ПРОГРАММИРОВАНИЕ, *2024*, *№ 2*, *с. 51–65*

КОМПЬЮТЕРНАЯ АЛГЕБРА

УДК 004.422.8.

РЕАЛИЗАЦИЯ АНАЛИТИЧЕСКОЙ ПРОЕКТИВНОЙ ГЕОМЕТРИИ ДЛЯ КОМПЬЮТЕРНОЙ ГРАФИКИ

© 2024 г. М. Н. Геворкян[a, *], А. В. Королькова[a, **], Д. С. Кулябов[a, b, *], Л. А. Севастьянов[a, b, ****]**

[a]*Российский университет дружбы народов, 117198 Москва, ул. Миклухо-Маклая, д. 6, Россия*

[b]*Объединенный институт ядерных исследований, 141980 Дубна, Московская обл., ул. Жолио-Кюри, д. 6, Россия*

**E-mail: gevorkyan-mn@rudn.ru*

***E-mail: korolkova-av@rudn.ru*

****E-mail: kulyabov-ds@rudn.ru*

*****E-mail: sevastianov-la@rudn.ru*

Поступила в редакцию 09.08.2023
После доработки 10.09.2023
Принята к публикации 01.10.2023

В своих исследованиях авторы активно используют разные разделы геометрии. Для геометрических построений используются подходы и системы компьютерной алгебры. В данный момент нас заинтересовала такая область, как компьютерная геометрия, и более узко, реализация машинной графики. Стандартом де-факто в современной компьютерной графике стало использование проективного пространства и однородных координат, то есть задача фактически сводится к применению аналитической проективной геометрии. Авторам не удалось подобрать систему компьютерной алгебры, которая могла бы реализовать проективную геометрию во всем объеме. Поэтому было принято решение реализовать применение компьютерной алгебры частично, для визуализации алгебраических ссотношений. Для этого предлагается использовать систему Asymptote.

Ключевые слова: проективная геометрия, система Asymptote, координаты Плюккера, собственные и несобственные точки, прямые и плоскости

DOI: 10.31857/S0132347424020089 **EDN:** ROPXHV

1. ВВЕДЕНИЕ

Проективная геометрия находит применение в разных областях компьютерных наук: в компьютерной графике, робототехнике, машинном зрении. Математическая теория служит основой для создания алгоритмов и их программной реализации. Авторы считают, что данный математический раздел имеет большие прикладные перспективы.

В проективной геометрии непропорционально большую роль играет визуализация геометрических структур. Авторам не удалось найти полной системы компьютерной алгебры, специализирующейся на методах проективной геометрии, что несомненно связано с ее спецификой. Ранее мы использовали методы проективной геометрии для обоснования применения гиперкомплексных чисел для описания движений в разных вариантах физических пространств [1–4]. В качестве следующего шага мы использовали подход геометрической алгебры (с использованием систем компьютерной алгебры) для исследования тех же объектов под другим углом зрения [5–7]. На следующем шаге мы решили вернуться к прямому применению теоретических основ к исследованиям. В качестве первого этапа данной научной программы предлагается частичная компьютеризация исследований. Алгебраические формулы реализованы в виде структур данных и наборов функций, связанных с геометрическими структурами. Затем с помощью языка Asymptote сделана программная реализация для визуализации полученных алгебраических структур. В данном случае к структурам относятся точки, прямые и плоскости.

Язык Asymptote имеет структуру системы компьютерной алгебры, но вместо символьного представления алгебраических объектов в результате работы Asymptote получается их визуальное представление.

Система Asymptote [8–12] — специализированный высокоуровневый язык программирования для создания векторных изображений научной направленности (геометрические чертежи, схемы, диаграммы и т.д.). Язык использует C-подобный синтаксис с некоторым числом дополнений и улучшений[1], что по замыслу разработчиков делает его более

[1] Сам язык в своем синтаксисе скорее ориентируется на C++ [13]. Например, переменные в Asymptote можно объявлять в любом месте программы и инициализировать сразу при объявлении, а функции могут принимать параметры по умолчанию.

удобным для освоения и использованием по сравнению с MetaPost [14] и PostScript.

Для наших целей крайне важно, что Asymptote поддерживает создание полноценных трехмерных изображений благодаря использованию OpenGL с автоматическим расчетом взаимного расположения геометрических тел, источников света и тени, что позволяет изображать пересечение поверхностей и кривых. Однако программный интерфейс для построения трехмерных изображений гораздо менее проработан и скупее документирован по сравнению с возможностями двумерной графики.

В данной статье мы не будем излагать основ языка Asymptote. Заинтересованный читатель может найти описание всех его возможностей в официальном руководстве на сайте [15], базовые примеры приведены в обучающих заметках [11], также множество примеров с комментариями на русском языке можно посмотреть в источниках [16–18]. Следует отметить, что большая часть примеров относится к двумерным изображениям. В статье акцент сделан на описании реализации посредством Asymptote структур данных, таких как проективные точки, прямые и плоскости для трехмерных графиков, приведены примеры исходного кода с пояснениями.

1.1. Структура статьи

В начале статьи приведена реализация структур данных, описывающих проективные точки (раздел 2), прямые (раздел 3) и плоскости (раздел 4). Изложение сопровождается примерами исходного кода с пояснением всех специфических конструкций, ключевых слов и функций языка Asymptote. В разделе 5 приводятся несколько примеров использования созданных структур, а также описываются некоторые технические тонкости создания именно трехмерных изображений, так как в официальном руководстве данный аспект освещен довольно скупо.

1.2. Обозначения и соглашения

Напомним основные используемые обозначения проективной геометрии. Формулы приведены в однородных координатах, обозначения взяты из монографии [19]:

• $\{\mathbf{v} \mid \mathbf{p} \times \mathbf{v}\}$ — прямая, проходящая через точку P по направлению $\mathbf{v}$;

• $\{\mathbf{p}_2 - \mathbf{p}_1 \mid \mathbf{p}_1 \times \mathbf{p}_2\}$ — прямая, проходящая через две точки P_1 и P_2;

• $\{\mathbf{p} \mid \mathbf{0}\}$ — прямая, проходящая через начало координат и точку P;

• $\{w_1\mathbf{p}_2 - w_2\mathbf{p}_1 \mid \mathbf{p}_1 \times \mathbf{p}_2\}$ — прямая, проходящая через две точки $\vec{\mathbf{p}}_1 = (\mathbf{p}_1 \mid w_1)$ и $\vec{\mathbf{p}}_2 = (\mathbf{p}_2 \mid w_2)$;

• $\{\mathbf{n}_1 \times \mathbf{n}_2 \mid d_1\mathbf{n}_2 - d_2\mathbf{n}_1\}$ — прямая линия пересечения двух плоскостей $[\mathbf{n}_1 \mid d_1]$ и $[\mathbf{n}_2 \mid d_2]$;

• $(m \times n + dv \mid -(n, v))$ — точка пересечения плоскости $[n \mid d]$ и прямой $\{v \mid m\}$;

• $[m_1 \times m_2 \mid (v_2, m_1)]$ — точка пересечения двух прямых $\{v_1 \mid m_1\}$ и $\{v_2 \mid m_2\}$;

• $(d_1 n_3 \times n_2 + d_2 n_1 \times n_3 + d_3 n_2 \times n_1 \mid (n_1, n_2, n_3))$ — точка пересечения трех плоскостей $[n_1 \mid d_1]$, $[n_2 \mid d_2]$ и $[n_3 \mid d_3]$;

• $(v \times m \mid (v, v))$ — точка, ближайшая к началу координат на прямой $\{v \mid m\}$;

• $(-dn \mid (n, n))$ — точка, ближайшая к началу координат на плоскости $[n \mid d]$;

• $[v \times u \mid -(u, m)]$ — плоскость, содержащая прямую $\{v \mid m\}$ и направление u;

• $[v \times p + m \mid -(p, m)]$ — плоскость, содержащая прямую $\{v \mid m\}$ и точку $(p \mid 1)$;

• $[m \mid 0]$ — плоскость, содержащая прямую $\{v \mid m\}$ и начало координат;

• $[v \times p + wm \mid -(p, m)]$ — плоскость, содержащая прямую $\{v \mid m\}$ и точку $(p \mid w)$;

• $[v \times u \mid (u, v, p)]$ — плоскость, содержащая точку $(p \mid 1)$ и направления v и u;

• $[m \times v \mid (m, m)]$ — плоскость с прямой $\{v \mid m\}$, наиболее отдаленная от O;

• $[-wp \mid (p, p)]$ — плоскость с точкой $(p \mid w)$, наиболее отдаленная от O;

• $\dfrac{(v_1, m_2) + (v_2, m_1)}{v_1 \times v_2}$ — расстояние между прямыми $\{v_1 \mid m_1\}$ и $\{v_2 \mid m_2\}$;

• $\dfrac{v \times p + m}{v}$ — расстояние между прямой $\{v \mid m\}$ и точкой $(p \mid 1)$;

• $m \,/\, v$ — расстояние от прямой $\{v \mid m\}$ до начала координат;

• $\dfrac{(n, p) + d}{n}$ — расстояние от плоскости $[n \mid d]$ до точки $(p \mid 1)$;

• $d \,/\, n$ — расстояние от плоскости $[n \mid d]$ до начала координат.

2. СОЗДАНИЕ СТРУКТУР НА ПРИМЕРЕ ОДНОРОДНЫХ КООРДИНАТ ТОЧКИ

Для создания пользовательских структур в языке Asymptote используется синтаксис, во многом похожий на соответствующий синтаксис языка Си, однако имеющий некоторую специфику, которую мы рассмотрим на примере структуры для однородных координат.

Напомним, что однородные координаты представляют собой систему координат, используемую в проективной геометрии и обладающую тем свойством, что определяемый ими объект не меняется при умножении всех координат на одно и то же ненулевое число. Из-за этого количество координат, необходимое для представления точек, всегда на одну больше, чем размерность пространства, в котором эти координаты используются.

Язык Asymptote имеет встроенный тип triple, который используется для задания координат точек в трехмерном пространстве. Структура данных для однородных координат отсутствует, по крайней мере недоступна пользователям, поэтому следует ее создать.

Назовем структуру Vec4, по аналогии с соответствующим типом данных из GLSL (OpenGL Shading Language, Graphics Library Shader Language):

```
struct Vec4 {
  real x; real y; real z; real w;
  pair xy; triple xyz;
  real[] xyzw;
}
```

Структура данных должна содержать всего 4 числа типа real, однако для удобства использования кроме полей x, y, z, w, соответствующих однородным координатам $x : y : z : w$, мы задаем поля, позволяющие получить только координаты xy в виде типа данных pair, трехмерные декартовы координаты xyz в виде типа данных triple и все координаты в виде массива $xyzw$. Кроме удобства при манипуляциях с указанными объектами такой подход также повторяет программный интерфейс языка GLSL.

После определения структуры Vec4 Asymptote автоматически создает одноименную функцию, обязательными аргументами которой являются все поля, перечисленные при определении структуры. Данная функция эквивалентна конструктору по умолчанию в терминах языка C++ или Java.

В нашем случае придется перечислить все семь полей, хотя они частично дублируют друг друга. Данная проблема решается заданием специализированных операторов инициализации (конструкторов), для чего следует перегрузить специальный оператор init. Внутри этого оператора можно использовать ключевое слово this, которое позволяет сослаться на экземпляр структуры.

Для нашего примера создадим три специализированных конструктора:

• Конструктор Vec4 из набора чисел:

```
void operator init(real x, real y,
↪  real z, real w) {
this.x = x;
this.y = y;
this.z = z;

  this.w = w;
  this.xy = (x, y);
  this.xyz = (x, y, z);
  this.xyzw = new real[] {this.x,
  ↪  this.y, this.z, this.w};
}
```

• Конструктор Vec4 из декартовых координат точки и отдельного скаляра w:

```
void operator init(triple p, real w)
↪  {
  this.x = p.x;
  this.y = p.y;
  this.z = p.z;
  this.w = w;
  this.xy = (p.x, p.y);
  this.xyz = (p.x, p.y, p.z);
  this.xyzw = new real[] {this.x,
  ↪  this.y, this.z, this.w};
}
```

• Конструктор Vec4 из массива:

```
void operator init(real[] v) {
  this.x = v[0];
  this.y = v[1];
  this.z = v[2];
  this.w = v[3];
  this.xy = (v[0], v[1]);
  this.xyz = (v[0], v[1], v[2]);
  this.xyzw = v[0:5];
}
```

Все перечисленные операторы следует размещать внутри пространства имен структуры (т.е. между открывающей и закрывающей фигурными скобками структуры Vec4).

Для удобства манипулирования данными можно также перегрузить оператор cast, который ответственен за неявное преобразование типов данных:

• Преобразование типа Vec4 в массив:

```
real[] operator cast(Vec4 v) {
  return v.xyzw;
}
```

• Преобразование типа Vec4 в трехмерные декартовы координаты:

```
triple operator cast(Vec4 v) {
  return v.xyz / v.w;
}
```

• Преобразование массива в тип Vec4:

```
Vec4 operator cast(real[] v) {
  return Vec4(v);
}
```

При преобразовании данных в тип triple осуществляется дополнительное деление на весовую координату w. В случае несобственной (идеальной) точки данное преобразование сработает некорректно, так как у идеальной точки координата $w=0$ и произойдет деление на ноль. Однако проверку на равенство нулю координаты w всегда можно произвести отдельно.

3. ПРОЕКТИВНЫЕ ПРЯМЫЕ В ВИДЕ СТРУКТУРЫ LINE3D

3.1. Структура и конструкторы

Прямая в пространстве полностью определяется координатами Плюккера, которые задаются направляющим вектором v и вектором момента m. Для вычислений удобно, чтобы вектор v был единичными. На основе векторов v и m можно вычислить любые другие характеристики прямой, поэтому в качестве полей структуры достаточно использовать только эти два вектора. Однако для вычислений часто необходимо знать некоторую точку P прямой, поэтому удобно также включить ее в поля структуры. Также для большей общности можно допустить хранение неединичного вектора v. В итоге структура для задания проективной прямой будет иметь следующий вид:

```
struct Line3D {
  // Направляющий вектор и момент:
  triple V, m;
  // Единичный направляющий вектор:
  triple v;
  // Произвольная точка прямой:
  triple P;
}
```

Для полноценного использования созданной структуры в ее области имен необходимо задать несколько дополнительных операторов инициализации. Так, прямая, проходящая через две собственные точки, заданные в декартовых координатах, вычисляется по формуле:

$$\mathbf{m} = \mathbf{p}_1 \times \mathbf{p}_2. \qquad (1)$$

Соответствующий конструктор будет выглядеть следующим образом:

```
void operator init (triple P2, triple
↪  P1) {
  this.V = P2 - P1;
  this.v = unit(this.V);
  this.m = cross(P1, P2) /
  ↪  length(this.V);
  this.P = P1;
}
```

Следует обратить внимание, что для вычисления единичного вектора v мы использовали встроенную в Asymptote функцию unit, для вычисления векторного произведения — функцию cross, а для вычисления нормы вектора — функцию length. Обратите внимание, что при вычислении m используется функция (1) в комбинации с соотношением

$$d_{OO_\perp} = OO_\perp =$$

$$= \frac{v \times m}{v} = \frac{vm\sin\frac{\pi}{2}}{v^2} = \frac{m}{v},$$

т.е. она делится на исходную длину вектора v, вычисленного как $\mathbf{p}_2 - \mathbf{p}_1$. В этом случае длина m имеет геометрический смысл расстояния от начала координат до прямой.

Следующий конструктор инициализирует структуру по заданным векторам $\mathbf{v}$ и $\mathbf{m}$:

```
void operator init (triple keyword V,
↪  triple keyword m) {
  this.V = V;
  this.v = unit(this.V);
  this.m = m / length(this.V);
  this.P = cross(this.v, this.m);
}
```

Вектор $\mathbf{v}$ может быть произвольной длины, его исходное значение записывается в поле V структуры, а его нормированное значение — в поле v. Здесь вектор $\mathbf{m}$ нормализован путем деления на собственную длину ненормированного касательного вектора $\|\mathbf{v}\|$.

Инструкция keyword указывает, что данный оператор инициализации можно вызывать только явно указав все аргументы, т.е. следующим образом:

```
triple V = (1,1,1); triple m = (1, 1,
↪  1);
Line3D L = Line3D(V=V, m=m);
```

Если осуществить вызов через Line3D L = Line(V, m), то будет вызван предыдущий конструктор по двум точкам, заданным в декартовых координатах.

В реализации конструктора, который инициализирует прямую, проходящую через две точки, заданные в проективных координатах, используется следующая формула:

$$\{w_1\mathbf{p}_2 - w_2\mathbf{p}_1 \mid \mathbf{p}_1 \times \mathbf{p}_2\}. \qquad (2)$$

Соответствующий конструктор будет выглядеть следующим образом:

```
void operator init (Vec4 P2, Vec4 P1) {
  this.V = P1.w * P2.xyz - P2.w *
  ↪  P1.xyz;
  this.v = this.V / length(this.V);
  this.m = cross(P1.xyz, P2.xyz)  /
  ↪  length(this.V);
  this.P = P1.xyz / P1.w;
}
```

Мы неявно подразумеваем, что по крайней мере P_1 — собственная точка, т.е. ее координата w не равна нулю. Подчеркнем, что формула (3) — универсальна и работает для всех возможных вариантов точек. Например, если $\vec{\mathbf{p}}_1 = (x, y, z, w)$ — собственная точка, а $\vec{\mathbf{p}}_2 = (\mathbf{v} \mid 0) = (v_x, v_y, v_z, 0)$ — несобственная точка, то координаты Плюккера прямой вычисляются как

$$\vec{\mathbf{l}} = \{w\mathbf{v} \mid \mathbf{p} \times \mathbf{v}\}.$$

Произведя нормализацию путем деления на $w|v|$, получим формулу:

$$\{\mathbf{v} \mid \mathbf{m}\} = \{v_x, v_y, v_z \mid m_x, m_y, m_z\} = \{\mathbf{v} \mid \mathbf{p} \times \mathbf{v}\}. \quad (3)$$

Формуле (3) соответствует следующий конструктор (прямая, проходящая через точку по направляющему вектору):

```
void operator init (Vec4 P, triple V) {
  this.V = V;
  this.v = this.V / length(this.V);
  this.m = cross(P.xyz, this.V) /
  ↪  length(this.V);
  this.P = P.xyz;
}
```

Если же обе точки $\vec{\mathbf{p}}_1 = (\mathbf{v}_1 \mid 0)$ и $\vec{\mathbf{p}}_2 = (\mathbf{v}_2 \mid 0)$ являются несобственными, то получаем несобственную прямую $\{\mathbf{0} \mid \mathbf{v}_1 \times \mathbf{v}_2\}$. Несобственные прямые могут встречаться при рассмотрении взаимного расположения плоскостей. Если плоскости параллельны, то они будут пересекаться как раз по несобственным прямым. Пример формулы (3) иллюстрирует преимущество проективного подхода, который позволяет одной формулой охватить сразу все возможные варианты.

Осталось создать еще два конструктора, которые реализуют формулу

$$\{\mathbf{v} \mid 0\}.$$

Соответствующие конструкторы имеют вид:

• Прямая, проходящая через начало координат и точку P (в декартовых координатах):

```
void operator init (Vec4 P) {
  this.V = P.xyz;
  this.v = unit(this.V);
  this.m = (0, 0, 0);
  this.P = P.xyz;
}
```

• Прямая, проходящая через начало координат и точку P:

```
void operator init (triple P) {
  this.V = P;
  this.v = unit(this.V);
  this.m = (0, 0, 0);
  this.P = P;
}
```

Для визуализации прямой необходимо знать хотя бы две ее точки. Для реализации параметрического уравнения прямой определим в пространстве имен структуры Line3D следующую функцию:

```
triple get_point(real t) {
  return this.P + this.v * t;
}
```

Удобно также иметь возможность вычислить сразу массив точек прямой, что крайне просто делается с помощью поэлементного цикла for:

```
triple[] get_points(real[] T) {
  triple[] res;
  for (real t : T) {
    res.push(get_point(t));
  }
  return res;
}
```

Данный вариант цикла for похож на аналогичный цикл из языка Python, позволяет перебирать элементы из массива и проводить с ними манипуляции.

3.2. Взаимное расположение прямых

В предыдущем пункте мы задали поля структуры, набор конструкторов и несколько методов. Теперь зададим несколько функций, аргументами которых будут экземпляры структуры Line3D. Данные функции разместим в том же файле, что и структура Line3D, но вне ее пространства имен (за закрывающей фигурной скобкой).

В первую очередь реализуем функцию вычисления *взаимного момента* двух прямых l_1 и l_2 по простой формуле

$$M = (\mathbf{v}_1, \mathbf{m}_2) + (\mathbf{v}_2, \mathbf{m}_1).$$

Реализация соответствующей функции имеет вид:

```
real reciprocal_moment(Line3D line01,
 ↪  Line3D line02) {
   return dot(line01.m, line02.v) +
    ↪  dot(line01.v, line02.m);
}
```

Здесь мы использовали встроенную функцию dot, которая вычисляет скалярное произведение двух векторов.

Несмотря на свою простоту, данная функция крайне полезна и позволяет определить взаимное положение двух прямых. Возможны три случая в зависимости от знака взаимного момента:

- если $M=0$, то прямые лежат в одной плоскости;
- если $M>0$, то переход от l_1 к l_2 осуществляется правым поворотом;
- если $M<0$, то переход от l_1 к l_2 получается левым поворотом.

Заметим, что делать проверку на равенство нулю следует с учетом погрешности представления вещественных чисел числами с плавающей точкой.

Далее определим функцию, реализующую формулу

$$d = d_2 - d_1 = \frac{(v_1, m_2) + (v_2, m_1)}{v_1 \times v_2}. \qquad (4)$$

Соотношение (4) позволяет вычислить длину d взаимного перпендикуляра между двумя скрещивающимися прямыми. Так как у нас уже есть функция вычисления взаимного момента, то вычисления длины перпендикуляра будет осуществляться также в одну строчку:

```
real reciprocal_perp_length(Line3D line01,
 ↪  Line3D line02) {
   return reciprocal_moment(line01, line02)
    ↪  / length(cross(line01.v, line02.v));
}
```

Предыдущую функцию можно было бы сделать более общей, добавив проверку скрещиваемости прямых, и в случае, если прямые лежат в одной плоскости, применить формулу

$$d_{QQ_\perp} = \frac{\|\mathbf{m} + \mathbf{v} \times \mathbf{q}\|}{\|\mathbf{v}\|} = \frac{\|(\mathbf{p} - \mathbf{q}) \times \mathbf{v}\|}{\|\mathbf{v}\|}$$

в виде

$$d = \frac{\mathbf{v} \times \mathbf{p}_2 + \mathbf{m}_1}{\mathbf{v}_1},$$

где $\vec{\mathbf{l}}_1 = \{\mathbf{v}_1 \,|\, \mathbf{m}_1\}$ — координаты Плюккера первой прямой, $\vec{\mathbf{p}}_2 = (\mathbf{p}_2 \,|\, 1)$ — однородные координаты произвольной точки второй прямой. Однако мы не стали усложнять функцию дополнительными внутренними проверками. Также реализуем формулу

$$[\mathbf{m}_1 \times \mathbf{m}_2 \,|\, (\mathbf{v}_2, \mathbf{m}_1)] = [\mathbf{m}_2 \times \mathbf{m}_1 \,|\, (\mathbf{v}_1, \mathbf{m}_2)]. \qquad (5)$$

При этом сделаем проверку на равенство нулю взаимного момента, так как в этом случае две прямые будут лежать в одной плоскости (будут компланарны):

```
// line-line-meet
Vec4 intersection(Line3D line01, Line3D
 ↪  line02) {
    real M = reciprocal_moment(line01,
     ↪  line02);
  assert( abs(M) < 1e-8, "Прямые не
   ↪  компланарны");
  return Vec4( cross(line01.m, line02.m),
   ↪  dot(line02.v, line01.m));
  // return Vec4( cross(line02.m,
   ↪  line01.m), dot(line01.v, line02.m));
}
```

Как указывалось выше, проверку на равенство нулю следует сделать с некоторой погрешностью. В нашей реализации мы допускаем довольно большую погрешность, так как основной целью является визуализация полученных объектов, где малые погрешности не критичны.

Для вычисления координат Плюккера прямой, содержащей взаимный перпендикуляр, воспользуемся довольно-таки громоздкой формулой:

$$\mathbf{v}_\perp = \mathbf{v}_1 \times \mathbf{v}_2,$$

$$\mathbf{m}_\perp = \mathbf{m}_1 \times \mathbf{l}_2 - \mathbf{m}_2 \times \mathbf{v}_1 +$$

$$+ \frac{(\mathbf{v}_1, \mathbf{m}_2) + (\mathbf{v}_2, \mathbf{m}_1)}{\mathbf{v}_1 \times \mathbf{v}_2} (\mathbf{v}_1, \mathbf{v}_2) \mathbf{v}_1 \times \mathbf{v}_2.$$

Компьютерная реализация имеет вид:

```
Line3D reciprocal_perp_line(Line3D line01,
↪  Line3D line02) {
  triple V = cross(line01.v, line02.v);
  triple m = cross(line01.m, line02.v) -
  ↪  cross(line02.m, line01.v) +
  ↪  reciprocal_perp_length(line01,
  ↪  line02) * dot(line01.v, line02.v) *
  ↪  cross(line01.v, line02.v) /
  ↪  length(cross(line01.v, line02.v));
  return Line3D(V=V, m=m);
}
```

4. ПРОЕКТИВНЫЕ ПЛОСКОСТИ В ВИДЕ СТРУКТУРЫ PLANE3D

4.1. Структура и конструкторы

Теперь рассмотрим структуру, реализующую задание плоскости в однородном виде. Для полного описания плоскости достаточно вектора $\vec{\boldsymbol{\pi}} = [\mathbf{n} \,|\, d]$, где $\mathbf{n}$ — единичный вектор нормали к плоскости, а d — ориентированное расстояние от начала координат до плоскости.

В модуле three языка Asymptote существует функция plane со следующей сигнатурой:

```
path3 plane(triple u, triple v,
↪  triple0=0);
```

Здесь u и v — направляющие векторы плоскости, а 0 — точка их приложения. Данная функция возвращает замкнутый прямоугольный контур 0 -- 0+u -- 0+u+v -- 0+v -- cycle, который затем можно отобразить в виде плоскости с помощью функции surface. Поэтому в структуре Plane3D полезно хранить два направляющих вектора $\mathbf{v}$ и $\mathbf{u}$, а также произвольную точку P. Полностью поля структуры будут выглядеть следующим образом:

```
struct Plane3D {
  // Вектор нормали (ненормированный):
  triple N;
  // Единичный вектор нормали:
  triple n;
  // Направляющие векторы (касательные
  ↪  векторы):
  triple U, V;
  // Единичные направляющие векторы:
  triple u, v;
  // Ориентированное расстояние от
  ↪  начала координат до прямой:
  real d;
  // Произвольная точка плоскости:
  triple P;
}
```

Зададим несколько сравнительно тривиальных операторов инициализации. Первый из них создает плоскость, содержащую прямую $\vec{\mathbf{l}} = \{\mathbf{v} \mid \mathbf{n}\}$ и направляющий вектор $\mathbf{v}$:

```
void operator init (Line3D line, triple
↪  U) {
  this.U = U;
  this.V = line.V;

  this.u = unit(U);
  this.v = line.v;

  this.N = cross(this.v, this.u);
  this.n = unit(this.N);
  this.d = -dot(this.u, line.m);

  this.P = -this.d * this.n;
}
```

В качестве точки плоскости берется ближайшая к началу координат. Вычисление проводится по формуле

$$[\mathbf{v} \times \mathbf{u} \mid -(\mathbf{u}, \mathbf{m})].$$

Второй конструктор инициализирует плоскость, проходящую через точку и прямую. Точка может быть задана однородными координатами, тогда работает формула

$$[\mathbf{v} \times \mathbf{p} + w\mathbf{m} \mid -(\mathbf{p}, \mathbf{m})].$$

Точка может быть задана декартовыми координатами, тогда вычисления проводятся по формуле

$$[\mathbf{v} \times \mathbf{p} + \mathbf{m} \mid -(\mathbf{p}, \mathbf{m})].$$

Так как случай однородных координат более общий, то мы ограничимся только им в реализации конструктора:

```
void operator init (Line3D line, Vec4 P)
↪  {
  this.U = line.P - P.xyz;
  this.V = line.V;
```

```
  this.u = unit(U);
  this.v = line.v;

  this.N = cross(line.v, P.xyz) + P.w *
  ↪  line.m;
  this.n = unit(this.N);
  this.d = -dot(P.xyz, line.m);

  this.P = P.xyz;
}
```

Третий конструктор инициализирует плоскость, проходящую через прямую и начало координат:

```
void operator init (Line3D line) {
  this.U = line.P;
  this.V = line.V;

  this.u = unit(U);
  this.v = line.v;

  this.N = line.m;
  this.n = unit(this.N);
  this.d = 0;

  this.P = line.P;
}
```

Вычисления проводятся по следующей формуле:

$$[\mathbf{m} \mid 0].$$

Четвертый конструктор создает плоскость, проходящую через заданную точку и два направляющих вектора. Это эквивалентно записи параметрического уравнения плоскости. Вычисления проводятся по формуле (8). В Asymptote нет смешанного произведения, но оно естественно заменяется на комбинацию скалярного и векторного произведений:

```
void operator init (triple P, triple U,
↪  triple V) {
  this.U = U;
  this.V = V;

  this.u = unit(U);
  this.v = unit(V);

  this.N = cross(U, V);
  this.n = unit(this.N);
  this.d = dot(V, cross(U, P)) /
  ↪  length(this.N);

  this.P = P;
}
```

Наконец пятый конструктор — более сложный по сравнению с двумя предыдущими. Он задает плоскость по коэффициентам общего уравнения. В этом случае довольно просто вычислить единичный вектор нормали и ориентированное расстояние от центра координат, но не так просто получить пару направляющих векторов:

```
void operator init (real A, real B, real
↪  C, real D) {
  this.N = (A, B, C);
  this.n = unit(this.N);
  triple w;
  if ( A <= B && A <= C) {
    w = this.n + (1, 0, 0);
  } else if (B <= A && B <= C) {
    w = this.n + (0, 1, 0);
  } else {
    w = this.n + (0, 0, 1);
  }
  this.u = unit(cross(w, this.n));
  this.v = cross(this.n, this.u);
  this.U = u;
  this.V = v;
  this.d = D / length(this.N);
  this.P = -this.d * this.n;
}
```

По известному единичному нормальному вектору плоскости $\mathbf{n}$ можно вычислить направляющие векторы плоскости $\mathbf{u}$ и $\mathbf{v}$. Для этого можно использовать простую процедуру, указанную в [20, с. 29]. Возьмем произвольный вектор $\mathbf{w}$, не коллинеарный $\mathbf{n}$. Чтобы найти такой вектор, можно взять наименьшую компоненту вектора $\mathbf{n}$ и увеличить ее на 1, например $\mathbf{w} = (n_x, n_y, n_z + 1)$, если $n_z < n_x$ и $n_z < n_y$. После этого можно вычислить вектор u:

$$\mathbf{u} = \frac{\mathbf{w} \times \mathbf{n}}{\|\mathbf{w} \times \mathbf{n}\|}.$$

Из свойств векторного произведения следует, что $\mathbf{u} \perp \mathbf{n}$. При вычислениях с помощью компьютера полезно также удостовериться, что значение $\|\mathbf{w} \times \mathbf{n}\|$ не слишком велико или мало, чтобы избежать лишних погрешностей, связанных с машинной точностью.

На последнем шаге находим $\mathbf{v}$ как $\mathbf{v} = \mathbf{n} \times \mathbf{u}$. Так как $\|\mathbf{n}\| = \|\mathbf{u}\| = 1$, то и $\mathbf{v}$ — единичный вектор. Таким образом мы получили ортонормированную тройку векторов $\langle \mathbf{n}, \mathbf{u}, \mathbf{v} \rangle$.

Точку плоскости можно найти как проекцию начала координат на плоскость $\mathbf{OO}_\perp = -d\mathbf{n}$ при условии, что $\mathbf{n}$ — единичный вектор:

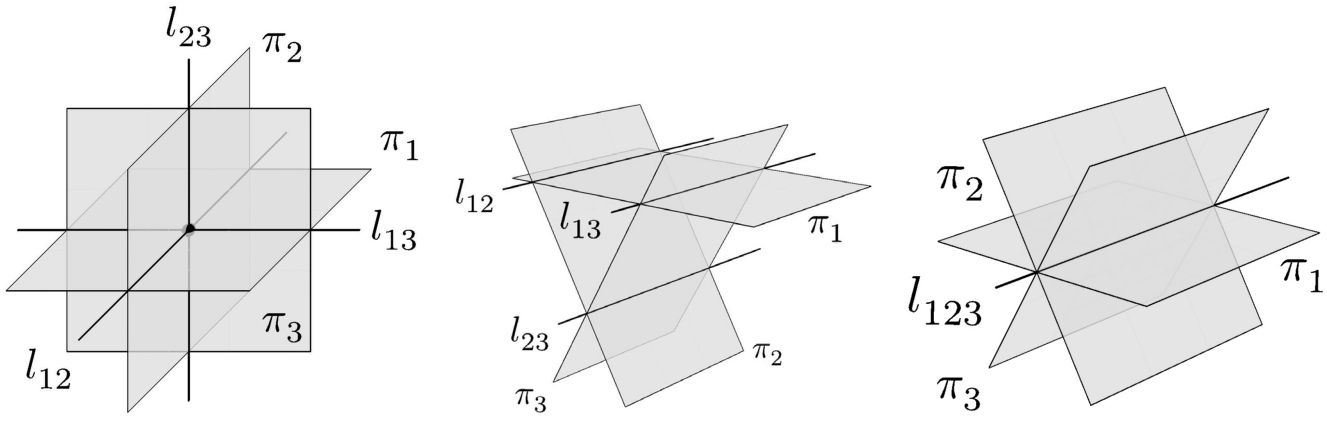


Рис. 1. Варианты непараллельных плоскостей.

$$(-d\mathbf{n} \mid \|\mathbf{n}\|^2).$$

В области имен структуры зададим еще две функции. Первая из них фактически является параметрическим уравнением плоскости:

```
triple get_point(real s, real t) {
  return this.P + this.u * s + this.v *
  ↪  t;
}
```

в виде прямоугольника:

```
guide3 get_contour() {
  triple P = this.P - 0.5(this.u +
  ↪  this.v);
  return  P -- P + this.u -- P + this.u
  ↪  + this.v -- P +this.v -- cycle;
}
```

При этом точка приложения направляющих векторов смещена так, чтобы визуально она лежала в центре прямоугольника, составляющего контур плоскости.

4.2. Взаимное расположение плоскостей

Имея в наличии структуру Plane3D, можно перейти к решению задачи определения взаимного положения двух и трех плоскостей.

В проективном пространстве две плоскости пересекаются всегда, так как параллельные плоскости считаются пересекающимися в бесконечности и прямая, по которой они пересекаются, является несобственной. Прямая пересечения вычисляется по формуле:

$$\{\mathbf{v} \mid \mathbf{m}\} = \{\mathbf{n}_1 \times \mathbf{n}_2 \mid d_1\mathbf{n}_2 - d_2\mathbf{n}_1\}. \qquad (6)$$

Возможны следующие варианты:

- если $\mathbf{v} \neq \mathbf{0}$, то прямая собственная;
- если $\mathbf{v} = \mathbf{0}$, то прямая несобственная;
  - если $d_1 = d_2$, то плоскости совпадают;
  - если $d_1 \neq d_2$, то плоскости параллельны.

Отметим также, что если две плоскости совпадают, то необязательно $\mathbf{n}_1 = \mathbf{n}_2$, так как возможен вариант, когда $\mathbf{n}_1 = -\mathbf{n}_2$. В этом случае лицевая и внутренняя стороны плоскостей отличаются.

В случае трех плоскостей вариантов может быть больше. Все они изображены на рис. 1 и 2.

Чтобы выписать формулы для вычислений, введем следующие обозначения. Плоскости π_1, π_2 и π_3 запишем в координатах Плюккера в следующем виде: $\vec{\boldsymbol{\pi}}_1 = [\mathbf{n}_1 | d_1]$, $\vec{\boldsymbol{\pi}}_2 = [\mathbf{n}_2 | d_2]$ и $\vec{\boldsymbol{\pi}}_3 = [\mathbf{n}_3 | d_3]$. Каждая пара этих трех плоскостей пересекается по некоторой прямой l_{ij}, где $i, j = 1, 2, 3$ и $i < j$.

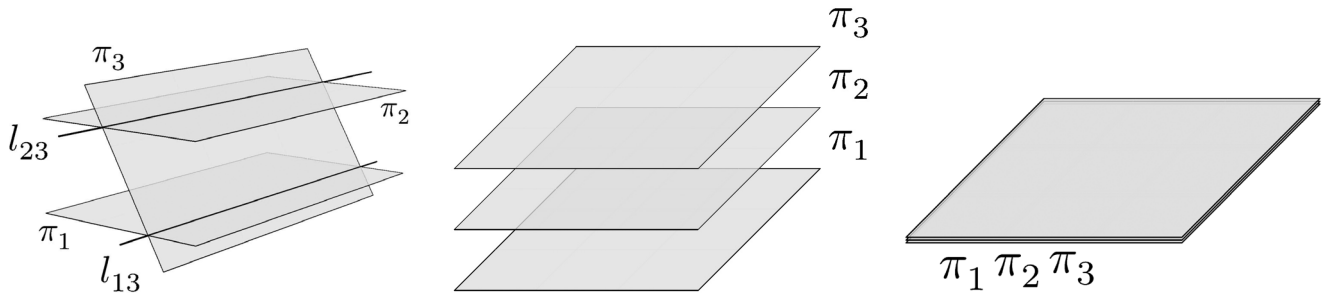
Тогда по формуле (14) запишем

$$\vec{\mathbf{l}}_{ij} = \{\mathbf{n}_i \times \mathbf{n}_j \mid d_i\mathbf{n}_j - d_j\mathbf{n}_i\}.$$

То есть получается следующее соотношение: $v_{ij} = \mathbf{n}_i \times \mathbf{n}_j$ и $\mathbf{m}_{ij} = d_1\mathbf{n}_2 - d_2\mathbf{n}_1$.

Все возможные варианты расположения трех плоскостей можно определить по следующей схеме:

- точка пересечения является собственной, т.е. $(\mathbf{n}_1, \mathbf{n}_2, \mathbf{n}_3) \neq 0$;
- точка пересечения является несобственной, т.е. $(\mathbf{n}_1, \mathbf{n}_2, \mathbf{n}_3) = 0$:
  - все прямые являются собственными, параллельными и различными (не совпадают), т.е. $\mathbf{v}_{12} \times \mathbf{v}_{13} = \mathbf{v}_{13} \times \mathbf{v}_{23} = \mathbf{v}_{23} \times \mathbf{v}_{12} = \mathbf{0}$ и $\mathbf{m}_{12} \neq \mathbf{m}_{23} \neq \mathbf{m}_{13}$;
  - все прямые являются собственными, параллельными и совпадают, т.е. $\mathbf{v}_{12} \times \mathbf{v}_{13} = \mathbf{v}_{13} \times \mathbf{v}_{23} = \mathbf{v}_{23} \times \mathbf{v}_{12} = \mathbf{0}$ и $\mathbf{m}_{12} = \mathbf{m}_{23} = \mathbf{m}_{13}$;



(а) (б) (в)

Рис. 2. Варианты параллельного расположения плоскостей. Вариант (в) — предельный случай варианта (б).

- две прямые являются собственными, параллельными и различными (не совпадают), т.е. $\mathbf{v}_{13} \times \mathbf{v}_{23} = \mathbf{0}$ и $\mathbf{m}_{13} \neq \mathbf{m}_{23}$, а одна прямая является несобственной, т.е. $\mathbf{v}_{12} = \mathbf{0}$;
- все прямые являются несобственными, т.е. $\mathbf{v}_{12} = \mathbf{v}_{23} = \mathbf{v}_{13} = \mathbf{0}$, а плоскости не совпадают, т.е. $d_1 \neq d_2 \neq d_3$;
- все прямые являются несобственными, т.е. $\mathbf{v}_{12} = \mathbf{v}_{23} = \mathbf{v}_{13} = 0$, а плоскости совпадают, т.е. $d_1 = d_2 = d_3$.

Однородные координаты точки пересечения трех плоскостей вычисляются по формуле:

$$p = \begin{pmatrix} d_1\mathbf{n}_3 \times \mathbf{n}_2 + d_2\mathbf{n}_1 \times \mathbf{n}_3 + \\ + d_3\mathbf{n}_2 \times \mathbf{n}_1 \mid (\mathbf{n}_1, \mathbf{n}_2, \mathbf{n}_3) \end{pmatrix} = \\ = (\Delta_x, \Delta_y, \Delta_z \mid \Delta). \quad (15)$$

Обозначения Δ_x, Δ_y, Δ_z представляют собой определители решения системы уравнений $(\vec{\mathbf{n}}, \vec{\mathbf{p}}) = 0$ методом Краммера:

$$\begin{cases} n_{1x}x + n_{1y}y + n_{1z}z = -d_1, \\ n_{2x}x + n_{2y}y + n_{2z}z = -d_2, \\ n_{3x}x + n_{3y}y + n_{3z}z = -d_3. \end{cases}$$

$$\Delta = \begin{vmatrix} n_{1x} & n_{1y} & n_{1z} \\ n_{2x} & n_{2y} & n_{2z} \\ n_{3x} & n_{3y} & n_{3z} \end{vmatrix}, \quad \Delta_x = \begin{vmatrix} -d_1 & n_{1y} & n_{1z} \\ -d_2 & n_{2y} & n_{2z} \\ -d_3 & n_{3y} & n_{3z} \end{vmatrix},$$

$$\Delta_y = \begin{vmatrix} n_{1x} & -d_1 & n_{1z} \\ n_{2x} & -d_2 & n_{2z} \\ n_{3x} & -d_3 & n_{3z} \end{vmatrix}, \quad \Delta_z = \begin{vmatrix} n_{1x} & n_{1y} & -d_1 \\ n_{2x} & n_{2y} & -d_2 \\ n_{3x} & n_{3y} & -d_3 \end{vmatrix}.$$

Обозначение $(\Delta_x, \Delta_y, \Delta_z \mid \Delta)$ задает точку в однородных координатах:

$$\vec{\mathbf{p}} = \left(\frac{\Delta_x}{\Delta}, \frac{\Delta_y}{\Delta}, \frac{\Delta_z}{\Delta} \,\middle|\, 1 \right) =: (\Delta_x, \Delta_y, \Delta_z \mid \Delta).$$

Рассмотрим теперь реализацию указанных выше формул в виде кода. Можно встроить непосредственно в функции проверки того, являются ли получаемые точки и прямые несобственными. Однако если оперировать только проективными объектами, то такая проверка излишняя и ее можно проводить непосредственно в момент визуализации, извлекая всю информацию из однородного представления точек, прямых и плоскостей.

Вычисление общей прямой, по которой пересекаются две плоскости, можно реализовать следующим образом:

```
Line3D intersection(Plane3D plane01,
 ↪  Plane3D plane02) {
  Line3D line;
  line.m = plane01.d * plane02.n -
   ↪  plane02.d * plane01.n;
  line.V = cross(plane01.n, plane02.n);
  // Если плоскости пересекаются по
   ↪  несобственной прямой, то v = 0
  if (abs(line.V) < 1e-8) {
    line.v = (0, 0, 0);
  } else {
    line.v = unit(line.V);
    // проведем нормировку, разделив на
     ↪  длину направляющего вектора
    line.m = line.m / length(line.V);
    line.P = cross(line.v, line.m) /
     ↪  dot(line.v, line.v);
  }
  return line;
}
```

Необходимые проверки были встроены непосредственно в код. Функция abs вычисляет норму вектора (можно заменить на функцию length). Еще раз следует заметить, что в случае использования чисел с плавающей запятой необходимо учитывать

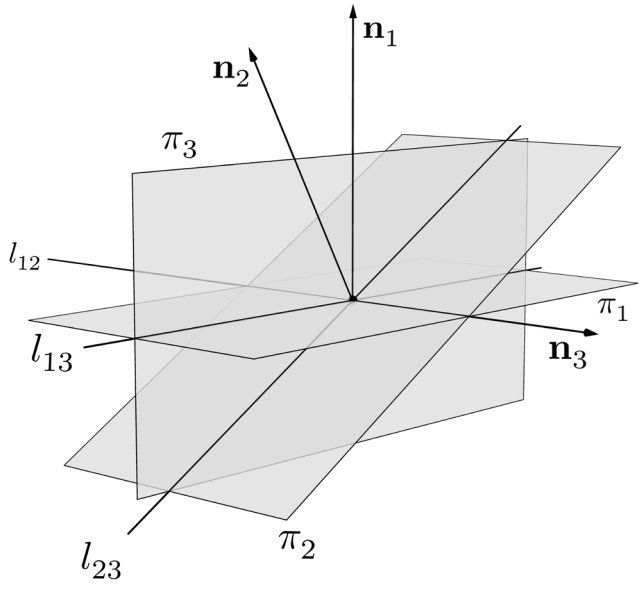


Рис. 3. Пересечение трех плоскостей.

машинную погрешность и использовать не строгое равенство, а проверять, что модуль числа не превосходит некоторое малое число.

Код для вычисления точки пересечения трех плоскостей будет выглядеть следующим образом:

```
Vec4 intersection(Plane3D plane01,
↪  Plane3D plane02, Plane3D plane03) {
  triple res;

  real n1n2n3 = dot(plane01.n,
  ↪  cross(plane02.n, plane03.n));

  res  = plane01.d * cross(plane03.n,
  ↪  plane02.n);
  res += plane02.d * cross(plane01.n,
  ↪  plane03.n);
  res += plane03.d * cross(plane02.n,
  ↪  plane01.n);
  return Vec4(res, n1n2n3);
}
```

В этом примере мы не делаем никаких проверок, так как никакая комбинация аргументов не приведет к исключительному случаю. Функция возвращает точку в однородных координатах. Проверку на собственность/несобственность можно сделать уже вне функции. В случае несобственной точки для дальнейшего анализа необходимо использовать функцию для определения прямых пересечения плоскостей.

Еще одна функция реализует формулу для вычисления однородных координат точки пересечения плоскости $[\mathbf{n} \mid d]$ и прямой $\{\mathbf{v} \mid \mathbf{m}\}$:

$$\left(-\frac{(\mathbf{m}\times\mathbf{n}+d\mathbf{v})}{(n,v)}\middle|1\right) = \big(-(\mathbf{m}\times\mathbf{n}+d\mathbf{v}) \mid (\mathbf{n},\mathbf{v})\big) = \\ = \big((\mathbf{m}\times\mathbf{n}+d\mathbf{v}) \mid -(\mathbf{n},\mathbf{v})\big). \quad (16)$$

Здесь также можно поступить двояко: вернуть тип triple и встроить все проверки внутрь функции, или же вернуть тип Vec4 и возложить ответственность за проверки на пользователя функции. Здесь мы приводим первый вариант функции:

```
triple intersection(Plane3D plane, Line3D
↪  line) {

  assert( dot(line.v, plane.n) > 1e-8,
  ↪  "Плоскость и прямая не
  ↪  пересекаются!");

  return -(cross(line.m, plane.n) +
  ↪  plane.d * line.v) / dot(line.v,
  ↪  plane.n);
}
```

5. ПРИМЕР ВИЗУАЛИЗАЦИИ

В данном разделе рассмотрим пример использования созданных структур и встроенных в Asymptote средств для визуализации комплексного изображения пересечения плоскостей и прямых. Приведем полный код программы и прокомментируем все его части.

Программа рисует три плоскости, которые пересекаются в одной точке. Изображаются прямые пересечения каждой пары плоскостей. Точка пересечения плоскостей одновременно является точкой пересечения прямых, по которым пересекаются данные плоскости. Результат показан на рис. 3.

Третья плоскость специально выбрана так, чтобы ее нормальный вектор совпадал с направляющим вектором прямой пересечения первой и второй плоскостей, т.е. $\mathbf{n}_3 = \mathbf{n}_1 \times \mathbf{n}_2 = \mathbf{v}_{12}$. Это изображение может служить иллюстрацией к доказательству формулы нахождения прямой, по которой пересекаются две плоскости.

Файл с исходным кодом Asymptote имеет расширение **.asy**. Для его запуска будем использовать команду **asy**. Вначале следует импортировать все используемые модули как встроенные, так и реализованные нами.

```
import settings;

settings.outformat = "pdf";
settings.render = 15;
```

```
include "Vec4.asy";
include "Line3D.asy";
include "Plane3D.asy";

import three;
import graph3;


unitsize(3cm);
```

Модуль settings позволяет настроить некоторые параметры работы Asymptote. В частности мы указываем, что следует создавать изображение в формате pdf, а также выставляем качество созданного трехмерного изображения. Значения больше нуля приведут к тому, что изображение получится растровым. В случае трехмерных чертежей мы вынуждены использовать растровый формат, так как иначе некорректно рассчитываются наложения поверхностей друг на друга. Качество растра следует подбирать опытным путем.

Далее импортируем три созданные нами структуры модуль three для поддержки трехмерных векторов и модуль graph3 для визуализации трехмерных поверхностей и кривых. Также выставляем единицу измерения по умолчанию в 3 сантиметра — такое значение было подобрано опытным путем исходя из критерия читаемости изображения при его публикации в статье или презентации.

В модуле three определена функция plane, которая принимает в качестве аргументов два направляющих вектора плоскости и точку их приложения, а в качестве результата возвращает контур плоскости в виде прямоугольника. Точка приложения векторов будет служить правым верхним углом данного прямоугольника. Далее этот контур можно использовать для отображения плоскости с помощью функции surface.

В первую очередь задаем направляющие векторы:

```
triple u_01 = X;
triple v_01 = Y;

triple u_02 = rotate(30, Y) * u_01;
triple v_02 = v_01;

triple u_03 = u_01;
triple v_03 = rotate(-90, X) * v_01;
```

Для первой плоскости в качестве направляющих выберем орты осей *Ox* и *Oy*, используя заданные в модуле graph3 векторы X, Y.

Точки приложения для каждой плоскости следует задавать отдельно, иначе все плоскости будут визуально исходить из одной точки, что сделает рисунок ненаглядным:

```
// Произвольная точка, к которой будем
↪  привязывать направляющие векторы
triple P_0 = (0, -0.5, 1);
// Точка плоскости, от которой будут
↪  откладываться направляющие векторы при
↪  рисовании

triple P_01 = P_0 - u_01;
triple P_02 = P_0 - u_02;
triple P_03 = P_0 - X + 0.5Y + 0.5Z;
```

Далее можно создать контуры и поверхности плоскостей:

```
// Создаем контуры плоскостей
path3 pl_01 = plane(u=2u_01, v=v_01,
↪  O=P_01);
path3 pl_02 = plane(u=2u_02, v=v_02,
↪  O=P_02);
path3 pl_03 = plane(u=2u_03, v=v_03,
↪  O=P_03);

surface pls_01 = surface(pl_01);
surface pls_02 = surface(pl_02);
surface pls_03 = surface(pl_03);
```

Чтобы созданные объекты были отрисованы, необходимо воспользоваться функцией draw. Покажем как отрисовать первую плоскость:

```
draw(
  s=pls_01,
  nu=1, nv=1,
  surfacepen=lightgrey+opacity(.8),
  meshpen=black+thin(),
  light=nolight
);
```

Разберем передаваемые в данную функцию параметры:

- в параметр s передается объект surface;
- параметры nu и nv задают количество точек сетки сплайновой поверхности;
- параметр surfacepen позволяет настроить перо для рисования поверхности;
- параметр meshpen отдельно настраивает перо для сетки, в нашем случае от сетки остается только контур плоскости;
- с помощью параметра light настраивается свет.

Мы хотим сделать подписи к элементам рисунка, в частности подписать обозначения для плоскостей. Для этого отображаем контуры плоскости отдельно:

```
draw(pl_01, p=0.25bp+black,
 ↪  L=Label("$\pi_1$",
 ↪  position=Relative(0.80)));
draw(pl_02, p=0.25bp+black,
 ↪  L=Label("$\pi_2$"));
draw(pl_03, p=0.25bp+black,
 ↪  L=Label("$\pi_3$",
 ↪  position=Relative(0.3)));
```

Здесь параметр p задает перо, параметр L — подпись к изображаемому объекту. Для создания подписей можно использовать нотацию LaTeX, что делает Asymptote удобным для создания иллюстраций к научным текстам.

Далее перейдем к использованию созданных нами структур. В первую очередь зададим все три плоскости:

```
Plane3D plane01 = Plane3D(P_01, u_01,
 ↪  v_01);
Plane3D plane02 = Plane3D(P_02, u_02,
 ↪  v_02);
Plane3D plane03 = Plane3D(P_03, u_03,
 ↪  v_03);
```

Затем найдем прямые пересечения данных плоскостей:

```
Line3D line12 = intersection(plane01,
 ↪  plane02);
Line3D line13 = intersection(plane01,
 ↪  plane03);
Line3D line23 = intersection(plane02,
 ↪  plane03);
```

Прямые отображаются в виде отрезков, поэтому следует знать две точки каждой прямой. Подбираются точки эмпирически, а чтобы получить координаты точек используем метод get_point. Например, для первой прямой:

```
triple l12_sp = line12.get_point(-1.6);
triple l12_ep = line12.get_point(+0.9);
```

Далее отображаем прямые. Так, для первой прямой:

```
draw(l12_sp--l12_ep, p=black,
 ↪  L=Label("$l_{12}$",
 ↪  position=BeginPoint));
```

Теперь находим точку пересечения и отображаем ее:

```
triple intersection_point =
 ↪  intersection(plane01, plane02,
 ↪  plane03);

dot(intersection_point);
```

Наконец, откладываем нормальные векторы всех трех плоскостей от точки их пересечения:

```
draw(
  intersection_point -- intersection_point
   ↪  + plane01.n,
  arrow=Arrow3(size=4),
  p=black,
  L=Label(s="$\vb{n}_1$", align=E,
   ↪  position=Relative(0.9))
);
```

6. ЗАКЛЮЧЕНИЕ

Мы рассмотрели реализацию аналитической проективной геометрии на языке Asymptote. Были реализованы однородные координаты, координаты Плюккера прямой и однородные координаты плоскости. Подробно описаны три созданные нами структуры, инициализирующие операторы и функции. Фундаментальным преимуществом по сравнению с классической аналитической геометрией является существенное упрощение вычислений, так как большинство функций просто повторяют проективные формулы и вычислительная часть зачастую умещается в несколько строк.

Следует особо отметить, что так как все функции Asymptote предназначены для работы с объектами трехмерного декартова пространства, то для окончательной визуализации полученных объектов мы все же вынуждены делать проверки на равенство нулю тех или иных величин, а также учитывать невозможность однозначно визуализировать несобственные точки, прямые и плоскости. Однако это уже ограничения технического характера.

Дальнейшая работа может иметь два направления. В математическом плане все используемые нами формулы могут быть записаны в терминах геометрической алгебры. С вычислительной точки зрения это не дает никаких преимуществ, однако с теоретической точки зрения дает более общий и гибкий взгляд на геометрические объекты и их взаимное расположение в пространстве.

Второе направление — техническое. Оно заключается в том, чтобы реализовать набор функций для

визуализации точек, прямых и плоскостей, заданных непосредственно в проективном виде.

ИСТОЧНИКИ ФИНАНСИРОВАНИЯ

Публикация выполнена в рамках проекта № 021934-0-000 Системы грантовой поддержки научных проектов РУДН (Геворкян М.Н., Королькова А.В., Севастьянов Л.А.) и при поддержке Программы стратегического академического лидерства РУДН (Кулябов Д.С.).

СПИСОК ЛИТЕРАТУРЫ

1. *Korolkova A.V., Gevorkyan M.N., Kulyabov D.S.* Implementation of hyperboliccomplex numbers in Julia language, Discrete Contin. Models Appl. Comput. Sci., 2022, vol. 30, no. 4, pp. 318–329.
2. *Kulyabov D.S., Korolkova A.V., Sevastianov L.A.* Complex numbers for relativistic operations, 2021.
3. *Kulyabov D.S., Korolkova A.V., Gevorkyan M.N.* Hyperbolic numbers as Einstein numbers, J Phys.: Conf. Ser., 2020, vol. 1557, p. 012027.
4. *Gevorkyan M.N., Korolkova A.V., Kulyabov D.S.* Approaches to the implementation of generalized complex numbers in the Julia language, Workshop on Information Technology and Scientific Computing in the framework of the X Int. Conf. Information and Telecommunication Technologies and Mathematical Modeling of High-Tech Systems (ITTMM), Kulyabov, D.S., Samouylov, K.E., and Sevastianov, L.A., Eds., 2020, vol. 2639, pp. 141–157.
5. *Геворкян М.Н., Королькова А.В., Кулябов Д.С.* Реализация геометрической алгебры в системах символьных вычислений // Программирование. 2023. № 1. С. 48–55.
6. *Королькова А.В., Геворкян М.Н., Кулябов Д.С., Севастьянов Л.А.* Средства компьютерной алгебры для геометризации уравнений Максвелла // Программирование. 2023. Т. 49, № 4. С. 33–38.
7. *Велиева Т.Р., Геворкян М.Н., Демидова А.В. и др.* Аппарат геометрической алгебры и кватернионов в системах символьных вычислений для описания вращений в евклидовом пространстве // Журнал вычислительной математики и математической физики. 2023. Т. 63. № 1. С. 31–42.
8. *Bowman J.C.* Hammerlindl A. Asymptote: A vector graphics language, 2008, vol. 29, no. 2, pp. 288–294.
9. *Bowman J.C.* Asymptote: Interactive TEX-aware 3D vector graphics, 2010, vol. 31, no. 2, pp. 203–205.
10. *Shardt O., Bowman J.C.* Surface parameterization of nonsimply connected planar Bzier regions, Comput.-Aided Des., 2012, vol. 44, no. 5, pp. 484.e1–484.e10.
11. *Bowman, J.C.* Asymptote: The vector graphics language, 2023. https://asymptote.sourceforge.io.
12. *Gevorkyan M.N., Korolkova A.V., Kulyabov D.S.* Asymptote-based scientific animation, Discrete Contin. Models Appl. Comput. Sci., 2023, vol. 31, no. 2, pp. 139–149.
13. *Страуструп Б.* Программирование. Принципы и практика с использованием C++. 2 изд. Вильямс, 2018. 1328 с.
14. *Hobby J., Knuth D.* MetaPost on the Web. https://www.tug.org/metapost.html.
15. *Staats C.* An Asymptote tutorial, 2022. https://asymptote.sourceforge.io/asymptote_tutorial.pdf.
16. *Крячков Ю.Г.* Asymptote для начинающих. http://mif.vspu.ru/books/ASYfb.pdf.
17. *Волченко Ю.М.* Научная графика на языке Asymptote. http://www.math.volchenko.com/AsyMan.pdf.
18. *Ивальди Ф.* Евклидова геометрия на языке векторной графики Asymptote.2015. http://mif.vspu.ru/books/geometry_new_ru.pdf.
19. *Lengyel E.* Foundations of game engine development, Terathon Software LLC, vol. 1. http://foundationsofgameenginedev.com.
20. *Marschner S., Shirley P.* Fundamentals of Computer Graphics, CRC Press, 5 ed.

IMPLEMENTATION OF ANALYTIC PROJECTIVE GEOMETRY FOR COMPUTER GRAPHICS

© 2024 M. N. Gevorkyan[a], A. V. Korol'kova[a], D. S. Kulyabov[a, b], L. A. Sevast'yanov[a, b]

[a] *RUDN University, 6 Miklukho-Maklaya St, Moscow, 117198 Russia*
[b] *Joint Institute for Nuclear Research, 6 ul. Zholio-Kyuri 6, Dubna, Moscow oblast, 141980 Russia*

In their research, the authors actively exploit different branches of geometry. For geometric constructions, computer algebra approaches and systems are used. Currently, we are interested in computer geometry, more specifically, the implementation of computer graphics. The use of the projective space and homogeneous coordinates has actually become a standard in modern computer graphics. In other words, the problem is reduced to the application of analytic projective geometry. The authors failed to find a computer algebra system that could implement projective geometry in its entirety. Therefore, it was decided to partially implement computer algebra for visualization of algebraic relations. For this purpose, the Asymptote system was employed.

Keywords: projective geometry, Asymptote system, Plücker coordinates, proper and improper points, lines and planes

REFERENCES

1. *Korolkova A.V., Gevorkyan M.N., Kulyabov D.S.* Implementation of hyperboliccomplex numbers in Julia language, Discrete Contin. Models Appl. Comput. Sci., 2022, vol. 30, no. 4, pp. 318–329.
2. *Kulyabov D.S., Korolkova A.V., Sevastianov L.A.* Complex numbers for relativistic operations, 2021.
3. *Kulyabov D.S., Korolkova A.V., Gevorkyan M.N.* Hyperbolic numbers as Einstein numbers, J Phys.: Conf. Ser., 2020, vol. 1557, p. 012027.
4. *Gevorkyan M.N., Korolkova A.V., Kulyabov D.S.* Approaches to the implementation of generalized complex numbers in the Julia language, Workshop on Information Technology and Scientific Computing in the framework of the X Int. Conf. Information and Telecommunication Technologies and Mathematical Modeling of High-Tech Systems (ITTMM), Kulyabov, D.S., Samouylov, K.E., and Sevastianov, L.A., Eds., 2020, vol. 2639, pp. 141–157.
5. *Gevorkyan M.N., Korol'kova A.V., Kulyabov D.S.* Implementation of geometric algebra in symbolic computation systems, Programmirovanie, 2023, no. 1, pp. 48–55.
6. *Korol'kova A.V., Gevorkyan M.N., Kulyabov D.S., Sevast'yanov L.A.* Computer algebra tools for geometrization of Maxwell's equations, Program. Comput. Software, 2023, vol. 49, pp. 336–371.
7. *Velieva T.R., Gevorkyan M.N., Demidova A.V., Korol'kova A.V., Kulyabov D.S.* Geometric algebra and quaternion techniques in computer algebra systems for describing rotations in Eucledean space, Comput. Math. Math. Phys., 2023, vol. 63, pp. 29–39.
8. *Bowman J.C.* Hammerlindl A. Asymptote: A vector graphics language, 2008, vol. 29, no. 2, pp. 288–294.
9. *Bowman J.C.* Asymptote: Interactive TEX-aware 3D vector graphics, 2010, vol. 31, no. 2, pp. 203–205.
10. *Shardt O., Bowman J.C.* Surface parameterization of nonsimply connected planar Bzier regions, Comput.-Aided Des., 2012, vol. 44, no. 5, pp. 484.e1–484.e10.
11. *Bowman, J.C.* Asymptote: The vector graphics language, 2023. https://asymptote.sourceforge.io.
12. *Gevorkyan M.N., Korolkova A.V., Kulyabov D.S.* Asymptote-based scientific animation, Discrete Contin. Models Appl. Comput. Sci., 2023, vol. 31, no. 2, pp. 139–149.
13. *Stroustrup B.* Programming: Principles and Practice Using C++, Addison-Wesley Professional, 2014, 2nd ed.
14. *Hobby J., Knuth D.* MetaPost on the web. https://www.tug.org/metapost.html.
15. *Staats C.* An Asymptote tutorial, 2022. https://asymptote.sourceforge.io/asymptote_tutorial.pdf.
16. *Kryachkov Yu.G.* Asymptote for beginners. http://mif.vspu.ru/books/ASYfb.pdf.
17. *Volchenko Yu.M.* Scientific graphics in the Asymptote language. http://www.math.volchenko.com/AsyMan.pdf.
18. *Ivaldi, Ph.*, Euclidean Geometry with ASYMPTOTE, 2011.
19. *Lengyel E.* Foundations of game engine development, Terathon Software LLC, vol. 1. http://foundationsofgameenginedev.com.
20. *Marschner S., Shirley P.* Fundamentals of Computer Graphics, CRC Press, 5 ed.