
  let whiteXform = this.mWhiteSq.getXform();
  let deltaX = 0.05;
  // Step A: test for white square movement
  if (engine.input.isKeyPressed(engine.input.keys.Right)) {
    if (whiteXform.getXPos() > 30) { // right-bound of the window
      whiteXform.setPosition(10, 60);
    }
    whiteXform.incXPosBy(deltaX);
  }
  // Step B: test for white square rotation
  if (engine.input.isKeyClicked(engine.input.keys.Up)) {
    whiteXform.incRotationByDegree(1);
  }

  let redXform = this.mRedSq.getXform();
  // Step C: test for pulsing the red square
  if (engine.input.isKeyPressed(engine.input.keys.Down)) {
    if (redXform.getWidth() > 5) {
      redXform.setSize(2, 2);
    }
    redXform.incSizeBy(0.05);
  }
}
```

В предыдущем коде шаг **A** гарантирует, что нажатие и удерживание клавиши со стрелкой вправо переместит белый квадрат вправо. **Шаг B** проверяет наличие события нажатия, а затем отпускания клавиши со стрелкой вверх. Белый квадрат поворачивается при обнаружении такого события. Обратите внимание, что нажатие и удерживание клавиши со стрелкой вверх не приведет к непрерывному нажатию клавиши и, следовательно, не приведет к непрерывному вращению белого квадрата. На **шаге C** выполняется проверка нажатия и удержания клавиши со стрелкой вниз для перемещения красного квадрата.

Вы можете запустить проект и включить дополнительные элементы управления для управления квадраты. Например, включите поддержку клавиш **WASD** для управления расположением красной площади. Еще раз обратите внимание, что, увеличивая / уменьшая величину изменения положения, вы эффективно контролируете скорость перемещения объекта.

Примечание Термин **“клавиши WASD”** используется для обозначения привязки клавиш популярных игровых элементов управления: клавиша **W** для перемещения вверх, **A** влево, **S** вниз и **D** вправо.

Управление ресурсами и асинхронная загрузка

Видеоигры обычно используют множество художественных средств или ресурсов, включая аудиоклипы и изображения. Ресурсы, необходимые для поддержки игры, могут быть большими. Кроме того, важно поддерживать независимость между ресурсами и самой игрой, чтобы их можно было обновлять независимо, например, изменяя фоновый звук без изменения самой игры. По этим причинам игровые ресурсы обычно хранятся извне, на системном жестком диске или сервере по сети. Хранящийся снаружи по отношению к игре ресурсы иногда называются **внешними ресурсами(external resources)** или **активами(assets)**

После начала игры внешние ресурсы должны быть явно загружены. Для эффективного использования памяти игра должна динамически загружать и выгружать ресурсы в зависимости от необходимости. Однако загрузка внешних ресурсов может включать операции устройств ввода/вывода или задержки сетевых пакетов и, таким образом, может занимать много времени и потенциально влиять на интерактивность в реальном времени. По этим причинам в любом случае в игре только часть ресурсов хранится в памяти, где операции загрузки выполняются стратегически, чтобы избежать прерывания игры.

В большинстве случаев ресурсы, необходимые на каждом уровне, сохраняются в памяти во время прохождения этого уровня. При таком подходе загрузка внешних ресурсов может происходить во время переходов на уровни, когда игроки ожидают новой игровой среды и с большей вероятностью допустят небольшие задержки загрузки.

После загрузки ресурс должен быть легкодоступен для поддержки интерактивности. Эффективное управление ресурсами имеет важное значение для любого игрового движка. Обратите внимание на четкое различие между управлением ресурсами, за которое отвечает игровой движок, и фактическим владением ресурсами.

Например, игровой движок должен поддерживать эффективную загрузку и воспроизведение фоновой музыки для игры, и именно игра (или клиент игрового движка) фактически владеет аудиофайлом для фоновой музыки и предоставляет его. При внедрении поддержки внешнего управления ресурсами важно помнить, что сами ресурсы не являются частью игрового движка. На данный момент игровой движок, который вы создавали, обрабатывает только один тип ресурсов — файлы **шейдеров GLSL**. Напомним, что объект **SimpleShader** загружается и компилирует файлы **simple_vs.glsl** и **simple_fs.glsl** в своем конструкторе. До сих пор загрузка файла шейдера осуществлялась с помощью синхронного **XMLHttpRequest.open()**.

Такая синхронная загрузка является примером неэффективного управления ресурсами, поскольку никакие операции не могут выполняться, пока браузер пытается открыть и загрузить файл шейдера. Эффективной альтернативой было бы выдать команду асинхронной загрузки и разрешить выполнение дополнительных операций во время открытия и загрузки файла. В этом разделе создается инфраструктура для поддержки асинхронной загрузки и эффективного доступ к загруженным ресурсам. Основываясь на этой инфраструктуре, в течение следующих нескольких проектов игровой движок будет расширен для поддержки пакетной загрузки ресурсов во время переходов сцен.

Проект карты ресурсов и загрузчика шейдеров

Этот проект поможет вам разработать компонент **resource_map**, инфраструктурный модуль для управления ресурсами, и продемонстрирует, как работать с этим модулем для асинхронной загрузки шейдерных файлов. Вы можете увидеть пример этого проекта, запущенного в Рисунок 4-3. Этот проект, по-видимому, идентичен предыдущему проекту, с той лишь разницей, что загружаются шейдеры GLSL. Исходный код этого проекта определен в папке главы 4/4.3.resource_map_and_shader_loader.

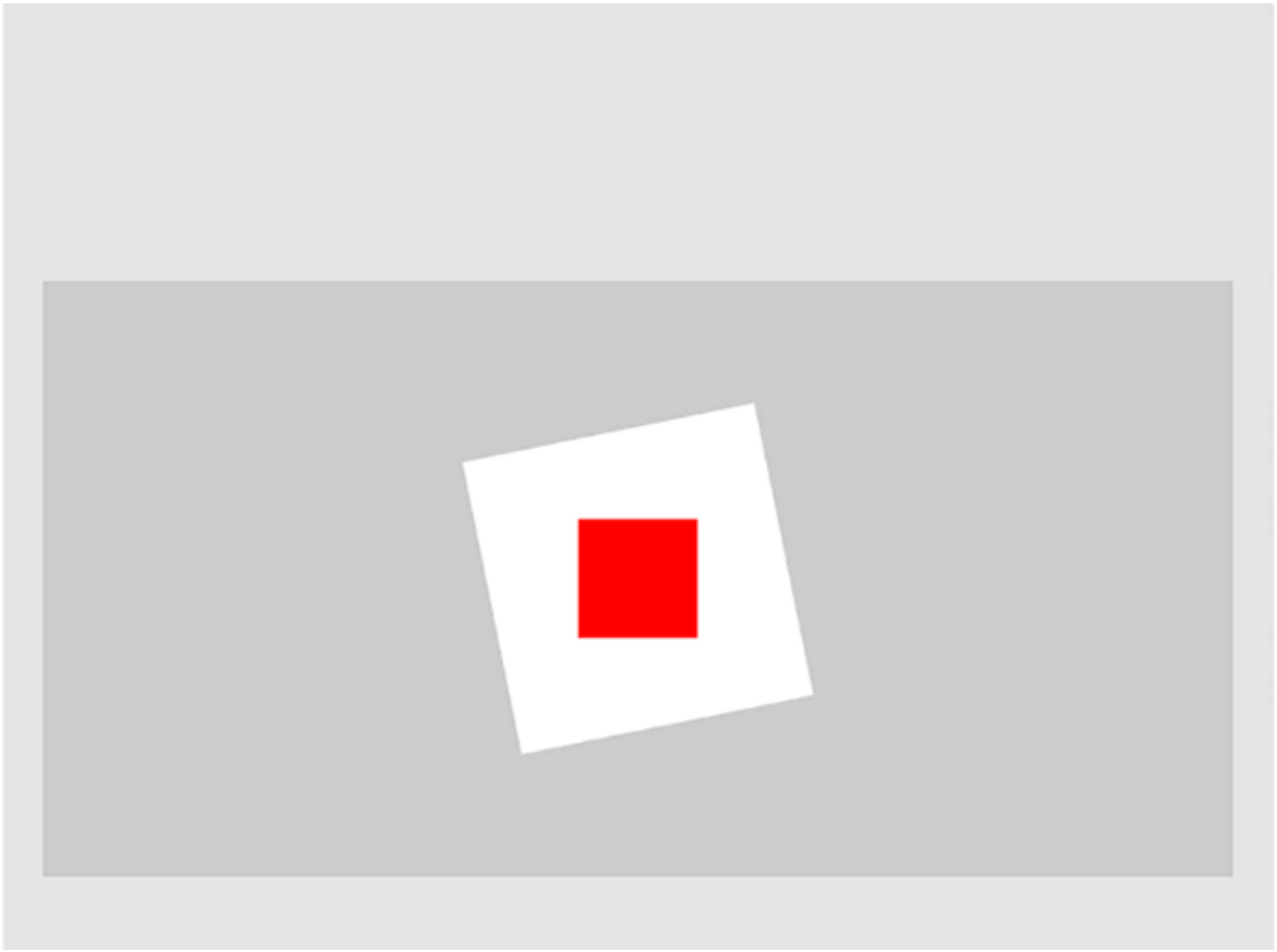

Рисунок 4-3. Запуск проекта карты ресурсов и загрузчика шейдеров

Элементы управления проекта идентичны предыдущему проекту следующим образом:

- **Клавиша со стрелкой вправо**: перемещает белый квадрат вправо и переносит его в левую часть игрового окна.
- **Клавиша со стрелкой вверх**: поворачивает белый квадрат
- **Клавиша со стрелкой вниз**: увеличивает размер красного квадрата, а затем сбрасывает его на пороговое значение.

Целями проекта являются следующие:

- Чтобы понять, как обрабатывать асинхронную загрузку
- Создать инфраструктуру, поддерживающую будущую загрузку ресурсов и доступ к ним
- Для асинхронной загрузки ресурсов через загрузку файлов шейдеров GLSL

Примечание Для получения дополнительной информации об асинхронных операциях Javascript вы можете обратиться ко многим отличным ресурсам в Интернете, например `https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous`

Добавьте компонент карты ресурсов в движок

Компонент движка **resource_map** управляет загрузкой, хранением и извлечением ресурсов после загрузки ресурсов. Эти операции являются внутренними для игрового движка и не должны быть доступны клиенту игрового движка. Как и в случае со всеми

компонентами ядра движка, например, **game loop**, файл исходного кода создается в папке **js/engine/core**. Подробности заключаются в следующем.

1. Создайте новый файл в папке **js/engine/core** и назовите его **resource_map.js** .
2. Определите класс **MapEntry** для поддержки подсчета ссылок на загруженные ресурсы. Подсчет ссылок необходим для того, чтобы избежать множественных загрузок или преждевременной выгрузки ресурса.

```
class MapEntry {
  constructor(data) {
    this.mData = data;
    this.mRefCount = 1;
  }
  decRef() { this.mRefCount--;}
  incRef() { this. mRefCount++;}
  set(data) { this.mData = data;}
  data() { return this.mData; }
  canRemove() { return (this.mRefCount == 0);}
}
```

3. Определите карту пары ключ-значение, **mMap**, для хранения и извлечения ресурсов и массив **mOutstandingPromises** , для учета всех невыполненных операций асинхронной загрузки:

```
let mMap = new Map();
let mOutstandingPromises = [];
```

Примечание Объект **Map** Javascript содержит коллекцию пар ключ-значение.

---

4. Определите функции для запроса о существовании, извлечения и настройки ресурса. Обратите внимание, что, как следует из имени переменной параметра **path**, ожидается, что полный путь к файлу внешних ресурсов будет использоваться в качестве ключа для доступа к соответствующему ресурсу, например, используя путь к файлу **js/glsl_shaders/simple_vs.glsl** в качестве ключа для доступа к файлу содержимое файла.