|  |
| --- |
| **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

Ассистент каф. АиКС ИК ТПУ Ф.Е. Татарский

**Допустить к защите**

Зав. кафедрой АиКС ИК ТПУ,

д.т.н., профессор Г.П. Цапко

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 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Г.П. Цапко | | |
|  | подпись |  |
| «\_\_\_\_\_\_» | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_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 27.02.2013 № 1028/с
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. Ознакомиться с существующими решениями и технологией создания интерактивных веб-приложений.
   3. Выявить требования к каркасу.
   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 графических 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

В ходе создания выпускной квалификационной работы на тему «Программный каркас для создания спрайтовой анимации на HTML5» решается проблема 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 графических приложений, а также для всех, кому интересна данная тематика. Работа содержит девять рисунков, две таблицы и одно приложение. Общий объем работы составляет 64 страницы. Структура работы представлена списком определений, введением, двумя главами, заключением, списком литературы, а также приложениями.

# СОДЕРЖАНИЕ

[РЕФЕРАТ 4](#_Toc358824384)

[СОДЕРЖАНИЕ 5](#_Toc358824385)

[ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ 6](#_Toc358824386)

[ВВЕДЕНИЕ 9](#_Toc358824387)

[1. ГРАФИЧЕСКОЕ ВЕБ-ПРИЛОЖЕНИЕ 10](#_Toc358824388)

[1.1. Современное веб-приложение 10](#_Toc358824389)

[1.2. Архитектура веб-приложения 11](#_Toc358824390)

[1.3. Клиентские технологии 12](#_Toc358824391)

[1.4. Технология 2d-context HTML5 14](#_Toc358824392)

[1.5. Сложности разработки с помощью 2d-context HTML5 20](#_Toc358824393)

[1.6. Постановка задачи 22](#_Toc358824394)

[2. АРХИТЕКТУРА ПРОГРАММНОГО КАРКАСА 23](#_Toc358824395)

[2.1. Обзор существующих решений 23](#_Toc358824396)

[2.2. Требования к программному каркасу 24](#_Toc358824397)

[2.3. Реализация 26](#_Toc358824398)

[2.3.1. Используемые алгоритмы и техники 26](#_Toc358824399)

[2.3.2. Диаграмма классов 29](#_Toc358824400)

[2.3.2.1. Класс ArmLib 29](#_Toc358824401)

[2.3.2.2. Класс Layer 31](#_Toc358824402)

[2.3.2.3. Класс EventQueue 33](#_Toc358824403)

[2.3.2.4. Класс Primitive 33](#_Toc358824404)

[2.3.2.5. Класс ArmObj 34](#_Toc358824405)

[2.3.2.6. Класс Skeleton 35](#_Toc358824406)

[2.3.2.7. Класс VisualObj 37](#_Toc358824407)

[2.3.2.8. Класс Object 40](#_Toc358824408)

[2.3.2.9. Класс Image 43](#_Toc358824409)

[2.3.2.10. Класс Rect 43](#_Toc358824410)

[2.3.2.11. Класс Circle 44](#_Toc358824411)

[2.3.2.12. Класс Line 44](#_Toc358824412)

[2.3.3. Используемые шаблоны проектирования 45](#_Toc358824413)

[2.3.4. Принятые стандарты кодирования 46](#_Toc358824414)

[2.3.5. Интеграция классов каркаса 49](#_Toc358824415)

[2.4. Средства разработки 54](#_Toc358824416)

[2.4.1. Используемые утилиты и библиотеки 54](#_Toc358824417)

[2.4.2. Средства разработки 55](#_Toc358824418)

[2.4.3. Интегрированная среда разработки 55](#_Toc358824419)

[2.4.4. Браузер 56](#_Toc358824420)

[2.4.5. Отладчик и профайлер 57](#_Toc358824421)

[2.4.6. Система контроля версий 58](#_Toc358824422)

[ЗАКЛЮЧЕНИЕ 59](#_Toc358824423)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 62](#_Toc358824424)

[ПРИЛОЖЕНИЕ А. ТЕХНИЧЕСКОЕ ЗАДАНИЕ 64](#_Toc358824425)

# ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

1. *Веб-приложение* – представляет собой клиент-серверное приложение, в котором клиентской частью является браузер, а серверной – веб-сервер. Логика приложения распределена между клиентской и серверной частями. Данные используемые приложением хранятся на сервере и используются приложением по запросу.
2. *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».
3. *JavaScript* – прототипно-оприентированный сценарный язык программирования, являющийся диалектом языка ECMAScript. Чаще всего используется в браузерах для доступа к программным объектам.
4. *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.
5. *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.
6. *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.
7. *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.
8. *Шейдер* – программа, которая используется в трёхмерной графике для определения окончательных параметров изображения или объекта.
9. *Программный каркас –* программное обеспечение, облегчающее разработку и объединение частей программного проекта через введение некоторой минимальной системы, через расширение которой ведется дальнейшая разработка.
10. *BOM –* Unicode-символ, используемый в качестве указателя порядка байтов текстового файла. Кодировка Unicode может использовать 16-разрядные или 32-разрядные числа и приложение должно иметь информацию как с ним поступать.
11. *Сокеты* – программного интерфейса для обеспечения обмена данными между процессами. Процессы при таком обмене могут исполняться как на одном компьютере, так на нескольких соединенных в сеть.
12. *UNIX* формат – Формат символа переноса строки используемый в UNIX-подобных операционных системах. В качестве разделителя используется символ LF.
13. *CamelCase нотация* – Стиль написания составных слов, при котором последовательность слов пишется слитно, при этом каждое слово пишется с заглавной буквы.
14. *Репозиорий* – Некоторое место, используемое для хранения и данных. В данной работе подразумевается хранилище исходного кода.
15. *Ветка* – В терминах систем контроля версий это некоторый вариант одного и того же документа или набора документов. Документы в некоторых смежных ветках имеют общую историю изменения до точки ветвления и разную после.

# ВВЕДЕНИЕ

На сегодняшний день создание программного обеспечения в виде веб-приложения 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 графикой в традиционных графических приложениях.

Обычно разработчик заинтересован упростить процесс создания, увеличить надежность, а так же сократить время разработки. Одним из способов достижения этих целей является создание программного каркаса.

Программный каркас это программное обеспечение, облегчающее разработку и объединение частей программного проекта через введение некоторой минимальной системы, с помощью расширения которой ведется дальнейшая разработка.

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.

Разрабатываемый каркас реализует различные функциональные части, повторяемые от приложения к приложению, и связывает их в некоторую систему, что позволяет уже на самом раннем этапе разработки приложения иметь реализованными базовые функции приложения.

# ГРАФИЧЕСКОЕ ВЕБ-ПРИЛОЖЕНИЕ

# Современное веб-приложение

С ростом возможностей современных веб-технологий, веб-приложение [16] 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 графических 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 при использовании таких технологий как Flash и HTML5 возможна разработка веб-приложения 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: Современное графическое веб-приложение может служить альтернативой настольным графическим приложениям, таким как игры.

# Архитектура веб-приложения

Веб-приложение представляет собой 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 [16]
* CSS [17]

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 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 графическая 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 [1];
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 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 [2]: 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 рисунок 1.

****

**Рисунок 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 [3] 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 [3].

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 для обрезки сцены по заданному контуру.
7. 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.
6. 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 графические веб-приложения.

Появление технологии 2d-context HTML5 позволяет создавать динамичные графические 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 решить следующие подзадачи:

* Подробнее ознакомится с интерфейсом 2d-context;
* Ознакомится с программным интерфейсом 2d-context;
* Ознакомиться с существующими решениями и технологией создания графических веб-приложений;
* Выявить требования к каркасу;
* Спроектировать и реализовать программный каркас.

Следующая глава посвящена реализации каркаса через решение, определенных в данной главе, подзадач.

# 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. 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 мышью и клавиатурой | классы обертки для примитивов | Загрузка мультимедиа-зависимостей объектов | Приоритеты очереди прорисовки | Модульность | Вызовы цепочкой |
| JCanvaScript | + | + | + | -\+ | + | + | - | + | - | - |
| KineticJS | + | + | + | + | + | + | - | + | + | - |
| LibCanvas | + | + | + | + | + | + | - | + | - | + |

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

1. jCanvaScript – Свободно распространяемая библиотека доступная по адресу [4] по лицензии MIT или GPL версии 2.
2. KineticJS – Свободно распространяемая библиотека доступная по адресу [5] по лицензии MIT или GPL версии 2.

LibCanvas – Свободно распространяемая библиотека доступная по адресу [6] по лицензии MIT или GPL.

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 мышью и клавиатурой;
6. Загрузка мультимедиа-зависимостей объектов;
7. Приоритеты очереди прорисовки;
8. Модульность;
9. Вызовы методов каркаса цепочкой.

# Требования к программному каркасу

В ходе анализа существующих решений и анализа процесса создания графических приложений были выявлены следующие требования:

**Объектное представление примитивов**, **Работа с мышью и клавиатурой**. В графическом приложении, интерактивность достигается за счет взаимодействия пользователя с отдельными элементами на экране, 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;

**Call-back функции для событий объектов.** В случае представления некоторой сущности на экране в виде некоторого объекта, нужно понимать, что объект может иметь не только свойства и методы, но и должен иметь возможность реагировать на некоторые асинхронные события, например: реагировать на щелчок мыши.

**Загрузка мультимедиа-зависимостей объектов**. Такие примитивы как изображение или видео перед выводом на экран требуют загрузки данных. Так как с помощью этих примитивов описывается некоторая сущность, то сущность должна быть описана максимально целостно. Сущность должна иметь возможность самостоятельно загрузить необходимые данные и в соответствующем обработчике среагировать на некоторые события связанные с загрузкой;

**Приоритеты очереди прорисовки.** Объекты могут пересекаться наэкране и определенные объекты должны находится выше или ниже других по этому нужен способ указывать приоритет объекта на прорисовку;

**Графический цикл**. Так как 2d-context представляет так называемый непосредственный графический режим, то программист имеет в распоряжении минимальный набор методов для создания некоторой картинки, но если есть потребность в изменяющемся динамичном изображении программист должен сам думать о очистке экрана, перед новым кадром, организации смены кадров и т.д.

Можно сделать вывод, что в каркасе должен быть реализован графический цикл, то есть программист не должен забоится о организации смены кадров и т.д.;

**Слои**. 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). Одним из способов увеличить производительность является использование нескольких тегов <canvas> расположенных друг над другом, определенные объекты располагаются на разных слоях.

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

**Модульность**. 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;

**Вызовы методов каркаса цепочкой**. При анализе существующих каркасов была отмечена такая функция как возможность вызывать методы объекта не только из самого объекта, но и сразу после вызова методов объекта не возвращающих значение. Данная функциональность используется и в крупных библиотеках и каркасах не связанных с графикой, например в jQuery [7].

# Реализация

# Используемые алгоритмы и техники

В каркасе были использованы следующие алгоритмы и техники:

* Угловой тест положения точки относительно полигона;
* Техника «redraw regions».

**Угловой тест положения точки относительно полигона**

Данный алгоритм используется для проверки попадания курсора мыши в ограничивающий примитив полигон. Алгоритм был взят из учебного пособия [8]

Алгоритм основан на вычислении и анализе алгебраических сумм углов между смежными векторами , соединящими точку q с вершиной , при обходе произвольного полигона P по замкнутому контуру в произвольном направлении. Тест основан на следующем опыте: наблюдатель, просматривающий веришны полигона из внутренней точки , совершает вокруг себя полный оборот (Рисунок 2, а), а из любой внешней точки – ни одного оборота (Рисунок 2, б).

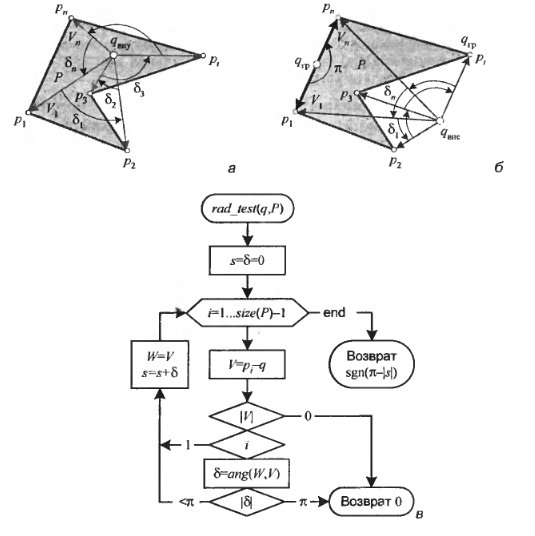


Рисунок 2 – Угловой тест положения точки относительно полигона

Алгоритм можно описать следующим выражением:

(1)

Граничная точка обнаруживается по следующим признакам:

* если при расчете векторов будет получен нулевой вектор длиной , то тестируемая точка совпадает с вершиной ;
* если при расчете углов будет получен развернутый угол с модулем , то тестируемая точка лежит на ребре (Рисунок 2, б)

**Техника «redraw regions»**

Цель данной техники состоит в уменьшении времени перерисовки кадра, за счет уменьшения времени затрачиваемого на очистку экрана для нового кадра.

Суть техники построена на том, факте что время, затрачиваемое на очистку или вывод изображения на некоторую площадь экрана тем больше, чем больше очищаемая площадь. Из этого факта можно сделать вывод, что уменьшить время перерисовки можно уменьшением стираемой для каждого объекта части экрана.

Техника реализована с помощью:

* Нахождения для каждого объекта на сцене, минимального прямоугольника, в который можно вписать данный объект и его положения на экране;
* Запоминании области экрана имеющей размер минимального прямоугольника перед выводом объекта на экран;
* Выводе запомненного изображения на экран при потребности перерисовать некоторый объект сцены.

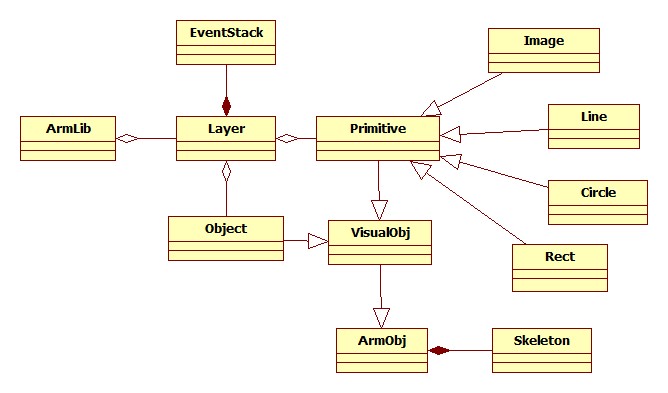
Данная техника эффективна при условии, что объекты сцены имеют небольшой размер в сравнении с размерами экрана в обратном случае возможно существенное замедление времени прорисовки. Время прорисовки может падать из-за неоднократной отчистки области экрана соизмеримой с размером экрана, при том, что при не использовании данной техники на перерисовку кадра выпадает только одна очистка экрана.

# 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.(Рисунок 3)

****

**Рисунок 3 – 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;

**object \_container** – данное поле содержит ссылку на тег являющийся контейнером для тега <canvas> текущего слоя;

**object \_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:*

**string \_name** – данное поле содержит имя объекта;

**bool \_isLoaded** – данное поле содержит булево значение характеризующее загружены ли мультимедиа-зависимости объекта или нет;

**object \_owner** – данное поле содержит ссылку на родительский по отношению к текущему, объект;

*Основные методы класса:*

**object load()** – метод инициирующий загрузку метода;

**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()** – 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()** – 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()** – 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()** – 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;

**void \_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** – поле задающее радиус;

**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 [9] 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.».

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 [10]:

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

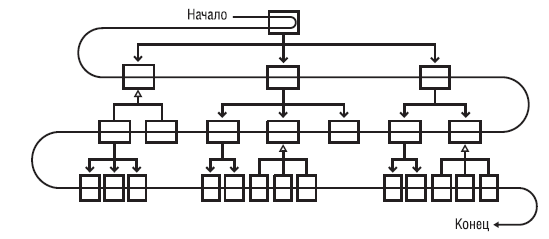


Рисунок 4 – Стратегии нисходящая интеграции

* **Восходящая интеграция** – Классы интегрируются от низа иерархии до вершины иерархии (Рисунок 5).

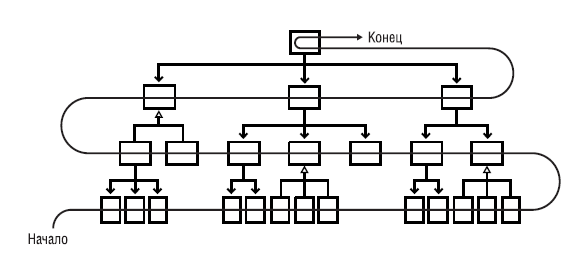


Рисунок 5 – Стратегия восходящей интеграции

* **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 6).

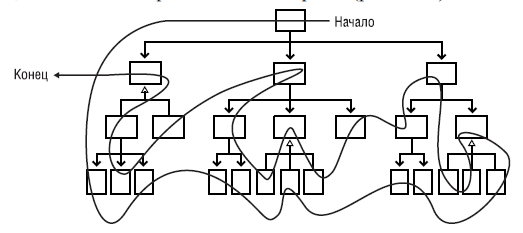


Рисунок 6 – Стратегия сендвич-интеграции

* **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 7).

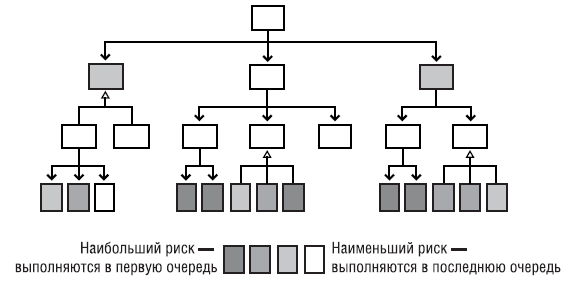


Рисунок 7 – Стратегия риск-ориентированной интеграции.

* **Функционально-ориентированная интеграция** – Первыми создаются группы классов, представляющие собой отдельные функции, поддающиеся определению, которые часто, но не всегда состоят из нескольких классов. (Рисунок 8).

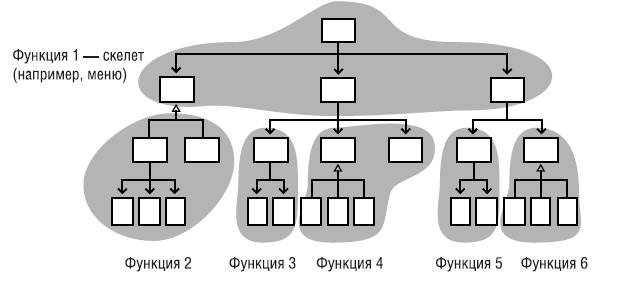


Рисунок 8 – Стратегия функционально-ориентированной интеграции.

* **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 9).

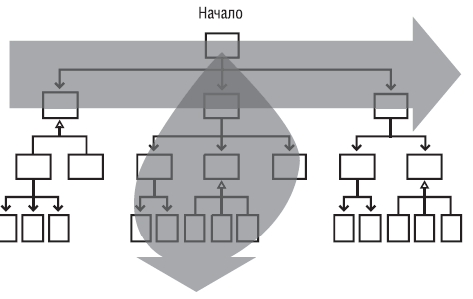


Рисунок 9 – Стратегия Т-образной интеграции.

Известно, что удачно выбранный способ интеграции может дать: [11]

* 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 [12] и библиотека общего назначения Gizmo [13].

**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 словам Стива Макконнелла до 40% рабочего времени программист тратит на редактирование исходного кода. [14]

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

Отладчик и профайлер и это основные инструменты программиста. С помощью отладчика, иногда его называют дебагером, выполняется поиск семантических ошибок, пожалуй, создание почти любого приложения не происходит без отладчика.

Профайлер это инструмент для выявления количественных характеристик работы приложения, таких время работы отдельных функций или количество используемой памяти. Профайлер используется для поиска так называемых «узких мест» программы, где происходят максимальные потери времени или памяти.

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 [15]. 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

В ходе выполнения данной работы была изучена спецификация на технологию 2d-context HTML5, проведен сравнительный анализ существующих программных каркасов для данной технологии и разработан и реализован программный каркас позволяющий упростить процесс разработки графического веб-приложения за счет реализации таких функций как:

* графический цикл;
* слои;
* объектное представление примитивов;
* call-back функции для событий объектов;
* работа с мышью и клавиатурой;
* классы-обертки для примитивов;
* загрузка мультимедиа-зависимостей объектов;
* приоритеты очереди прорисовки;
* модульность;
* вызовы методов каркаса цепочкой.

В ходе работы были достигнуты следующие результаты:

* Разработан и реализован программный каркас для упрощения разработки графического веб-приложения с помощью технологии 2d-context HTML5.
* Написана статья на тему: «Программный каркас для создания спрайтовой анимации на HTML5». Данная статья получила третье место в секции «Технологии разработки и проектирования информационных систем» на конференции «Технологии Microsoft в теории и практике программирования».

В дальнейшем работа по данной теме будет продолжена. В частности планируется существенно пересмотреть архитектуру каркаса с целью повышения гибкости разработки с использованием каркаса.

**CONCLUSION**

In the course of this work was studied specification for the technology 2d-context HTML5, a comparative analysis of the existing policy frameworks for the technology, and designed and implemented a software framework to simplify the process of developing a multimedia web-based applications through the implementation of such things as:

• Graphical loop;

• Layers;

• Object representation of primitives;

• Call-back function for the events of objects;

• Work with the mouse and keyboard;

• Wrapper classes for primitives;

• Downloading dependencies of multimedia object;

• Priorities of drawing turns;

• Modularity;

• Calling of framework methods one by one.

During the following results were achieved:

• Developed and implemented a software framework to simplify the development of graphical web applications using technologies 2d-context HTML5.

• Write an article on "The software framework for creating sprite animation on HTML5».

Further work on this topic will continue. In particular architecture will significantly revise carcass to improve design flexibility by using the framework.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Context. WebGL – a new dimension for browser exploitation *[Электронный ресурс].* – Режим доступа: http://www.contextis.com/research/blog/webgl-new-dimension-browser-exploitation/, свободный. Дата обращения: 25.05.2012.
2. Joel Oughton. SVG vs Canvas Performance *[Электронный ресурс].* – Режим доступа: http://joeloughton.com/blog/web-applications/svg-vs-canvas-performance/, свободный. Дата обращения: 04.03.2013.
3. W3C. 2d-context specification *[Электронный ресурс].* – Режим доступа: http://www.w3.org/TR/2010/WD-2dcontext-20101019/, свободный. Дата обращения: 04.03.2012.
4. Официальный сайт каркаса JCanvaScript *[Электронный ресурс].* – Режим доступа: http://jcscript.com/, свободный. Дата обращения: 26.05.2012.
5. Официальный сайт каркаса KineticJS *[Электронный ресурс].* – Режим доступа: http://kineticjs.com/, свободный. Дата обращения: 26.05.2012.
6. Павел Пономаренко. LibCanvas *[Электронный ресурс].* – Режим доступа: https://github.com/theshock/libcanvas, свободный. Дата обращения: 26.05.2012.
7. Официальный сайт библиотеки jQuery *[Электронный ресурс].* – Режим доступа: http://jquery.com/, свободный. Дата обращения: 11.06.2012.
8. Никулин Е.А. Компьютерная геометрия и алгоритмы машинной графики – Спб:: БХВ-Петербург, 2005. – С. 34.
9. Александреску А., Саттер Г. Стандарты программирования на C++. :Пер. с англ. – М.: Издательский дом «Вильямс», 2005. – С. 10.
10. Стив Макконнелл. Совершенный код, Мастер-класс / Пер. с англ. М.: Издательство «Русская редакция», 2010, С. 678.
11. Стив Макконнелл. Совершенный код, Мастер-класс / Пер. с англ. М.: Издательство «Русская редакция», 2010, С. 674.
12. Официальный сайт jsLint *[Электронный ресурс].* – Режим доступа: http://www.jslint.com/, свободный. Дата обращения: 3.06.2012.
13. Лизин, Александр Сергеевич. Gizmo *[Электронный ресурс].* – Режим доступа: http://github.com/sogimu/Gizmo.git, свободный. Дата обращения: 29.05.2012.
14. Стив Макконнелл. Совершенный код, Мастер-класс / Пер. с англ. М.: Издательство «Русская редакция», 2010, С. 695.
15. GitHub. Социальная сеть. Веб-сервис для хостинга проектов *[Электронный ресурс].* – Режим доступа: http://gtihub.com/, свободный. Дата обращения: 3.06.2012.
16. HTML5. Living Standard *[Электронный ресурс].* – Режим доступа: http://www.whatwg.org/specs/web-apps/current-work/, свободный. Дата обращения: 3.06.2012.
17. CSS. Specification *[Электронный ресурс].* – Режим доступа: http://www.w3.org/TR/CSS21/, свободный. Дата обращения: 3.06.2012.

# ПРИЛОЖЕНИЕ А. ТЕХНИЧЕСКОЕ ЗАДАНИЕ