РАЗРАБОТКА ПРОГРАММЫ НА ЯЗЫКЕ RUST ДЛЯ МОДЕЛИРОВАНИЯ ТРАЕКТОРИИ ПОЛЁТА ОБЪЕКТА С УЧЁТОМ ИЗМЕНЯЮЩИХСЯ ПАРАМЕТРОВ

Джахед Рахмани, *старший преподаватель*
И.С. Александров, *студент*
Московский технический университет связи и информатики
(Россия, г. Москва)

DOI:10.24412/2500-1000-2025-3-1-242-252

Аннотация. *Данная статья представляет разработку программного обеспечения для моделирования траектории полёта объекта на языке программирования Rust с учётом изменяющихся параметров. Описана реализация алгоритмов расчёта траектории, проверки пересечений с регионами и вычисления ускорения с учётом изменяющихся параметров, таких как лобовое сопротивление и гравитационное ускорение. Было продемонстрировано практическое применение программы на конкретных примерах. Программа демонстрирует высокую производительность и обладает широким спектром применения, включая научные исследования, где она может быть использована для изучения динамики движения тел в различных условиях и сценариях различной сложности, а также прикладные области, такие как разработка видеоигр, где она позволяет реализовывать реалистичное моделирование физического взаимодействия объектов. В заключении рассмотрены преимущества и ограничения системы, предложены направления для дальнейшего развития, а также подведены итоги, подчеркивающие значимость и потенциал данного программного обеспечения.*

Ключевые слова: *расчёт траектории, лобовое сопротивление, гравитационное ускорение, движение тел, язык программирования Rust, разработка компьютерных программ.*

Технологии моделирования и прогнозирования траекторий движения объектов играют ключевую роль в различных областях, включая аэрокосмическую отрасль, где они применяются для расчёта траекторий полёта космических тел [1], таких как первичный расчёт траектории взлёта космических аппаратов, индустрию видеоигр, где обеспечивают реалистичное моделирование физики движения и падения объектов, а также расчёт баллистики игровых снарядов, и научные исследования, где служат инструментом для изучения динамики движения тел в разнообразных условиях. Кроме того, данные технологии активно используются при создании образовательных симуляторов, направленных на изучение физических законов. Одной из важнейших задач в данной области является разработка точных и эффективных систем моделирования траекторий полёта объекта, которые учитывают как воздействие переменных факторов окружающей среды, таких как плотность воздуха и гравитационное ускорение, так и влияние постоянных параметров, включая массу объекта и коэффициент аэродинамического сопротивления.

Современные задачи требуют все более точных и быстрых расчетов в реальном времени, что вынуждает использовать современные языки программирования, способные обеспечить высокую производительность и минимальные задержки. В последние годы язык Rust привлекает все больше внимания благодаря своей высокой производительности, а также эффективному и безопасному управлению памятью [2]. Эти преимущества делают Rust крайне перспективным языком программирования для разработки надежных и высокопроизводительных решений в области программного обеспечения. Rust позволяет избежать многих ошибок, связанных с управлением памятью, что особенно важно при работе с ресурсоемкими вычислениями. Именно поэтому Rust (rustc версии 1.86.0-nightly) был выбран для создания данной системы, что позволяет продемонстрировать его возможности и особенности в контексте решения сложных вычислительных задач.

В настоящей статье рассматривается программное обеспечение, предназначенное для моделирования траектории движущегося объекта с учетом множества факторов, часть из

которых может изменяться при попадании объекта в определённые регионы. Кроме того, система способна рассчитывать точку пересечения траектории объекта с препятствиями, представленными в виде границ регионов.

Анализ существующих решений

Разработанная программа для моделирования траекторий на языке Rust обладает рядом уникальных особенностей, которые отличают её от существующих решений в области моделирования физики и траекторий, таких как Unity Physics и Bullet. Ниже представлено краткое сравнение с указанными решениями.

Unity Physics представляет собой физический движок, интегрированный в игровой движок Unity, который применяется для моделирования физических процессов в реальном времени, включая движение объектов, столкновения и гравитацию. В отличие от программы, представленной в данной статье, Unity Physics ориентирован на универсальные задачи моделирования физики, тогда как разработанная программа специализируется на моделировании баллистических траекторий с учётом динамически изменяющихся параметров среды, таких как плотность воздуха и гравитационное ускорение, что не является приоритетом для Unity Physics. Кроме того, программа, описанная в статье, обладает меньшей ресурсоёмкостью и более высокой производительностью, поскольку не требует использования игрового движка.

Bullet представляет собой широко известную библиотеку для моделирования физики, которая применяется в игровой индустрии, симуляциях и научных исследованиях. Она поддерживает расчёты столкновений, динамику твёрдых и мягких тел. В отличие от программы, представленной в данной статье, Bullet не ориентирована на учёт динамически изменяющихся параметров среды, таких как плотность воздуха и гравитационное ускорение, что является ключевой особенностью разработанной программы. Кроме того, использование языка Rust обеспечивает более безопасное управление памятью, что минимизирует риск ошибок в процессе разработки.

Выбор фреймворка для визуального интерфейса

В рамках разработки программного обеспечения первостепенное внимание уделяется выбору инструментов для создания пользовательского интерфейса. В данной работе для реализации визуального интерфейса был использован фреймворк eframe (версии 0.11.7), который представляет собой официальный фреймворк, предназначенный для разработки приложений с использованием библиотеки egui. Библиотека egui предоставляет лёгкий подход к созданию интерфейсов, отличающихся отличной отзывчивостью, что делает её особенно привлекательной для систем реального времени, требующих минимальных временных задержек. Фреймворк eframe, предоставляет удобные абстракции для управления окнами и отрисовки пользовательского интерфейса на различных платформах. eframe интегрирует egui с библиотеками, отвечающими за управление окнами и графическую отрисовку.

Выбор фреймворка eframe обусловлен его отличной совместимостью с языком программирования Rust, поскольку как сам eframe, так и библиотека egui, полностью разработаны на Rust. Архитектура данного фреймворка полностью соответствует принципам языка Rust, обеспечивая безопасное и эффективное управление памятью. Кроме того, немаловажным фактором выбора eframe стала его поддержка кроссплатформенности, которая позволяет использовать разработанное программное обеспечение на различных операционных системах без необходимости внесения существенных изменений в исходный код.

Библиотека egui реализует графический пользовательский интерфейс в немедленном режиме (immediate mode), в отличие от множества библиотек, основанных на сохранённом режиме (retained mode).

В рамках немедленного режима интерфейс полностью перестраивается каждый цикл отрисовки, что исключает необходимость сохранения состояния виджетов между итерациями рендеринга. Это требует явного описания логики интерфейса на каждом проходе. В отличие от сохранённого режима, где реакция на события обеспечивается через привязку callback-функций, в немедленном режиме обработка событий осуществляется посредством проверки их наличия в каждом цикле обновления интерфейса.

Подобная архитектура обладает рядом преимуществ. Во-первых, она позволяет эффективнее использовать память, поскольку отсут-

ствует необходимость хранения состояния всех элементов интерфейса. Во-вторых, снижаются временные задержки, связанные с обработкой событий, что повышает отзывчивость приложения. Однако ключевым недостатком данного подхода является повышенное потребление процессорного времени, обусловленное необходимостью описания логики интерфейса и его полной отрисовкой на каждом цикле рендеринга.

Описание алгоритма программы

Далее будет рассмотрен и проанализирован принцип работы моделирования траектории объекта с учётом динамически изменяющихся факторов окружающей среды, таких как гравитационное ускорение и лобовое сопротивление, в рамках разработанного программного обеспечения.

Для упрощения структуры исходного кода и облегчения последующих вычислений были разработаны несколько структур данных, среди которых ключевой является структура, описывающая двумерный вектор. В языке Rust подобные структуры данных реализуются с использованием ключевого слова «struct». Для данной структуры реализованы методы, отвечающие за математические операторы. По мимо прочего эта структура используется для описания позиции и размеров регионов, что обеспечивает универсальность её применения в рамках программы. Реализация структур данных показана на листинге 1.

Листинг 1 – Структуры данных

```
#[derive(Clone, Copy)]
struct Vec2d {
    x: f64,
    y: f64,
}
impl Vec2d {
    fn new(x: f64, y: f64) -> Vec2d {
        Vec2d { x, y }
    }
    fn from_angle(angle: f64) -> Vec2d {
        Vec2d { x: angle.cos(), y: angle.sin() }
    }
}
impl Sub<Vec2d> for Vec2d {
    type Output = Vec2d;
    fn sub(self, rhs: Vec2d) -> Self::Output {
        Vec2d::new(self.x - rhs.x, self.y - rhs.y)
    }
}
impl Add<Vec2d> for Vec2d {
    type Output = Vec2d;
    fn add(self, rhs: Vec2d) -> Self::Output {
        Vec2d::new(self.x + rhs.x, self.y + rhs.y)
    }
}
impl Mul<f64> for Vec2d {
    type Output = Vec2d;
    fn mul(self, rhs: f64) -> Self::Output {
        Vec2d::new(self.x * rhs, self.y * rhs)
    }
}
impl Div<f64> for Vec2d {
    type Output = Vec2d;
    fn div(self, rhs: f64) -> Self::Output {
```

```
        Vec2d::new(self.x / rhs, self.y / rhs)
    }
}
struct PolyBox {
    pos: Vec2d,
    rotation: f64,
    size: Vec2d,
    density: f64,
    acceleration: f64,
    id: usize,
    vertices: [[f64; 2]; 4],
}
impl Default for PolyBox {
    fn default() -> Self {
        Self {
            pos: Vec2d::new(2.0, 0.0),
            size: Vec2d::new(1.0, 2.0),
            rotation: 0.0,
            density: 1.225,
            acceleration: 9.81,
            id: 0,
            vertices: [[0.0; 2]; 4],
        }
    }
}
struct Intersection<'a> {
    _x: f64,
    _y: f64,
    distance: f64,
    poly_box_info: &'a PolyBox,
}
```

Рассмотрим основной алгоритм расчёта траектории. При запуске программы и при каждом изменении значений в полях ввода данных активируется процедура перерасчёта траектории объекта.

Для начала программа запускает цикл, в котором перебираются все регионы. Для каждого региона вычисляются координаты вершин прямоугольника, которые затем сохраняются в соответствующую структуру данных. Эта структура добавляется в массив для последующего использования. Функция, которая отвечает за этот цикл представлена на листинге 2.

Листинг 2 – Функция расчёта координат вершин региона

```
fn get_boxes_poses(&mut self) {
    for poly_box in self.boxes.iter_mut() {
        let rad = poly_box.rotation.to_radians();
        let rad1 = rad + PI / 2.0;
        let vec1 = Vec2d::from_angle(rad1);
        let vec2 = Vec2d::from_angle(rad);
        let pos1 = poly_box.pos + vec1 * poly_box.size.y - vec2 * poly_box.size.x;
        let pos2 = poly_box.pos + vec1 * poly_box.size.y + vec2 * poly_box.size.x;
        let pos3 = poly_box.pos - vec1 * poly_box.size.y + vec2 * poly_box.size.x;
        let pos4 = poly_box.pos - vec1 * poly_box.size.y - vec2 * poly_box.size.x;
        poly_box.vertices = [
            [pos1.x, pos1.y],
            [pos2.x, pos2.y],
            [pos3.x, pos3.y],
            [pos4.x, pos4.y],
        ]
    };
}
```

На следующем этапе программа проверяет, находится ли начальная (нулевая) точка внутри одного или нескольких из заданных регионов. Для этого выполняется цикл, в рамках которого осуществляется перебор всех регионов. На каждой итерации цикла проводится проверка принадлежности точки региону, заданному в виде прямоугольника. Для этого проверяется положение точки относительно граней прямоугольника: если точка располагается по одну сторону от всех граней, это означает, что она находится внутри региона. Несмотря на существование альтернативных методов проверки принадлежности точки прямоугольнику, данный подход был выбран благодаря минимальным временным затратам на выполнение вычислений. На листинге 3 представлена функция проверки положение точки относительно отрезка.

Листинг 3 – Функция расчёта положение точки относительно отрезка

```
    fn get_denom(a0: f64, a1: f64, b0: f64, b1: f64, c0: f64, c1: f64, d0: f64, d1: f64) -> f64 {
   (a0 - a1) * (b0 - b1) - (c0 - c1) * (d0 - d1)
}
```

Информация о регионе, в котором находится точка, добавляется в динамический массив со всеми регионами, через которые сейчас проходит объект, который в языке программирования Rust реализован как структура данных типа «Vec».

На следующем этапе программа запускает основной цикл расчёта траектории. В рамках этого цикла сначала из массива регионов, через которые в данный момент проходит траектория объекта, извлекается первый регион. Его характеристики используются для последующих вычислений ускорения объекта. Затем рассчитываются обновлённые координаты объекта и его текущая скорость по двум осям. Для повышения точности расчётов был выбран метод Рунге-Кутты 4-го порядка [3, 4]. Формулы, используемые для расчёта ускорения и координат, представлены ниже:

$$\mathrm{k}of = \frac{\rho C_d A}{2m} \quad (1)$$

$$\frac{dv_x}{dt} = -\mathrm{k}of\sqrt{v_x^2 + v_y^2} * v_x \quad (2)$$

$$\frac{dv_y}{dt} = -\text{k}of\sqrt{v_x^2 + v_y^2} * v_y - g \quad (3)$$

$$\frac{dx}{dt} = v_x, \frac{dy}{dt} = v_y \quad (4)$$

Для расчёта коэффициентов k_1, k_2, k_3, k_4 используемых при вычислении ускорения, была разработана специализированная функция, реализация которой приведена на листинге 4.

Листинг 4 – Функция расчёта коофицента

```
 fn get_cur_state(kof: f64, down_acceleration: f64, x_vel: f64, y_vel: f64) -> (f64, f64) {
    let v = (x_vel.powi(2) + y_vel.powi(2)).sqrt();
    let x_acceleration = -kof * v * x_vel;
    let y_acceleration = -kof * v * y_vel - down_acceleration;
    (x_acceleration, y_acceleration)
}
```

После вычисления новой позиции объекта программа повторно активирует цикл, перебирающий все регионы. Для каждого региона выполняется проверка на пересечение отрезка, соединяющего предыдущую и обновлённую позиции объекта, с границами региона. Все грани, пересекаемые данным отрезком, добавляются в специальный динамический массив, который затем сортируется по возрастанию расстояния от предыдущей позиции объекта до точки пересечения с соответствующей гранью. Далее выполняется цикл по отсортированному массиву, в рамках которого осуществляется поиск региона, соответствующего грани, с которой произошло пересечение, в массиве регионов, через которые в данный момент проходит траектория объекта. Если регион присутствует в массиве, он удаляется из него, а в противном случае регион добавляется в этот массив.

На заключительном этапе цикла расчёта траектории выполняется проверка, пересекает ли текущая траектория ось абсцисс. В случае пересечения вычисляется точка пересечения, которая принимается в качестве конечной точки траектории, после чего основной цикл расчёта траектории завершается. Если пересечение с осью абсцисс не обнаружено, выполнение цикла продолжается для последующих итераций.

И наконец в массив добавляются координаты текущей точки и на этом завершается текущая итерация цикла расчёта траектории. После выполнения всех итераций программа получит полную траекторию объекта. На листинге 5 представлен код итерации цикла, в котором находится основной алгоритм расчёта траектории полёта объекта.

Листинг 5 – Тело цикла расчёта траектории

```
let old_x = x;
let old_y = y;
let old_x_vel = x_vel;
let (kof, down_acceleration) = match inside_boxes.first() {
        None => (self.default_density * self.cof * self.area / (2.0 * self.mass), self.default_acceleration),
        Some(&poly_box) => (poly_box.density * self.cof * self.area / (2.0 * self.mass),
poly_box.acceleration)
};
let (k1_vel_x, k1_vel_y) = get_cur_state(kof, down_acceleration, x_vel, y_vel);
let (k1_x, k1_y) = (x_vel, y_vel);
```

```
let (k2_vel_x, k2_vel_y) = get_cur_state(kof, down_acceleration, x_vel + self.time_step / 2.0 *
k1_vel_x, y_vel + self.time_step / 2.0 * k1_vel_y);
let (k2_x, k2_y) = (x_vel + self.time_step / 2.0 * k1_vel_x, y_vel + self.time_step / 2.0 * k1_vel_y);

let (k3_vel_x, k3_vel_y) = get_cur_state(kof, down_acceleration, x_vel + self.time_step / 2.0 *
k2_vel_x, y_vel + self.time_step / 2.0 * k2_vel_y);
let (k3_x, k3_y) = (x_vel + self.time_step / 2.0 * k2_vel_x, y_vel + self.time_step / 2.0 * k2_vel_y);

let (k4_vel_x, k4_vel_y) = get_cur_state(kof, down_acceleration, x_vel + self.time_step * k3_vel_x,
y_vel + self.time_step * k3_vel_y);
let (k4_x, k4_y) = (x_vel + self.time_step * k3_vel_x, y_vel + self.time_step * k3_vel_y);

(x_vel, y_vel, x, y) = (x_vel + self.time_step / 6.0 * (k1_vel_x + 2.0 * k2_vel_x + 2.0 * k3_vel_x +
k4_vel_x), y_vel + self.time_step / 6.0 * (k1_vel_y + 2.0 * k2_vel_y + 2.0 * k3_vel_y + k4_vel_y), x +
self.time_step / 6.0 * (k1_x + 2.0 * k2_x + 2.0 * k3_x + k4_x), y + self.time_step / 6.0 * (k1_y + 2.0 *
k2_y + 2.0 * k3_y + k4_y));

if (old_x_vel >= 0.0) != (x_vel >= 0.0) {
	x_vel = 0.0;
}
first_intersection.clear();
for polygon in self.boxes.iter() {
	let vertices = polygon.vertices;
	for [edge1, edge2] in [[vertices[0], vertices[1]], [vertices[1], vertices[2]], [vertices[2], verti-
ces[3]], [vertices[3], vertices[0]]] {
		let denom = get_denom(old_x, x, edge1[1], edge2[1], old_y, y, edge1[0], edge2[0]);
		if denom == 0.0 {
			continue;
		}
		let t = get_denom(old_x, edge1[0], edge1[1], edge2[1], old_y, edge1[1], edge1[0],
edge2[0]) / denom;
		let s = -get_denom(old_x, x, old_y, edge1[1], old_y, y, old_x, edge1[0]) / denom;

		if (0.0..=1.0).contains(&t) && (0.0..=1.0).contains(&s) {
			let intersection_x = old_x + t * (x - old_x);
			let intersection_y = old_y + t * (y - old_y);

			let distance = (intersection_x - old_x).powi(2) + (intersection_y - old_y).powi(2);
			first_intersection.push(Intersection { _x: intersection_x, _y: intersection_y, dis-
tance, poly_box_info: polygon });
			self.important_points.push([intersection_x, intersection_y]);
		}
	}
}
first_intersection.sort_by(|a, b| if a.distance < b.distance {Ordering::Less} else {Ordering::Greater});
for intersection in first_intersection.iter() {
	let i = inside_boxes.iter().position(|x: &&PolyBox| x.id == intersection.poly_box_info.id);
	if let Some(i) = i {
		inside_boxes.remove(i);
	} else {
		inside_boxes.push(intersection.poly_box_info);
	}
```

```
}
if y <= 0.0 {
        self.points.push([old_x + (x - old_x) * -old_y / (y - old_y), 0.0]);
        self.important_points.push([old_x + (x - old_x) * -old_y / (y - old_y), 0.0]);
        break
}
self.points.push([x, y]);
```

Примеры применения

Далее, рассмотрим несколько примеров использования программы для расчёта траектории.

Пример 1: Рассмотрим задачу расчёта траектории объекта массой 1 кг, коэффициентом сопротивления 1 и площадью поперечного сечения 1 м². Начальная скорость объекта составляет 15 м/с по обеим осям. Параметры окружающей среды заданы следующим образом: плотность воздуха – 1,225 кг/м³ (что соответствует плотности воздуха на Земле при стандартных условиях), гравитационное ускорение – 9,81 м/с² (приблизительно равное земному). Дополнительные регионы не учитываются. Результаты работы программы представлены на рисунке 1.

Пример 2: Рассмотрим расчёт траектории объекта с аналогичными параметрами массы, коэффициента сопротивления и площади поперечного сечения, а также с теми же стандартными параметрами окружающей среды. Однако в данном случае изменены начальные скорости, а также добавлены регионы, внутри которых плотность воздуха и гравитационное ускорение уменьшаются с увеличением высоты. Такие регионы могут воспроизводить различные слои атмосферы, что представляет значительную ценность для расчёта траекторий полёта космических аппаратов, учитывая изменение параметров среды на разных высотах. Для обеспечения корректности расчётов временной шаг был уменьшен. Результаты работы программы представлены на рисунке 2.

Пример 3: Рассмотрим моделирование траектории объекта с использованием стандартных параметров объекта и окружающей среды, но с добавлением региона, для которого задано отрицательное значение гравитационного ускорения. Данный пример демонстрирует возможности программы для моделирования сложных траекторий, включая нестандартные сценарии, что позволяет использовать алгоритм данной программы в разработке игр с акцентом на реалистичное и сложное физическое взаимодействие объектов. Результаты работы программы показаны на рисунке 3.

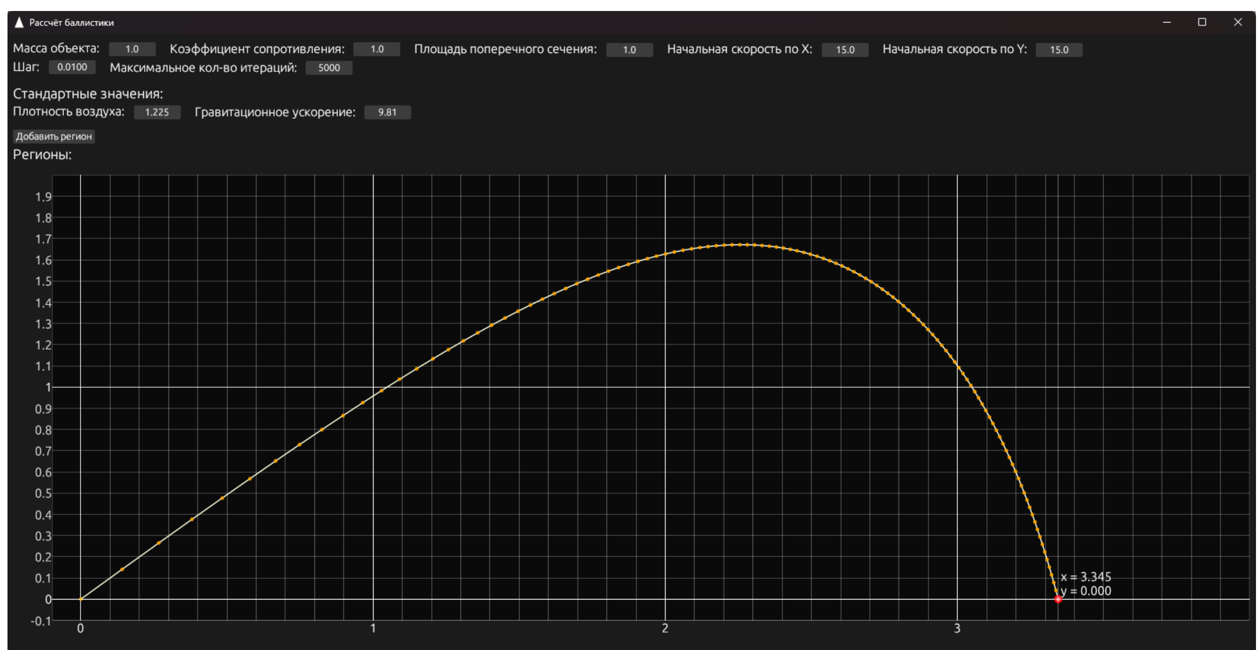

Рис. 1. Первый пример работы программного обеспечения

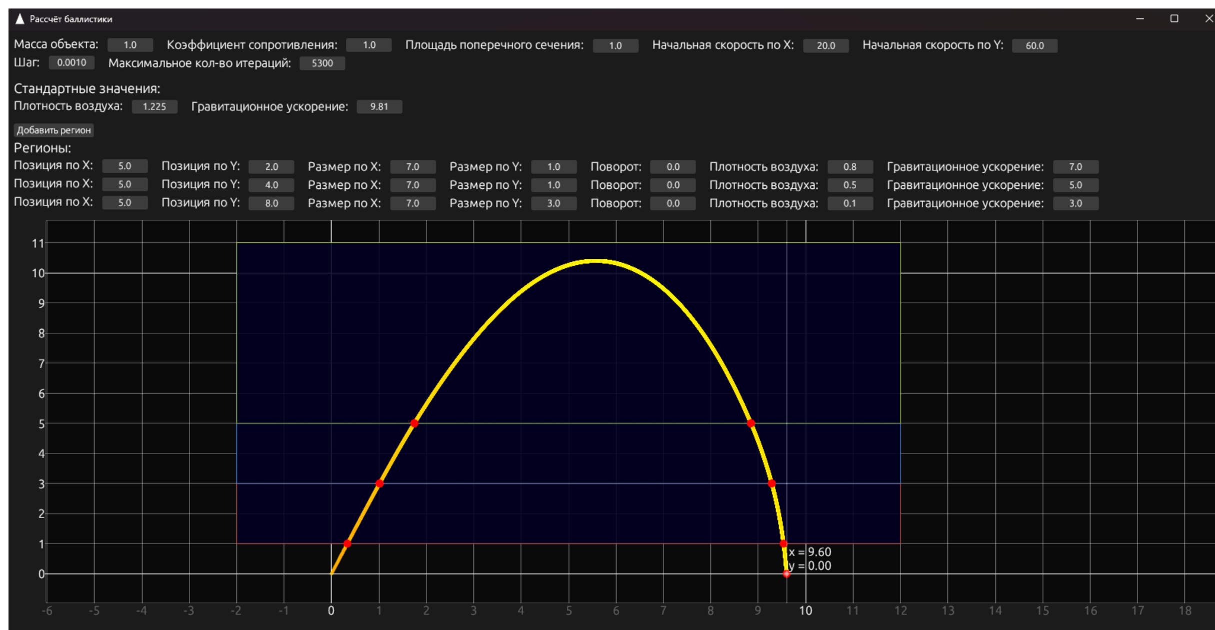

Рис. 2. Второй пример работы программного обеспечения

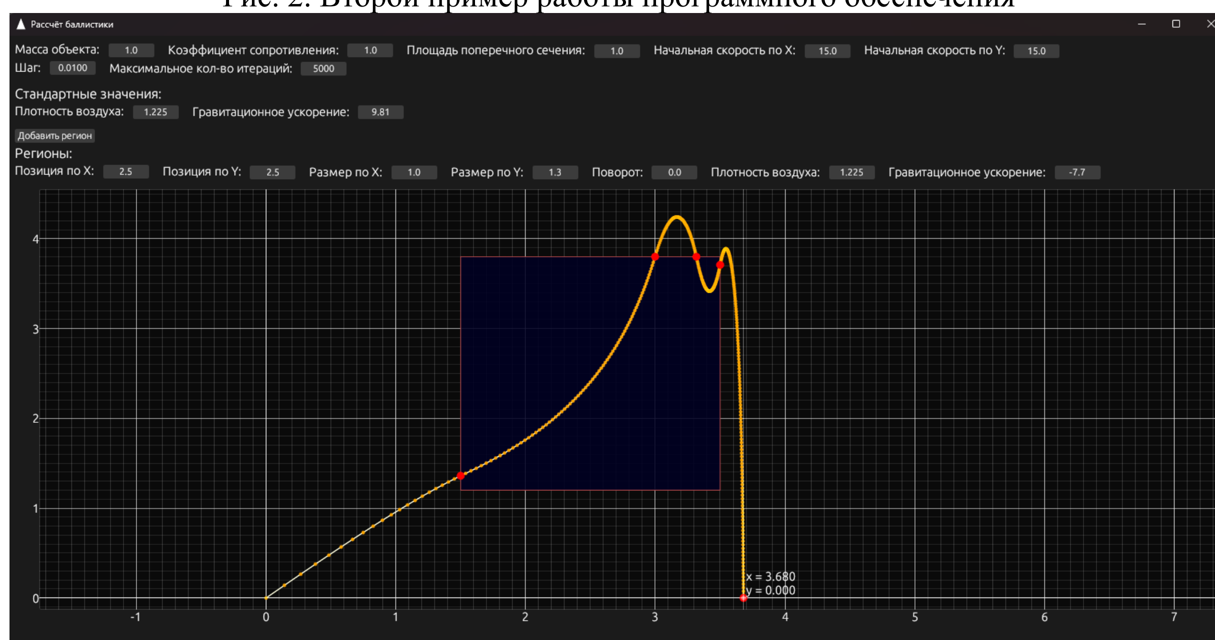

Рис. 3. Третий пример работы программного обеспечения

Представленные примеры иллюстрируют широкий спектр возможностей программы для моделирования траекторий объектов в различных условиях. Данное программное обеспечение может служить эффективным инструментом для исследователей, разработчиков игр и специалистов, занимающихся моделированием траекторий полёта объектов.

Тестирование производительности

Язык программирования Rust представляет собой относительно низкоуровневый компилируемый язык, не использующий сборщик мусора, что обеспечивает высокую производительность, сопоставимую с C++ [5], при минимальных затратах оперативной памяти и повышенной безопасности работы с памятью.

Далее проведём анализ временных затрат на выполнение алгоритма расчёта траектории падения объекта с использованием стандартных параметров: плотность воздуха – 1,225 кг/м³, гравитационное ускорение – 9,81 м/с², масса объекта – 1 кг, коэффициент сопротивления – 1 и площадь поперечного сечения – 1 м². Начальная скорость объекта составляет 1500 м/с по обеим осям. Исследование проводилось при различном количестве итераций и регионов на основе 200 попыток. Результаты временных затрат представлены в таблице.

Таблица. Анализ времени работы алгоритма

		Количество регионов			
		0	5	10	20
Максимальное количество итераций	1000	~69 278 нс	~96 271 нс	~131 938 нс	~202 417 нс
	5000	~341 775 нс	~467 512 нс	~641 617 нс	~978 868 нс
	10000	~677 519 нс	~929 238 нс	~1 275 136 нс	~1 962 894 нс
	100000	~7 614 688 нс	~10 316 289 нс	~14 099 795 нс	~20 849 297 нс

Как видно из результатов, даже при значительных значениях максимального количества итераций и количества регионов время выполнения алгоритма составляет доли секунды, что подтверждает высокую эффективность данного алгоритма с точки зрения скорости выполнения.

Заключение. В данной статье была разработано программное обеспечение, которое моделирует траекторию полёта объекта, с учетом изменяющихся во времени параметров.

Программа обладает рядом ключевых преимуществ. Гибкость и адаптивность обеспечиваются возможностью настройки множества параметров, влияющих на моделирование траектории. Простота использования достигается за счёт интуитивно понятного интерфейса, что делает программу доступной для широкого круга пользователей. Высокая производительность программы обусловлена эффективной реализацией алгоритмов расчёта траектории, что во многом стало возможным благодаря выбору языка программирования Rust, обеспечивающего оптимальную скорость выполнения вычислений.

Однако программа имеет ряд ограничений. При выборе слишком большого значения временного шага возможны некорректные расчёты траектории. Точность вычислений может быть повышена за счёт внедрения динамического временного шага или перерасчёта траектории непосредственно в точке столкновения с границей региона, а не на следующей итерации цикла. Программа обладает значительным потенциалом для дальнейшего улучшения и оптимизации.

Программное обеспечение для моделирования траектории полёта объектов представляет собой важный инструмент, который может быть использован как в академических исследованиях для анализа траекторий движения объектов в условиях различной степени сложности, так и в образовательных целях для углублённого изучения физических законов и принципов, а также в аэрокосмической отрасли для расчёта траекторий полёта космических аппаратов во время взлёта или посадки. Кроме того, его использование в игровой индустрии способствует созданию реалистичных и интерактивных симуляций. Программа предоставляет удобные и эффективные средства для расчёта траекторий с учётом динамически изменяющихся параметров, что делает её доступной и полезной как для начинающих, так и для опытных исследователей.

Библиографический список

1. Егорова Л.А., Лохин В.В. Баллистика и разрушение космических тел в атмосфере планет // Вестник ННГУ. – 2011. – №4-2.
2. Кайшев Д.А. Новое поколение языков программирования – RUST // StudNet. – 2021. – №6.
3. Аминов Р.Ш., Страхов В.В. Применение метода Рунге-Кутты для решения уравнения // Universum: технические науки. – 2022. – №10-1 (103).
4. Новиков Е.А., Кнауб Л.В. Алгоритм интегрирования на основе явного трехстадийного метода Рунге-Кутта // Вестник КрасГАУ. – 2009. – №3.
5. Астанов Х., Атаджанов Дж, Байлыев С. Тренды в разработке языков программирования: удобство и безопасность // Вестник науки. – 2024. – №10 (79).

DEVELOPMENT OF A RUST-BASED PROGRAM FOR MODELING OBJECT FLIGHT TRAJECTORIES WITH TIME-VARYING PARAMETERS

Jahid Rahmani, *Senior Lecturer*
I.S. Alexandrov, *Student*
Moscow Technical University of Communications and Informatics
(Russia, Moscow)

Abstract. *This article presents the development of software for modeling object flight trajectories using the Rust programming language, taking into account time-varying parameters. The implementation of algorithms for trajectory calculation, intersection checks with regions, and acceleration computation considering factors such as drag and gravitational acceleration is described. The practical application of the program is demonstrated through specific examples. The program exhibits high performance and has a wide range of applications, including scientific research, where it can be used to study the dynamics of object motion under various conditions and scenarios of varying complexity, as well as applied fields such as video game development, where it enables realistic simulation of physical interactions between objects. The conclusion discusses the advantages and limitations of the system, proposes directions for further development, and summarizes the significance and potential of this software.*

Keywords: *trajectory calculation, frontal resistance, gravitational acceleration, object motion, Rust programming language, software development.*