|  |
| --- |
| **МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ**  Федеральное государственное бюджетное образовательное учреждение  высшего профессионального образования  **«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ**  **ТОМСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»** |

Институт кибернетики  
Направление – информатика и вычислительная техника  
Кафедра автоматизации и компьютерных систем

**ПРОГРАММНЫЙ КАРКАС ДЛЯ СОЗДАНИЯ СПРАЙТОВОЙ АНИМАЦИИ НА HTML5**

**Выпускная квалификационная работа**

**(на соискание квалификации бакалавр)**

Студент гр. 8В94 ИК ТПУ  А.С. Лизин

Руководитель

Аспирант каф. АиКС ИК ТПУ Ф.Е. Татарский

**Допустить к защите:**

Заведующий кафедрой АиКС ИК ТПУ

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
| (Подпись) |  | (Дата) |  | (Фамилия И.О.) |

ТОМСК – 2013 г.

**МИНИСТЕРСТВО НАУКИ И ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ**

Федеральное государственное бюджетное образовательное учреждение

высшего профессионального образования

**«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ**

**ТОМСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»**

|  |  |  |
| --- | --- | --- |
| Кафедра АиКС ИК ТПУ  **УТВЕРЖДАЮ** | | |
| Зав. кафедрой \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_ | | |
|  | подпись | И.О.Фамилия |
| «\_\_\_\_\_\_» | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_20\_\_ г. | |

**З А Д А Н И Е**

**на выполнение выпускной квалификационной работы**

Студенту гр. 8В94 Лизину Александру Сергеевичу

1. **Тема выпускной квалификационной работы** «Программный каркас для создания спрайтовой анимации на html5»,утверждена приказом от \_\_\_\_\_\_\_\_\_ № \_\_\_\_\_\_\_
2. **Срок сдачи студентом готовой работы:** 11 июня 2013 г.
3. **Исходные данные к работе:**
   1. Ознакомится с программным интерфейсом технологи 2d-context HTML5.
   2. Выявить требования к каркасу.
   3. Изучить необходимые алгоритмы и техники.
   4. Спроектировать и реализовать программный каркас.
4. **Содержание расчётно-пояснительной записки (перечень вопросов, подлежащих разработке):**
   1. Процесс создания веб-приложений.
   2. Программный интерфейс технологии 2d-context HTML5.
   3. Проблема создания мультимедийных веб-приложений с использованием 2d-context HTML5.
   4. Архитектура программного каркаса.
5. **Дата выдачи задания на выполнение выпускной квалификационной работы:**

20 января 2013 г.

Руководитель

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(подпись, дата, И.О.Фамилия)

#### Задание принял к исполнению (студент)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(подпись, дата, И.О.Фамилия)

# РЕФЕРАТ

Дипломная работа на тему «Программный каркас для создания спрайтовой анимации на HTML5» решает проблему сложности создания интерактивных приложений с двумерной графикой с применением 2d-context HTML5. Исходя из поставленной задачи, в работе подробно рассмотрена область компьютерной графики в целом и проблема сложности использования 2d-context в частности. В работе предлагается метод решения упомянутой проблемы с помощью создания специализированного каркаса.

Работа представляет интерес для специалистов, работающих в области создания мультимедийных приложений(# игр), а также для всех, кому интересна данная тематика. Работа содержит восемь рисунков, две таблицы и два приложения. Общий объем работы составляет 41 страницу. Структура работы представлена списком определений, введением, двумя главами, заключением, списком литературы, а также приложениями.

# СОДЕРЖАНИЕ

[РЕФЕРАТ 4](#_Toc358541986)

[СОДЕРЖАНИЕ 5](#_Toc358541987)

[ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ 7](#_Toc358541988)

[ВВЕДЕНИЕ 9](#_Toc358541989)

[1. МУЛЬТИМЕДИЙНОЕ ВЕБ-ПРИЛОЖЕНИЕ 10](#_Toc358541990)

[1.1. Современное веб-приложение 10](#_Toc358541991)

[1.2. Архитектура веб-приложения 11](#_Toc358541992)

[1.3. Клиентские технологии 12](#_Toc358541993)

[1.4. Технология 2d-context HTML5 14](#_Toc358541994)

[1.5. Сложности разработки с помощью 2d-context HTML5 19](#_Toc358541995)

[1.6. Постановка задачи 21](#_Toc358541996)

[2. АРХИТЕКТУРА ПРОГРАММНОГО КАРКАСА 22](#_Toc358541997)

[2.1. Обзор существующих решений 22](#_Toc358541998)

[2.2. Требования к программному каркасу 23](#_Toc358541999)

[2.3. Реализация 24](#_Toc358542000)

[2.3.1. Парадигма программирования 24](#_Toc358542001)

[2.3.2. Используемые алгоритмы 24](#_Toc358542002)

[2.3.3. Диаграмма классов 24](#_Toc358542003)

[2.3.3.1. Класс ArmLib 25](#_Toc358542004)

[2.3.3.2. Класс Layer 27](#_Toc358542005)

[2.3.3.3. Класс EventQueue 29](#_Toc358542006)

[2.3.3.4. Класс Primitive 29](#_Toc358542007)

[2.3.3.5. Класс ArmObj 30](#_Toc358542008)

[2.3.3.6. Класс Skeleton 31](#_Toc358542009)

[2.3.3.7. Класс VisualObj 33](#_Toc358542010)

[2.3.3.8. Класс Object 35](#_Toc358542011)

[2.3.3.9. Класс Image 36](#_Toc358542012)

[2.3.3.10. Класс Rect 36](#_Toc358542013)

[2.3.3.11. Класс Circle 37](#_Toc358542014)

[2.3.3.12. Класс Line 37](#_Toc358542015)

[2.3.4. Используемые шаблоны проектирования 38](#_Toc358542016)

[2.3.5. Принятые стандарты кодирования 39](#_Toc358542017)

[2.3.6. Интеграция классов каркаса 41](#_Toc358542018)

[2.4. Средства разработки 46](#_Toc358542019)

[2.4.1. Используемые утилиты и библиотеки 46](#_Toc358542020)

[2.4.2. Средства разработки 47](#_Toc358542021)

[2.4.3. Интегрированная среда разработки 47](#_Toc358542022)

[2.4.4. Браузер 48](#_Toc358542023)

[2.4.5. Отладчик и профайлер 49](#_Toc358542024)

[2.4.6. Система контроля версий 49](#_Toc358542025)

[ЗАКЛЮЧЕНИЕ 51](#_Toc358542026)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 52](#_Toc358542027)

[ПРИЛОЖЕНИЯ 53](#_Toc358542028)

[ТЗ 53](#_Toc358542029)

[Руководство программиста (ГОСТ 19.504) 53](#_Toc358542030)

# ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

**RIA** (от англ. Rich Internet application) — веб-приложение, доступное через Интернет, насыщенное функциональностью традиционных настольных приложений, которое предоставляется либо уникальной спецификой браузера, либо через плагин, либо путём «песочницы» (виртуальной машины).

**API** (англ. application programming interface) — набор готовых классов, процедур, функций, структур и констант, предоставляемых приложением (библиотекой, сервисом) для использования во внешних программных продуктах. Используется программистами для написания всевозможных приложений.

**SVG** (от англ. Scalable Vector Graphics) — язык разметки масштабируемой векторной графики, созданный W3C и входящий в подмножество расширяемого языка разметки XML, предназначен для описания двумерной векторной и смешанной векторно/растровой графики в формате XML.

**DOM** (от англ. Document Object Model) — не зависящий от платформы и языка программный интерфейс, позволяющий программам и скриптам получить доступ к содержимому HTML, XHTML и XML-документов, а также изменять содержимое, структуру и оформление таких документов.

**OpenGL ES** — подмножество графического интерфейса OpenGL, разработанное специально для встраиваемых систем — мобильных телефонов, карманных компьютеров, игровых консолей. OpenGL ES определяется и продвигается консорциумом Khronos Group, в который входят производители программного и аппаратного обеспечения, заинтересованные в открытом API для графики и мультимедиа.

**Шейдер** – программа, которая используется в трёхмерной графике для определения окончательных параметров изображения или объекта.

**Программный каркас –**

**Трехмерная графика реального времени –**

**BOM –**

**UNIX формат –**

**CamelCase нотация –**

**Репозиорий –**

**Ветка –**

**UML –**

# ВВЕДЕНИЕ

Использование насыщенных веб-приложений позволяет создавать приложения с функциональностью сравнимой с традиционными приложениями, а по таким параметрам как сохранность личных данных и системные требования даже выгодно превосходить традиционные приложения. Такая технология как 2d-context HTML5 позволяет создавать графические приложения с двумерной графикой сравнимой с традиционными графическими приложениями.

Традиционно разработчик заинтересован упростить процесс создания, увеличить надежность, а так же сократить время разработки. Распространенным способом реализации этого является создание программного каркаса. Программный каркас это программное обеспечение, облегчающее разработку и объединение частей программного проекта.

Технология 2d-context в силу относительной новизны, а так же незаконченной спецификации еще не располагает богатым набором развитых программных каркасов. В связи, с чем встает необходимость создании специализированного каркаса.

Реализованный каркас упрощает работу программиста за счет:

1. Предоставления готовых, согласованных реализации частей мультимедийнго приложения использующего 2d-context HTML5.
2. Упрощения взаимодействия с интерфейсом 2d-context.
3. Модульности каркаса.

# МУЛЬТИМЕДИЙНОЕ ВЕБ-ПРИЛОЖЕНИЕ

# Современное веб-приложение

С ростом возможностей современных веб-технологий, веб-приложение становится серьезной альтернативой для традиционных настольных приложений. Это связано с эволюцией браузеров, увеличением скорости и качества интернет соединения у среднего пользователя, а так же с появлением новых веб-технологий. Для исполнения веб-приложение нуждается только в наличии современного браузера, что позволяет считать веб-приложение кросплатформенным. Данный факт делает целевой платформой большинство современных вычислительных устройств, включая мобильные телефоны и даже некоторые виды бытовой техники. Хорошей оценкой возможностей мультимедийных веб-приложений могут послужить игры, так для большинства игр требуется достаточно большая производительность платформы, что включает в себя не только аппаратные, но и программные возможности. Программной платформой для веб-приложений, в первую очередь, служат браузеры и технологии доступные для программиста при их использовании. На данный момент с помощью таких технологий как Flash и HTML5 существует возможность разработки веб-приложений с трехмерной графикой реального времени. Это очень важный показатель программных возможностей платформы, так как для поддержки трехмерной графики реального времени требуется частота обновления изображения порядка 60 кадров в секунду с просчетом всей экранной сцены примерно за 16 миллисекунд. Это стало возможно благодаря появлению поддержки данными технологиями вычислительных возможностей видеокарт.

**Вывод: Современной мультимединое веб-приложение может служить альтернативой настольным графическим приложениям, таким как игры.**

# Архитектура веб-приложения

Веб-приложение представляет клиент-серверное приложение, в котором клиентом является браузер, а сервером – веб-сервер. Логика приложения распределена между клиентской и серверной частями. Данные используемые приложением хранятся на сервере и используются приложением по запросу.

Клиентская часть приложения реализует пользовательский интерфейс, обрабатывает действия пользователя, формирует запросы к серверной части и обрабатывает его ответы.

Серверная часть получает, обрабатывает и отправляет запросы клиентской части либо по протоколу http, либо с помощью сокетов.

В зависимости от того какая часть приложения выполняется на стороне клиента, клиентскую часть называют толстым или тонким клиентом.

Толстым клиентом называют клиентскую часть приложения, в которой выполняется большая часть логики приложения, а серверная сторона занимается в основном передачей и хранением данных.

Тонкий клиент по аналогии с толстым, имеет не равную функциональную нагрузку, но большая часть работы происходит на серверной стороне.

Большинство современных веб-приложений имеют толстые клиенты.

Клиентская часть реализуется с помощью таких технологий как:

Интерфейс:

* HTML\ XHTML
* CSS

Логика приложения и независимый от браузера интерфейс:

* Adobe Flash, Adobe Flex
* Java
* JavaScript
* Silverlight

Серверная часть может быть реализована практический с помощью любого языка программирования.

**Вывод: Веб-приложение состоит из клиентской и серверной частей между которыми разделена логика приложения. Существует множество технологий для реализации клиентской и серверной части.**

# Клиентские технологии

На данный момент существует ряд технологий позволяющих вести разработку графических веб-приложений, приведем наиболее известные технологии:

1. **Flash** – мультимедийная платформа компании Adobe для создания веб-приложений или мультимедийных презентаций. Широко используется для создания рекламных баннеров, анимации, игр, а также воспроизведения на веб-страницах видео- и аудиозаписей.

Платформа включает в себя ряд средств разработки, прежде всего en:Adobe Flash Professional и Adobe Flash Builder (ранее Adobe Flex Builder); а также программу для воспроизведения flash-контента — Adobe Flash Player;

1. **2d-context HTML5** – это контекст тега <canvas> предоставляющего интерфейс для создания двумерной растровой графики. 2d-context позволяет манипулировать изображением на плоской двумерной системе координат с центром в левом верхнем углу экрана. Интерфейс представлен набором методов и свойств, определяющих графические примитивы их внешний вид и аффинные преобразования над экранной плоскостью. Данная технология реализует непосредственный режим графики, то есть программист должен сам заботится о перерисовки каждого кадра, реализации необходимых графических алгоритмов и т.д.

Данный контекст можно считать простой альтернативой технологии Flash и вероятно будет подходить для создания мультимедийных веб-приложений, игр, анимированных интерфейсов и т.д.;

1. **Webgl-context** – Это контекст тега <canvas> предоставляющего интерфейс для создания трехмерной графики. Интерфейс является производным от OpenGL ® ES 2.0 и имеет схожие возможности, включая работу с вершинными и пиксельными шейдерами. Webgl-context позволяет задействовать вычислительные мощности видеокарты, что может подвергать пользователя риску, через открытие доступа к привилегированному режиму видеокарты и оборудования[13];
2. **SVG** – язык описания векторных изображений, построенный на языке XML;
3. **Манипуляции DOM-моделью HTML документа** – С самого появления языка HTML, HTML документ состоял из набора тегов определявших внешний вид и назначение различных элементов на странице. Манипуляция тегами, т.е. структурными элементами DOM–модели, можно рассматривать как возможность создания спрайтовой анимации. Этот метод мало применим для достаточно сложной игры из-за «подрагивания» или «мелькания» при достаточно частом обновления страницы. Манипуляции с DOM-моделью не предназначены для частых динамичных изменений.

Для сравнения 2d-context и SVG, можно использовать тест, приведенный по ссылке[14]: тест состоял в отслеживании числа кадров в секунду (FPS) при изменении количества объектов (движущиеся с максимальной скоростью квадраты) на экране при использовании 2d-context и при использовании SVG. Результаты теста на ОС Windows 7, приведены на рисунке1.

****

**Рис. 1. Сравнительный тест технологий 2d-context и SVG**

Как можно видеть на графике (Рисунок 1), 2d-contex более производителен при большом количестве объектов и так как основными задачами, решаемыми с помощью графических возможностей HTML5 и Flash, чаще всего являются реализация спрайтовой анимация и работа с изображениями, можно сделать вывод, что для замены технологии Flash больше всего подходит 2d-context ведь он, как и Flash, реализует работу с растровой графикой и позволяет работать с большим числом объектов. Очевидно, что выбирая между 2d-context и SVG для реализации приложения, которое должно генерировать динамичное изображение состоящее из большого количества объектов, следует выбрать 2d-context.

**Вывод: Существует достаточно обширный выбор графических технологий для веб-приложений.**

# Технология 2d-context HTML5

2d-context HTML5 – это контекст тега <canvas> предоставляющего интерфейс для создания двумерной растровой графики. 2d-context позволяет манипулировать изображением на плоской двумерной системе координат с центром в левом верхнем углу экрана. Интерфейс представлен набором методов и свойств, определяющих графические примитивы их внешний вид и аффинные преобразования над экранной плоскостью. Данная технология реализует непосредственный режим графики, то есть программист должен сам заботится о перерисовки каждого кадра, реализации необходимых графических алгоритмов и т.д.

Спецификация на данную технологию создается W3C, но на сегодняшний день спецификация еще не дописана.

Ниже приведен программный интерфейс доступный для использования с помощью языка программирования JavaScript:

**Методы сохранения и восстановления состояния контекста:**

* void save();
* void restore();

**Методы трансформации сцены**

* void scale( double x, double y);
* void rotate( double angle);
* void translate( double x, double y);
* void transform( double a, double b, double c, double d, double e, double f);
* void setTransform( double a, double b, double c, double d, double e, double f);

**Атрибут характеризующий уровень прозрачности**

* double globalAlpha;

**Атрубут задающий тип пересечения примитивов**

* DOMString globalCompositeOperation;

**Атрибут устанавливающий размытие изображения**

* boolean imageSmoothingEnabled;

**Атрибуты характеризующие внешний вид контуров и заливки**

* DOMString or CanvasGradient or CanvasPattern strokeStyle;
* DOMString or CanvasGradient or CanvasPattern fillStyle;

**Метод создающий линейный градиент**

* CanvasGradient createLinearGradient(double x0, double y0, double x1, double y1);

**Метод создающий радиальный градиент**

* CanvasGradient createRadialGradient(double x0, double y0, double r0, double x1, double y1, double r1);

**Атрибуты устанавливающие смещение тени по оси x и y**

* double shadowOffsetX;
* double shadowOffsetY;

**Атрибут характеризующий степень размытия тени**

* double shadowBlur;

**Атрибут характеризующий цвет тени**

* DOMString shadowColor;

**Метод стирающий область**

* void clearRect( double x, double y, double w, double h);

**Метод заполняющий область цветом**

* void fillRect( double x, double y, double w, double h);

**Метод рисующий рамку вокруг области**

* void strokeRect( double x, double y, double w, double h);

**Метод начала графического пути**

* void beginPath();

**Метод закраски пути**

* void fill(optional CanvasWindingRule w = "nonzero");
* void fill(Path path);

**Метод закраски контуров в пути**

* void stroke();
* void stroke(Path path);

**Метод рисующий рамку вокруг переданного элемента**

* void drawSystemFocusRing(Element element);
* void drawSystemFocusRing(Path path, Element element);

**Метод рисующий рамку вокруг выбранного элемента**

* boolean drawCustomFocusRing(Element element);
* boolean drawCustomFocusRing(Path path, Element element);

**Метод обрезающий вывод пути по контору**

* void clip(optional CanvasWindingRule w = "nonzero");
* void clip(Path path);

**Метод проверки принадлежности точки пути**

* boolean isPointInPath( double x, double y, optional CanvasWindingRule w = "nonzero");
* boolean isPointInPath(Path path, double x, double y);

**Методы выводящий текст с заданными парамтетрами**

* void fillText(DOMString text, double x, double y, optional double maxWidth);
* void strokeText(DOMString text, double x, double y, optional double maxWidth);

**Метод прорисовки изображении**

* void drawImage((HTMLImageElement or HTMLCanvasElement or HTMLVideoElement) image, double dx, double dy);
* void drawImage((HTMLImageElement or HTMLCanvasElement or HTMLVideoElement) image, double dx, double dy, double dw, double dh);
* void drawImage((HTMLImageElement or HTMLCanvasElement or HTMLVideoElement) image, double sx, double sy, double sw, double sh, double dx, double dy, double dw, double dh);
* void addHitRegion(HitRegionOptions options);
* void removeHitRegion(HitRegionOptions options);

**Метод создающий массив пикселей**

* ImageData createImageData( double sw, double sh);
* ImageData createImageData(ImageData imagedata);

**Метод получения массива пикселей с экрана**

* ImageData getImageData(double sx, double sy, double sw, double sh);

**Метод вывода на экран массива пикселей**

* void putImageData(ImageData imagedata, double dx, double dy, double dirtyX, double dirtyY, double dirtyWidth, double dirtyHeight);
* void putImageData(ImageData imagedata, double dx, double dy);

**Атрубуты с информацией о параметрах щрифта для вывод на экран**

* DOMString font;
* DOMString textAlign;
* DOMString textBaseline;

**Метод закрывающий путь**

* void closePath();

**Метод устанавливающий начальную точку для вывода линий**

* void moveTo( double x, double y);

**Метод проводящий линию**

* void lineTo( double x, double y);

**Методы рисующие кривые**

* void quadraticCurveTo( double cpx, double cpy, double x, double y);
* void bezierCurveTo( double cp1x, double cp1y, double cp2x, double cp2y, double x, double y);

**Метод рисующий дугу по точкам и радиусу**

* void arcTo( double x1, double y1, double x2, double y2, double radius);

**Метод рисующий прямоугольник**

* void rect( double x, double y, double w, double h);

**Метод рисующий дугу**

* void arc( double x, double y, double radius, double startAngle, double endAngle, optional boolean anticlockwise = false);

**Метод рисующий элипс**

* void ellipse( double x, double y, double radiusX, double radiusY, double rotation, double startAngle, double endAngle, boolean anticlockwise);

**Вывод: Программный интерфейс 2d-context предоставляет минимально необходимые методы для создания двумерного статического изображения. Методов направленных на реализацию анимации интерфейс 2d-context не содержит.**

# Сложности разработки с помощью 2d-context HTML5

Для оценки проблем и сложностей разработки с помощью 2d-context необходимо ознакомится с самим процессом разработки.

Для использования любой технологии или инструмента необходимо ознакомится со спецификацией, из прочтения которой можно выяснить возможности, требования и концепции использования. Для технологии 2d-context спецификация доступна по адресу [5].

В ходе анализа спецификации методы и атрибуты 2d-context можно условно разделить на:

1. Методы работы с путями.
2. Методы и атрибуты для визуализации текста.
3. Методы для визуализации графических примитивов:
   1. Прямоугольник.
   2. Изображение.
   3. Линия.
   4. ломаная линия.
   5. Дуга.
   6. Кривые Безье.
   7. Эллипс.
4. Методы для работы с массивом пикселей.
5. Методы сохранения и восстановления контекста.
6. Метод определения принадлежности точки пути.

Можно заметить, что методы и атрибуты 2d-context позволяют:

* Программировать изображение, но не позволяют работать с отдельными частями экрана как с объектами.
* Для создания анимации требуются перерисовка экрана для изменения изображения. Это вызвано отсутствием методов для создания анимации.
* Описание несложных сцен требуют достаточно большое количество кода. Это вызвано тем фактом, что изображение создается с помощью минимального набора графических примитивов.

В ходе проверки заявленных методов в основных браузерах, было замеченно, что браузеры поддерживают спецификацию не в равной степени, так же ряд методов не имею реализации. Вероятно, это было вызвано не законченностью стандарта.

**Можно сделать вывод, что использование 2d-context влечет за собой следующие трудности:**

1. **Нет возможности работать с частями изображения как с самостоятельными объектами.**
2. **Для создания анимации требуются перерисовка экрана для изменения изображения.**
3. **Описание несложных сцен требуют достаточно большое количество кода.**
4. **Неготовый стандарт. Стандарт в стадии тестирования – W3C объявил о планах, согласно которым окончательная версия стандарта HTML5 будет утверждена лишь к 2014 году[6].**
5. **Отсутствие визуальных сред, вроде Flash Professional CS6.**

**Слабое развитие специализированных каркасов, вызванные, скорее всего, незавершенностью стандарта.**

# Постановка задачи

Постоянное совершенствование браузеров, появление поддержки браузерами более производительных графических технологий, а так же рост аппаратных возможностей дает разработчикам возможность создавать более динамичные мультимедийные веб-приложения.

Появление технологии 2d-context HTML5 позволяет создавать динамичные мультимедийные веб-приложения, но разработка без специализированного программного каркаса требует много времени на разработку и более подвержено ошибкам в силу объема и сложности реализуемого кода. В связи с данной проблемой возникает потребность в специализированном программном каркасе, реализующем базовую функциональность графического приложения.

Для решения поставлено задачи необходимо:

* Подробнее ознакомится с программным интерфейсом 2d-context.
* Выявить требования к каркасу.
* Изучить необходимые алгоритмы и техники.
* Спроектировать и реализовать программный каркас.

# АРХИТЕКТУРА ПРОГРАММНОГО КАРКАСА

# Обзор существующих решений

Перед разработкой каркаса был совершен обзор существующих свободно распространяемым каркасов и библиотек для работы с 2d-context.

Цель обзора – получение представлений о свойствах существующих каркасов для получения общего представления о функциях современных каркасов.

Обзор был совершен для трех каркасов: jCanvaScript, KineticJS, LibCanvas.

1. jCanvaScript – Свободно распространяемая библиотека доступная по адресу [20] по лицензии MIT или GPL версии 2.
2. KineticJS – Свободно распространяемая библиотека доступная по адресу [21] по лицензии MIT или GPL версии 2.
3. LibCanvas – Свободно распространяемая библиотека доступная по адресу [22] по лицензии MIT или GPL.

Результаты обзора функций каркасов представлены в таблице 1.

**Таблица 1 – Возможностей существующих решений**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Графический цикл | Слои | Объектное представление примитивов | Call-back функции для событий объектов | Работа с мышю и клавиатурой | классы обертки для примитивов | Загрузка мультимедиа-зависимостей объектов | Приоритеты очереди прорисовки | Модульность | Вызовы цепочкой |
| jCanvaScript | + | + | + | -\+ | + | + | - | + | - | - |
| KineticJS | + | + | + | + | + | + | - | + | + | - |
| LibCanvas | + | + | + | + | + | + | - | + | - | + |

Результаты обзора функций каркасов были использованы при создании каркаса.

**Вывод: Современный программный каркас для создания двумерной графики с помощью 2d-context должен реализовывать следующую функциональность:**

1. **Графический цикл**
2. **Слои**
3. **Объектное представление примитивов**
4. **Call-back функции для событий объектов**
5. **Работа с мышю и клавиатурой**
6. **Классы-обертки для примитивов**
7. **Загрузка мультимедиа-зависимостей объектов**
8. **Приоритеты очереди прорисовки**
9. **Модульность**

# Требования к программному каркасу

В ходе анализа существующих решений и анализа процесса создания мультимедийных приложений было выявлено:

В мультимедиа приложении, интерактивность достигается за счет взаимодействия пользователя с отдельными элементами на экране (# кнопки, поя ввода и т.д. ), при этом элементы представлены набором графических примитивов. Можно сделать вывод, что для изменения внешнего вида элемента необходимо изменять свойства каждого примитива входящего в состав элемента, то есть для разработчика важно иметь возможность представлять элемент интерфейса с помощью совокупности графических примитивов, возможность управлять внешними видом совокупности примитивов как одной сущности.

Так как 2d-context представляет так называемый непосредственный графический режим[]

Анализ игровых приложений, а также личный опыт создания, подталкивают к выводу, что каркас должен обеспечивать различные оптимизации с точки зрения производительности, для оценки качества оптимизации можно использовать такой параметр как FPS (#число кадров в секунду).

Программный каркас это инструмент программиста, то есть он должен быть удобен программисту, а значит должен обладать интуитивным, лаконичным интерфейсом.

Необходимо учитывать, что размер каркаса влияет на общее время загрузки и старта приложения, а так как при создании достаточно универсального каркаса, в разных приложениях могут быть задействованы далеко не все возможности, необходимо реализовать модульности каркаса. Должна быть возможности собирать каркас для конкретного приложения, из необходимых для конкретного приложения, модулей.

# Реализация

# Парадигма программирования

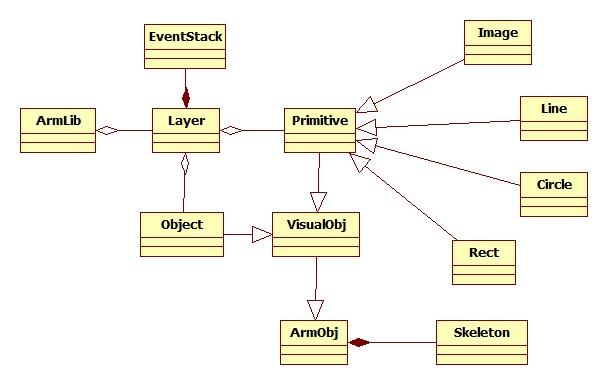
# Используемые алгоритмы

# Диаграмма классов

Описать структурное строение каркаса можно с помощью UML-диаграммы классов.

Каркас содержит 12 классов: ArmLib, Layer, EventStack, Primitive, VisualObj, Image, Rect, Circle, Line, Object, ArmObj, Skeleton.

В ходе разработки приложения была создана диаграмма классов. На Рисунке 2 для наглядности приведен ее не полный аналог.

****

**Рисунок 2 – UML диаграмма классов**

# Класс ArmLib

Данный класс является головным для каркаса. Посредством методов данного класса происходит как управление каркасом (# запуск, остановка и т.д.) так и реакция на события клавиатуры, мыши и т.д.

Объект данного класса содержит в себе объекты класса Layer.

Основные поля класса:

* **Array \_type** – данное поле содержит информацию о классе данного объекта. Данное поле создано для возможности проверки типа объекта. Потребность в поле существует из-за отсутствия в языке JavaScript проверки типов;
* **String \_name** – данное поле содержит имя объекта;
* **Array \_list** – в данном поле, виде массива хранятся объекты класса Layer;
* **Object \_class** – в данном поле хранятся ссылки на классы используемые в каркасе;

**О**сновные методы класса:

* **Void bindWithTag()** – метод для привязки каркаса к некоторому тегу, который может содержать в себе другие теги, а именно теги <canvas>;
* **Object run()** – метод для старта каркаса;
* **Object stop()** – метод для остановки работы каркаса;
* **Void listenMouseKeyboardEvents()** – метод для вызова методов \_listenKeybordEvents и \_listenMouseEvents;
* **Void notListenMouseKeyboardEvents()** – метод вызова методов \_notListenKeybordEvents и \_notListenMouseEvents;
* **Object \_addLayer()** – метод для привязки передаваемого объекта класса Layer к каркасу. Данный метод стартует в конструкторе класса Layer.
* **Void \_listenKeybordEvents()** – метод для установки обработчиков событий клавиатуры. Через обработчики каркас реагирует на события клавиатуры;
* **Void \_notListenKeybordEvents()** – метод для удаления обработчиков клавиатуры;
* **Void \_listenMouseEvents()** – метод для установки обработчиков событий мыши. Через обработчики каркас реагирует на события мыши;
* **Void \_notListenMouseEvents()** – метод для удаления обработчиков мыши;
* **Void \_sendEvent()** – метод для «отправки события» о котором необходимо известить слои каркаса. Метод формирует объект, содержащий в себе информацию о событии и добавляет данный объект в стек событий.

# Класс Layer

Данный класс описывает слой. Слой это объект, который хранит в себе либо объекты класса Object, либо объекты классов производных от класса Primitive и отвечает за их прорисовку на экран. Слой может быть перекрыт другим слоем.

Основные поля класса:

* **Array \_type** – данное поле содержит информацию о классе данного объекта. Данное поле создано для возможности проверки типа объекта. Потребность в поле существует из-за отсутствия в языке JavaScript проверки типов;
* **String \_name** – данное поле содержит имя объекта;
* **Bool \_isRuning** – данное поле содержит булево значение, характеризующие работает ли слой данный момент;
* **CanvasRenderingContext2D \_context** – данное поле содержит объект 2d-context связанный с тегом <canvas> данного слоя;
* **\_container** – данное поле содержит ссылку на тег являющийся контейнером для тега <canvas> текущего слоя;
* **\_canvas** – данное поле содержит ссылку на тег <canvas> текущего слоя;
* **Number \_width** – ширина слоя в пикселях;
* **Number \_height** – высота слоя в пикслеях;
* **Number \_fps** – число, равное требуемому числу кадров в секунду;
* **Number \_zindex** – число характеризующее степень удаленности слоя относительно других слоев. Чем большее число – тем ближе к пользователю;
* **Array \_list** – в данном поле, в виде массива хранятся объекты классов Object и объекты классов производных от класса Primitive;
* **EventStack \_eventStack** – поле содержит объект класса EventStack;

Основные методы класса:

* **Void run()** – метод для старта обработки объектов слоя;
* **Void stop()** – метод для остановки обработки объектов слоя;
* **Object addChild(Object)** – метод для добавления объекта класса Object или объекта классов производных от класса Primitive;
* **Object removeChild(Object)** – метод для удаления некоторого объекта из слоя;
* **Object setFunc(String, Function)** – метод для установки обработчика на некоторое событие слоя;
* **Object getFunc(String)** – метод для получения обработчика события слоя;
* **Void \_init()** – метод для инициализации объекта слоя;
* **Void \_begin()** – в данном методе для каждого объекта слоя вызывается метод \_begin. Данный метод выполняет инициализацию каждого объекта слоя;
* **Void \_clear()** – в данном методе для каждого объекта слоя вызывается метод \_clear. Данный метод выполняет удаление каждого объекта слоя с экрана;
* **Void \_update()** – в данном методе для каждого объекта слоя вызывается метод \_update. Данный метод выполняет обновление каждого объекта слоя;
* **Void \_draw()** – в данном методе для каждого объекта слоя вызывается метод \_draw. Данный метод выполняет прорисовку каждого объекта слоя на экране;
* **Void \_doEvents()** – данный метод вызывает обработку событий накопившихся в объекте доступном по ссылке \_eventStack за итерацию игрового цикла.

# Класс EventQueue

Данный класс реализует работу с массивом как с очередью. Класс описывает очередь сообщений клавиатуры, мыши и т.д.

Основные поля класса:

* **Array \_queue** – данное поле содержит массив содержащий объекты, описывающие события;

Основные методы класса:

* **Void push(Object)** – метод для добавления события;
* **Object Pop()** – метод для получения последнего события.

# Класс Primitive

Абстрактный класс. Данный класс является общим описанием для любого графического примитива, то есть описывает свойства и методы содержащиеся в любом примитиве.

Основные поля класса:

* **Array \_type** – данное поле содержит информацию о классе данного объекта. Данное поле создано для возможности проверки типа объекта. Потребность в поле существует из-за отсутствия в языке JavaScript проверки типов;
* **String \_fill** – поле содержит строку характеризующую цвет заливки;
* **String \_stroke** – поле содержит строку характеризующую цвет контура;
* **Object \_drawRect** – представляет собой объект состоящий из двух полей drawingRectPos (объект из двух полей описывающих координаты двух точек прямоугольника) и drawingRectImage, содержащим сохраненную часть экрана. В поле сохраняется некоторая часть экрана и информация об положении данной части на экране;

Основные методы класса:

* **Void saveDrawingRectImage()** – метод сохраняет часть экрана занимаемую, данным примитивом. Метод вызывается перед прорисовкой примитива, то есть изображение примитива не содержится в сохраняемой области;
* **Void removeDrawingRectImage()** – метод восстанавливает изображение под примитивом, то есть удаляет примитив с экрана;
* **Void updateDrawingRectPos()** – метод высчитывает прямоугольную область экрана занимаемую примитивом;
* **Void showRemovingRect()** – метод отображает прямоугольную область экрана занимаемую примитивом;
* **Void \_begin()** – данный метод выполняет инициализацию каждого объекта;
* **Void \_clear()** – данный метод выполняет удаление объекта с экрана;
* **Void \_update()** – данный метод выполняет обновление объекта;
* **Void \_draw()** – данный метод выполняет прорисовку объекта на экране;

# Класс ArmObj

Абстрактный класс. Данный класс является наиболее общим описанием объекта каркаса;

Основные поля класса:

* **\_name** – данное поле содержит имя объекта;
* **\_isLoaded** – данное поле содержит булево значение характеризующее загружены ли мультимедиа-зависимости объекта или нет;
* **\_owner** – данное поле содержит ссылку на родительский по отношению к текущему, объект;

Основные методы класса:

* **Object load()** –
* **Object setFunc(String, Function)** – метод для установки обработчика на некоторое событие объекта;
* **Object getFunc(String)** – метод для получения обработчика события объекта;
* **Bool haveOwner()** – метод возвращающий булево значение отвечающее на вопрос существует ли объект являющийся родительским по отношению к данному объекту;
* **Void \_begin()** – виртуальный метод инициализации объекта;
* **Void \_update()** – виртуальный метод обновления объекта;
* **Void \_load()** – виртуальный метод загрузки зависимостей объекта;
* **Void \_\_onLoad()** – метод первым выполняемый по загрузке зависимостей объекта. В данном методе происходит вызов пользовательского обработчика события загрузки, сообщение родительскому объекту о загрузке текущего. После вызова данного метода объект становится загруженным.

# Класс Skeleton

Данный класс описывает полигон таких точек, что если через них провести отрезки получившейся многоугольник опишет геомерическое представления некоторого визуального объекта каркаса.

Класс совершает параллельный перенос и поворот точек полигона в соответствии с матрицей преобразования.

Основные поля класса:

* **Array \_unchangedPoints** – поле содержит массив объектов класса точка. Поле описывает не измененные точки;
* **Array \_translatedPoints** – поле содержит массив объектов класса точка. Поле описывает перемещенные точки;
* **Array \_transformedPoints** – поле содержит массив объектов класса точка. Поле описывает точки, повернутые на некоторый угол;
* **Object \_matrixOfTranslatePoints** – поле содержит объект класса Matrix. Поле описывает перемещенные точки ввиде матрицы;
* **Object \_matrixOfTransformedPoints** – поле содержит объект класса Matrix. Поле описывает точки повернутые на некоторый угол, ввиде матрицы;
* **Object \_polygoneOfTransformedPoints** – поле содержит объект класса Vector2D. Поле описывает вектор точек перемещенных и повернутых на некоторый угол;

Основные методы класса:

* **Object addPoint(Point)** – данный метод принимает на вход объект класса Point, описывающий точку, и добавляет переданную точку к полигону;
* **Bool hasPoint(Point)** – данный метод возвращает булево значение говорящее от том есть ли переданная точка в полигоне;
* **Void translate(Number, Number)** – метод перемещает точки полигона параллельным переносом по оси икс и игрек в соответствии с переданными значениями;
* **Void transform(Matrix)** – метод преобразует точки полигона в соответствии с переданной матрицей преобразования;
* **Void show(Layer)** – метод отображает на переданном слое многоугольник проходящий через точки полигона;
* **Void \_updateMatrixOfUnchangedPoints()** – метод обновляющий матрицу не преобразованных точек;
* **Void \_updatePolygoneOfTransformedPoints()** – метод создающий полигон из преобразованных точек;

# Класс VisualObj

Абстрактный класс. Данный класс является наиболее общим описанием визуального объекта каркаса;

Основные поля класса:

* **Array \_type** – данное поле содержит информацию о классе данного объекта. Данное поле создано для возможности проверки типа объекта. Потребность в поле существует из-за отсутствия в языке JavaScript проверки типов;
* **CanvasRenderingContext2D \_context** – данное поле содержит объект 2d-context связанный с тегом <canvas> для слоя данного объекта;
* **Object \_layer** – данное поле содержи ссылку на объект слоя текущего объекта;
* **Number \_x** – данное поле характеризует положение объекта по оси икс;
* **Number\_y** – данное поле характеризует положение объекта по оси игрек;
* **Number \_angle** – данное поле характеризует угол поворота объекта;
* **Object \_centralPoint** – данное поле описывает точку относительно которой происходит поворот объекта;
* **Object \_scale** – данное поле содержит объект содержащий два поля характеризующих масштабирование объекта по оси икс и игрек;
* **Number \_zindex** – число характеризующее степень удаленности объекта относительно других объектов. Чем большее число – тем ближе к пользователю;
* **Number \_globalAlpha** – данное поле характеризует степень прозрачности объекта;
* **Object \_\_transformMatrix** – данное поле содержит обекта класса Matrix и представляет собой матрицу преобразования;
* **Array \_paramsTransformMatrix** – данное поле содержит матрицу преобразования в виде массива;
* **Object \_skeleton** – данное поле содержит объект класса Skeleton;
* **Bool \_haveChanges** – данное поле содержит булеву переменную характеризующую есть ли изменения в объекте;

Основные методы класса:

* **Bool haveLayer()** – метод отвечающий на вопрос прикреплен ли данный объект к некоторому слою. Метод возвращает булево значение;
* **Bool haveChanges()** – метод отвечающий на вопрос имеет ли объект изменения. Метод возвращает булево значение;
* **Void \_begin()** – виртуальный метод. Данный метод выполняет инициализацию каждого объекта слоя;
* **Void \_clear()** – виртуальный метод. Данный метод выполняет удаление каждого объекта слоя с экрана;
* **Void \_draw()** – виртуальный метод. Данный метод выполняет прорисовку каждого объекта слоя на экране;
* **Void initTransformMatrix()** – метод создает матрицу из свойств текущего объекта: угол поворота, координаты и т.д.;
* **Void updateTransformMatrix()** – метод создает матрицу из свойств текущего объекта: угол поворота, координаты и т.д.;
* **Void initSkeleton(Array)** – метод создает и инициализирует объект класса Skeleton;
* **Void updateSkeleton()** – метод обновляет объект класса Skeleton в соответствии с матрицей преобразования и координатами текущего объекта;
* **set x(O)** – сеттер для свойства \_x. В методе запоминает величину изменения свойства \_x и объект считается измененным после изменения величины поля;
* **get x()** – геттер для свойства \_x;
* **set y(O)** – сеттер для свойства \_y. В методе запоминает величину изменения свойства \_y и объект считается измененным после изменения величины поля;
* **get y()** – геттер для свойства \_y;
* **set width(O)** – сеттер для свойства \_width. В методе запоминает величину изменения свойства \_width и объект считается измененным после изменения величины поля;
* **get width()** – геттер для свойства \_width;
* **set height(O)** – сеттер для свойства \_height. В методе запоминает величину изменения свойства \_height и объект считается измененным после изменения величины поля;
* **get height()** – геттер для свойства \_height;
* **set angle()** – сеттер для свойства \_angle. В методе запоминает величину изменения свойства \_angle и объект считается измененным после изменения величины поля. Если выставляемая величина больше, чем 360 то в поле записывается остаток от деления на 360;
* **get angle()** – геттер для свойства \_angle;
* **set centralPoint(O)** – сеттер для свойства \_centralPoint. В методе запоминает величину изменения свойства \_centralPoint и объект считается измененным после изменения величины поля;
* **get centralPoint()** – геттер для свойства \_centralPoint;
* **set scale()** – сеттер для свойства \_scale. В методе запоминает величину изменения свойства \_scale и объект считается измененным после изменения величины поля;
* **get scale()** – геттер для свойства \_scale;
* **set zindex()** – сеттер для свойства \_zindex. В методе запоминает величину изменения свойства \_zindex и объект считается измененным после изменения величины поля;
* **get zindex()** – геттер для свойства \_zindex;
* **set globalAlpha()** – сеттер для свойства \_globalAlpha. В методе запоминает величину изменения свойства \_globalAlpha и объект считается измененным после изменения величины поля;
* **get globalAlpha()** – геттер для свойства \_globalAlpha.

# Класс Object

Данный класс описывает объект контейнер. Данный класс описывает объект, который хранит в себе либо объекты класса Object, либо объекты классов производных от класса Primitive и отвечает за их прорисовку на экран.

Основные поля класса:

* **Array \_type** – данное поле содержит информацию о классе данного объекта. Данное поле создано для возможности проверки типа объекта. Потребность в поле существует из-за отсутствия в языке JavaScript проверки типов;
* **String \_name** – данное поле содержит имя объекта;
* **Array \_list** – в данном поле, виде массива хранятся либо объекты класса Object, либо объекты классов производных от класса Primitive;
* **Number \_numberNotLoadedChilds** – данное поле содержит число объектов в объекте Object, зависимости которых еще небыли загружены.

Основные методы класса:

* **Object addChild(Object)** – метод для добавления объекта класса Object или объекта классов производных от класса Primitive;
* **Object removeChild(Object)** – метод для удаления некоторого объекта из объекта;
* **Object load()** – метод для загрузки зависимостей всех дорчерних объектов;
* **Number getNumberNotLoadedChilds()** – метод возвращающий число не загруженных дочерних объектов;
* **\_begin()** – в данном методе для каждого объекта вызывается метод \_begin. Данный метод выполняет инициализацию каждого объекта объекта;
* **Void \_clear()** – в данном методе для каждого объекта вызывается метод \_clear. Данный метод выполняет удаление каждого объекта объекта с экрана;
* **Void \_update()** – в данном методе для каждого объекта вызывается метод \_update. Данный метод выполняет обновление каждого объекта объекта;
* **Void \_draw()** – в данном методе для каждого объекта вызывается метод \_draw. Данный метод выполняет прорисовку каждого объекта объекта на экране;
* **Void \_sortByZindex()** – в данном методе происходит сортировка обектов в массиве \_list согласно полю \_zindex каждого объекта;
* **Void \_loadedChild()** – данный метод выполняется при загрузке очередного объекта. Если загружены все объекты, то выполнятся обработчик данного объекта \_\_onLoad();
* **Void \_\_onKeyDown()** – метод выполняемый либо при получении события onKeyDown слоем данного объекта, либо при вызове метода onKeyDown объекта являющимся родительским по отношению к текущему объекту. В данном методе происходит вызов пользовательского обработчика события onKeyDown и вызов метода \_\_onKeyDown дочерних объектов;
* **Void \_\_onKeyPress()** – метод выполняемый либо при получении события onKeyPress слоем данного объекта, либо при вызове метода onKeyPress объекта являющимся родительским по отношению к текущему объекту. В данном методе происходит вызов пользовательского обработчика события onKeyPress и вызов метода \_\_ onKeyPress дочерних объектов;
* **Void \_\_onKeyUp()** – метод выполняемый либо при получении события onKeyUp слоем данного объекта, либо при вызове метода onKeyUp объекта являющимся родительским по отношению к текущему объекту. В данном методе происходит вызов пользовательского обработчика события onKeyUp и вызов метода \_\_ onKeyUp дочерних объектов;
* **Void \_\_onMouseDown()** – метод выполняемый либо при получении события onMouseDown слоем данного объекта, либо при вызове метода onMouseDown объекта являющимся родительским по отношению к текущему объекту. В данном методе происходит вызов пользовательского обработчика события onMouseDown и вызов метода \_\_ onMouseDown дочерних объектов;
* **Void \_\_onMouseUp()** – метод выполняемый либо при получении события onMouseUp слоем данного объекта, либо при вызове метода onMouseUp объекта являющимся родительским по отношению к текущему объекту. В данном методе происходит вызов пользовательского обработчика события onMouseUp и вызов метода \_\_ onMouseUp дочерних объектов;
* **Void \_\_onMouseMove()** – метод выполняемый либо при получении события onMouseMove слоем данного объекта, либо при вызове метода onMouseMove объекта являющимся родительским по отношению к текущему объекту. В данном методе происходит вызов пользовательского обработчика события onMouseMove и вызов метода \_\_ onMouseMove дочерних объектов.

# Класс Image

Данный класс описывает примитив, выводящий на экран изображение.

Основные поля класса:

* **Number \_width** – ширина изображения в пикселях;
* **Number \_height** – высота изображения в пикселях;
* **String\_src** – путь к изображению;
* **Image \_image** – объект изображения;
* **Bool \_loaded** – поле отвечающие загружен или нет объект.

Основные методы класса:

* **Void \_load()** – метод инициирующий загрузку изображения;
* **Void \_draw()** – реализация метода вывода примитива на экран.

# Класс Rect

Данный класс описывает примитив, выводящий на экран прямоугольник.

Основные поля класса:

* **Number \_width** – ширина изображения в пикселях;
* **Number \_height** – высота изображения в пикселях;
* **Number \_lineWidth** – толщина линии;
* **Bool \_loaded** – поле отвечающие загружен или нет объект.

Основные методы класса:

* **Void \_load()** – метод инициирующий загрузку зависимостей;
* **Void \_draw()** – метод обрисовывающий примитив на экран;
* **Void updateDrawingRectPos()** – метод высчитывает прямоугольную область экрана занимаемую примитивом.

# Класс Circle

Данный класс описывает примитив, выводящий на экран окружность.

Основные поля класса:

* **Number \_radius**
* **Number \_lineWidth** – толщина линии;
* **Bool \_loaded** – поле отвечающие загружен или нет объект.

Основные методы класса:

* **Void \_load()** – метод инициирующий загрузку зависимостей;
* **Void \_draw()** – метод обрисовывающий примитив на экран;
* **Void updateDrawingRectPos()** – метод высчитывает прямоугольную область экрана занимаемую примитивом.

# Класс Line

Данный класс описывает примитив, выводящий на экран линию.

Основные поля класса:

* **Number \_x2** – данное поле характеризует положение второй точки по оси икс, через которую проходит линия;
* **Number\_y2** – данное поле характеризует положение второй точки по оси игрек, через которую проходит линия;
* **Number \_lineWidth** – толщина линии;
* **Number \_loaded** – поле отвечающие загружен или нет объект.

Основные методы класса:

* **Void \_load()** – метод инициирующий загрузку зависимостей;
* **Void \_draw()** – метод обрисовывающий примитив на экран;
* **Void updateDrawingRectPos()** – метод высчитывает прямоугольную область экрана занимаемую примитивом.

# Используемые шаблоны проектирования

В ходе разработки сложного программного обеспечения важно создавать качественный, готовый к сопровождению другими программистами, код. Одной из методик уменьшения сложности сопровождения, является использование, так называемых, шаблонов проектирования программного обеспечения.

Шаблоны проектирования описывают повторимые архитектурные конструкции, представляющие собой решение проблемы проектирования в рамках некоторого часто возникающего контекста. Обычно шаблон не является законченным образцом, который может быть прямо преобразован в код; это лишь пример решения задачи, который можно использовать в различных ситуациях.

Шаблоны показывают отношения и взаимодействия между классами или объектами, без определения того, какие конечные классы или объекты приложения будут использоваться. Использование широко известных шаблонов, делает код более «читаемым».

В каркасе были использованы ряд шаблонов, а именно:

**Порождающие шаблоны:**

• «Одиночка», данный шаблон использован в классе ArmLib, для гарантирования существования только одного объекта класса ArmLib;

• «Фасад», данный шаблон использован в классах графических примитивов: Rect, Line, Circle, Image. В данных классах данный шаблон дает возможность более удобно работать с API контекста рисования HTML5;

• «Компоновщик», данный шаблон использован в классах ArmLib, Layer и Object для создания древовидной структуры отношений примитивов и объектов сцены.

**Фундаментальные шаблоны**

• «Делегирование», данный шаблон является базовым для приложения созданного в объектно-ориентированном стиле. Данный шаблон используется почти во всех классах каркаса.

**Поведенческие шаблоны**

• «Наблюдатель», данный шаблон используется в классах ArmLib, Layer и Object для оповещения объектов класса Layer в классе ArmLib, объектов класса Object, Image, Rect, Line в класах Object и Layer о событиях клавиатуры и мыши.

**Вывод: Использование шаблонов проектирования позволило стандартизировать некоторые части каркаса, что может упростить работу с исходным кодом другим разработчиками.**

# Принятые стандарты кодирования

Общепринятым мнением и в частности мнением А. Александреску и Г. Саттера [17] признанно, что принятие стандарта кодирования положительно сказывается на процессе разработки:

«Хорошие стандарты кодирования могут принести не малую выгоду с различных точек зрения.

* *Повышение качества кода.* Работа в соответствии со стандартом приводит к однотипному решению одинаковых задач, что повышает ясноть кода и упрощает его сопровождение.
* *Повышение скорости разработки*. Разработчику не приходится решать все задачи и принимать решения «с нуля»»
* *Повышение уровня взаимодействия в команде*. Наличие стандарта позволяет уменьшить разногласия в команде и устранить ненужные дебаты по мелким вопросам, облегчает понимание и поддержку чужого кода членами команды.
* *Согласованность в работе*. При использовании стандарта разарботчики направляют свои услилия в верном направлении, на решение действительно важных задач.» [17].

Исходя из выше озвученных доводов для разработки был сформирован стандарт кодирования, изложенный в таблице 2.

**Таблица 2. Принятый стандарт кодирования**

|  |  |
| --- | --- |
| Объект стандартизации | Стандарт |
| Кодировка текста программы | UTF-8 без BOM |
| Символ перевода строки | LF (UNIX формат) |
| Пробелы | Операторы и аргументы функций отделяются одним пробелом. |
| Именование идентификаторов (переменные, называния функций, классы) | * CamelCase нотация * Имена классов с большой буквы * Имена открытых методов и переменных с маленькой буквы * Имена закрытых методов и переменных с нижнего подчеркивания. |
| Блоки и отступы | Скобка начала блока на той же строке, что оператор владелец блока. Операторы в блоке сдвинуты от начала блока на один символ <Tab>. |
| Структура директорий | Каждый класс расположен в отдельном файле. Файлы классов-наследников располагаются на директорию глубже класса родителя. |

**Вывод: Принятие стандарта кодирования позволило увеличить читаемость, однородность внешнего вида кода. Продуманная структура файлов проекта упростила сборку каркаса из отдельных компонентов, а также потенциально сделала проще процесс дальнейшего развития каркаса.**

# Интеграция классов каркаса

Термин «интеграция» обозначает операцию по объединению отдельных программных компонентов в функционирующую систему.

Термин интеграции тесно переплетается с вопросом последовательности конструирования. Выбранный способ интеграции определяет порядок кодирования и объединения компонентов.

Виды интеграции главным образом можно разделить по частоте на поэтапную и инкрементную.

**Поэтапная интеграция**

Поэтапная интеграция представляет собой последовательное, полное создание всех компонентов системы: кодирование, тестирование и отладка. После полного создания компонентов происходит их интеграция в систему.

**Преимущества поэтапной интеграции**

* Если проектируемая система небольшого размера, то поэтапная интеграция может уменьшить время разработки из-за того, что сборка всей системы будет проходить существенно меньшее количество раз, чем в случае с инкрементной интеграцией.

**Проблемы поэтапной интеграции**

* При объединении системы возможны ошибки, чей источник трудно определить ведь потенциально источником ошибки может быть любой компонент системы.

**Инкрементная интеграция**

Инкрементная интеграция представляет собой создание системы через добавление небольших фрагментов кода и тестирование получившейся системы. Процесс наращивания системы фрагментом кода называется итерацией. Наращиваемый за итерацию код, может определять некоторую, небольшую функциональную часть системы.

**Преимущества инкрементной интеграции**:

* Упрощается процесс нахождения ошибок из-за определенности, какая именно часть кода вызывает ошибку. Это вызвано тем, что ошибки возникают только после интеграции очередного фрагмента кода.
* В процессе разработки, всегда можно оценить на какой стадии готовности находится система.

**Проблемы инкрементной интеграции**

* Не всегда применима из-за особенностей интегрируемой системы. Например, если система может функционировать только при всех готовых компонентах.

**Стратегии инкрементной интеграции**

Инкрементная интеграция может осуществляться с помощью ряда стратегий [22]:

* **Нисходящая интеграция** – Классы интегрируются от вершины иерархии до низа иерархии (Рисунок 3).

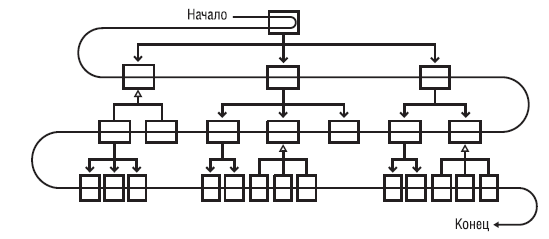


Рисунок 3 – Стратегии нисходящая интеграции

* **Восходящая интеграция** – Классы интегрируются от низа иерархии до вершины иерархии (Рисунок 4).

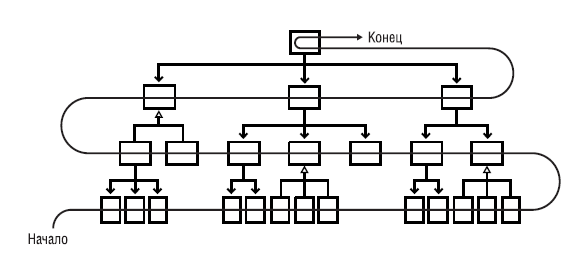


Рисунок 4 – Стратегия восходящей интеграции

* **Сендвич-интеграция** – Первыми объединяются высокоуровневые классы на вершине иерархии. Затем добавляются классы, взаимодействующие с аппаратной частью, и широко используемые вспомогательные классы в низу иерархии. В последнюю очередь интегрируются классы среднего уровня (Рисунок 5).

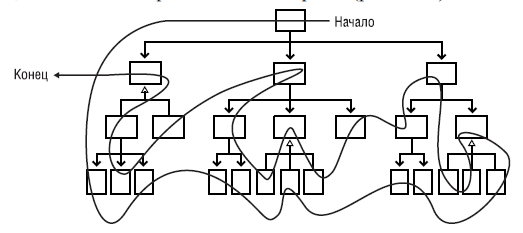


Рисунок 5 – Стратегия сендвич-интеграции

* **Риск-ориентированная интеграция** – Первыми определяется степень риска, связанная с каждым классом. Под риском понимается риск переделки или наличия ошибок. Затем решается, какие части системы будут самыми трудными, и реализуются в первую очередь (Рисунок 6).

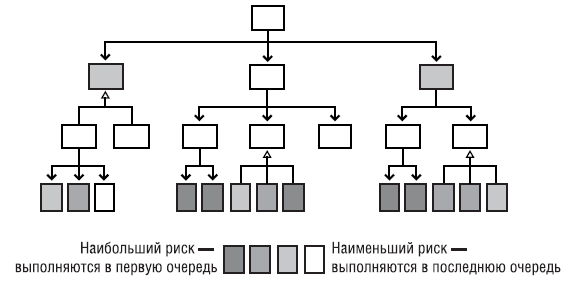


Рисунок 6 – Стратегия риск-ориентированной интеграции.

* **Функционально-ориентированная интеграция** – Первыми создаются группы классов, представляющие собой отдельные функции, поддающиеся определению, которые часто, но не всегда состоят из нескольких классов. (Рисунок 7).

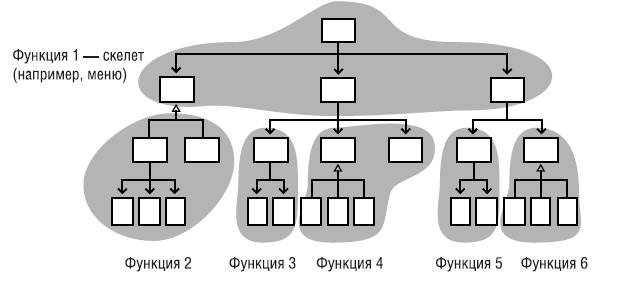


Рисунок 7 – Стратегия функционально-ориентированной интеграции.

* **Т-образная интеграция** – Первым создается и интегрируется вертикальный срез системы, чтобы проверить архитектурные допущения. После этого создается и интегрируется основная горизонталь системы, чтобы предоставить каркас для разработки остальной функциональности. (Рисунок 8).

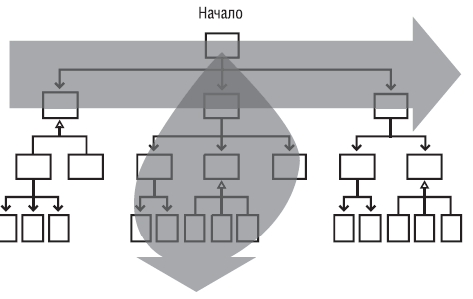


Рисунок 8 – Стратегия Т-образной интеграции.

Известно, что удачно выбранный способ интеграции может дать: [18]

* Возможность выявить принципиальные ошибки в архитектуре приложения при минимальном количестве закодированных компонентов системы.
* Упрощение диагностики дефектов.
* Уменьшение числа возможных ошибок.
* Уменьшение общего времени разработки.
* Улучшение качества кода.

Для интеграции классов каркаса была выбрана так называемая Т-интеграция.

Инкрементная интеграция была выбрана из-за того, что реализуемый каркас уже нельзя отнести к небольшой системе, то есть в процессе могут появляться трудноустранимые ошибки связанные с объединением целого ряда компонентов системы за один раз.

Стратегия Т-инкрементной интеграции была выбрана для того, что бы проверить работоспособность системы при создании минимально необходимого количества компонентов каркаса.

Система реализовывалась так, что бы в первую очередь обеспечить работу компонентов Object и Image. После получения работающей системы с минимальной функциональностью дальнейшая функциональность наращивалась по аналогии с уже реализованным компонентам (Rect, Line, Circle по аналогии с Image).

В рамках каркаса интеграция шла через реализацию и объединение классов каркаса в порядке:

* ArmLib.
* Классы Layer и EventStack.
* Классы ArmObj и Skeleton.
* Класс VisualObj.
* Класс Object и Primitive.
* Класс Image.
* Класс Rect, Line, Circle.

**Вывод: Осознанный выбор способа интеграции позволил реализовывать каркас с помощью ряда итерации, в ходе которых постепенно наращивалась функциональность каркаса, что давало возможность проверить удачность реализованной за итерацию части.**

# Средства разработки

# Используемые утилиты и библиотеки

В ходе разработки была использована утилита jsLint[19] и библиотека общего назначения Gizmo [20].

**Утилита jsLint** проверяет код на наличие синтаксических и широко распространенных семантических ошибок. Возможность подсветки семантических ошибок намного ускорило процесс кодирования из-за возможности быстро исключать многие ошибки данного класса с помощью данной утилиты.

Была использована версия jsLint в виде плагина для среды разработки Netbeans так как разработка велась с помощью среды Netbeans. Использование jsLint в виде плагина являлось более удобным путем использования, чем классический путь использование утилиты в виде веб или консольного приложения.

**Библиотека Gizmo** была реализована в ходе кодирования каркаса. Данная библиотека была создана для хранения функций, чей функционал напрямую не относится к функциональности реализуемого каркаса, но необходим для его реализации. Примером таких функции может служить реализация обертки имитирующей синтаксис ООП для языка C++.

Это было сделано из-за отсутствия в языке JavaScript языковых конструкций для создания классов в их классическом, для таких языков как C++ или С#, понимании.

**Вывод: Использование утилиты для анализа кода помогло упростить процесс кодирования, а написание своей библиотеки общего назначения позволило повысить количество повторного использования уже реализованных решений и накопить набор потенциально полезных в других проектах, функций.**

# Средства разработки

# Интегрированная среда разработки

«По некоторым оценкам до 40% рабочего времени программист тратит

на редактирование исходного кода» [16]

Среда разработки является инструментом призванным упрощать разработку программного продукта, за счет интеграции таких инструментов как: текстовый редактор, компилятор, средства оптимизации сборки, система контроля версий и т.д, в одном продукте.

Выбор среды разработки, является критичным как с точки зрения времени выполнения проекта, так и с точки зрения качества проекта.

Грамотный выбор подходящей среды разработки может позволять: уменьшать время кодирования и количество ошибок, за счет:

• Подсветки синтаксиса используемого языка программирования;

• Наглядном представлении файлов проекта;

• Интеграции с системами контроля версий;

• Интеграция с системами тестирования;

• Использовании различных плагинов, например jsLint и т.д.

Для разработки была выбрана интегрированная среда разработки Netbeans. Выбор Netbeans обусловлен:

• Стоимостью, среда бесплатна;

• Большим количеством пользователей, в случае возникновения проблем проще найти решение;

• Поддержкой синтаксиса языка JavaScript;

• Поддержка распределенной системы контроля версий Git.

• Возможность использовать утилиту jsLint в виде плагина, что позволяет использовать данную утилиту совместно со средой разработки.

**Вывод: Использование интегрированной среды разработки дало возможность сосредоточиться на процессе кодирования, что вероятно уменьшило количество возможных ошибок и улучшило качество кода.**

# Браузер

В силу заложенных требований к кросбраузерности, каркас тестировался на трех основных браузерах:

* Chrome 26.0.1410.64;
* Opera 12.13;
* FireFox 18.0.1.

В первую очередь тестирование проводилось на браузере Chrome из-за наличия встроенного отладчика и профилировщика.

**Вывод: выделение трех целевых браузеров позволило оптимизировать каркаса только к наиболее востребованным браузерам.**

# Отладчик и профайлер

В ходе разработки использовался встроенные отладчик и профайлер браузера Chrome. Это было вызвано тем, что браузер Chrome был одним из целевых браузеров и содержал субъективно наиболее удобные отладчик и профайлер среди встроенных инструментов других браузеров.

Использование отладчика браузера было более предпочтительным, чем использование отладчика среды Netbeans, так как процесс отладки можно было вести полностью в браузере.

**Вывод: использование отладчика и профайлера позволило выявлять ошибки и оценивать производительность отдельных участков кода, что позволило повысить производительность каркаса**

# Система контроля версий

В проекте была использована распределенная система контроля версий Git и бесплатный сервис для хостинга репозиториев под управлением Git – GitHub. Это позволило обезопасить проект от возможной порчи диска и утраты файлов проекта, упростило возможность переключатся между версиями кода.

Разработка велась через создание ветки для разработки отдельного компонента системы. В случае если разработка компонента признавалась удачно законченной – ветка сливалась с базовой веткой репозиория, если неудачной – совершался окат либо к предыдущим версиям кода в ветке, либо создание новой ветки для разрабоки компонента с учетом предыдущих ошибок.

Выбор распределенной, а не централизованной системы контроля версий позволил не зависеть от наличия доступа в интернет.

**Вывод: Использование системы контроля версий уменьшило риск потери исходного кода проекта и упростило процесс кодирования.**

# ЗАКЛЮЧЕНИЕ

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

# ПРИЛОЖЕНИЯ

# ТЗ

# Руководство программиста (ГОСТ 19.504)