Здесь будет титульный лист. TODO

РЕФЕРАТ

Выпускная квалификационная работа по теме «Система физического моделирования на основе априорного подхода обнаружения столкновений» содержит 34 страниц текстового документа, TODOиллюстраций, TODOтаблицы, TODOприложений, 54 использованных источников.

ФИЗИЧЕСКИЙ ДВИЖОК, ОБНАРУЖЕНИЕ СТОЛКНОВЕНИЙ, АПРИОРНЫЙ ПОДХОД, ЧИСЛЕННЫЕ МЕТОДЫ, OCAML.

Цель работы: TODO.

Задачи: TODO.

TODO

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Здесь будет введение. TODO

1 Теоретические сведения

1.1 Определения

Физический движок – программное обеспечение, предназначенное для приближённой симуляции физический систем реального мира в виртуальном, например динамику твёрдого тела (включая обнаружение столкновений и обработку ударов).

Иными словами, система физического моделирования (т.е. моделирующая физику – в данном контексте симуляция синонимична моделированию) и есть физический движок.

Традиционно, физические движки делятся на 2 типа: игровые и научные. Отличительная черта первых – возможность работы в режиме реального времени, то есть воспроизводить физические процессы в игре с той же самой скоростью, в которой они происходят в реальном мире. Отличительная черта вторых – важна точность вычислений, скорость вычислений не играет существенной роли.

Кроме того, в игровых движках зачастую достаточно симулировать только текущее состояние виртуального мира без сохранения результата, а в научных движках необходимо сохранять все смоделированные в виртуальном мире взаимодействия для последующего анализа.

Одна из задач данной работы – создать движок, имеющий свойства и научного, и игрового – чтобы присутствовала возможность и моделирования в реальном времени, и эффективной «перемотки».

1.2 Апостериорный и априорный подход

В философском смысле, апостериори означает знание, полученное из опыта; а априори означает знание, полученное до опыта и независимо от него [1, с. 105-106].

В аналогичном порядке и делят подходы к реализации физических моделей: определение столкновения апостериори и априорный подход (обнаружение до происхождения столкновения).

В апостериорном случае физическое моделирование продвигается на небольшой шаг, а затем проверяется, пересекаются ли какие-либо объекты или,

по видимости, считаются пересекающимися. На каждом шаге моделирования создается список всех пересекающихся тел, а положения и траектории этих объектов «фиксируются» для учета столкновения. Этот метод называется апостериорным, поскольку он, как правило, не учитывает фактический момент столкновения, а фиксирует столкновение только после того, как оно произошло.

В априорных методах существует алгоритм обнаружения столкновений, который сможет очень точно предсказать траектории физических тел. Моменты столкновения рассчитываются с высокой точностью, при этом физические тела никогда фактически не пересекаются. Это называется априорным, потому что алгоритм обнаружения столкновений вычисляет моменты столкновения до того, как он обновит конфигурацию физических тел.

Основные преимущества апостериорных методов заключаются в том, алгоритму обнаружения столкновений не нужно знать о множестве физических переменных; алгоритму подается простой список физических тел, и программа возвращает список пересекающихся тел. Алгоритму обнаружения столкновений не нужно понимать трение, упругие столкновения или, что еще хуже, неупругие столкновения и деформируемые тела. Кроме того, апостериорные алгоритмы в действительности на одно измерение проще, чем априорные алгоритмы. Априорный алгоритм должен иметь дело с переменной времени, которая отсутствует в апостериорной задаче.

С другой стороны, у апостериорных алгоритмы существует шаг «исправления», где пересечения (которые физически не являются правильными) должны быть исправлены и это может создать проблему. Более того, если дискретный шаг слишком велик, столкновение может остаться незамеченным, в результате чего объект пройдет сквозь другой, если он достаточно быстрый или маленький.

Преимуществами априорных алгоритмов являются повышенная точность и стабильность. Физическое моделирование трудно отделить от алгоритма обнаружения столкновений. Однако во всех случаях, кроме самых простых, проблема определения времени когда два тела столкнутся заранее (учитывая некоторые начальные данные), не имеет аналитического решения – обычно требуется численный поиск корней (что и описано в 1.6).

Обычно для игровых физических движков используется апостериорный подход потому что не важна точность, а дискретное моделирование хорошо ложиться на расчёты в реальном времени. Однако данная работа призвана исследовать априорный подход и при этом создать движок, способный на симуляцию в реальном времени.

Как указано ранее, априорный подход должен учитывать моделируемые законы физики. Но при этом много что трудно учесть, поэтому симулируемая модель будет упрощённая, без учёта деформации, вращения, формы. Подвижные объекты – только тела идеально круглой формы, движение равноускоренное (равнозамедленное, но в дальнейшем везде называется равноускоренным), и т.д. Подробнее модель описана под пунктом 1.3.

1.3 Описание модели

Тело. Абсолютно твёрдое тело в форме круга равномерной плотности (центр масс в центре круга) обладающее массой (m), коэффициентом трения (μ), радиусом (r), начальной скоростью ($\vec{v_0}$), положением (координаты x и y или радиус-вектор $\vec{r}$). На тело действует сила трения ($F_{\text{тр.}}$).

Точка. Неподвижная точка в пространстве, определена через координаты.

Линия. Неподвижная прямая линия в пространстве, может быть ограничена точкой с двух или одной сторон образуя отрезок или луч соответственно. Определена через общее уравнение прямой.

Сцена. Множество тел, линий, точек и постоянных (например, ускорение свободного падения).

Обновлённая сцена – сцена, в которой обновлены параметры тел, линий, точек или постоянных.

Сцена через время Δt – обновлённая сцена, в которой все тела обновлены так, что новая начальная скорость равна скорости в этот момент времени (1).

$$\vec{v_{0_{\text{new}}}} = \vec{v}(\Delta t) \tag{1}$$

где $\vec{v_{0_{\text{new}}}}$ – новая начальная скорость;

$\vec{v}(\Delta t)$ – старая скорость в момент времени Δt.

Модель. Множество пар (t, S), где t – момент времени, а S – сцена. Иными словами, модель представляет собой цепочку сцен, для каждой из которой указан момент времени.

Сцена в момент времени t_1 – такая сцена S_0 через время $t_1 - t_0$, где пара (t_0, S_0) является членом модели, при этом соблюдается (2).

$$\forall (t, S) \in M \;\; (t \leqslant t_0 \vee t > t_1) \tag{2}$$

где M – модель;

t_0 – время, выбранное для получения модели в момент времени t_1;

Иными словами, для того чтобы получить сцену в момент времени, надо из цепочки сцен найти такую, у которой время будет максимально, но при этом меньше требуемого момента времени и получить сцену через разность требуемого и найденного времени по формуле (1).

Столкновение. Так как тела не могут пересекаться, и при этом передвигаются, могут происходить столкновения. Так же тела не могут пересекаться с точками и линиями. Т.е. тела могут сталкиваться с телами, или линиями, или точками. Уравнение столкновения тела с телом (3)-через радиус-вектор, (4)-через координаты.

$$|\vec{r_1}(t) - \vec{r_2}(t)| = r_1 + r_2 \tag{3}$$

где $\vec{r_1}(t)$ – радиус-вектор положения первого тела;

$\vec{r_2}(t)$ – радиус-вектор положения второго тела;

r_1 – радиус первого тела;

r_2 – радиус второго тела.

$$\sqrt{(x_1(t) - x_2(t))^2 + (y_1(t) - y_2(t))^2} = r_1 + r_2 \tag{4}$$

где $x_1(t)$ – координата положения первого тела по оси X;

$y_1(t)$ – координата положения первого тела по оси Y;

$x_2(t)$ – координата положения второго тела по оси X;

$y_2(t)$ – координата положения второго тела по оси Y.

Эти уравнения получены исходя из того что разность векторов является вектором из центра одного тела в центр другого [3, с. 39]. И тогда, если его длина равна сумме радиусов этих тел, значит тела столкнулись.

Подобным образом можно определить уравнение столкновения тела с точкой: (5)-через радиус-вектор, (6)-через координаты.

$$|\vec{r}(t) - \vec{p}| = r \tag{5}$$

где $\vec{r}(t)$ – радиус-вектор положения тела;

$\vec{p}$ – радиус-вектор точки;

r – радиус тела.

$$\sqrt{(x(t) - p_x)^2 + (y(t) - p_y)^2} = r \tag{6}$$

где $x(t)$ – координата положения тела по оси X;

$y(t)$ – координата положения тела по оси Y;

p_x – координата точки по оси X;

p_y – координата точки по оси Y.

С обнаружение столкновением тела и линии ситуация несколько иная, надо воспользоваться формулой расстояний от точки до прямой (7) [4, с. 452] и тогда получится (8).

$$\frac{|Ax + By + C|}{\sqrt{A^2 + B^2}} \tag{7}$$

где A, B, C – коэффициенты общего уравнения прямой;

x, y – координаты точки.

$$\frac{|Ax(t) + By(t) + C|}{\sqrt{A^2 + B^2}} = r \tag{8}$$

где $x(t)$, $y(t)$ – координаты тела в момент времени t.

В дальнейшем будут определены формулы для нахождения положения тела, а именно: (17), (18).

1.4 Определение формул скорости и траектории тела

Далее, под моментом времени t будет подразумеваться время относительно сцены, а не модели.

Как указано выше, у тела есть начальная скорость и на него действует сила трения. По второму закону Ньютона [2, с. 114], у тела есть ускорение так как на него действует сила. Такое движение называется равноускоренным.

Скорость при равноускоренном движении определяется формулой (9)[2, с. 96].

$$\vec{v}(t) = \vec{v_0} + \vec{a}t \tag{9}$$

где $\vec{v}(t)$ – вектор скорости тела в момент времени t;

$\vec{v_0}$ – вектор начальной скорости тела;

$\vec{a}$ – вектор ускорения тела;

t – момент времени.

При этом вектор ускорения сонаправлен вектору силы (по второму закону Ньютона, (10)).

$$\vec{a} = \frac{\vec{F}}{m} \tag{10}$$

где $\vec{a}$ – вектор ускорения тела;

$\vec{F}$ – вектор силы действующей на тело;

m – масса тела.

Но вектор силы трения $\vec{F_{\text{тр.}}}$ противонаправлен вектору скорости тела [2, с. 21]. Поэтому, и вектор ускорения тела будет противонаправлен вектору скорости тела.

При этом вектор скорости тела $\vec{v}(t)$ должен быть сонаправлен вектору $\vec{v_0}$ потому что тело не может поменять направление движения при воздействии силы трения. Для того чтобы выяснить, при каких t сонаправленность векторов $\vec{v}(t)$ и $\vec{v_0}$ в уравнении (9) соблюдается, достаточно увидеть, что длина вектора $\vec{v_0}$ должна быть больше длине вектора $\vec{a}t$ и получить неравенство для t (11).

$$t < \frac{|\vec{v_0}|}{|\vec{a}|} \tag{11}$$

А для остальных t, $\vec{v}(t)$ следует принять нулю. Тогда скорость выражается через (12):

$$\vec{v}(t) = \begin{cases} \vec{v_0} + \vec{a}t, & 0 \leqslant t < \frac{|\vec{v_0}|}{|\vec{a}|}, \\ 0, & t \geqslant \frac{|\vec{v_0}|}{|\vec{a}|}. \end{cases} \tag{12}$$

Проекции на ось абсцисс (13) и ординат (14):

$$v_x(t) = \begin{cases} v_{0_x} + a_x t, & 0 \leqslant t < \frac{|\vec{v_0}|}{|\vec{a}|}, \\ 0, & t \geqslant \frac{|\vec{v_0}|}{|\vec{a}|}. \end{cases} \tag{13}$$

где $v_x(t)$ – проекция вектора скорости тела $\vec{v}(t)$ в момент времени t на ось X;
v_{0_x} – проекция вектора начальной скорости тела $\vec{v_0}$ на ось X;
a_x – проекция вектора ускорения тела $\vec{a}$ на ось X.

$$v_y(t) = \begin{cases} v_{0_y} + a_y t, & 0 \leqslant t < \frac{|\vec{v_0}|}{|\vec{a}|}, \\ 0, & t \geqslant \frac{|\vec{v_0}|}{|\vec{a}|}. \end{cases} \tag{14}$$

где $v_y(t)$ – проекция вектора скорости тела $\vec{v}(t)$ в момент времени t на ось Y;
v_{0_y} – проекция вектора начальной скорости тела $\vec{v_0}$ на ось Y;
a_y – проекция вектора ускорения тела $\vec{a}$ на ось Y.

Теперь найдём формулу для траектории движения тела. Формуле, соответствующей (9), только для траектории, соответствует (15):

$$\vec{r}(t) = \vec{r_0} + \vec{v_0}t + \frac{\vec{a}t^2}{2} \tag{15}$$

где $\vec{r}(t)$ – радиус-вектор положения тела в момент времени t;
$\vec{r_0}$ – радиус-вектор начального положения тела.

Исходя из (12), уравнение для траектории с учётом того, что вектор скорости должен быть противонаправлен вектору ускорения, будет (16):

$$\vec{r}(t) = \begin{cases} \vec{r_0} + \vec{v_0}t + \frac{\vec{a}t^2}{2}, & 0 \leqslant t < \frac{|\vec{v_0}|}{|\vec{a}|}, \\ \vec{r_0}, & t \geqslant \frac{|\vec{v_0}|}{|\vec{a}|}. \end{cases} \tag{16}$$

Соответствующие проекции на ось абсцисс (17) и ординат (18):

$$x(t) = \begin{cases} x_0 + v_{0_x} t + \frac{a_x t^2}{2}, & 0 \leqslant t < \frac{|\vec{v_0}|}{|\vec{a}|}, \\ x_0, & t \geqslant \frac{|\vec{v_0}|}{|\vec{a}|}. \end{cases} \tag{17}$$

где $x(t)$ – координата положения тела $\vec{r}(t)$ в момент времени t на ось X;

x_0 – координата начального положения тела $\vec{v_0}$ на ось X.

$$y(t) = \begin{cases} y_0 + v_{0_y} t + \frac{a_y t^2}{2}, & 0 \leqslant t < \frac{|\vec{v_0}|}{|\vec{a}|}, \\ y_0, & t \geqslant \frac{|\vec{v_0}|}{|\vec{a}|}. \end{cases} \tag{18}$$

где $y(t)$ – координата положения тела $\vec{r}(t)$ в момент времени t на ось Y;

y_0 – координата начального положения тела $\vec{v_0}$ на ось Y.

1.5 Определение уравнений для обнаружения столкновений

Подставив формулы (17) и (18) в уравнения (4), (6), (8) можно получить алгебраические уравнения от t четвёртой степени.

Например, для случая, когда $0 \leqslant t < \frac{|\vec{v_0}|}{|\vec{a}|}$ частичный вывод уравнения времени столкновения тела с телом будет таким (19):

$$\begin{aligned} \sqrt{(x_1(t) - x_2(t))^2 + (y_1(t) - y_2(t))^2} &= r_1 + r_2 \\ (x_1(t) - x_2(t))^2 + (y_1(t) - y_2(t))^2 &= (r_1 + r_2)^2 \\ (x_{0_1} + v_{0_{x_1}} t + \frac{a_{x_1} t^2}{2})^2 - (x_{0_2} + v_{0_{x_2}} t + \frac{a_{x_2} t^2}{2})^2 + & \\ +(y_{0_1} + v_{0_{y_1}} t + \frac{a_{y_1} t^2}{2})^2 - (y_{0_2} + v_{0_{y_2}} t + \frac{a_{y_2} t^2}{2})^2 &= (r_1 + r_2)^2 \end{aligned} \tag{19}$$

где $x_1(t)$ – координата первого тела в момент времени t по оси X;

$x_2(t)$ – координата второго тела в момент времени t по оси X;

$y_1(t)$ – координата первого тела в момент времени t по оси Y;

$y_2(t)$ – координата второго тела в момент времени t по оси Y;

x_{0_1} – начальная координата первого тела по оси X;

x_{0_2} – начальная координата второго тела по оси X;

y_{0_1} – начальная координата первого тела по оси Y;

y_{0_2} – начальная координата второго тела по оси Y;

$v_{0_{x_1}}$ – проекция вектора начальной скорости первого тела на ось X;
$v_{0_{y_1}}$ – проекция вектора начальной скорости первого тела на ось Y;
$v_{0_{x_2}}$ – проекция вектора начальной скорости второго тела на ось X;
$v_{0_{y_2}}$ – проекция вектора начальной скорости второго тела на ось Y;
a_{x_1} – проекция вектора ускорения первого тела на ось X;
a_{y_1} – проекция вектора ускорения первого тела на ось Y;
a_{x_2} – проекция вектора ускорения второго тела на ось X;
a_{y_2} – проекция вектора ускорения второго тела на ось Y;
r_1 – радиус первого тела;
r_2 – радиус второго тела.

Из (19) видно, что дальнейшее раскрытие квадратов, сокращение, вынос t за скобки, перенос из правой части в левую приведёт к тому, что уравнение примет вид (20).

$$P(t) = 0 \tag{20}$$

где $P(t)$ – многочлен от t.

Значит, уравнение алгебраическое одной переменной. Теперь найдём степень уравнения (т.е. максимальную степень одночлена). Возьмём одночлен $\frac{a_{x_1}t^2}{2}$, который является частью многочлена, который возведён в квадрат. Следовательно, в результирующем многочлене будет присутствовать одночлен (21).

$$(\frac{a_{x_1}t^2}{2})^2 = \frac{{a_{x_1}}^2t^4}{4} \tag{21}$$

Как видно, в результирующем многочлене t возведён в 4 степень. При этом в результирующем уравнении этот одночлен не сократится потому что у других подобных одночленов другие коэффициенты. Значит, в результирующем уравнении будет присутствовать одночлен степени 4.

Так как для каждого случая подстановки формул (17) и (18) в уравнения (4), (6), (8) придётся производить 8 таких выводов (для каждого из 2 случаев из (17) следует рассмотреть 2 случая из (18) при подстановке в (4) и по 2 случая при подстановке в (6) и (8)), проще составить программный алгоритм (который

описан позднее в 3.2.1), который применяя методы компьютерной алгебры, сможет строить выражения и вычислять их значения, подставляя переменные.

1.6 Решение алгебраических уравнений четвёртой степени

Так как для работы создаваемого физического движка не требуется обнаруживать время столкновения без погрешности и корни могут быть только действительными, можно воспользоваться численными методами. К тому же, аналитические методы решения алгебраических уравнений четвёртой степени обладают громоздкостью. И при этом, предложенный далее метод численного решения алгебраических уравнений позволяет решать уравнения любой степени, что позволит теоретически внести в модель и реализацию, без существенного переписывания кода, например разноускоренное движение (т.е. величину рывок, от которой зависит ускорение) и тогда уравнение, подобное (19), получится уже пятой степени, что не имеет решений в радикалах вовсе по теореме Абеля о неразрешимости уравнений в радикалах [5, с. 112].

1.6.1 Метод бисекции

Метод бисекции или метод деления отрезка пополам – простейший численный метод для решения нелинейных уравнений вида $f(x) = 0$, является частным случаем бинарного поиска [7].

Метод бисекции позволяет найти корень функции на отрезке $[x_l, x_r]$, если $sign(f(x_l)) \neq sign(f(x_r))$. При этом, если функция $f(x)$ не монотонна, она имеет несколько корней и метод найдёт лишь один корень [7]. Поэтому, метод бисекции не подходит для общего случая поиска решений алгебраических уравнений без предварительной подготовки, которая описана в пункте 1.6.2.

При этом можно обобщить этот метод для поиска корней на промежутках вида $(-\infty, x_r]$, $[x_l, +\infty)$. Недостающую границу поиска можно подобрать кратным (например, с каждой итерацией увеличивать изменение в 2 раза) увеличением (в случае бесконечности справа) или уменьшением (в случае бесконечности слева) относительно известной границы до того момента, пока знак функции на левой и правой границе не станет разным. Но при этом надо учесть, что корня на промежутке может не быть в случае когда при

изменении (увеличении или уменьшении, как указано ранее) неизвестной границы, значение функции не становится ближе к нулю.

Описание метода: на каждой итерации берётся середина отрезка (22).

$$x_m = \frac{x_l + x_r}{2} \tag{22}$$

где x_m – середина отрезка $[x_l, x_r]$.

Теперь следует вычислить значение функции в середине отрезка и если (23), то корень (равный x_m) найден; иначе следует разбить отрезок $[x_l, x_r]$ на два отрезка $[x_l, x_m]$ и $[x_m, x_r]$.

$$|f(x_m)| \leqslant \varepsilon \tag{23}$$

где ε – заданная точность по оси Y.

Далее, если x_l и x_m имеют разный знак, провести итерацию с отрезком $[x_l, x_m]$. Иначе, разный знак должен быть у x_m и x_r, значит провести итерацию с отрезком $[x_m, x_r]$.

1.6.2 Описание реализованного метода

Допустим, нам известны точки экстремума (т.е. локальные минимумы и максимумы) функции $P(x)$. Тогда, можно получить промежутки возрастания и убывания функции. Если на концах этих промежутках функция принимает разный знак, значит она имеет один корень и его можно найти методом бисекции, описанном в 1.6.1.

Но как получить экстремумы функции? Т.к. возрастание меняется на убывание (или наоборот), значит производная функции меняет знак и в точки экстремума имеет значение 0. Значит, надо найти корни производной. Так как мы работает с многочленом, производная многочлена имеет степень на единицу меньше. Поэтому можно применять рекурсивно применять этот же метод, пока степень не уменьшиться до того как уравнение можно решить аналитическим методом, например линейное или квадратное.

1.7 Обработка ударов

Формула расчёта скорости удара двух движущихся объектов на двухмерной плоскости в без-угловом виде выглядит следующим образом (24), согласно [6]:

$$\vec{v_1'} = \vec{v_1} - \frac{2m_2}{m_1 + m_2} \frac{\langle \vec{v_1} - \vec{v_2}, \vec{r_1} - \vec{r_2} \rangle}{|\vec{r_1} - \vec{r_2}|^2} (\vec{r_1} - \vec{r_2}) \tag{24}$$

где $\vec{v_1'}$ – вектор скорости первого тела после удара;

$\vec{v_1}$ – вектор скорости первого тела до удара;

$\vec{v_2}$ – вектор скорости второго тела до удара;

$\vec{r_1}$ – радиус-вектор положения первого тела;

$\vec{r_2}$ – радиус-вектор положения второго тела;

m_1 – масса первого тела;

m_2 – масса второго тела.

Для вычисление скорости второго тела, надо поменять местами нижние индексы, обозначающие первое и второе тело.

Эту же формулу можно и применить для обработки ударов с точками и линиями. Достаточно принять в бесконечность в качестве массы второго тела, нулевой вектор в качестве скорости и точку соприкосновения в качестве радиус-вектора положения второго тела.

1.8 Процесс моделирования

Модель можно реализовать как стейт-машину, которая принимает на вход текущую модель и действие и возвращает новую модель. Действие может быть, например, таким:

- придание скорости телу;
- добавление тела, или точки, или линии;
- удаление тела, или точки, или линии;
- изменение параметров тела;
- изменение глобальных параметров (ускорение свободного падения).

Подробности реализации описаны в пункте 3.2.

2 Проектирование

Кроме создания физического движка, важно продемонстрировать его работу. Желательно в интерактивном режиме. В XXI веке сложилась ситуация, что веб-приложения обладают преимуществом, таким что для того чтобы им воспользовался пользователь, достаточно браузера и просто перейти по ссылке. Поэтому, решено сделать Single Page Application (т.е. веб-приложение, которое загружает только один веб-документ, а затем обновляет содержимое тела этого документа с помощью JavaScript API, когда необходимо показать другое содержимое [8]), которое в интерактивном режиме отображает состояние модели (описанной в пункте 1.3), меняющееся с течением времени.

Так как создаваемый движок будет способен работать в реальном времени, существует возможность создать многопользовательский режим, для которого потребуется серверная часть, на которую перенесётся работа движка, а клиенты будут получать лишь изменения в модели (подробнее в 3.4.1). Соответственно, браузерное интерактивное SPA приложение с графическим интерфейсом назовём клиентской часть.

Так как мы хотим, чтобы движок работал и на клиентской части, и на серверной, следует выбрать язык программирования, который позволяет создавать и браузерные приложения, и нативные.

При этом для того чтобы сделать веб-приложение интерактивным, браузеры могут работать лишь с JavaScript с возможность подключения WebAssembly-модулей. WebAssembly – относительно новый способ создавать приложения на языке, отличным от JavaScript; он представляет из себя байткод стековой машины, который может быть получен из множества языков [9], например C/C++ [10], Rust [11], C# [12], F# [13] и т.д. [14].

Однако существует и другой способ писать на языке отличном от JavaScript – это использования компилятора, который исходный код на нужном языке компилирует в код на языке JavaScript, пригодный для исполнения в браузере. Поэтому существуют языки программирования, пригодные для исполнения и в браузере, и на сервере, что приводит к переиспользованию кода и уменьшении времени разработки. Примеры таких языков: OCaml (подробнее в пункте 2.2.1), F# [15, с. 48], TypeScript и т.д. [16].

При выборе языка программирования важно руководствоваться не только его кроссплатформенностью (в данном случае кроссплатформенностью можно назвать возможность работать и в браузере, и на сервере), но ещё и опытом работы с ним; стоит обратить внимание на выразительность, способность к обнаружению ошибок, производительность, экосистему. В качестве такого языка программирования выбран OCaml. OCaml является функциональным языком программирования, что можно рассматривать как преимущество при использовании как инструмента для решения задач численного моделирования [17].

2.1 Язык программирования OCaml

OCaml (до 2011 – Objective Caml [18]) – промышленный язык программирования общего назначения, в котором особое внимание уделяется выразительности и надёжности [19]. Обладает мультипарадигменностью, но в первую очередь преподноситься как функциональный язык программирования.

Ключевые достоинства и черты OCaml, согласно [20, с. 3] и [21]:

- **выразительность**. на OCaml можно создавать более компактные, простые и понятные системы, чем на таких языках, как Java или C#;

- **обнаружение ошибок**. Существует удивительно широкий круг ошибок, против которых система типов эффективна, включая многие ошибки, которые довольно трудно обнаружить с помощью тестирования;

- **производительность**. Производительность OCaml находится на одном уровне или лучше, чем у Java, и на расстоянии вытянутой руки от таких языков, как C или C++. В добавок к высококачественной генерации нативного кода, OCaml имеет инкрементальный сборщик мусора, который может быть настроен на выполнение небольших фрагментов работы за раз, что делает его более подходящим для приложений мягкого реального времени, таких как трейдинг;

- сборка мусора для автоматического управления памятью, которая сейчас является общей чертой современных языков высокого уровня;

- функции первого порядка, которые можно передавать как обычные значения, как в JavaScript, Common Lisp и C#;

- статическая типизация увеличивает производительность и уменьшает число ошибок во время исполнения, как в Java или C#;

- параметрический полиморфизм, позволяющий создавать абстракции, работающие с различными типами данных, подобно джереникам из Java, Rust, C# или шаблонам из C++;

- хорошая поддержка иммутабельного программирования, т.е. программирования без деструктивных обновлений в структурах данных. Такой подход представлен в традиционных функциональных языках программирования, таких как Scheme, а так же часто встречается во всём от распределенных фреймворков для работы с большими данными до UI-тулкитов;

- вывод типов, который позволяет не указывать тип каждой переменной в программе; Вместо этого, типы выводятся из того, как используется значение. В мейнстримных языках такое есть на уровне локальных переменных, например ключевое слово «var» в C# или ключевое слово «auto» в C++.

- алгебраические типы данных и паттерн-матчинг позволяют определять и манипулировать сложными структурами данных; Так же доступны в Scala, Rust, F#.

2.2 Обзор экосистемы языка OCaml

Под экосистемой можно подразумевать совокупность пакетного менеджера, системы сборки, компилятора/интерпретатора, библиотек, фреймворков и т.д. В XXI веке экосистемы языков программирования развиваются под действием opensource-сообщества. Качество и размер экосистемы могут зависеть от размеров сообщества. Банально, чем больше разработчиков – тем больше библиотек. OCaml не является мейнстримным языком, т.е. его сообщество относительно мало. Соответственно, и размер экосистемы мал. Для сравнения: количество пакетов (на момент мая 2022), доступных через пакетный менеджер opam для OCaml: 3860 единиц [22]; через npm для JavaScript: 1,97 млн [23]. Несмотря на такой разброс, в экосистеме OCaml достаточно нужного для выполнения проекта. Все рассмотренные далее библиотеки распространяются с открытым исходным кодом.

2.2.1 Компиляторы OCaml в JavaScript

Существует мнение, что JavaScript это Lisp в одежде Си: несмотря на си-подобный синтаксис, делающий его похожим на обычный процедурный язык, он больше имеет общего с функциональными языками, такими как Lisp или Scheme, а не с Си или Java [24]. Благодаря такой семантической схожести и потребности разработчиков, которые пишут на функциональных языках писать для браузера, и появляются компиляторы в JavaScript. Примеры таких компиляторов или диалектов для функциональных языков помимо OCaml: Fable (F#), ClojureScript (Clojure), RacketScript (Racket), ghcjs (Haskell). Рассмотрим такие компиляторы для OCaml в порядке появления.

Ocamljs. Ранее существовал компилятор, который для генерации JavaScript использовал внутреннее «lambda-представление» компилятора OCaml [25]. Так как такое представление является нестабильным (т.е. меняется от релиза к релизу языка OCaml) [26], такой подход серьёзно усложняет поддержку последних версий языка. В отличие от подхода js_of_ocaml, который кроме того, помогает добиться большей производительности [27, с. 13].

Js_of_ocaml. Использует иной подход, основанный на том, что компилятор OCaml может генерировать кроме нативного кода, ещё и байткод (генерацию и интерпретацию байткода можно сравнить с тем как работают платформы Java И .NET – главное различие, что там используется ещё и JIT-компиляция), что позволяет интегрироваться с текущей экосистемой[27, с. 1].

ReScript (ранее – BuckleScript). Использует подобный подход, что и ocamljs [28], но с прицелом на читаемый получаемый код и интеграцией в существующую JavaScript-экосистему [29]. При этом у ReScript своя система сборки, а так же устаревшая версия OCaml, что не даёт полноценно разрабатывать фуллстек приложения на OCaml. Решить эту проблему может форк под название Melange [30], но он находится в состоянии разработки.

Ещё один подход – использовать WebAssembly при работе с OCaml, однако инструментарий для такого подхода сейчас находиться в зачаточном состоянии [31].

2.2.2 Стандартные библиотеки

Встроенная стандартная библиотека OCaml отличается тем, что её фундаментальная роль служить для самогенерации (bootstrapping) компилятора, поэтому она небольшая и портативная, что делает её не совсем инструментом общего назначения [21]. Поэтому для её замены есть библиотеки Base, Core, Batteries, Containers [32].

Base – разработанная внутри Jane Street и протестированная в деле реализация стандартной библиотеки [32]. Core – это надстройка для Base, которая значительно расширяет её функциональность [33]. Библиотека для построения пользовательского интерфейса, выбранная в 2.2.4, уже имеет в своих зависимостях Core, поэтому желательно не использовать другую стандартную библиотеку чтобы не создавать лишних зависимостей.

2.2.3 Библиотеки конкурентного программирования

В своей книге [34] Я. Мински, А. Мадхавапедди, Дж. Хикки отметили, что: «Логика работы программ, взаимодействующих с внешним миром, часто предполагает ожидание: ожидание щелчка мышью, ожидание завершения операции чтения данных с диска или ожидание освобождения места в выходном сетевом буфере. Даже в не самых сложных интерактивных приложениях с успехом можно использовать приемы конкурентного программирования, например для ожидания наступления нескольких событий или немедленной реакции на первое наступившее событие. Один из подходов к организации конкурентного выполнения заключается в использовании системных потоков выполнения. Данный подход является доминирующим в таких языках программирования, как Java или C#. В этой модели для каждой задачи, которой может потребоваться приостановиться в ожидании некоторого события, выделяется отдельный поток выполнения, который можно заблокировать, не останавливая работу самой программы. Другой подход применяется в однопоточных программах и заключается в использовании цикла событий, в рамках которого реализуется реакция на внешние события, такие как тайм-ауты или щелчки мышью, путем вызова функций, специально зарегистрированных для этого. Этот подход

часто используется в языках программирования, подобных языку JavaScript, имеющему однопоточную среду выполнения, а также во многих библиотеках поддержки графического интерфейса пользователя. Каждый из данных механизмов имеет собственные достоинства и недостатки. Система потоков требует значительного объема памяти и других системных ресурсов. Кроме того, операционная система может произвольно прерывать выполнение одного потока, чтобы передать управление другому, что требует от программиста проявлять особую осторожность и защищать совместно используемые ресурсы с применением блокировок и условных переменных, вследствие чего легко допустить ошибку. Однопоточные системы, управляемые событиями, с другой стороны, в каждый момент времени решают только одну задачу и не требуют применения сложных механизмов синхронизации. Однако перевернутая организация программ, управляемых событиями, часто приводит к тому, что вам нередко приходится прибегать к весьма неуклюжим уловкам, чтобы обеспечить некоторое подобие конкурентного выполнения в цикле событий, что может завести в труднопроходимый лабиринт разнообразных ситуаций обработки событий.»

При этом асинхронность в JavaScript работает так же через цикл событий (event loop). Это тоже роднит семантику OCaml с семантикой JavaScript, как указано в пункте 2.2.1.

Основные библиотеки – Lwt и Async. Lwt («lightweight threads» – легковесные потоки) позволяет писать программы с участием потоков в монадическом стиле, что позволяет писать асинхронный код как обычный OCaml-код [35, с. 1]. Async создавалась с оглядкой на Lwt, но компании Jane Street, которая создала Async, требовалась другая обработка ошибок, более лучший контроль над параллелизмом [36]. Но при этом, эти две библиотеки не совместимы между собой, т.е. один проект не может полноценно использовать обе библиотеки вместе полноценно [37]. Из-за этого в экосистеме существует деление среди библиотек, использующих асинхронность – часть библиотек используют Lwt, часть Async. Получается, следует учитывать что одни библиотеки и фреймворки могут быть не совместимы с другими.

2.2.4 Библиотеки для построения пользовательского интерфейса

Рассмотрим бибилиотеки для построения пользовательского интерфейса, совместимые с js_of_ocaml.

jsoo-react. Предоставляет биндинги для JavaScript-библиотеки React, что позволяет использовать React в коде на OCaml [38].

ocaml-vdom. Реализует Elm-архитектуру – функциональный путь для описания пользовательского интерфейса. Отличается тем, что алгоритм обнаружения изменений в виртуальном DOM, не создаёт структуры данных, представляющие изменения, а применяет их к реальному DOM «на лету» [40].

virtual_dom. Предоставляет биндинги для JavaScript-библиотеки virtual-dom, позволяющей производить полноценные операции обнаружения изменений на виртуальном DOM и т.д. [41]. Часто не используется непосредственно, а используется через нижеописанные библиотеки.

incr_dom. Используя библиотеку Incremental для построения самокорректирующихся вычислений, или вычислений, которые могут быть эффективно обновлены при изменении их входных данных, и вышеописанную virtual_dom, позволяет создавать отзывчивые веб-интерфейсы [33].

bonsai. Расширяет библиотеку incr_dom, ввода концепцию компонента [42], представляющих из себя чисто функциональные стейт-машины, из которых можно составлять композицию [43].

Вообще, использование виртуального DOM нужно так как при взаимодействии с SPA, JavaScript разбирает и выстраивает новый UI с помощью DOM API, при этом обновлять или изменять новые элементы относительно легко, но процесс вставки новых элементов идёт крайне медленно [44, с. 72]. Используя виртуальный DOM можно не взаимодействовать с DOM напрямую. Преимущество incr_dom и bonsai в том, что благодаря инкрементальным вычислениям, процесс обнаружения изменений в виртуальном DOM может иметь быстрее [45]; а концепция композиции стейт-машин библиотеки bonsai,хорошо ложиться на разрабатываемый проект потому что и создаваемый движок является стейт-машиной (согласно пункту 1.8).

2.2.5 Веб-фреймворки

Для взаимодействия сервера и клиента требуется обмен сообщений в реальном времени, что лучше всего реализовать с помощью WebSocket [**?**]. Можно рассмотреть библиотеку async_rpc_websocket, которая позволяет описывать протоков удалённого вызова процедур, однако для своей работы она использует библиотеку Async, что нежелательно.

Поэтому можно использовать веб-фреймворк, поддерживающий технологию WebSocket и реализовать обмен сообщений вручную. Самым современным таким фреймворком является dream, который отличается идеоматичным для функциональных языков API: HTTP-обработчики представлены функциями, которые можно композировать в последовательном стиле (умножение), или в альтернативном (сложение), что можно интерпретировать как алгебраическое кольцо [47].

2.2.6 Библиотеки для тестирования

ppx_inline_test. Позволяет специальным синтаксисом создавать тесты для проверки конкретных свойств прямо в коде создаваемых библиотек [48].

ppx_expect. В предыдущем абзаце упомянут подход, в основном проверяются некоторые конкретные свойства в определенном сценарии. Иногда, однако, хочется не проверить то или иное свойство, а зафиксировать и сделать видимым поведение кода. Expect-тесты позволяют сделать именно это. [48]

2.2.7 Библиотеки сериализации и десериализации

Во многих языках программирования (де)сериализация построена на том, что информация о типе значения доступна во время выполнения программы и с помощью рефлексии можно получить например информацию о, например названии поля объекта и его значение и записать их в сериализуемый результат. В OCaml, напротив, во время исполнения отсутствует всякое представление о типе, т.е. например значение «0» типа «int» будет иметь то же самое представление в памяти что и значение «None» типа «option» или значение «false» типа «bool» [49].

Поэтому (де)сериализация в OCaml построена на другом принципе – должна быть определена функция которая принимает сериализуемый объект и возвращает сериализованное представление (в случае сериализации), или принимает сериализованное представление и возвращает десериализованный объект (в случае десериализации). Такие функции программист может определить вручную, но это утомительно и при каждом структуры типа данных, требуется изменять код функций. Решить эту проблему помогают препроцессорные расширения для OCaml, позволяющие на этапе компиляции из определения типа данных сгенерировать функции для десериализации и сериализации. Основные такие библиотеки, предоставляющие сериализованное представление и препроцессорные расширения, являются Yojson (для JSON [50]) и Sexplib (для S-выражений [51]).

Использование S-выражений является более традиционным в функциональных языках программирования и их уже используют выбранные библиотеки Core и Bonsai, поэтому для консистентности принято решение не использовать JSON в проекте.

2.2.8 Среда разработки и система сборки

После многолетней запутанности в системах сборки OCaml, стала стандартом де-факто система сборки Dune; подобное можно сказать про пакетный менеджер Opam [52].

Поддерживается интегрированная среда разработки Visual Studio Code через плагин OCaml Platform [53].

2.3 CSS-фреймворк

Для оформления пользовательского интерфейса требуется CSS-фреймворк, обеспечивающий отзывчивую вёрстку (responsive layout) – колонки, контейнеры и т.д.; готовые компоненты – кнопки, таблицы, формы и т.д. Самым известным таким фреймворком является Bootstrap, однако выбор сделан в пользу Bulma, который отличается оригинальностью и простотой [54].

2.4 Выбор библиотек и подходов

Окончательный перечень используемых средств разработки таков:

- OCaml – язык программирования;
- Js_of_ocaml – компилятор OCaml в JavaScript;
- Lwt – библиотека для конкурентного программирования;
- Core – стандартная библиотека;
- Dream – web-фреймворк;
- ppx_inline_test, ppx_expect – библиотеки юнит-тестирования;
- Sexplib – библиотека для сериализации и десериализации S-выражений;
- Bulma – CSS-фреймворк;
- Dune, opam – система сборки и пакетный менеджер;
- VS Code, OCaml Platform – среда разработки и плагин для работы с OCaml.

3 Программная реализация

Здесь будет третья глава. TODO

3.1 Решатель алгебраических уравнений

TODO

3.2 Движок

TODO

3.2.1 Символьные вычисления

TODO

3.3 Клиентская часть, одиночный режим

TODO

3.4 Серверная часть

3.4.1 Получение отличий модели

TODO

3.5 Клиентская часть, многопользовательский режим

TODO

4 Примеры использования и перспективы

TODO

4.1 Визуализация броуновского движения

TODO

4.2 Получение числа π методом Гальперина

TODOhttps://habr.com/ru/post/533454/

4.3 Перспективы и дальнейшее возможное развитие

4.3.1 Расширение возможностей многопользовательского режима

TODOСохранение реплеев на сервере, чат.

4.3.2 Клон игры «Смешарики» (может быть известна как «Чапаев»)

TODOhttps://shararam.fandom.com/wiki/Смешарики_(игра)

4.3.3 Обобщение TODO

TODOВоздействие нескольких сил, неупругие столкновения, применение действий при взаимодействии или даже трёхмерное пространство и т.д.

4.3.4 Формальная верификация частей алгоритма

TODO

5 Заключение

Здесь будет заключение. TODO

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Критика чистого разума Сочинения в шести томах. Том 3 Академия наук СССР Институт философии Издательство социально-экономической литературы «Мысль» Москва - 1964. TODO

2. Роуэлл, Г. Физика : учебное издание / Г. Роуэлл, С. Герберт. – Москва : Просвещение, 1994. – 576 с. – ISBN 5-09-002920-2.

3. Math for Progammers TODOhttps://www.manning.com/books/math-for-programmers

4. Larson R., Hostetler R. . Precalculus: A Concise Course. — Boston: Houghton Mifflin, 2007. — xvii + 526 + 102 p. — ISBN 0-618-62719-7. TODO

5. Алексеев, В. Б. Теорема Абеля в задачах и решениях. — М.: МЦНМО, 2001. — 192 с. — ISBN 5-900916-86-3. TODO

6. https://en.wikipedia.org/wiki/Elastic_collision TODO

7. TODOAutar K Kaw Numerical Methods with Applications Chapter 03.03 Bisection Method

8. https://developer.mozilla.org/en-US/docs/Glossary/SPA TODO

9. Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the web up to speed with WebAssembly. In Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017). Association for Computing Machinery, New York, NY, USA, 185–200. https://doi.org/10.1145/3062341.3062363 TODO

10. https://emscripten.org/docs/introducing_emscripten/about_emscripten.html TODO

11. https://rustwasm.github.io/docs/book/why-rust-and-webassembly.html TODO

12. https://docs.microsoft.com/ru-ru/aspnet/core/blazor TODO

13. https://fsbolero.io/docs/ TODO

14. https://webassembly.org/getting-started/developers-guide/ TODO

15. https://fsharp.org/history/hopl-final/hopl-fsharp.pdf Don Syme. 2020. The early history of F#. Proc. ACM Program. Lang. 4, HOPL, Article 75 (June 2020), 58 pages. https://doi.org/10.1145/3386325 TODO

16. Майоров, А. TypeScript для PHP-разработчика. Как писать на JavaScript большие приложения и не сойти с ума / А. Майоров // Системный администратор. – 2015. – № 7-8(152-153). – С. 95-99. – EDN UBPAGH.

17. Шутов, В. С. Функциональное программирование для решения математических задач / В. С. Шутов // Фундаментальные исследования основных направлений технических и физико - математических наук : сборник статей Международной научно-практической конференции, Челябинск, 01 июня 2018 года. – Челябинск: Общество с ограниченной ответственностью ”Аэтерна”, 2018. – С. 120-122. – EDN XPWRYL.

18. https://caml.inria.fr/about/history.en.html TODO

19. https://ocaml.org/ TODO

20. Yaron Minsky. OCaml for the Masses. ACM Queue, Sep 27, 2011 TODO

21. Real World OCaml. 2nd Edition. Yaron Minsky. Anil Madhavapeddy. https://dev.realworldocaml.org/prologue.html TODO

22. https://opam.ocaml.org/ TODO

23. https://www.npmjs.com/ TODOРежим доступа: для зарегистрированных пользователей.

24. https://www.crockford.com/javascript/javascript.html

25. https://jaked.org/ocamljs/Jscomp.html TODO

26. https://dev.realworldocaml.org/compiler-backend.html TODO

27. https://www.irif.fr/ balat/publications/vouillon_balat-js_of_ocaml.pdf TODO

28. https://github.com/ocsigen/js_of_ocaml/issues/338 TODO

29. https://rescript-lang.org/docs/manual/latest/introduction TODO

30. https://anmonteiro.com/2021/03/on-ocaml-and-the-js-platform/ TODO

31. https://okcdz.medium.com/run-ocaml-in-the-browser-by-webassembly-31ce464594c6 TODO

32. https://ocamlverse.github.io/content/standard_libraries.html TODO

33. https://opensource.janestreet.com/ TODO

34. Мински Я., Мадхавапедди А., Хикки Дж. М57 Программирование на языке OCaml / пер. с анг.л А. Н. Киселева. – М.: ДМК Пресс, 2014. – 536 с.: ил. ISBN 978-5-97060-102-0 TODO

35. Jerome Vouillon Lwt: a Cooperative Thread Library TODOhttps://www.irif.fr/~vouillon/publi/lwt.pdf

36. https://blog.janestreet.com/announcing-async/ TODO

37. http://rgrinberg.com/posts/abandoning-async/ TODO

38. https://github.com/ml-in-barcelona/jsoo-react TODO

39. https://guide.elm-lang.org/architecture/ TODO

40. https://github.com/LexiFi/ocaml-vdom TODO

41. https://github.com/janestreet/virtual_dom TODO

42. https://github.com/janestreet/bonsai/blob/master/docs/blogs/history.md TODO

43. https://opensource.janestreet.com/bonsai/ TODO

44. А. Бэнкс, Е. Порселло. React и Redux: функциональная веб-разработка ISBN 978-5-4461-0668-4 TODO

45. https://www.youtube.com/watch?v=R3xX37RGJKE TODO

46. https://developer.mozilla.org/ru/docs/Web/API/WebSocket TODO

47. https://aantron.github.io/dream TODO

48. https://dev.realworldocaml.org/testing.html

49. https://dev.realworldocaml.org/runtime-memory-layout.html TODO

50. https://dev.realworldocaml.org/json.html TODO

51. https://dev.realworldocaml.org/data-serialization.html TODO

52. https://www.infoq.com/presentations/ocaml-browser-iot/ TODO

53. https://dev.realworldocaml.org/platform.html TODO

54. https://bulma.io/alternative-to-bootstrap/ TODO