Правительство Российской Федерации
Федеральное государственное автономное образовательное учреждение высшего профессионального образования

«Национальный исследовательский университет»
«Высшая школа экономики»

Отделение программной инженерии
Кафедра Управления разработкой программного обеспечения

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

На тему: Программа вычисления анимации 3D персонажа методом инверсной кинематики

Студента группы № 471ПИ
Охтеров Е. А.

Научный руководитель:
к.т.н., доцент
кафедры УРПО,
Гринкруг Е.М.

Москва, 2014 г.

Аннотация

Сегодня алгоритмы инверсной кинематики находятся на краю человеческих познаний о создании реалистичной анимации. Придя в компьютерную графику из робототехники, они претерпели множество изменений, получив новый толчёк для своего развития. В представленной работе описываются два общепринятых метода, которые пришли из робототехники: метод координатного спуска (CCD – Cyclic Coordinate Descent) и нахождение обратной матрицы Якоби (Jacobian Inverse). Но эти алгоритмы не идеальны. Чтобы была возможность разработать более мощные алгоритмы, нам необходимо понять связь между этой задачей и проблемой оптимизации. Преследуя эту цель, в этой работе алгоритмы CCD, Inverse Jacobian и BFGS описываются как эволюция мысли, что, вероятно, вдохновит читателя сделать какое-то обобщение и улучшить характеристики представленных алгоритмов.

Кроме того, данную работу сопровождает демонстрационное приложение, поэтому, помимо описания алгоритмов инверсной кинематики, здесь можно найти описание того, как происходит отображение объектов и описание архитектуры приложения. Эти знания должны помочь заинтересованному читателю быстрее разобраться с кодом программы и расширить её функционал.

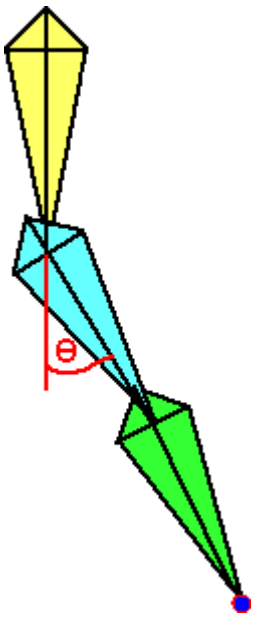
Содержание

1. Введение

Компьютерная графика – это многомиллиардная индустрия. Важной составляющей компьютерной графики является анимация. На сегодняшний день одна из самых технически сложных частей компьютерной анимации – это инверсная кинематика. Она используется в таких редакторах 3D графики, как 3DS MAX, Blender, Maya, CINEMA 4D, Vue и т.д. Кроме того её часто используют в играх и даже при создании фильмов. Чтобы понять, чем является инверсная кинематика, важно осознать саму концепцию кинематики.

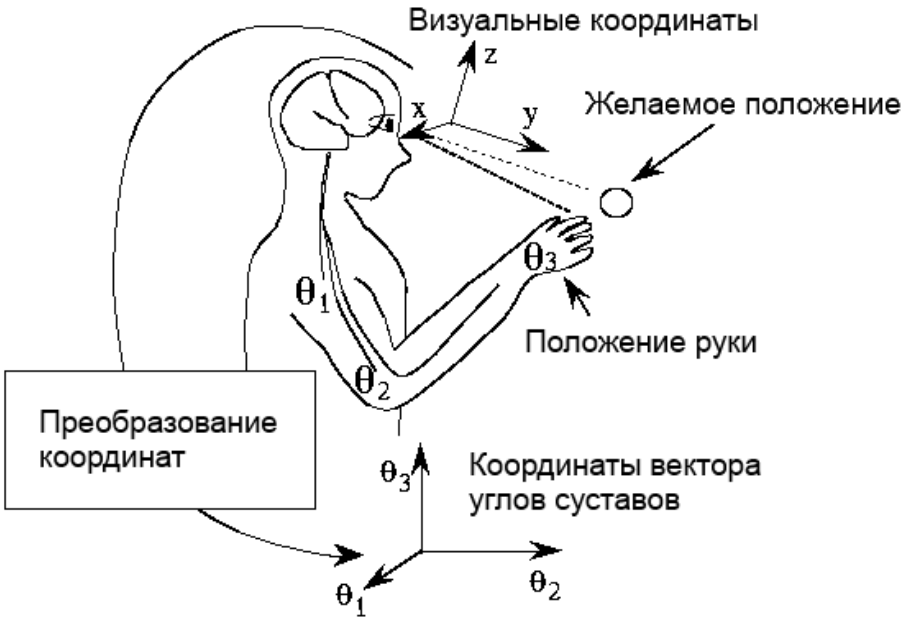
Кинематика – это один из разделов механики. Другой раздел, который исследует причины возникновения механического движения – динамика. На самом абстрактном уровне, механика оперирует с системой твёрдых тел (rigid multibody system), которая состоит из множества абсолютно твёрдых тел меньшего размера. Динамика системы твёрдых тел изучает движение под воздействием сил, приложенных к этой системе. Кинематика в свою очередь изучает математическое описание движения идеализированных тел, без рассмотрения причин движения. Другими словами, кинематика абстрагируется от законов физики и учитывает только геометрию движения. В компьютерной анимации кинематика делится на 2 вида: прямую и инверсную.

Прямую и инверсную кинематику можно описать, рассматривая работу аниматора. Аниматор создаёт видео с помощью специальных программ, которые позволяют ему создавать скелет объекта (т.е. структуру, состоящую из системы иерархических костей). Целью аниматора является изменение положения костей и запечатление поз объекта в виде снимков. Последовательность этих снимков и составляет анимацию. Прямая и инверсная кинематика определяют то, как аниматор управляет скелетом. Изображение 1 показывает часть скелета (кинематическую цепь). Если аниматор использует прямую кинематику, то ему придётся двигать кость за костью, чтобы установить объект в нужное положение. К примеру, если повернуть синюю кость на угол тэта, то заодно повернётся и её ребёнок – зелёная кость. Родительские кости прямой кинематикой не затрагиваются, поэтому создавать анимации сцен с большим числом костей этой техникой довольно трудно.



Изображение 1. Кинематическая цепь.

Инверсная кинематика, как и предполагает название, делает обратное. Эта техника позволяет аниматору определить желаемое положение end-effector'а (последняя кость в цепи), а углы всех родительских костей рассчитываются автоматически. Инверсная кинематика даёт возможность думать естественно. Например, если мы хотим нашей рукой дотянуться до стакана с водой, то нам не приходится думать о том, на какой угол нужно повернуть плечо, чтобы можно было повернуть локоть так, чтобы кисть достала до стакана. Мы просто думаем о том, что конец нашей руки должен соприкоснуться со стаканом, а мозг рассчитывает, как для этого нужно повернуть плечевой и локтевой суставы.



Изображение 2. Инверсная кинематика внутри нашей головы.

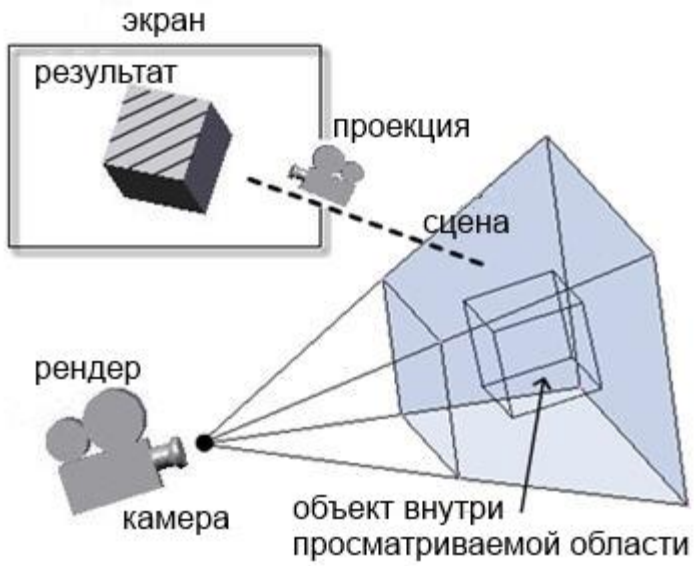
Проблема инверсной кинематики относится к области математического программирования. Следовательно, любые методы из теории оптимизации могут быть применимы к решению этой задачи. В качестве представителей 3-х разных классов таких методов для теоретического исследования были выбраны метод координатного спуска (CCD), Jacobian Inverse и BFGS.

Сложно думать о концепции инверсной кинематики в отрыве от того, как строится само приложение. Важно понимать, как она соотносится с базовыми концепциями компьютерной графики и как кинематика (инверсная кинематика в частности) встраивается в общую систему понятий компьютерной графики. Поэтому вначале будут даны основы компьютерной графики, на которой будут базироваться алгоритмы анимации.

2. Теория компьютерной графики

В этом разделе представлена необходимая теория компьютерной графики, которая использовалась при построении приложения.

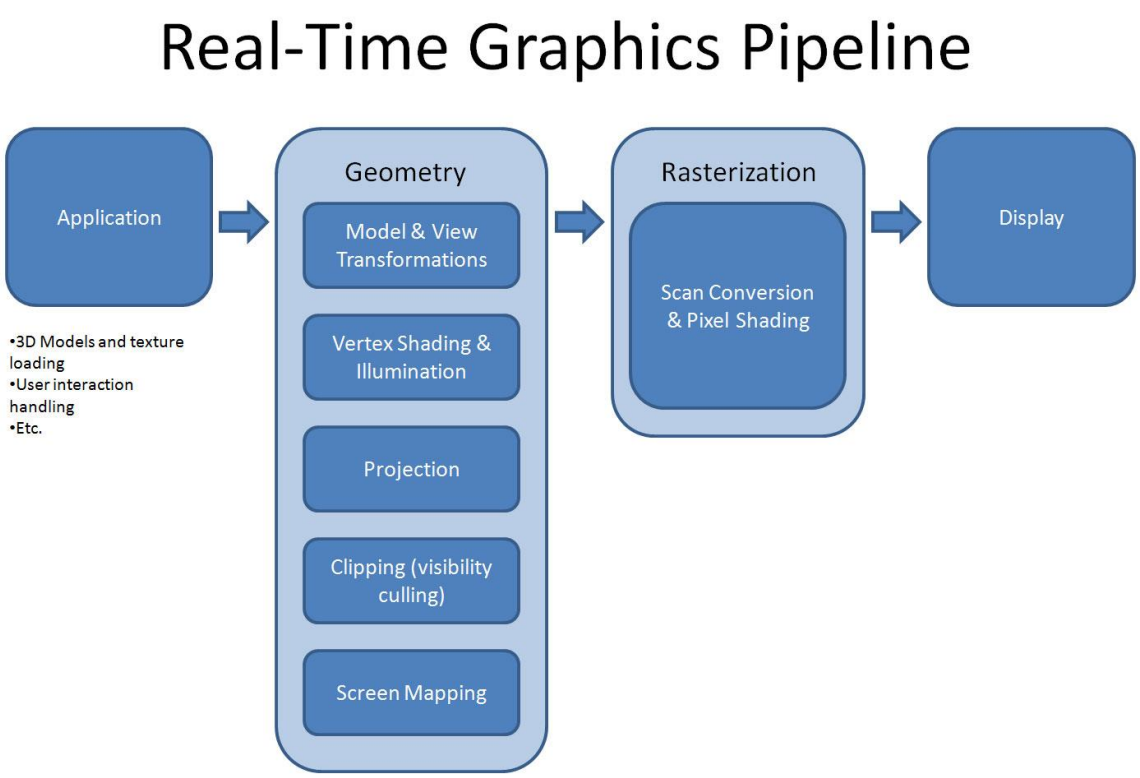
Ядро компьютерной графики – конвейер рендеринга графики (graphics rendering pipeline). Основная функция конвейера – генерация (рендеринг) двухмерного изображения из следующих составляющих: виртуальная камера, трёхмерные объекты, источники света, виртуальные материалы, текстуры и т.д. Процесс отображения виден на следующем изображении.



Изображение 3. Процесс рендеринга кубика на экран.

Расположение и форма объектов на экране определяются их геометрией, характеристиками среды и расположением камеры в этом виртуальном пространстве. Внешний облик объектов зависит от свойств материала, источников света, текстур и моделей распространения света (shading).

Графический конвейер представляет линейный процесс «сборки» примитивов из исходных данных о геометрии объектов. Этот процесс состоит из нескольких стадий. Пока не выполнена предыдущая стадия, мы не можем перейти к следующей, так как выходные данные предыдущего шага – это входные данные для следующего. Весь процесс идёт в сторону упрощения обработки данных. Чем дальше мы продвигаемся по конвейеру, тем меньше у нас контроля над обработкой данных. Концептуально конвейер можно представить в виде трёх больших процессов: прикладная стадия (Application), стадия обработки геометрии (Geometry) и растеризация (Rasterization).



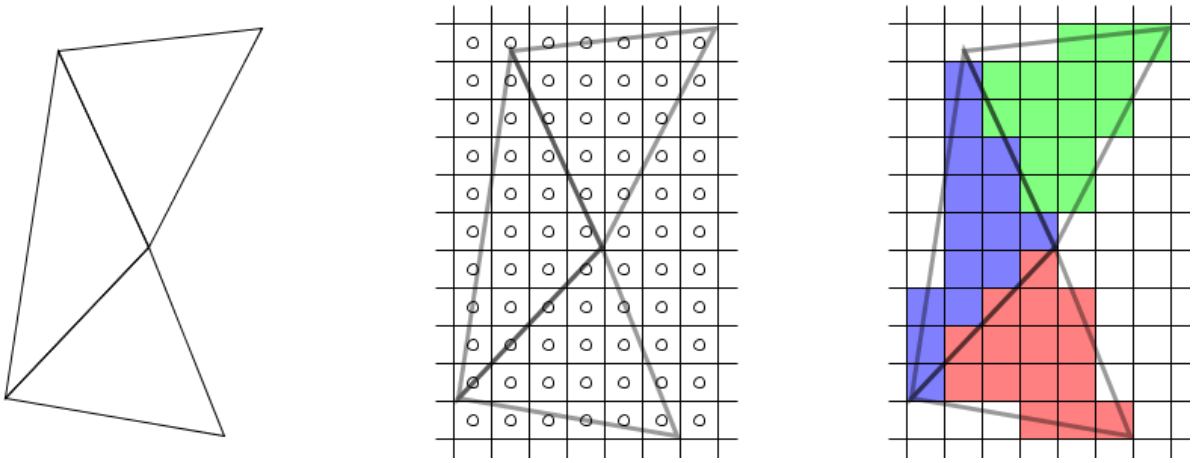
Изображение 4. Логические стадии графического 3D конвейера.

На прикладной стадии выполняется логика программы. Это самый свободный для программиста этап конвейера. Тут мы оперируем, например, такими объектами, как человек или камерой, смотрящей на человека. Можно, получив ввод от пользователя, как-то изменить положение этой камеры на виртуальной сцене. В результате работы всей этой логики мы получаем какое-то состояние сцены или по-другому – статический 3D кадр. Чтобы его увидеть, этот кадр нужно передать следующему этапу конвейера.

На стадии обработки геометрии мы мыслим языком полигонов (чаще треугольников) и вершин. Вся сцена на этом этапе представляется просто множеством примитивов, составленных из вершин. Основное предназначение этого шага – упрощение представления 3D данных и сведение их к представлению в 2D. Происходит избавление от иерархий зависимостей между объектами сцены (Model & View Transformations), освещение этих объектов с помощью источников света (Vertex Shading & Illumination), проекция полученных вершин на плоскость виртуальной камеры (Projection), отсечение вершин за пределами видимости камеры (Clipping, visibility culling) и, наконец, масштабирование проекции под размеры экрана (Screen Mapping). Сегодня все эти операции выполняются на процессоре видеокарты (GPU – Graphical Processing Unit) и чтобы управлять этим этапом пишутся специальные шейдерные программы на одном из трёх доступных языков – GLSL для API OpenGL, HLSL для API Direct3D или CG, который

работает как в API OpenGL, так и в API Direct3D. После того, как отработают вершинные шейдеры, получается множество 2D точек (образы 3D вершин) на экране и их цвет.

На последней стадии 2D точки с предыдущего этапа рассматриваются как вершины треугольников и цвета трёх вершин интерполируются, чтобы закрасить внутренность каждого из треугольников. Треугольники заполняются пиксель за пикселем и чтобы определить цвет текущего пикселя, выполняется ещё одна шейдерная программа – fragment shader, который также пишется на одном из языков – GLSL, HLSL или CG.

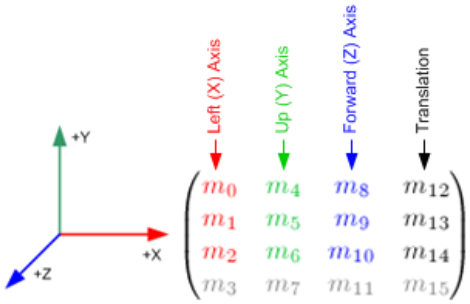


Изображение 5. Процесс растеризации треугольников на последней стадии.

2.1 Преобразования в трёхмерном пространстве

Важную роль в трёхмерной графике играют преобразования пространства. Они помогают позиционировать объекты на сцене, устанавливать пространственные взаимоотношения между этими объектами и перемещать предметы по виртуальной сцене. Три преобразования настолько важны и повсеместны, что у них есть свои названия – Model , View и Projection.

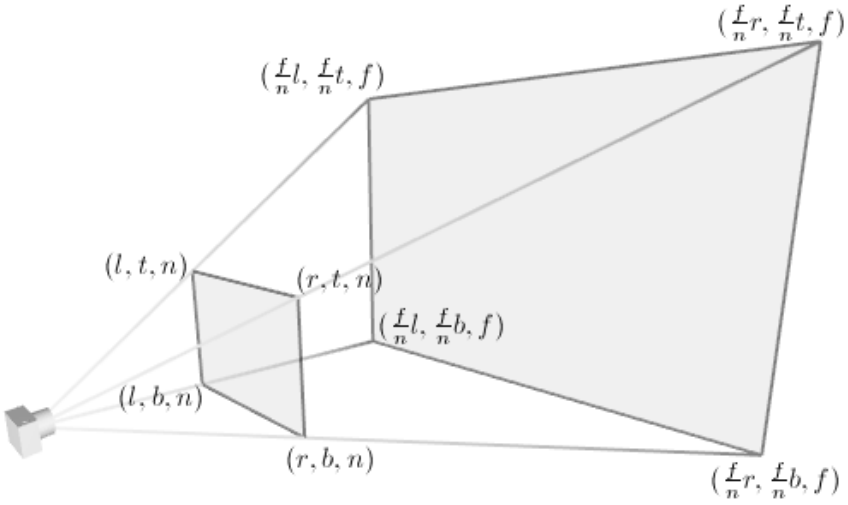
Вершины каждого объекта определяются в своём локальном пространстве (model space). Когда мы помещаем какой-то объект на сцену, нам нужно поставить его относительно других объектов в этой сцене. Для этого мы представляем, что у нас есть какая-то глобальная точка отсчёта и тогда относительно неё зададим положение и ориентацию нашего объекта с помощью матрицы Model. Таким образом, каждый предмет должен иметь свою матрицу, которая будет определять, как он расположен в глобальном пространстве.



Изображение 6. Матрица Model хранит в себе координаты осей системы координат, относительно которой определяются вершины объекта.

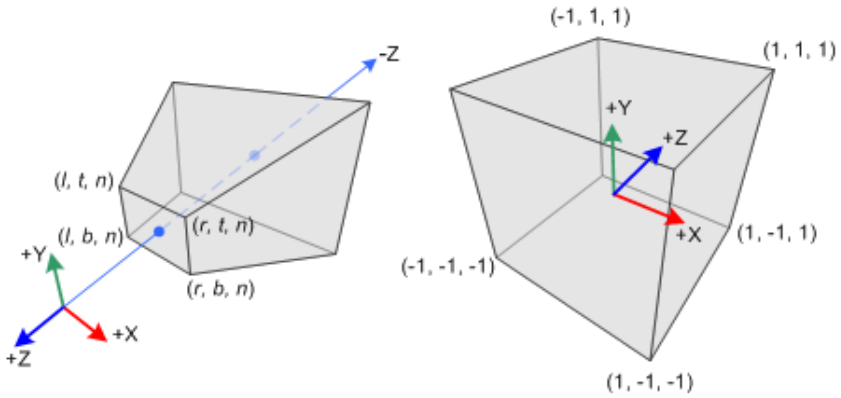
После того, как объекты на сцене размещены, нужно понять, как они располагаются относительно камеры. Для этого для камеры определяется своя матрица – такая же, как и для обычных объектов, но из-за того, что с ней нужно работать по-другому она имеет другое название – View.

Также с камерой ассоциируется матрица Projection, которая определяет характеристики камеры и выполняет проективное преобразование вершин. Эта матрица определяет размеры усечённой пирамиды, представленной на рисунке ниже.



Изображение 7. Матрица Projection определяет пространство видимости камеры – view frustum.

Действие этой матрицы можно представить как растяжение ближней к камере плоскости и одновременное сжатие дальней плоскости. Таким образом, те объекты, которые находятся ближе к камере, увеличатся в размерах, а те объекты, что дальше – уменьшатся.



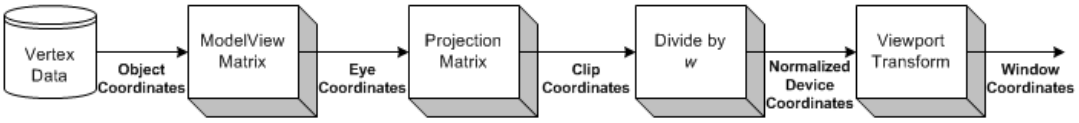
Изображение 8. Матрица Projection искажает пространство таким образом, чтобы усечённая пирамида (слева) стала кубом со стороной длиной два (справа).

Кроме того, матрица Projection определяет пределы видимости камеры. До преобразования эти пределы определяются точками (l, t, n), (l, b, n), (r, b, n), (r, t, n) – для передней плоскости и точками $(\frac{f}{n}l, \frac{f}{n}t, f)$, $(\frac{f}{n}l, \frac{f}{n}b, f)$, $(\frac{f}{n}r, \frac{f}{n}b, f)$, $(\frac{f}{n}r, \frac{f}{n}t, f)$ – для задней плоскости. После преобразования точек с помощью этой матрицы границы области видимости уже определяются вершинами единичного куба. Чтобы матрица Projection вела себя подобным образом, она строится по следующей формуле:

$$\begin{pmatrix} \frac{2n}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2n}{t-b} & 0 & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix},$$

где числа l, r, b, t, n, f – представляют собой координаты точек, показанных на изображении 7.

Теперь геометрические преобразования вершин на сцене можно представить в виде перемножения их координат на матрицы Model, View, Projection. На следующем рисунке этот процесс представлен более подробно.



Изображение 9. Преобразование 3D вершин из мировых координат в 2D координаты на экране.

После преобразования матрицей проекции получаются четырёхмерные координаты, которые затем нормализуются (если не были отсечены) путём деления на последний компонент вектора – w.

2.2 Задание иерархий с помощью кватернионов

Кватернионы в компьютерной графике используются для вращения объектов и задания их ориентации. Так как вращения не коммутативны, их нельзя считать векторами. Допустим, $T_x(d)$ – это смещение вдоль оси x на расстояние d и $R_x(\vartheta)$ – это поворот вокруг оси x на угол ϑ. Пусть то же будет верно для T_y, T_z, R_y и R_z. Тогда $T_y(45\text{ см})T_x(90\text{ см})$ коммутативно, то есть эквивалентно $T_x(90\text{ см})\ T_y(45\text{ см})$. Но $R_y(45^\circ)R_x(90^\circ) \neq R_x(90^\circ)R_y(45^\circ)$. Если же взять композицию $R_y(180^\circ)R_x(180^\circ)$ мы получим то же вращение, что и $R_z(180^\circ)$.

Эти элементарные наблюдения имеют далеко идущие последствия. Множество всех трёхмерных вращений организовано не как простое трёхмерное пространство, а как замкнутое трёхмерное многообразие (обобщение понятия поверхности), что также является группой. Эта группа также известна как $SO(3)$, от слов special и orthogonal.

Единичные кватернионы обладают таким свойством, что они охватывают геометрию, топологию и структуру группы трёхмерных вращений простейшим способом (они формируют универсальное покрытие).

Кватернионы могут быть определены несколькими эквивалентными способами, но полезно знать все из них. Исторически кватернионы осознавались Гамильтоном как расширение комплексных чисел $w + ix + jy + kz$, где $i^2 = j^2 = k^2 = -1$, $ij = k = -ji$. Единственное отличие от обычной арифметики действительных чисел – некоммутативность умножения. У Гамильтона также было более абстрактное представление о кватернионах как о простых четвёрках действительных чисел $[w, x, y, z]$. Так получилось, что компоненты группируются на 2 части – чисто мнимую (x, y, z), её он назвал *вектором*, и действительную часть w – её он назвал *скаляром*. Позже Гиббс, используя терминологию Гамильтона, вытащил из операций над кватернионами операции над векторами и сформировал векторную арифметику. Из-за этого сегодня мы обычно записываем кватернионы как $[w, \vec{\boldsymbol{v}}] = [w, x, y, z]$.

Центральный факт, который заставляет нас пользоваться кватернионами для задания ориентации и вращения объектов состоит в следующем. Если кватернион определён так $q = [\cos\vartheta\,, \sin\vartheta\,\vec{\boldsymbol{v}}]$, то операция qpq^{-1} выполнит вращение вокруг оси $\vec{\boldsymbol{v}}$ на угол 2ϑ.

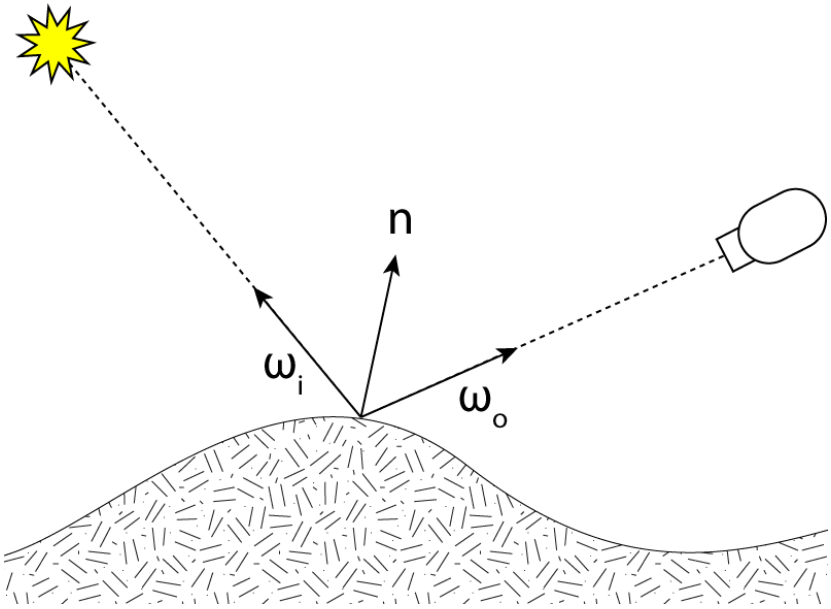
2.3 Освещение сцены

Для того чтобы видеть объекты на сцене, нам нужно использовать освещение и модели распространения света. В 1965 году Эдвард Никодемус определил функцию, моделирующую отражение света от непрозрачной поверхности

$$f_r(\omega_i, \omega_o) = \frac{dL_r(\omega_o)}{dE_i(\omega_i)} = \frac{dL_r(\omega_o)}{L_i(\omega_i)\cos\vartheta_i \, d\omega_i},$$

где L – это яркость, E – это освещённость и ϑ_i – угол между ω_i и нормалью n.

Она получила название как двулучевая функция отражательной способности (ДФОС или BRDF – bidirectional reflectance distribution function).

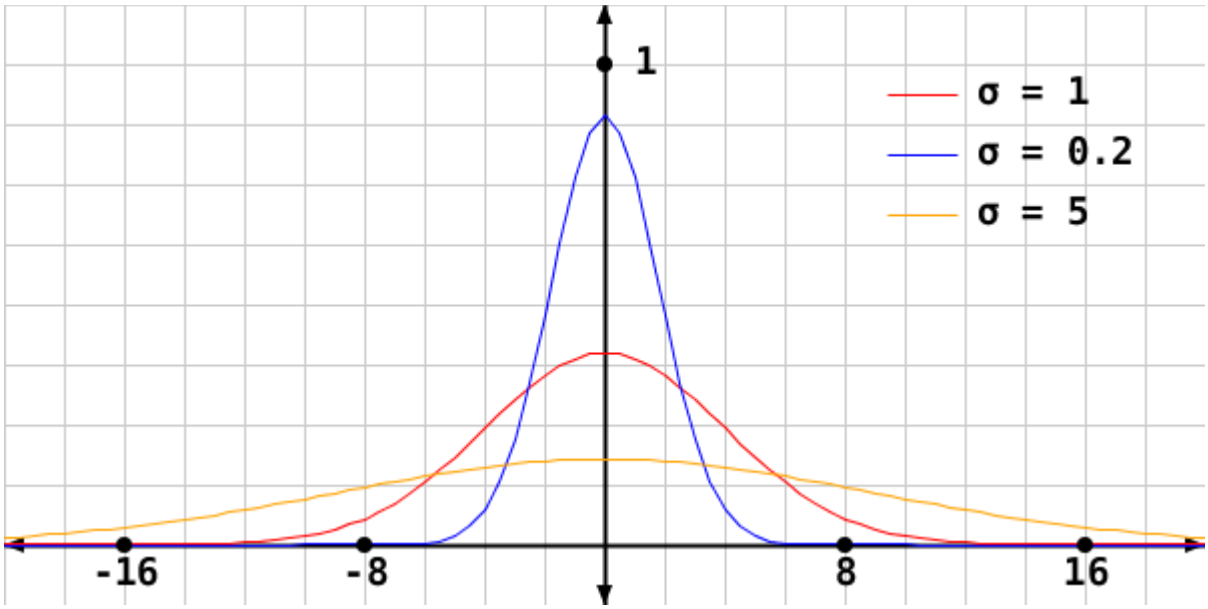


Изображение 10. Векторы, используемые в функции BRDF.

В большинстве случаев, используются модели функции BRDF – модель Ламберта и Блинна-Фонга. Свет разбивается на две компоненты – diffuse и specular. Соответственно, появляются компоненты diffuse и specular как у источника света, так и у материала, на который этот свет будет падать. Также используются 3 модели самого источника света, как объекта: 1. бесконечно удалённый источник света (например, Солнце), 2. точечный источник света (лампа в комнате), 3. конус света (настольная лампа). Цвет и яркость пикселя определяется как характеристиками источника света и материала, так и физическими законами распространения света.

Модели Ламберта и Блинна-Фонга очень быстры, но довольно примитивны. Они позволяют моделировать металлическую и пластиковую поверхности (даже они моделируются чересчур идеально, чтобы казаться реальными). Поэтому в компьютерной графике разработаны более сложные модели, которые позволяют моделировать широкий спектр материалов.

Простейшее усложнение модели, которое можно произвести с моделью Ламберта. Представим, что поверхность материала состоит из множества более маленьких поверностей. И каждая из этих поверхностей повёрнута в свою сторону (соответственно и нормаль). И тогда зададимся вопросом – какое количество микроповерхностей ориентированы таким образом, что они отражают свет в сторону камеры? Чем больше микроповерхностей ориентировано соответствующим образом, тем сильнее будет отражённый свет. Распределение микроповерхностей можно задать с помощью с помощью гауссовской функции $f(x) = \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$.



Изображение 11. Функция плотности гауссовского распределения для разных параметров σ.

Математическое ожидание μ для нас – это нормаль к поверхности. Значения функции распределения X представляют собой отклонения в градусах от нормали.

Дальнейшее усложнение diffuse модели приводит к технике Subsurface Scattering (когда свет также распространяется под поверхностью материала, как, например, у кожи или бумаги; эта модель учитывает, что свет может выйти из места, отличного от того, куда он зашёл и даже сразу из нескольких мест). Более того, усложняется исходная функция BRDF, и она становится

функцией двунаправленного поверхностного рассеивания отражения (BSSRDF – bidirectional scattering distribution function).

Усложнение модели распространения света приводит к технике Occlusion Mapping, когда учитывается, что объект может затенять сам себя.

На гране вычислительной мощности сегодня находятся техники глобального освещения (Global Illumination), когда мы считаем, что свет от объекта не сразу отражается к нам в камеру, а также распространяется в среду, где он взаимодействует с поверхностями других объектов, меняя цвет и постепенно угасая в интенсивности, пока не перестанет быть видимым глазу. В такой максимально приближенной к реальности модели источниками света становятся все объекты. Что ещё хуже – каждая точка поверхности рассматривается как источник света.

Теперь, когда стало понятно, как устроено 3D приложение, как размещаются объекты на сцене и затем отображаются на экране, описание контроля объектов на сцене кажется естественным продолжением общей теории 3D графики.

2.4 Теория контроля объектов на сцене

Виртуальная сцена может содержать тысячи объектов. Если вообразить статическую сцену, где объекты не меняют своего положения, то никаких проблем не возникает (кроме бесполезности этой сцены). Но если появляется потребность в перемещении объектов друг относительно друга, то не хотелось бы держать в голове положение всех 100 000 предметов на сцене. По этой же причине в армии существует иерархия. Сложно генералу отдавать приказ каждому солдату в отдельности, тем более что для какой-то группы солдат приказ будет один и тот же.

В компьютерной графике иерархию объектов моделирует *граф сцены* (scene graph). Положение каждого объекта должно быть определено относительно какого-то родительского объекта. Если обратиться к абстракции из математики, то *граф сцены* – это иерархия систем координат. Каждая система координат определяет своё пространство, в котором живёт объект (относительно этой системы координат задаются положения вершин). Взаимоотношение между двумя пространствами можно определить с помощью матрицы. Но для 3D мира матрица мира матрица 3×3 будет излишней. Достаточно знать d – расстояние до начала системы координат и ориентацию (w, q_x, q_y, q_z) дочернего узла относительно родительского. Применив ещё одну математическую абстракцию, можно сказать, что есть некоторая функциональная зависимость

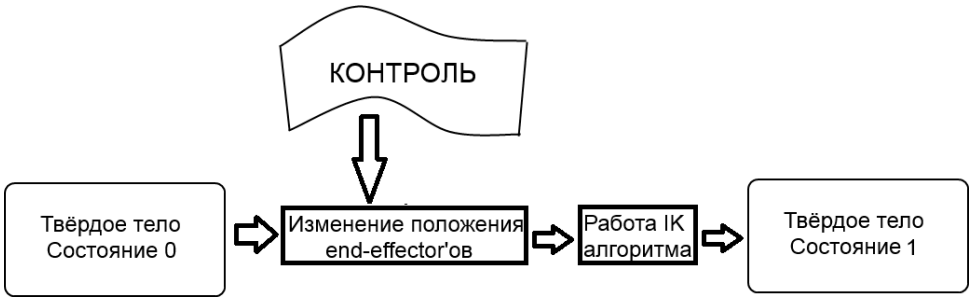
дочернего узла от родительского (не смотря на то, что дочерний узел сам «выбирает» как ему располагаться).

Как было написано во введении, кинематика оперирует с системой твёрдых тел. Эта система обычно моделируется с помощью набора костей (bones), соединённых суставами (joints). Есть несколько типов суставов: раздвижные, винтовые, поступательные и т.д. Но самый популярный тип – это вращающийся сустав (rotational joint). Для моделирования состояния одного вращающегося сустава нам нужно 4 числа – кватернион, либо ось вращения и угол поворота. Если смотреть более абстрактно, то каждый вращающийся сустав представляет собой систему координат. Так что можно задавать состояние каждого сустава матрицей. Множество таких кватернионов (либо матриц) формирует конфигурацию кинематической цепи.

Теперь можно совершить последний шаг абстракции и перейти от слов к символам.

Задача инверсной кинематики изначально предполагает управление положением конечного сустава. Поэтому у него даже есть особое название – end-effector. Положение каждого end-effector’а – это функция от состояний суставов кинематической цепи $s_i(\theta_1, \theta_2, \dots, \theta_n)$.

Вся система твёрдых тел (multibody) контролируется путём указания положения для end-effector’ов.



Изображение 12. Процесс контроля состоянием твёрдого тела.

Каждый сустав (или кость) в кинематической цепи определяет свою систему координат. Положение и ориентация дочерних костей задаются в родительской системе координат. Положение и ориентация – это набор преобразований: перенос $T(x, y, z)$ и поворот $R(\theta)$. Мы можем думать о суперпозиции этих преобразований как о разнице между родительским и дочерним звеном:

$$M_i = T(x_i, y_i, z_i)R(\theta_i), \qquad (1)$$

где x_i, y_i, z_i – это положение дочерней кости в родительской системе координат, а θ_i – это угол поворота и ось вращения.

В этой нотации, если у нас есть «i-1»-ый сустав, тогда дочерний сустав «i» конструируется путём преобразования «i-1»-ого сустава с помощью матрицы M_i. В общем случае соотношение между двумя любыми звеньями «i» и «j» в кинематической цепи находится с помощью следующей матрицы преобразования:

$$M_i^j = M_j M_{j-1} \dots M_{i+1} M_i. \quad (2)$$

Эта формула говорит нам, что положение (x, y, z) и ориентация (α, β, γ) последнего звена в кинематической цепи находятся простым перемножением матриц каждого звена всей цепи.

Допустим q – это -вектор параметров: $q = ((x_1, y_1, z_1, \alpha_1, \beta_1, \gamma_1), \dots, (x_1, y_1, z_1, \alpha_1, \beta_1, \gamma_1))$. Тогда проблема прямой кинематики вычисления положения и ориентации X последнего звена в цепи может быть записана следующим образом:

$$X = F(q), \quad (3)$$

где $F(q)$ – это функция, в которой производится перемножение матриц, связанных соответствующими параметрами вектора q.

Теперь легко поставить задачу инверсной кинематики:

$$q = F^{-1}(X), \quad (4)$$

где $F^{-1}(X)$ – это преобразование, которое находит параметры всей кинематической цепи, основываясь на заданных параметрах последнего звена в цепи X.

Задача инверсной кинематики гораздо более сложная, потому что нам нужно определить $n - 1$ преобразование вместо одного, в случае (3). Также часто случается, что уравнение (4) имеет бесконечное множество решений или не имеет решений вообще. Общего аналитического решения для произвольной кинематической цепи не существует. Все популярные методы решения уравнения (4) основываются на методах, которые предполагают математическое программирование.

2.4.1 Метод координатного спуска (CCD)

Самый лёгкий способ реализации инверсной кинематики – это метод координатного спуска, впервые предложенный Вангом и Ченом [3]. Основная идея оптимизационных методов заключается в построении задачи минимизации по уравнению (4). Рассмотрим расположение последнего звена в цепи X в положении P. Расстояние от X до P будет служить в качестве меры ошибки:

$$E(q) = (P - X)^2. \quad (5)$$

Положение и ориентация звена X зависит от конфигурации всей цепи q. Изменяя параметры цепи, мы либо продвигаемся в сторону целевого положения P (тем самым уменьшая ошибку $E(q)$), либо отдаляемся от него. Теперь можно поставить задачу формально:

$$\begin{cases} \min E(q), \\ \text{при ограничениях на параметры: } l_i \le q_i \le u_i, i = 1 \dots n, \end{cases} \quad (6)$$

где l_i и u_i – это нижняя и верхняя границы на параметры кости «i».

В такой постановке – это обычная задача нелинейной оптимизации, которая может быть решена итеративно. На каждой итерации алгоритм двигается от последнего звена в кинематической цепи до её корня. После изменения параметров конфигурации q, алгоритм CCD оценивает, насколько хорошо мы приблизились с помощью функции (5). Значение этой функции помогает нам принять решение, делать ли следующую итерацию или нет. Алгоритм останавливается, когда $E(q)$ становится меньше некоторого определённого значения ε. Над каждым суставом мы выполняем преобразование, которое приближает X как можно ближе к P.

Можно понять всю работу алгоритма на примере одной итерации. Допустим, мы знаем положение текущего звена $\boldsymbol{p}$, положение конечного звена $\boldsymbol{e}$ и целевое положение конечного звена $\boldsymbol{g}$. Тогда мы можем построить следующие векторы: $\vec{u} = \frac{\boldsymbol{e}-\boldsymbol{p}}{|\boldsymbol{e}-\boldsymbol{p}|}, \vec{v} = \frac{\boldsymbol{g}-\boldsymbol{p}}{|\boldsymbol{g}-\boldsymbol{p}|}$. Теперь мы можем вычислить ось вращения и угол поворота по следующим формулам:

$$axis = \vec{u} \times \vec{v}, \quad (7)$$

$$angle = \operatorname{acos}(\vec{u} \cdot \vec{v}). \quad (8)$$

Теперь мы можем построить кватернион, с помощью которого мы будем вращать сустав

$$Q = \left(\cos\left(\frac{angle}{2}\right), \sin\left(\frac{angle}{2}\right) * \left(\frac{axis}{|axis|}\right)\right). \quad (9)$$

2.4.2 Jacobian Inverse

Матрица Якоби – это матрица J, состоящая из частных производных первого порядка векторозначной функции. Допустим, $F: \mathbb{R}^n \to \mathbb{R}^m$ – это некоторая матрица преобразования. Мы можем её записать в другой нотации, близкой к определению конфигурации кинематической цепи $\boldsymbol{q}$, $F_1(q_1, \dots, q_n), \dots, F_m(q_1, \dots, q_n)$. Тогда матрица Якоби будет выглядеть так:

$$J = \begin{pmatrix} \frac{\partial F_1}{\partial q_1} & \cdots & \frac{\partial F_1}{\partial q_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial F_m}{\partial q_1} & \cdots & \frac{\partial F_m}{\partial q_1} \end{pmatrix}. \quad (10)$$

Матрица Якоби описывает ориентацию касательной плоскости к графику функции в заданной точке. В этом контексте эта матрица – обобщение градиента функции, значения которой – скаляр (на самом деле, каждый столбец матрицы J – это градиент). О матрице J можно ещё думать как о том, насколько сильно функция растянется после применения преобразования (10).

Важность матрицы Якоби заключается в том, что она позволяет нам построить лучшее линейное приближение функции вокруг заданной точки. Так как уравнение (4) не линейно (из-за синусов и косинусов в матрице вращения), обратная задача довольно сложна. Естественным будет такой подход, когда мы линеаризуем задачу вокруг текущей конфигурации кинематической цепи. Для начала нам нужно выполнить разложение в ряд Тейлора функции $F(\boldsymbol{q} + \Delta\boldsymbol{q})$:

$$F(\boldsymbol{q} + \Delta\boldsymbol{q}) = F(\boldsymbol{q}) + J(\boldsymbol{q})\Delta\boldsymbol{q} + o(|\Delta\boldsymbol{q}|). \quad (11)$$

Теперь мы откидываем радикал, чтобы сделать уравнение линейным и таким образом, мы сформировали основу для контроля над движением:

$$\Delta\boldsymbol{q} = J(\boldsymbol{q})^{-1}\big(F(\boldsymbol{q} + \Delta\boldsymbol{q}) - F(\boldsymbol{q})\big). \quad (12)$$

Выражение $\big(F(\boldsymbol{q} + \Delta\boldsymbol{q}) - F(\boldsymbol{q})\big) = \boldsymbol{X_1} - \boldsymbol{X_0}$ представляет собой разницу между желаемым и текущим решением. Это означает, что у нас теперь есть итеративная схема для нахождения корректной конфигурации. Мы не можем получить решение за один шаг, потому что приближение Якобианом верно лишь в эпсилон окрестности. Чем мы идём дальше от нашей начальной конфигурации, тем большую роль начинают играть нелинейные члены в нашем разложении Тейлора.

Чтобы получить преобразование из $\Delta\boldsymbol{q}$, нам нужно делать шаг интегрирования в конце каждой итерации. Простейший метод интегрирования – метод Эйлера:

$$\boldsymbol{q} = \boldsymbol{q} + h\,\Delta\boldsymbol{q}. \quad (13)$$

Этот метод очень может быть довольно неточным, так как он предполагает константную скорость на всём шаге интегрирования. Тем не менее, метод может неплохо работать для маленьких кинематических цепей. Для длинных цепей лучше использовать метод Рунге-Кутта с адаптивным размером шага.

У нас также есть проблема с инвертированием Якобиана, так как он не квадратный. По этой причине $J(\boldsymbol{q})^{-1}$ заменяют на псевдообратную матрицу $J(\boldsymbol{q})^{+}$, которая вычисляется с

помощью сингулярного разложения (SVD – Singular Value Decomposition). Сам Якобиан можно легко построить, рассматривая саму геометрию задачи: $\vec{\boldsymbol{a}} \times (\boldsymbol{e} - \boldsymbol{p})$, где $\vec{\boldsymbol{a}}$ – ось вращения, $\boldsymbol{e}$ – end-effector, $\boldsymbol{p}$ – положение сустава (кости).

2.4.3 Метод Broyden-Fletcher-Goldfarb-Shanno (BFGS)

Метод BFGS – это численный метод для решения задач нелинейной оптимизации. BFGS – это приближение метода Ньютона, и он принадлежит к классу квазиньютоновских методов.

Метод Ньютона итеративный и специализируется на нахождении стационарных точек дифференцируемых функций. Он пытается построить последовательность x_n (из начального предположения x_0), которая сходится к x_*, такой что $f'(x_*) = 0$. Разложение Тейлора второго порядка функции вокруг x_n:

$$f_T(x_n + \Delta x) = f_T\big(x_n + (x - x_n)\big) = f_T(x) = f(x_n) + f'(x_n)\Delta x + \frac{1}{2}f''(x_n)\Delta x^2. \quad (14)$$

Это разложение представляет собой квадратичную функцию, где $f(x_n)$, $f'(x_n)$, $\frac{1}{2}f''(x_n)$ – это всё константы. Так что эта функция достигает своего экстремума, когда её производная по Δx равна нулю.

$$f'(x_n) + f''(x_n)\Delta x = 0. \quad (15)$$

$$\Delta x = x - x_n = -\frac{f'(x_n)}{f''(x_n)}. \quad (16)$$

Уравнение (16) можно интерпретировать как нахождение размера шага от точки x_n, который нужно совершить, чтобы переместиться к локальному экстремуму функции f вокруг точки x_n. Если вдруг окажется, что функция f квадратична, тогда f_T будет точным представлением и уравнение (16) сойдётся за один шаг. Иначе, это всего лишь приближение и нам нужно сходиться к экстремуму итерациями:

$$x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)},\ n = 0,1, \dots . \quad (17)$$

Эту идею нельзя применить напрямую к задаче инверсной кинематики, так как целевая функция $F(\boldsymbol{q})$ – это вектор-функция. Поэтому метод Ньютона был расширен для больших размерностей [4]:

$$\boldsymbol{x_{n+1}} = \boldsymbol{x_n} - [Hf(\boldsymbol{x_n})]^{-1}\nabla f(\boldsymbol{x_n}),\ n = 0, 1, \dots . \quad (18)$$

Когда мы не достаточно близко к минимуму, то шаг итерации даже с положительно определённым гессианом $Hf(\boldsymbol{x_n})$ не обязательно уменьшит целевую функцию. Мы можем перескочить слишком далеко из-за плохого приближения квадратичной функции. Поэтому

метод Ньютона обычно модифицируется с помощью константы $\alpha \in (0, 1)$, определяющей размер шага:

$$\boldsymbol{x_{n+1}} = \boldsymbol{x_n} - \alpha[Hf(\boldsymbol{x_n})]^{-1}\nabla f(\boldsymbol{x_n}),\ n = 0, 1, \ldots . \quad (19)$$

Инвертирование Гессиана при больших размерностях – это довольно дорогая операция. Более того, нет гарантий, что Гессиан положительно определён. Полный ньютоновский шаг с настоящим Гессианом может перекинуть нас в точку, где целевая функция увеличивается в значении. Идея квазиньютоновских методов состоит в том, чтобы начать с положительно определённого, симметричного приближения H и достраивать дальнейшие приближения таким образом, чтобы они оставались положительно определёнными и симметричными. Находясь далеко от минимума, эта стратегия гарантирует нам, что мы всегда спускаемся вниз. Вблизи к минимуму, наша формула обновления матрицы будет приближаться к истинному Гессиану и у нас будет квадратичная скорость сходимости метода Ньютона.

В итоге, алгоритм BFGS выглядит следующим образом:

Из начального предположения $\boldsymbol{x_0}$ и приближения инвертированного Гессиана B_0^{-1} делать следующие шаги до тех пор, пока $|\nabla f(x_n)| < tolerance\_value$, где n – это номер последнего шага.

1. Получаем вектор, направляющий нас вниз $\boldsymbol{p_k} = -B_k^{-1}\nabla f(\boldsymbol{x_k})$.
2. Делаем «backtracking line search», чтобы определить размер шага α_k и делаем обновление $\boldsymbol{x_{k+1}} = \boldsymbol{x_k} - \alpha_k \boldsymbol{p_k}$.
3. $\boldsymbol{s_k} = \alpha_k \boldsymbol{p_k}$.
4. $\boldsymbol{y_k} = \nabla f(\boldsymbol{x_{k+1}}) - \nabla f(\boldsymbol{x_k})$.
5. $B_{k+1}^{-1} = B_k^{-1} + \frac{(\boldsymbol{s_k^T y_k} + \boldsymbol{y_k^T} B_k^{-1} \boldsymbol{y_k})(\boldsymbol{s_k s_k^T})}{(\boldsymbol{s_k^T y_k})^2} - \frac{B_k^{-1}\boldsymbol{y_k s_k^T} + \boldsymbol{s_k y_k^T} B_k^{-1}}{\boldsymbol{s_k^T y_k}}$.

3. Проектирование архитектуры приложения

3.1 Философия проектирования архитектуры

Раньше, когда проекты были маленькими, было достаточно немного подумать и сразу начать писать приложение. Так как мы склонны применять один и тот же инструмент для всё более широкого круга задач до тех пор пока он не даст сбой, так и я применил то, что работало на маленьких проектах, но не сработало в этот раз. Попробовав написать код сразу, я столкнулся с тем, что мне приходилось переписывать большие участки кода по десять или более раз. Чем больше становилось кода, тем более громоздкой и хрупкой становилась эта конструкция. Со временем весь код перестал помещаться в моей голове, и я перестал понимать, куда я двигаюсь и что мне делать дальше.

После небольшого отчаяния я вспомнил, откуда появилось ООП и что именно оно решает. Я удалил половину написанного кода и начал проектировать архитектуру, начиная рассуждения с того, как обеспечить функциональность верхнего уровня.

Что должна делать программа? В наиболее общем представлении – отображать параллелепипеды, сцеплённые друг с другом. При этом должна быть возможность изменять положение конечного параллелепипеда и на это изменение программа должна автоматически пересчитать положение промежуточных параллелепипедов в цепочке. С этим верхнеуровневым описанием функциональности уже можно начать планировать архитектуру.

Воспользовавшись практикой ООП, разобъём общую функциональность на более мелкие функциональные блоки.

Теперь разобъём эти верхнеуровневые блоки на как можно более маленькие функциональные блоки. Каждый блок должен отвечать только за одно очень простое действие, но должен делать это хорошо и независимо от других блоков (желательно).

Сперва опишу блоки представления данных.

Mesh – это описание одного «параллелепипеда». Будет хранить координаты точек $(v_1, \dots, v_n)$ и материал, из которого сделан объект.

Bone – кость используется для представления иерархии, как и у человека. С костью ассоциируется «мясо» (mesh). Если смотреть математически, то bone задаёт систему координат для вершин mesh. Это позволит, изменив параметры одной кости, косвенно изменить сразу n вершин.

Skeleton – скелет имеет верхнеуровневое представление об иерархии костей, о последовательности их сцепления. Он также знает, какая кость первая, а какая последняя. Также skeleton – это интерфейс для других модулей, которые будут работать с этой иерархией.

Теперь можно описать блоки, представляющие алгоритмы инверсной кинематики и отображение данных.

Solver – решатель содержит в себе различные алгоритмы для изменения координат костей, а также будет интерфейсом для сущности, которая пожелает изменить end-effector.

Graphics – модуль, который умеет отображать иерархию костей, используя для этого разные модели освещения и шейдерные программы GLSL для API OpenGL.

Application – сущность, которая умеет пользоваться созданными функциональными блоками.

3.2 Архитектура системы модулей OGRE + IK

После того, как была реализована архитектура, описанная в разделе 3.1, стало понятно, что, так как алгоритмы инверсной кинематики выделены в отдельный модуль, их можно попробовать интегрировать в какую-нибудь существующую графическую библиотеку.

В открытом доступе есть множество библиотек, которым не помешала бы моя реализация модуля *Solver*. Я стал исследовать разные графические библиотеки на предмет чистоты и понятности кода, чтобы можно было легко внедрить реализацию алгоритма

инверсной кинематики в виде отдельного модуля. В итоге самыми понятными движками оказались IrrLicht и OGRE. Фреймворк IrrLicht очень мощный – он охватывает как видеообработку, так и обработку звука, джойстиков и много других устройств. Не смотря на то, что для его разработки используется C++, его архитектура больше напоминает C с классами. Для того, чтобы внедрить свои алгоритмы, пришлось бы изменять немало когда самого IrrLicht, а это плохая практика программирования.

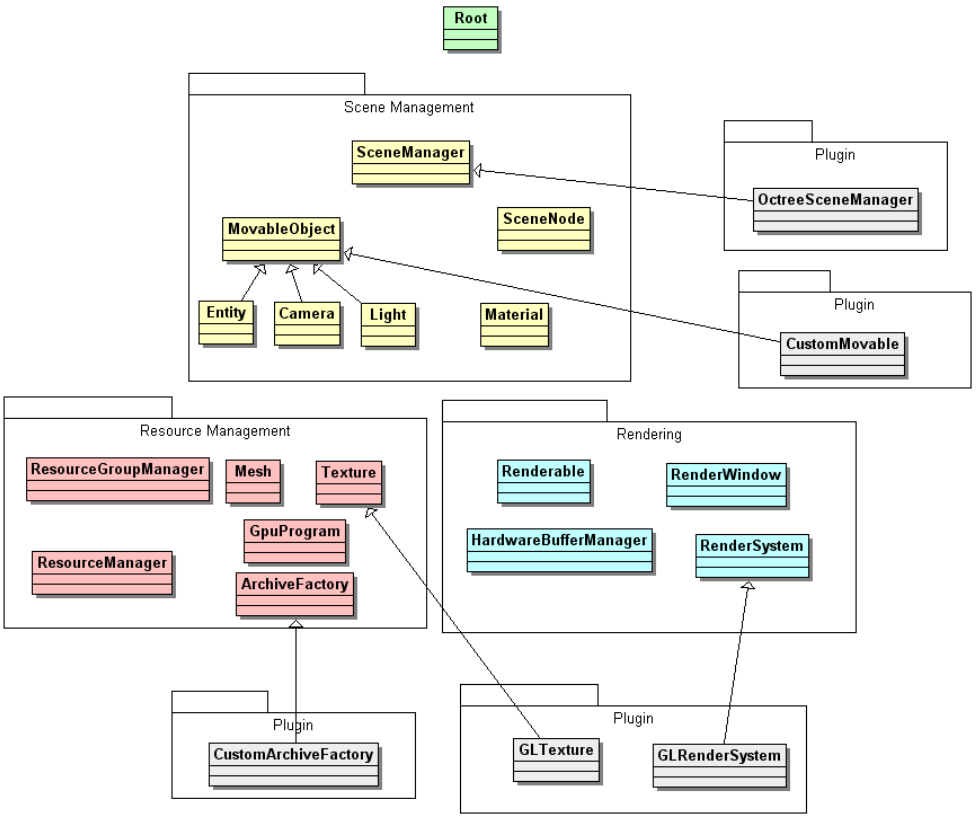
Не смотря на то, что 3D библиотека OGRE уступает по возможностям управления различными ресурсами компьютера, заложенная в ней модульная архитектура позволяет с лёгкостью дополнять её различными функциями (например, моим модулем IK).

OGRE позволяет использовать объектно-ориентированный подход в том, что изначально представляет собой процедурную обработку данных: рендеринг простых геометрических примитивов на плоскость рендеринга (render target – обычно буфер экрана, который хранится в памяти видеокарты). Обычно, когда используется OpenGL или Direct3D для рендеринга сцены, нам нужно выполнить последовательность действий - процедурный поток обработки данных: установить состояние рендеринга с помощью API вызовов, потом сказать GPU отрендерить геометрические примитивы с помощью пары других API вызовов. Повторять последнее действие до тех пор, пока кадр не будет полностью построен. Начать всё заново для следующего кадра.

С объектно-ориентированным подходом к рендерингу геометрии, нам вообще не нужно работать с геометрией. Теперь мы оперируем объектами, которые составляют сцену: двигающиеся (movable) объекты на сцене, статические объекты, представляющие геометрию объектов, освещение, камеры и так далее. Вызовы к API больше не используются – просто помещаем объект на сцену и OGRE трансформирует эти действия либо в обращения к OpenGL, либо к Direct3D. Также не нужно предрасчитывать самому в голове матрицы для трансформирования объектов, чтобы позиционировать их на сцене.

Библиотека OGRE использует множество общих паттернов проектирования, которые делают библиотеку более гибкой и лёгкой в использовании. Например, OGRE информирует приложение (Application) обо всех действиях с помощью паттерна Observer. Клиентский код регистрирует себя для получения извещений о событиях или изменениях состояния библиотеки (например, с помощью FrameListener приложение получает сообщения о начале рендеринга кадра (frame started event) и об окончании рендеринга кадра (frame ended event)). Паттерн Singleton используется для того, чтобы обеспечить возможность создания единственного экземпляра класса. Паттерн Iterator используется для того, чтобы пройтись по содержимому

структуры. Паттерн Visitor используется там, где нужно выполнить какие-то операции над объектом без изменения самого объекта (например, над узлами графа сцены).



Изображение 13. Архитектура графической библиотеки OGRE.

Для того, чтобы встроить модуль IK в OGRE, достаточно понимать как работают следующие модули: Root, SceneManager, SceneNode, MovableObject и Mesh. Они позволяют разместить скелет на сцене и задать для него оболочку (Mesh). После этого в Solver нужно поставить этот скелет, и он начнёт его модифицировать после установки целей (target) для end-effector’ов и вызова метода Solver::solve().

4. Заключение

Как видно из моего изложения, 3D графика – очень богатая область для исследований. Кроме своей визуальной красоты в играх и фильмах, она также привлекает интересными проблемами, которые порой оказываются на стыке фундаментальных наук. Был довольно трудный выбор между изучением законов распространения света и их моделей, физикой тканей и жидкостей, параллельными вычислениями реалистичных виртуальных сцен на кластере (алгоритм Reyes).

Оборачиваясь назад и видя проделанную работу и количество изученной информации, я понимаю, насколько правильным был мой выбор. Я изучил чуть глубже аналитическую геометрию, математическую оптимизацию и даже немного познакомился с физикой (динамикой) некоторых явлений.

Кроме того, инверсная кинематика ведёт меня естественным образом к ещё одной дисциплине, о которой мне пока известно, что она способна производить мурашки на моей коже, и что когда-нибудь она изменит облик Земли и человечества – к искусственному интеллекту.

5. Библиография

1. Fletcher Dunn, Ian Parberry, 3D Math Primer For Graphics And Game Development, ISBN-13: 978-1556229114.
2. Tomas Akenine-Moller, Eric Haines, Natty Hoffman, Real-Time Rendering, Third edition, ISBN 978-1568814247.
3. L.T. Wang and C.C. Chen (1991) *A Combined Optimization Method for Solving the Inverse Kinematics Problem of Mechanical Manipulators*. IEEE Transactions On Robotics and Automation, v.7, 489-498.
4. W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery (2007) Numerical Recipes: The Art of Scientific Computing (3rd ed.). New York: Cambridge University Press. ISBN 978-0-521-88068-8.
5. Chris Welman (1993) INVERSE KINEMATICS AND GEOMETRIC CONSTRAINTS FOR ARTICULATED FIGURE MANIPULATION. Simon Fraser University 1989.
6. Kenny Erleben, Jon Sporring, Knud Henriksen and Henrik Dohlmann (2005) *Physics based animation*. Charles River Media.