4 РАЗРАБОТКА ПРОГРАММНЫХ МОДУЛЕЙ

4.1 Класс `Particle`

Класс `Particle` представляет частицу. Частица - это самый простой объект, который можно моделировать в физической системе. Она имеет данные о местоположении (без ориентации), вместе со скоростью. Она может быть интегрирована вперед во времени и иметь приложенные к ней силы и импульсы. Частица управляет своим состоянием и предоставляет доступ через набор методов.

Есть два набора данных у частицы: характеристики и состояние.

Характеристики являются свойствами частицы, не зависящей от ее текущих кинематических свойств. Сюда входят масса, момент инерции и демпфирующие свойства. Две одинаковые частицы будут иметь одинаковые значения для своих характеристик.

Состояние включает в себя все характеристики, а также включает кинематические свойства частицы в текущем моделировании.

Когда данные состояния изменяются, производные значения необходимо обновлять: это может быть достигнуто либо путем интегрирования, либо путем вызова функции `calculateInternals`. Такой двухэтапный процесс используется, потому что пересчет внутренних элементов может быть дорогостоящим процессом. Все изменения состояния должны выполняться одновременно, что позволяет пересчитать их одним вызовом.

Класс содержит следующие поля:

1) `inverseMass`. Хранит обратную массу частицы. Полезнее хранить обратную массу, потому что интегрирования становиться проще, и в режиме реального времени более полезно иметь объекты с бесконечной массой, чем с нулевой массой (абсолютно неустойчивая при численном моделировании).

2) `damping`. Удерживает величину демпфирования, приложенного к линейному движению.

3) `position`. Хранит линейное положение частицы в мировом пространстве.

4) `velocity`. Хранит линейную скорость частицы в мировом пространстве.

5) `forceAccum`. Хранит накопленную силу, которая будет применяться только на следующей итерации моделирования. Это значение обнуляется на каждом этапе интеграции.

6) `acceleration`. Хранит ускорение частицы. Это значение может использоваться для установки ускорения из-за силы тяжести (его основная цель) или любого другого постоянного ускорения.

Класс `Particle` реализует следующие методы:

1) `integrate`. Интегрирует частицу вперед по времени на заданную

величину. Эта функция использует метод интегрирования Ньютона-Эйлера, который является линейным приближением к правильному интегралу. По этой причине могут возникать неточности в некоторых случаях.

2) `velocity`. Хранит линейную скорость частицы в мировом пространстве.

3) `setMass`. Устанавливает массу частицы. Метод делает недействительными внутренние данные частицы. Либо функцию интеграции, либо функцию `calculateInternals` следует вызывать перед попыткой получить какие-либо настройки частицы.

4) `setAcceleration`. Устанавливает постоянное ускорение частицы.

5) `setVelocity`. Устанавливает скорость частицы.

6) `clearAccumulator`. Убирает все приложенные силы к частице. Вызывается каждый раз после интегрирования.

7) `setMass`. Устанавливает массу частицы. Новая масса не может быть равна нулю, также очень маленькие значения могут вызвать странное поведение. Метод делает недействительными внутренние данные частицы. Либо функцию интеграции, либо функцию `calculateInternals` следует вызывать перед попыткой получить какие-либо настройки частицы.

8) `setPosition`. Устанавливает текущую мировую позицию частицы.

9) `setInverseMass`. Устанавливает обратную массу частицы. Метод делает недействительными внутренние данные частицы. Следует вызывать либо функцию интегрирования, либо функцию `calculateInternals` перед попыткой получить какие-либо настройки частицы.

Метод `integrate` выглядит следующим образом:

```
// We don't integrate things with zero mass.
if (inverseMass <= 0.0f)
     return;

assert(duration > 0.0);

// Update linear position.
position.addScaledVector(velocity, duration);

// Work out the acceleration from the force
Vector3 resultingAcc = acceleration;
resultingAcc.addScaledVector(forceAccum,
inverseMass);

// Update linear velocity from the acceleration.
velocity.addScaledVector(resultingAcc, duration);
// Impose drag.
velocity *= real_pow(damping, duration);

// Clear the forces.
```

```
clearAccumulator();
```

Сначала мы проверяем массу частицы, так как не имеет смысла интегрировать частицы с бесконечной массой. Затем мы добавляем к текущей позиции частицы ее скорость умноженную на время интегрирования. Далее нам необходимо пересчитать скорость для следующего интегрирования. К текущему ускорению мы добавляем результирующую силу, действующую на частицу, умноженную на обратную массу. Результат умножаем на время интегрирования и прибавляем к текущей скорости. Это и будет скорость частицы на следующем этапе интеграции.

4.2 Класс `ParticleWorld`

Основная задача класса это хранить частицы и предоставляет средства для их обновления.

Класс `ParticleWorld` реализует следующие методы:

1) `ParticleWorld`. Создает новый симулятор частиц, который может обрабатывать до заданного количества контактов на кадр. Также можно указать количество итераций для разрешения контактов. Если не указано количество итераций, будет использовано число в два раза большее количества контактов.

2) `generateContacts`. Позволяет каждому зарегистрированному контактному генератору сообщать о своих контактах. Возвращает количество сгенерированных контактов.

3) `integrate`. Интегрирует все частицы в этом мире вперед во времени на заданный период.

4) `runPhysics`. Обрабатывает всю физику для мира частиц.

5) `startFrame`. Инициализирует мир для симуляции кадра. Это очищает аккумуляторы силы для частиц в мире. После вызова этого метода, к частицам можно прикладывать силы.

```
unsigned limit = maxContacts;
ParticleContact *nextContact = contacts;

for(ContactGenerators::iterator
g = contactGenerators.begin();
             g != contactGenerators.end();
             g++)
{
     unsigned used =(*g)->addContact(nextContact, limit);
     limit -= used;
     nextContact += used;
// We've run out of contacts to fill. This means we're missing
// contacts.
     if (limit <= 0)
          break;
```

```
}

// Return the number of contacts used.
return maxContacts - limit;
```

Метод `generateContacts` перебирает все зарегистрированные генераторы контактов и через интерфейс класса `ParticleContactGenerator` вызывает у них метод `addContact`, пока не будет достигнут лимит контактов для фрейма или все генераторы не будут обработаны.

4.3 Класс `ParticleForceGenerator`

Этот класс предоставляет интерфейс для подсчета и обновления силы приложенной к частице. Также является базовым для всех генераторов силы, таких как `ParticleGravity`, `ParticleDrag`, `ParticleAnchoredSpring`, `ParticleAnchoredBungee`, `ParticleFakeSpring`, `ParticleSpring` и других. Его метод `updateForce` необходимо переопределить в производных классах.

4.4 Класс `ParticleGravity`

Силовой генератор, который прикладывает гравитационную силу. Один экземпляр может быть использован для нескольких частиц. Этот класс реализует интерфейс `ParticleForceGenerator`

Класс `ParticleGravity` реализует следующие методы:

1) `ParticleGravity`. Создает новый генератор с заданной силой тяжести.

2) `updateForce`. Прикладывает гравитационную силу к заданной частице.

Метод `updateForce` реализован следующим образом:

```
// Check that we do not have infinite mass
if (!particle->hasFiniteMass())
     return;

// Apply the mass-scaled force to the particle
particle->addForce(gravity * particle->getMass());
```

Сначала мы проверяем является ли конечной масса частицы(гравитация не будет воздействовать на частицы с бесконечной массой), а затем прикладываем к частице силу, равную произведению ускорения гравитации на массу частицы.

4.5 Класс `ParticleContactGenerator`

Это основной полиморфный интерфейс для контактных генераторов, применяющихся к частицам. Его метод `addContact` заполняет заданную контактную структуру `ParticleContact` созданным контактом. Указатель контакта должен указывать на первый доступный контакт в массиве контактов. Метод возвращает количество записей, которые были записаны.

4.6 Класс `ParticleLink`

Класс `ParticleLink` соединяет две частицы вместе, генерируя контакт, если они нарушают ограничения их связи. Этот класс используется в качестве базового класса для кабелей и стержней, и может использоваться в качестве базового класса для пружин с ограничением их расширения. Класс `ParticleLink` является наследником `ParticleContactGenerator` и реализует интерфейс контактных генераторов.

Класс `ParticleLink` реализует метод `addContact`. Генерирует контакты, чтобы эта связь частиц не нарушалась. Этот класс может генерировать только один контакт, поэтому указатель может быть указателем на один элемент, предельный параметр считается равным как минимум одному, а возвращаемое значение равно 0, если расстояние не было чрезмерно увеличенным, или 1, если контакт необходим.

4.7 Класс `ParticleRod`

Этот класс представляет стержень, связывающий две частицы. `ParticleRod` создает контакты, если частицы находятся слишком далеко, или слишком близко.

Класс содержит одно поле `length`, которое хранит длину, на которой необходимо держать частицы.

Класс реализует интерфейс `ParticleLink` и переопределяет его метод `addContact`. Этот метод заполняет структуру контакта, необходимую для предотвращения расхождения связанных частиц.

```
// Find the length of the rod
real currentLen = currentLength();

// Check if we're over-extended
if (currentLen == length)
{
     return 0;
}
// Otherwise return the contact
contact->particle[0] = particle[0];
contact->particle[1] = particle[1];
```

```
    // Calculate the normal
    Vector3 normal = particle[1]->getPosition() -
         particle[0]->getPosition();
    normal.normalise();

    // The contact normal depends on whether we're extending or
compressing
    if (currentLen > length)
    {
         contact->contactNormal = normal;
         contact->penetration = currentLen - length;
    }
    else
    {
         contact->contactNormal = normal * -1;
         contact->penetration = length - currentLen;
    }

    // Always use zero restitution (no bounciness)
    contact->restitution = 0;

    return 1;
```

Сначала мы вычисляем расстояние между частицами и проверяем отличается ли оно от необходимой длины стержня, так как если нет, то контакт генерировать нет нужды. Далее мы вычисляем нормаль контакта в зависимости от того, нужно ли нам сблизить или растянуть частицы, и глубине проникновения. Коэффициент восстановления для таких контактов всегда будет равен нулю.

4.8 Класс `ParticleCable`

Кабели связывают пары частиц, создавая контакт, если они оказываются слишком далеко друг от друга.

Класс содержит следующие поля:

1) `maxLength`. Хранит максимальную длину кабеля.

2) `restitution`. Хранит коэффициент восстановления кабеля или его упругость.

Класс реализует интерфейс `ParticleLink` и переопределяет его метод `addContact`. Этот метод заполняет структуру контакта, необходимую для предотвращения растяжения кабеля.

```
    real length = currentLength();
    // Find the length of the cable
    // Check if we're over-extended
    if (length < maxLength)
    {
```

```
            return 0;
        }

        // Otherwise return the contact
        contact->particle[0] = particle[0];
        contact->particle[1] = particle[1];

        // Calculate the normal
        Vector3 normal = particle[1]->getPosition() - particle[0]-
>getPosition();
        normal.normalise();
        contact->contactNormal = normal;

        contact->penetration = length-maxLength;
        contact->restitution = restitution;

        return 1;
```

Сначала мы вычисляем расстояние между частицами и проверяем превышает ли оно допустимую длину кабеля, так как если нет, то контакт генерировать нет нужды. Далее мы вычисляем нормаль контакта, равную нормализованной разнице между позициями частиц, и записываем ее вместе с коэффициентом восстановления в контактную структуру.

4.9 Класс `ParticleConstraint`

Класс `ParticleConstraint` выполняет ту же роль, что и `ParticleLink`, за исключением того, что он является базовым для генераторов контактов для соединений частиц с неподвижной точкой в пространстве.

Класс содержит следующие поля:

1) `particle`. Хранит частицу которую, для которой нужно применить ограничение.

2) `anchor`. Хранит координаты точки в пространстве для соединения с частицей.

Класс `ParticleConstraint` реализует метод `addContact`. Генерирует контакты, чтобы эта связь частиц не нарушалась. Этот класс может генерировать только один контакт, поэтому указатель может быть указателем на один элемент, предельный параметр считается равным как минимум одному, а возвращаемое значение равно 0, если расстояние не было чрезмерно увеличенным, или 1, если контакт необходим.

4.10 Класс `ParticleCableConstraint`

Класс представляет собой кабель, соединяющий частицу и точку в пространстве. Класс `ParticleCableContraint` является наследником

ParticleConstraint.

Класс содержит следующие поля:

1) maxLength. Хранит максимальную длину кабеля.

2) restitution. Хранит коэффициент восстановления кабеля или его упругость.

Класс реализует интерфейс ParticleConstraint и переопределяет его метод addContact. Этот метод заполняет структуру контакта, необходимую для предотвращения растяжения кабеля.

```
// Find the length of the cable
real length = currentLength();

// Check if we're over-extended
if (length < maxLength)
{
     return 0;
}

// Otherwise return the contact
contact->particle[0] = particle;
contact->particle[1] = 0;

// Calculate the normal
Vector3 normal = anchor - particle->getPosition();
normal.normalise();
contact->contactNormal = normal;

contact->penetration = length-maxLength;
contact->restitution = restitution;

return 1;
```

Сначала мы вычисляем расстояние между частицей и опорной точкой, и проверяем превышает ли оно допустимую длину кабеля, так как если нет, то контакт генерировать нет нужды. Далее мы вычисляем нормаль контакта, равную нормализованной разнице между позициями частиц, и записываем ее вместе с коэффициентом восстановления в контактную структуру.

4.11 Класс ParticleRodConstaint

Этот класс представляет стержень, связывающий частицу с неподвижной точкой в пространстве. ParticleRodConstraint создает контакты, если частицы находятся слишком далеко, или слишком близко.

Класс содержит одно поле length, которое хранит длину, на которой необходимо держать частицу от точки.

Класс реализует интерфейс ParticleConstraint и переопределяет

его метод `addContact`. Этот метод заполняет структуру контакта, необходимую для предотвращения расхождения связанных частиц.

```
    // Find the length of the rod
    real currentLen = currentLength();

    // Check if we're over-extended
    if (currentLen == length)
    {
        return 0;
    }
    // Otherwise return the contact
    contact->particle[0] = particle;
    contact->particle[1] = 0;

    // Calculate the normal
    Vector3 normal = anchor - particle->getPosition();
    normal.normalise();

    // The contact normal depends on whether we're extending or
compressing
    if (currentLen > length)
    {
        contact->contactNormal = normal;
        contact->penetration = currentLen - length;
    }
    else
    {
        contact->contactNormal = normal * -1;
        contact->penetration = length - currentLen;
    }
    // Always use zero restitution (no bounciness)
    contact->restitution = 0;

    return 1;
```

Сначала мы вычисляем расстояние между частицей и точкой, и проверяем отличается ли оно от необходимой длины стержня, так как если нет, то контакт генерировать нет нужды. Далее мы вычисляем нормаль контакта в зависимости от того, нужно ли нам сблизить или растянуть частицы, и глубине проникновения. Коэффициент восстановления для таких контактов всегда будет равен нулю.