РАЗРАБОТКА ЯДРА ИГРОВОГО ДВИЖКА

Т.П. Канатуш
УО «Гродненский государственный университет им. Я. Купалы», факультет математики и информатики, специальность «Программное обеспечение информационных технологий», кафедра современных технологий программирования.

Научный руководитель – Л.В. Рудикова, кандидат физико-математических наук, доцент, заведующий кафедрой современных технологий программирования.

В статье рассмотрен пример разработки игрового движка, который позволяет экономить на написании кода в игровом приложении, вынося повторяющиеся из игры в игру элементы и используя высокоуровневые операции. Это работа с физикой, графикой, звуком, сетью и т.д. Концепция игровых движков является неотъемлемой частью в проектировании игровых приложений. Движок может быть определен как неигровая специфическая технология, которая развивается независимо от геймплейной составляющей. Развитие данной технологии позволяет достичь высокого результата и ускорить процесс разработки компьютерных игр. Во введении рассмотрены: задача и особенности разработки игрового движка. В основной части подробнее описан функционал алгоритм функционирования разработки. В заключении описаны преимущества предлагаемой разработки и возможные варианты улучшения.

Ключевые слова: архитектура движка, ядро движка, компьютерные игры, игровое приложение.

Введение. В наше время компьютерные игры стали неотъемлемой частью нашей культуры, а также полноценным продуктом на потребительском рынке в сфере развлечений. Данная, быстро развивающаяся, информационная отрасль заслуживает особого внимания и понимания того, как она функционирует.

Любое игровое приложение состоит из двух независимых частей: это игровой движок и сама игра. Игровой движок – это, по сути, сложный конвейер, преобразующий входящие ресурсы компьютера, выводя их в виде графики. От игры же требуется грамотно распорядиться данными ресурсами и задать им поведение, так называемые скрипты.

Игровой движок позволяет экономить на написании кода в игровом приложении вынося повторяющиеся из игры в игру элементы, используя высокоуровневые операции. Это работа с физикой, графикой, звуком, сетью и так далее.

Как можно понять, концепция игровых движков является неотъемлемой частью в проектировании игровых приложений. Движок может быть определен как неигровая специфическая технология, которая развивается независимо от геймплейной составляющей. Развитие данной технологии позволяет достичь высокого результата и ускорить процесс разработки компьютерных игр.

Основная часть. ***Компоненты игрового движка.*** Игровой движок – программное ядро комплексной программной системы (игры), содержащее базовую функциональность игры, но при этом, не включающее код, специфичный для геймплейной функциональности конкретной игры.

В современных моделях программирования игровые движки используются в качестве механической основы игр. Игровые движки состоят из множества компонентов-модулей, которые реализуют игровой функционал в виде отображения и обработки графики, звука искусственного интеллекта и прочего. В дальнейшем остаётся дополнить движок контентом, который будет уже соответствовать конкретно разрабатываемой игре или программе. Модульный дизайн игровых движков позволяет игрокам и программистам легко заменять его части модифицировать их с целью создания новых игр с новыми моделями, улучшенной графикой, звуками, иным сценарием, изменять существующий материал и добавлять новые функции.

В результате построения диаграммы компонентов была получена общая структура игрового движка (рисунок 1) и список всех компонентов. Она отображает зависимость одних компонентов от других компонентов и список используемых библиотек или пакетов каждым из них. Компоненты представляют из себя отдельные модули, чаще всего разрабатывающиеся независимо друг от друга. Вместе они образуют единую сеть зависимостей, собираясь в один корневой компонент: ядро игрового движка, который используется игрой через API игрового движка.

Рассмотрим компоненты игрового движка.

Game – это сама игра. В этот модуль входит не только код игры, но и ресурсы. Есть разные варианты реализации этого модуля. Иногда язык написания игры совпадает с языком движка. К примеру, движок написан на C++ и сама игра также написана на C++. В других случаях между модулем Game и движком появляется еще один слой-прослойка. Которая предоставляет другие варианты разработки. К примеру, на

языке Lua, CSharp и другие. Пример движки LOVE, Corona SDK - разработка игры реализуется на Lua, основной код движка скрыт и не доступен.

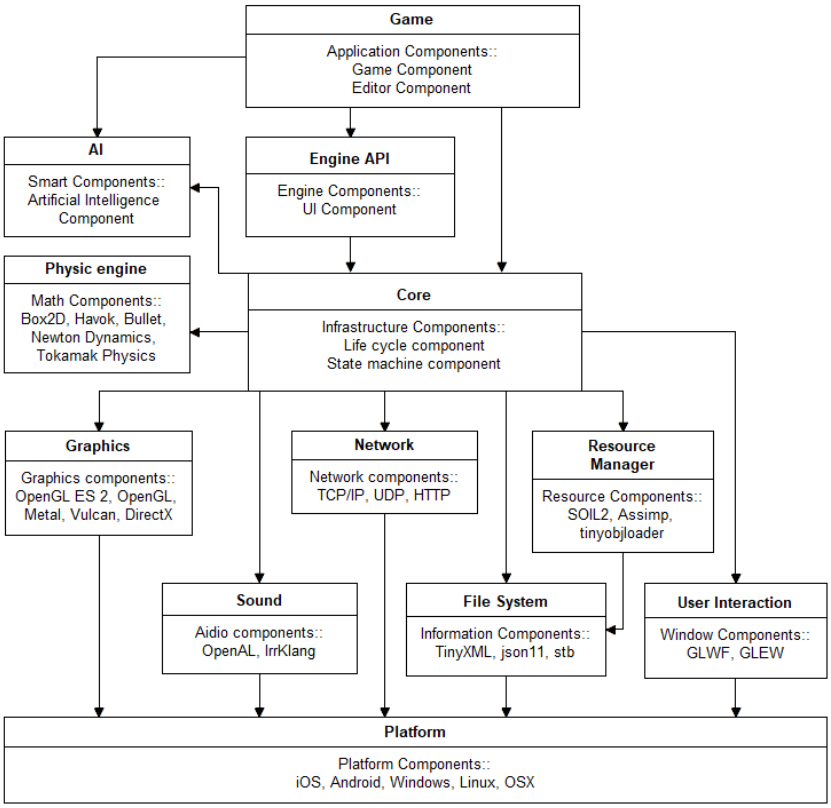

Рисунок 1 – Общая структура игрового движка

Engine API – программный интерфейс движка, с помощью которого можно работать с движком. Набор функций, классов и структур.

Core – ядро движка, тут происходит жизненный цикл игры. Это основной компонент, который связывает все остальные.

AI – искусственный интеллект, который может использоваться в игре. Это может быть часть движка, часть игры. В некоторых играх модуля может не быть или не использоваться.

Physic engine – физический движок. Это может быть какое-нибудь самописное или готовое решение. Среди используемых Box2D, Havok, Bullet, Newton Dynamics, Tokamak Physics.

Resource manager – менеджер ресурсов. Довольно важный компонент движка. Позволяет работать с ресурсами – звуками, графикой и другими файлами. Позволяет оптимизировать работу с загрузкой и переиспользованием. К примеру, нужно использовать в 10 разных местах один и тот же звук - мы будет хранить только 10 ссылок и один загруженный файл, вместо 10 выделенных блоков одних и тех же данных. При нехватке памяти, данные будет выгружены. И другие полезные функции по работе с ресурсами.

Graphics – модуль графики. Здесь у нас будет реализован рендер. Для разных систем будет разные API для графики. До недавнего времени это был OpenGL ES 2 (и просто OpenGL). Сейчас идет разделение графических подсистем. Для iOS\OSX это Metal, для остальных пока OpenGL, Vulkan.

Sound – звуковой модуль. Позволяет воспроизводить звуки и музыку. В качестве форматов используется несжатый wav и сжатый mp3 для музыки. Но все потребности может покрыть открытый формат Ogg. Это сжатый и открытый формат аудиофайлов. В качестве связующего звена выступает OpenAL или платформозависимые решения (IrrKlang).

Network – по названию понятно, что работа модуля связана с сетью и обменом данных. Это может быть TCP\UDP пакеты и в виде готовый протоколов HTTP.

File system – работа с файловой системой. Каждый платформа имеет свои особенности работы с данными. Для многих платформ можно просто обратится к данным со стандартными функциями, в некоторых только с API платформы. К примеру, при работе с файлами, которые хранятся в APK - необходимо использовать встроенные функции в NDK.

User interaction – это события от пользователя. Нажатие кнопок или движение и клик мыши. Этот модуль получает от пользователя события при помощи оконной системы.

Platform – это уже целевая платформа на которой будет запускаться движок (игра): iOS, Android, Windows, Linux, OSX и возможно другие варианты. К сожалению, Windows Phone уже рассматривать не стоит, хотя платформа была интересная. Каждый движок имеет разный баланс погружения в нативную

реализацию. Некоторые движки максимально пытаются отделится от платформы и используют максимально узкую прослойку между системой и движком. Для примера если движок на C++, а платформа iOS - то используемый язык будет Objective C и чтобы реализовать к ним доступ нужно к нужным методам реализовать некоторую обвертку. Некоторые движки пишут большую часть реализации в нативном коде. Например, вывод звуков реализована с помощью языка Java для платформы Android, а вызов происходит с помощью нескольких функций.

Конечный автомат. Конечный автомат – это некоторая абстрактная модель, содержащая конечное число состояний. Используется для представления и управления потоком выполнения каких-либо команд [3]. Многие проблемы в программировании игр могут быть решены естественным путём с помощью конечных автоматов. По этой причине многие игровые движки создают концепцию конечных автоматов прямо в ядре игровой объектной модели.

Конечный автомат представляет собой граф, вершины которого являются состояниями, а рёбра – переходы между состояниями. Каждое ребро имеет метку, хранящую условие перехода из одного состояния в другое. В разработке используется простая форма FSM для управления различными состояниями, которые движку необходимо обработать в определённый момент времени.

Реализация конечного автомата начинается с выявления его состояний и переходов между ними.

Отправной точкой в нашем случае является состояние «Loading», которое исполняет роль загрузочного окна, из которого мы попадаем в главное меню программы (рисунок 2) Находясь в меню, мы можем перейти либо в игровое состояние, либо в меню настроек. Из игрового состояния, в случае какой-либо исключительной ситуации, переходим в состояние ошибки программы, а там следом обратно в главное меню программы (рисунок 2).

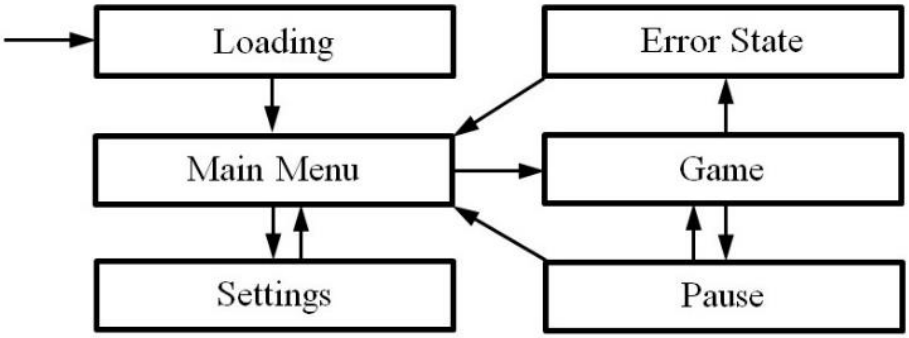

Рисунок 2 – Конечный автомат

При реализации данного конечного автомата возникает проблема. Допустим, что текущее состояние это «Pause». Если мы выходим из меню паузы, то куда мы должны попасть далее: обратно в игру или в главное меню?

Решением такой проблемы является конечный автомат, основанный на стеке. Данный вид FSM использует стек для управления состояниями. В верхней части стека находится активное состояние, а переходы возникают при добавлении или удалении состояний из стека.

Можно выделить три перехода в FSM, основанном на стеке: добавление с заменой, добавление без замены (перекрытие) и удаление состояний.

Добавление состояния в стек конечного автомата с заменой проходит в три этапа: удаление из вершины стека текущего состояния, добавление нового состояния, установка метки как активного состояния, то есть запуск работы состояния в вершине стека (рисунок 3).

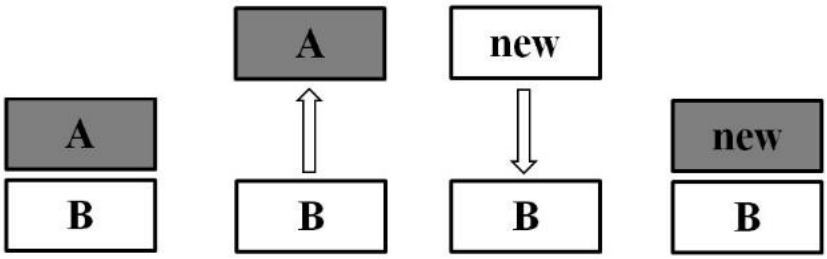

Рисунок 3 – Добавление состояния в стек с заменой

Добавление состояния в стек конечного автомата без замены, другими словами можно сказать перекрытие текущего состояния проходит в три этапа: остановка текущего состояния, добавление нового состояния в вершину стека, установка метки как активного состояния, то есть запуск работы состояния в вершине стека (рисунок 4).

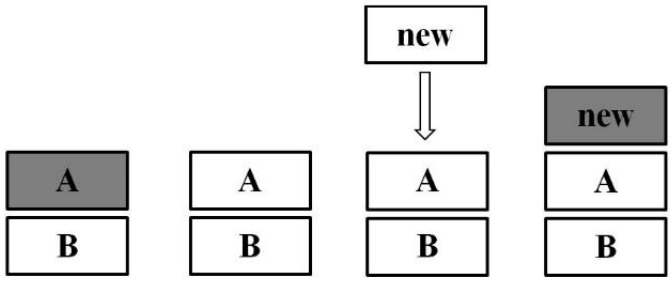

Рисунок 4 – Добавление состояния в стек без замены (перекрытие)

Удаление текущего состояния из стека конечного автомата проходит в два этапа: удаление из вершины стека текущего состояния, возобновление работы состояния в вершине стека (рисунок 5).

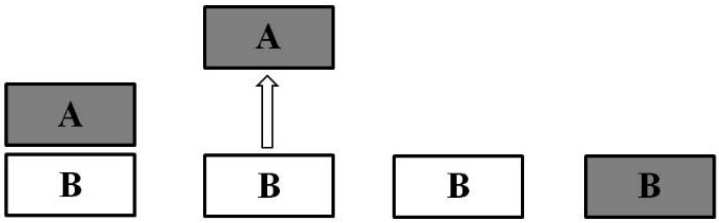

Рисунок 5 – Удаление состояния из стека

Конечные автоматы, безусловно, полезны для реализации управления состояниями программы. Они могут быть легко представлены в виде графа, что позволяет разработчику увидеть все возможные варианты.

Реализация конечного автомата с функциями-состояниями является простым, но в то же время мощным методом. Даже более сложные переплетения состояний могут быть реализованы при помощи FSM.

Состояния игрового приложения. В результате построения диаграммы состояний был сформирован конечный автомат, по сути граф, вершины которого являются состояниями игрового приложения и рёбрами, являющимися переходами между ними с имеющимися для этого условиями соответственно для каждого. На ней видны основные состояния приложения и возможные переходы между ними. Каждое состояние представляет собой конкретную отображаемую сцену и интерфейс взаимодействия с пользователем или внутренний процесс приложения (рисунок 6).

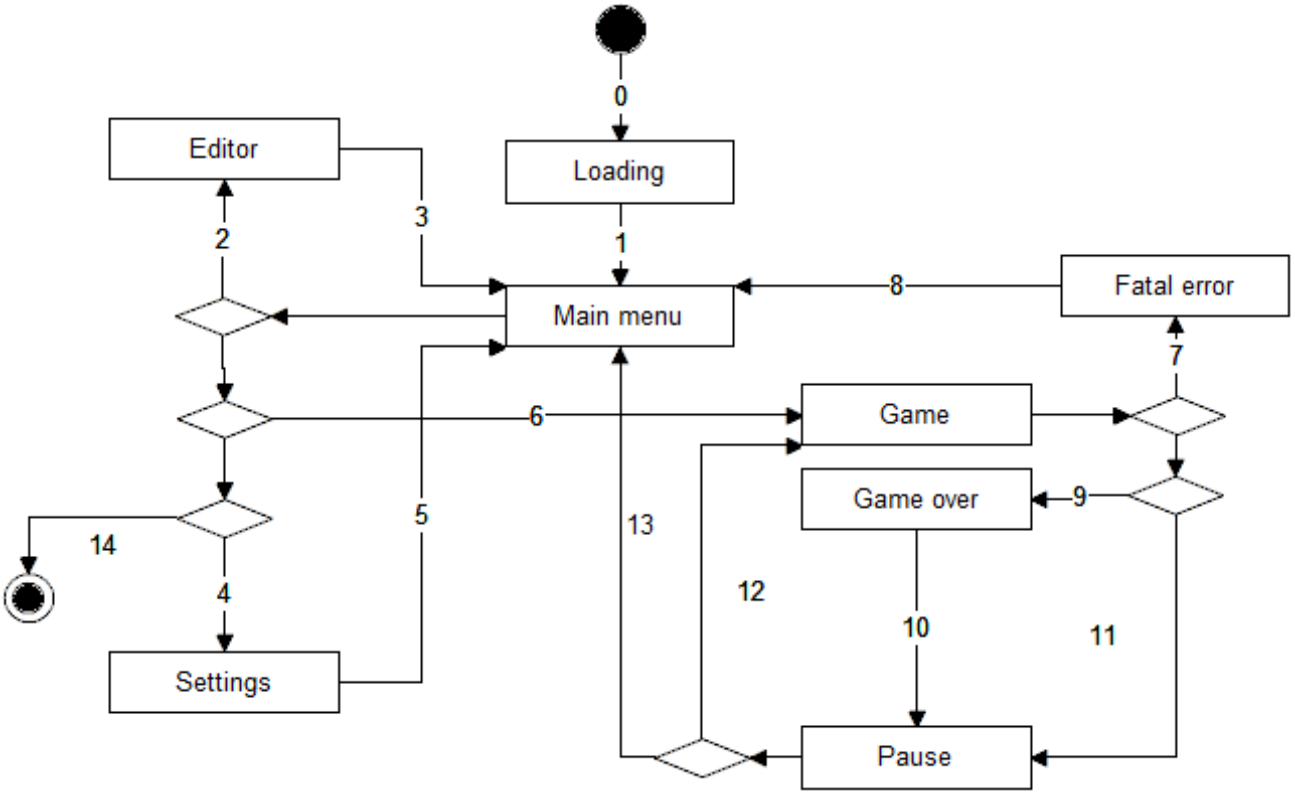

Рисунок 6 – Диаграмма состояний игрового приложения

Состояния:

1. Loading – состояние загрузки игрового приложения: приложение загружает требуемые для его старта начальные ресурсы.
2. Main menu – главное меню игрового приложения, в котором можно выбрать дальнейшие действия.
3. Editor – редактор уровней: позволяет редактировать загружаемые уровни для игры.
4. Settings – меню настроек игрового приложения: задаются различные параметры игры, а также самого приложения.
5. Game – состояние игры: игровой процесс приложения, при котором интерактивное взаимодействие с пользователем (игроком) происходит в реальном времени.
6. Fatal error – состояние ошибки возникает в случае непредвиденных ошибок в процессе игры, которое позволит, вернувшись в главное меню, перезапустить игровое приложение.
7. Game over – состояние проигрыша наступает после логического конца игры в случае достижения цели или потери игровых очков и так далее.
8. Pause – состояние игровой паузы: позволяет приостановить игровой процесс, соответственно его возобновить или покинуть игровую сессию, вернувшись в главное меню игрового приложения.

Условия перехода между состояниями:

0. Точка входа в граф состояний, обозначающая запуск пользователем игрового приложения.
1. Загрузка основных ресурсов завершена, что инициализирует запуск меню игрового приложения.
2. Нажатие кнопки «Editor» в главном меню открывает редактор уровней.
3. Нажатие клавиши Escape в редакторе уровней возвращает в главное меню.
4. Нажатие кнопки «Settings» в главном меню откроет меню настроек игрового приложения.
5. Нажатие кнопки «Back» в меню настроек возвращает в главное меню.
6. Нажатие кнопки «Start game» в главном меню запускает игровой процесс.
7. Возникновение любой фатальной ошибки или исключительной ситуации перенаправит в состояние, которое прекратит работу игровой сессии.
8. Подтверждение пользователем наличия проблемы нажатием на кнопку «Ок» вернёт в главное меню.
9. Полная потеря очков жизни всех игроков завершает игровой процесс, выводя счёт игры.
10. Нажатие клавиши Escape после проигрыша открывает меню паузы.
11. Нажатие клавиши Escape во время игровой сессии останавливает игровой процесс и открывает меню паузы.
12. Нажатие кнопки «Resume» в меню паузы возобновит игровую сессию.
13. Нажатие кнопки «To menu» в меню паузы закончит игровую сессию и откроет главное меню.

Минимальный жизненный цикл игры. В результате создания диаграммы активностей была получена модель поведения отдельно взятого состояния игрового движка, а также все условия изменения текущего поведения (рисунок 7). На ней отображены основные этапы жизненного цикла приложения, которые позволят понять схему поведения при обработке других таких же состояний. Также отобразили три этапа обработки логических данных и построения графического контекста (рендеринга) перед финальной отрисовкой.

Активные элементы:

- Load Resource – запрос на получение внешних ресурсов: изображения, сетки, звука, карты, с последующей загрузкой их в оперативную память.
- Open Resource – запрос на получение внешних ресурсов: изображения, сетки, звука, карты, хранящихся во внутреннем хранилище игрового движка.
- Unload Resource – запрос на выгрузку игровых ресурсов: изображения, сетки, звука, карты, путём освобождения памяти или замещения новыми внешними ресурсами.
- Create State – Callback метод, вызывающийся перед добавлением в стек обработки конечного автомата. Отвечает за начальное создание состояния и инициализацию внутренних ресурсов и компонентов.
- Start State – Callback метод, вызывающийся после размещения состояния в стеке обработки конечного автомата. Отвечает за запуск работы состояния, инициализирующий атрибуты компонентов перед их работой.
- Resume State – Callback метод, вызывающийся при возобновлении состояния после паузы, вызванной, например, перекрытием его другим состоянием, работающего в фоновом режиме. Отвечает за возобновление снимка текущего состояния перед продолжением его работы.
- Pause State – Callback метод, вызывающийся при временной остановке состояния, например, путём перекрытия его другим состоянием, работающего в фоновом режиме.
- Stop State – Callback метод, вызывающийся при полной остановке состояния, например, при извлечении из стека обработки конечного автомата и помещении в специальный пул состояний для дальнейшего его возобновления. Отвечает за полную остановку всех процессов, связанных с данным состоянием.

- Restart State – Callback метод, вызывающийся при перезапуске состояния в после извлечения из пула состояния и помещения его в стек обработки конечного автомата. Отвечает за сброс значений всех атрибутов и установки в изначальное состояние, как будто состояние только что создали.
- Destroy State – Callback метод, вызывающийся при уничтожении состояния в момент удаления из стека обработки конечного автомата. Отвечает за освобождение и удаление используемых состоянием ресурсов.
- Input – обработка внешних событий, связанных с пользовательским вводом при помощи внешних физических устройств: клавиатуры, мыши, джойстиков, геймпадов и других манипуляторов. Отвечает за взаимодействия с графическим интерфейсом, вызывая внутренние события, ведущие к внутреннему изменению состояния или его смены.
- PreUpdate – пред обновление логики осуществляется непосредственно перед основным обновлением. Включает в себя подготовку и обработку специфических данных, которые пригодятся в дальнейшем для основного обновления состояния.
- Update – обновление логики, которое может повториться 1 или 2 раза за одну итерацию цикла, включающее в себя изменение внутренних значений состояния на основании внешнего пользовательского ввода и/или данных с предыдущих итераций.
- PostUpdate – пост обновление логики, то есть финальная доработка и корректировка данных входящих в итоговое формирование логического контекста.
- PreRender – пре рендер графики.
- Render – рендер графики отвечает за создание плоской картинки – цифрового растрового изображения – по разработанной 3D или 2D сцене на основании геометрических данных, положения точки зрения наблюдателя.
- PostRender – пост рендер графики включает в себя пост эффекты – какие-либо визуальные эффекты, накладываемые поверх видео изображения на основании информации об освещённости или затемнения, наличия какого-то вещества, наличии преломляющих свет свойств и так далее.
- Drawing – отрисовка, как финальный этап формирования кадра, включает в себя все результаты рендеринга, а также отрисовка поверх этого элементов графического интерфейса и других управляющих элементов.

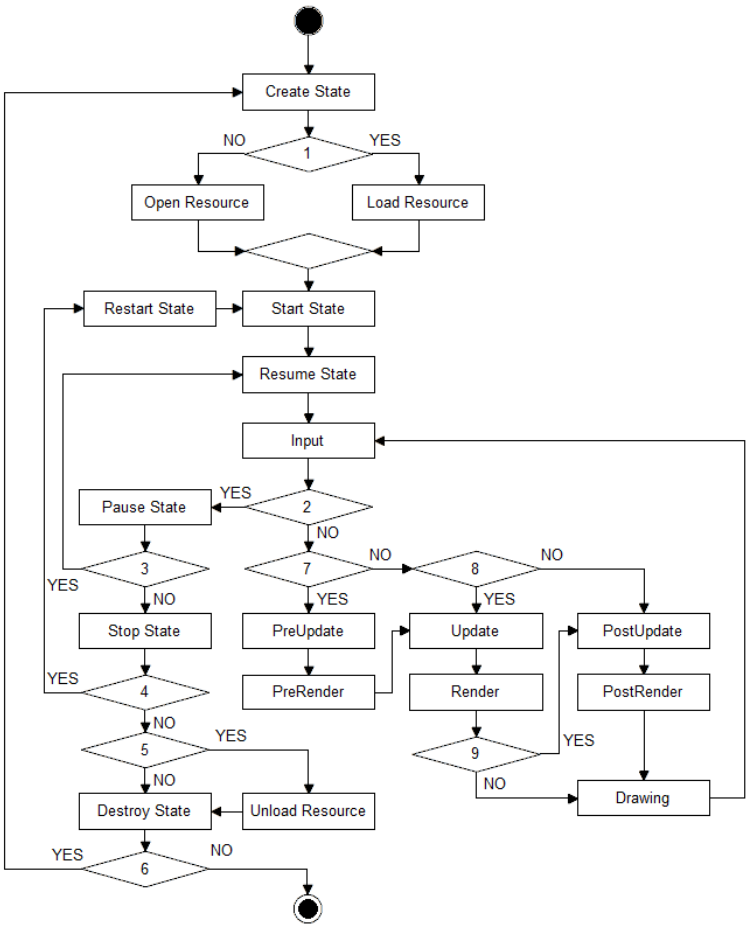

Рисунок 7 – Диаграмма минимального жизненного цикла игрового приложения

Условия изменения поведения:

1. Момент создания текущего состояния включает в себя инициализацию внутренних ресурсов, где решается нужно ли их загружать из вне или открыть уже имеющиеся во внутреннем хранилище.
2. Когда дело доходит до пользовательского ввода, который влияет на текущее состояние приложения, пользователь может повлиять на то, что нужно ли приостанавливать текущее состояние или продолжать обновление логики.
3. В приостановленном режиме мы можем возобновить работу текущего состояния, либо полностьюего остановить. Это конечно скажется на том, что все значения хранящихся логических данных будут сброшены, в следствии чего утеряны, если конечно не сохранить снимок текущего состояния.
4. Сохранённое в специальном пуле состояние может быть снова помещено в стек обработки конечного автомата и перезапущено без подгрузки, что ускоряет его возвращение в активное состояние, либо его попросту безвозвратно уничтожают.
5. В случае уничтожения текущего состояния следует остановить все процессы, сбросить данные, а также следует решить нужно ли выгружать из памяти используемые ресурсы или оставить для повторного использования другими состояниями.
6. После окончательного уничтожения состояния на основании пользовательского ввода или решения системы создаётся новое состояние или приложение завершает свою работу в случае, если больше не имеется состояний в стеке обработки конечного автомата.
7. В случае пассивного пользовательского ввода (ввода не влияющего на поведение самого состояния, а только на контекст логических данных) или его отсутствия система подготавливает специфические данные для последующей их обработки, либо просто пропускает данный этап формирования логических данных.
8. Если текущее состояние не включает в себя обработку сложных логических данных и является промежуточным состоянием для настройки и/или корректировки данных, то оно может пропустить этап обновления логики и перейти сразу к пост обновлению.
9. В зависимости от внутренних параметров системы и/или пользовательских настроек система может в дальнейшем корректировать и изменять основные логические данные для формирования специфических ситуаций или сразу же приступить к рендерингу и финальной отрисовке.

Заключение. Как можно понять, концепция игровых движков является неотъемлемой частью в проектировании игровых приложений. Движок может быть определен как неигровая специфическая технология, которая развивается независимо от геймплейной составляющей. Развитие данной технологии позволяет достичь высокого результата и ускорить процесс разработки.

В будущем, наш проект может представлять как практическую, так и академическую пользу. Поскольку в случае его удачной реализации, его можно будет развивать дальше, а релизная версия, написанная в максимально простом виде, может в дальнейшем использоваться как шаблон, которым смогут воспользоваться все желающие для того, чтобы ознакомиться с архитектурой игровых движков на конкретном примере.

Совершенствование, исследование и использование данной концепции в игровых технологиях, является актуальной темой.

Список литературы

1. GameDev.ru [Электронный ресурс] / Программирование/Термины/Движок. – Режим доступа: http://www.gamedev.ru/code/terms/Engine. – Дата доступа: 25.03.2020.

2. 3DNews.ru Daily Digital Digest [Электронный ресурс] / Анатомия игровых движков. – Режим доступа: https://3dnews.ru/games/engines. – Дата доступа: 25.03.2020.

3. Tproger.ru [Электронный ресурс] / Конечный автомат: теория и реализация. – Режим доступа: https://tproger.ru/translations/finite-state-machines-theory-and-implementation. – Дата доступа: 25.03.2020.

4. Википедия [Электронный ресурс] / Физический движок. – Режим доступа: https://clck.ru/DQExy. – Дата доступа: 25.03.2020.

5. Robert Nystrom. Game Programming Patterns / Robert Nystrom – GB genever benning 2014

6. Jason Gregory. Game Engine Architecture / Jason Gregory – CRC Press 2009

7. Jason Gregory. Game Engine Architecture. Second edition / Jason Gregory – CRC Press 2015

8. Джошуа Глейзер, Санджай Мадхав. Многопользовательские игры Разработка сетевых приложений / Джошуа Глейзер, Санджай Мадхав – Питер 2017