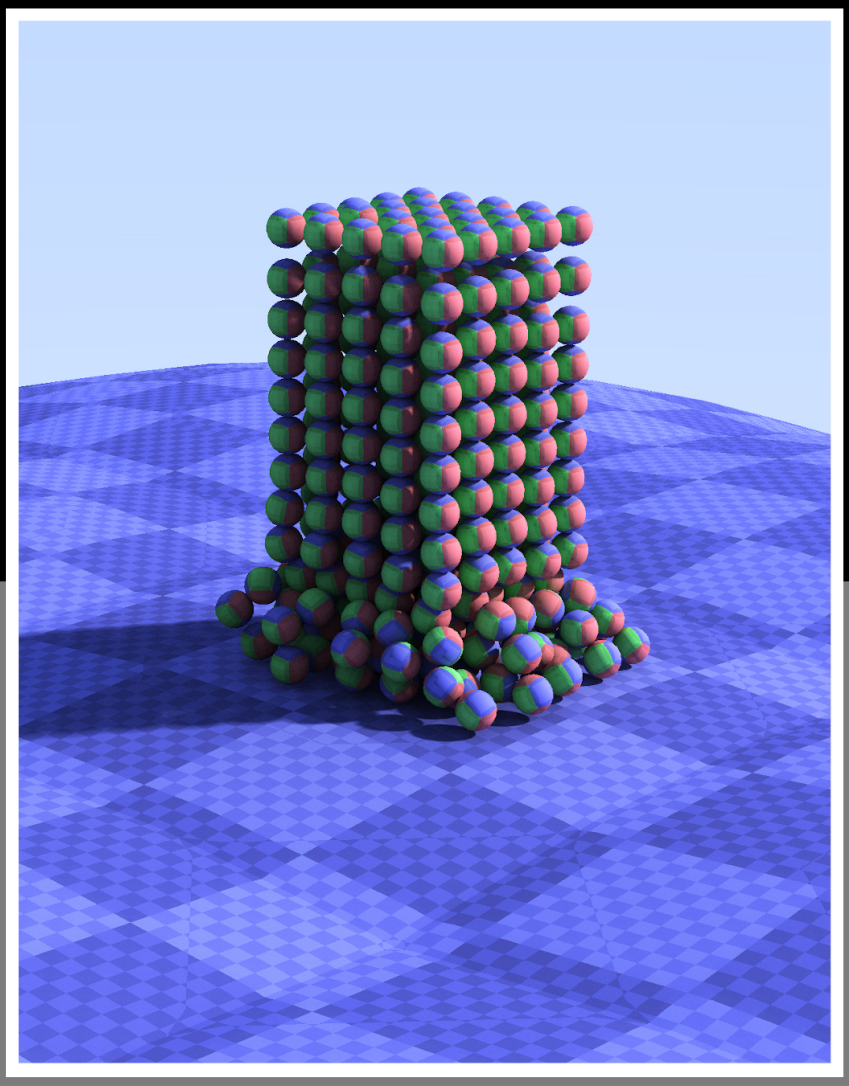
Game Physics

In One Weekend



Gregory Hodges

Game Physics: In One Weekend

Gregory Hodges
Version 1.08

Contents

1 Overview

This book series was inspired by the similarly named "Ray Tracing In One Weekend" by Peter Shirley. His books enable someone, with little to no knowledge of graphics programming, to build a working homebrew path tracer.

After reading his books, I thought there should be more series similar to that for as many subjects as possible. After all, when starting to learn a subject, often times the most difficult part is figuring out how to begin.

Eventually, I would have the thought to contribute in some small way. And the result is this book. Now, while this book is not meant to make you an expert in rigid body physics for games, it is meant to be a large springboard into the subject. And if you work through the entirety of the series, then at the end of it, you will have continuous collision detection, constraints, motors, ragdolls, and you'll be able to simulate a stable stack of blocks.

Since this is a book on game physics, you will need a renderer. If you have one of your own, then excellent. If you do not have one, then I have prepared a very meager Vulkan renderer. There will be instructions on the companion website, https://gamephysicsweekend.github.io/, for how to use it. Specifically, this "renderer" is just a skeleton project for the book. And there are comments in the provided project that read "// TODO: add code", this is where you would manually transcibe the code from the book to the project.

The code in this book is written in C++. Because this is a book with intended application in games, I'm assuming that you either work on games or are a student aimed at working in the industry. And C++ is still rather ubiquitous for game development, so it is a natural choice for this book series.

It is also assumed that you have some understanding of vector calculus, linear algebra, and quaternions. If you do not, then that's fine. I recommend Schaum's outlines as an introduction. Those books are both affordable and pleasant overviews of the subjects. I also recommend 3blue1brown's video lectures to build an intuition for mathematics.

Speaking of math, this book assumes the +z-axis is up; other people like to use +y-axis as up, but not here. And for basic book keeping, vectors are written with a little arrow above the variable, $\vec{v}$, matrices will always be upper case bold, **M**, quaternions are lower case bold, **q**, and scalars can be upper or lower case, s S.

$$\begin{aligned} \vec{v} &- vector \\ \mathbf{M} &- matrix \\ \mathbf{q} &- quaternion \\ s &- scalar \end{aligned}$$

Finally, I've never found a textbook that was completely void of errors. So, if you find any mistakes, have constructive feedback, or just a question, then feel free to hit me up on twitter @ghodges_dev. And if you don't have social media, then you can also email me, greg@gamephysicsweekend.com. Some people have also raised issues on the github page, which is also a great way to get in contact. However, I may not be able to immediately respond, but I assure you that I eventually will.

It is my hope, that whether you're a student or a professional programmer, that you'll learn something from this book. And with that, let's begin.

1.1 Acknowledgements

Peter Shirley, Zachary Williams, Adam Petrone, Nick Korn, Mincho Paskalev, Yuki Nishidate

2 The Vector Classes

We will need a way of representing positions and directions in 2D, 3D, and 4D space. For that, we will use the Vec2 class to represent 2D vectors, the Vec3 class to represent 3D vectors, and the Vec4 class to represent 4D vectors.

Although, we will mostly only be concerned with 3D vectors, since we're creating a 3D physics simulation. And the Vec2 and Vec4 classes are mainly here as support for the matrix classes that will be introduced in the next chapter.

These classes implement the very basic features that you'd expect for vectors, such as dot products, cross products, addition, multiplication, etc. If you have familiarity with any shading language or open source math library, such as glsl or glm, then these classes should seem very familiar.

Also, if you're using the renderer from the github page, then this code is included and you won't need to manually transcribe it.

```
/*
 ================================
 Vec2
 ================================
 */
class Vec2 {
public:
  Vec2();
  Vec2( const float value );
  Vec2( const Vec2 & rhs );
  Vec2( float X, float Y );
  Vec2( const float * xy );
  Vec2 & operator = ( const Vec2 & rhs );

  bool       operator == ( const Vec2 & rhs ) const;
  bool       operator != ( const Vec2 & rhs ) const;
  Vec2       operator + ( const Vec2 & rhs ) const;
  const Vec2 &    operator += ( const Vec2 & rhs );
  const Vec2 &    operator -= ( const Vec2 & rhs );
  Vec2       operator - ( const Vec2 & rhs ) const;
  Vec2       operator * ( const float rhs ) const;
  const Vec2 &    operator *= ( const float rhs );
  const Vec2 &    operator /= ( const float rhs );
  float      operator [] ( const int idx ) const;
  float &    operator [] ( const int idx );

  const Vec2 & Normalize();
  float GetMagnitude() const;
  bool IsValid() const;
  float Dot( const Vec2 & rhs ) const { return x * rhs.x + y * rhs.y; }

  const float * ToPtr() const { return &x; }

public:
  float x;
  float y;
};

inline Vec2::Vec2() :
x( 0 ),
y( 0 ) {
}

inline Vec2::Vec2( const float value ) :
x( value ),
y( value ) {
}

inline Vec2::Vec2( const Vec2 & rhs ) :
x( rhs.x ),
y( rhs.y ) {
}

inline Vec2::Vec2( float X, float Y ) :
```

```
x( X ),
y( Y ) {
}

inline Vec2::Vec2( const float * xy ) :
x( xy[ 0 ] ),
y( xy[ 1 ] ) {
}

inline Vec2& Vec2::operator=( const Vec2 & rhs ) {
  x = rhs.x;
  y = rhs.y;
  return *this;
}

inline bool Vec2::operator==( const Vec2 & rhs ) const {
  if ( x != rhs.x ) {
    return false;
  }
  if ( y != rhs.y ) {
    return false;
  }

  return true;
}

inline bool Vec2::operator != ( const Vec2 & rhs ) const {
  if ( *this == rhs ) {
    return false;
  }

  return true;
}

inline Vec2 Vec2::operator + ( const Vec2 & rhs ) const {
  Vec2 temp;
  temp.x = x + rhs.x;
  temp.y = y + rhs.y;
  return temp;
}

inline const Vec2 & Vec2::operator += ( const Vec2 & rhs ) {
  x += rhs.x;
  y += rhs.y;
  return *this;
}

inline const Vec2 & Vec2::operator -= ( const Vec2 & rhs ) {
  x -= rhs.x;
  y -= rhs.y;
  return *this;
}

inline Vec2 Vec2::operator - ( const Vec2 & rhs ) const {
  Vec2 temp;
  temp.x = x - rhs.x;
  temp.y = y - rhs.y;
  return temp;
}

inline Vec2 Vec2::operator * ( const float rhs ) const {
  Vec2 temp;
  temp.x = x * rhs;
  temp.y = y * rhs;
  return temp;
}

inline const Vec2 & Vec2::operator *= ( const float rhs ) {
  x *= rhs;
  y *= rhs;
    return *this;
```

```
}

inline const Vec2 & Vec2::operator /= ( const float rhs ) {
  x /= rhs;
  y /= rhs;
    return *this;
}

inline float Vec2::operator [] ( const int idx ) const {
  assert( idx >= 0 && idx < 2 );
  return ( &x )[ idx ];
}

inline float & Vec2::operator [] ( const int idx ) {
  assert( idx >= 0 && idx < 2 );
  return ( &x )[ idx ];
}

inline const Vec2 & Vec2::Normalize() {
  float mag = GetMagnitude();
  float invMag = 1.0f / mag;
  if ( 0.0f * invMag == 0.0f * invMag ) {
    x = x * invMag;
    y = y * invMag;
  }

    return *this;
}

inline float Vec2::GetMagnitude() const {
  float mag;

  mag = x * x + y * y;
  mag = sqrtf( mag );

  return mag;
}

inline bool Vec2::IsValid() const {
  if ( x * 0.0f != x * 0.0f ) {
    // x is NaN or Inf
    return false;
  }

  if ( y * 0.0f != y * 0.0f ) {
    // y is NaN or Inf
    return false;
  }

  return true;
}

/*
 ================================
 Vec3
 ================================
 */
class Vec3 {
public:
  Vec3();
  Vec3( float value );
  Vec3( const Vec3 & rhs );
  Vec3( float X, float Y, float Z );
  Vec3( const float * xyz );
  Vec3 & operator = ( const Vec3 & rhs );
  Vec3 & operator = ( const float * rhs );

  bool      operator == ( const Vec3 & rhs ) const;
  bool      operator != ( const Vec3 & rhs ) const;
  Vec3      operator + ( const Vec3 & rhs ) const;
  const Vec3 &  operator += ( const Vec3 & rhs );
```

```
  const Vec3 &  operator -= ( const Vec3 & rhs );
  Vec3      operator - ( const Vec3 & rhs ) const;
  Vec3      operator * ( const float rhs ) const;
    Vec3      operator / ( const float rhs ) const;
  const Vec3 &  operator *= ( const float rhs );
    const Vec3 &  operator /= ( const float rhs );
  float     operator [] ( const int idx ) const;
  float &     operator [] ( const int idx );

  void Zero() { x = 0.0f; y = 0.0f; z = 0.0f; }

  Vec3 Cross( const Vec3 & rhs ) const;
  float Dot( const Vec3 & rhs ) const;

  const Vec3 & Normalize();
  float GetMagnitude() const;
  float GetLengthSqr() const { return Dot( *this ); }
  bool IsValid() const;
  void GetOrtho( Vec3 & u, Vec3 & v ) const;

  const float * ToPtr() const { return &x; }

public:
  float x;
  float y;
  float z;
};

inline Vec3::Vec3() :
x( 0 ),
y( 0 ),
z( 0 ) {
}

inline Vec3::Vec3( float value ) :
x( value ),
y( value ),
z( value ) {
}

inline Vec3::Vec3( const Vec3 &rhs ) :
x( rhs.x ),
y( rhs.y ),
z( rhs.z ) {
}

inline Vec3::Vec3( float X, float Y, float Z ) :
x( X ),
y( Y ),
z( Z ) {
}

inline Vec3::Vec3( const float * xyz ) :
x( xyz[ 0 ] ),
y( xyz[ 1 ] ),
z( xyz[ 2 ] ) {
}

inline Vec3 & Vec3::operator = ( const Vec3 & rhs ) {
  x = rhs.x;
  y = rhs.y;
  z = rhs.z;
  return *this;
}

inline Vec3& Vec3::operator=( const float * rhs ) {
  x = rhs[ 0 ];
  y = rhs[ 1 ];
  z = rhs[ 2 ];
  return *this;
}
```

```
inline bool Vec3::operator == ( const Vec3 & rhs ) const {
  if ( x != rhs.x ) {
    return false;
  }

  if ( y != rhs.y ) {
    return false;
  }

  if ( z != rhs.z ) {
    return false;
  }

  return true;
}

inline bool Vec3::operator != ( const Vec3 & rhs ) const {
  if ( *this == rhs ) {
    return false;
  }

  return true;
}

inline Vec3 Vec3::operator + ( const Vec3 & rhs ) const {
  Vec3 temp;
  temp.x = x + rhs.x;
  temp.y = y + rhs.y;
  temp.z = z + rhs.z;
  return temp;
}

inline const Vec3 & Vec3::operator += ( const Vec3 & rhs ) {
  x += rhs.x;
  y += rhs.y;
  z += rhs.z;
  return *this;
}

inline const Vec3 & Vec3::operator -= ( const Vec3 & rhs ) {
  x -= rhs.x;
  y -= rhs.y;
  z -= rhs.z;
  return *this;
}

inline Vec3 Vec3::operator - ( const Vec3 & rhs ) const {
  Vec3 temp;
  temp.x = x - rhs.x;
  temp.y = y - rhs.y;
  temp.z = z - rhs.z;
  return temp;
}

inline Vec3 Vec3::operator * ( const float rhs ) const {
  Vec3 temp;
  temp.x = x * rhs;
  temp.y = y * rhs;
  temp.z = z * rhs;
  return temp;
}

inline Vec3 Vec3::operator / ( const float rhs ) const {
  Vec3 temp;
  temp.x = x / rhs;
  temp.y = y / rhs;
  temp.z = z / rhs;
  return temp;
}
```

```
inline const Vec3 & Vec3::operator *= ( const float rhs ) {
  x *= rhs;
  y *= rhs;
  z *= rhs;
  return *this;
}

inline const Vec3 & Vec3::operator /= ( const float rhs ) {
  x /= rhs;
  y /= rhs;
  z /= rhs;
  return *this;
}

inline float Vec3::operator [] ( const int idx ) const {
  assert( idx >= 0 && idx < 3 );
  return ( &x )[ idx ];
}

inline float & Vec3::operator [] ( const int idx ) {
  assert( idx >= 0 && idx < 3 );
  return ( &x )[ idx ];
}

inline Vec3 Vec3::Cross( const Vec3 & rhs ) const {
  // This cross product is A x B, where this is A and rhs is B
  Vec3 temp;
  temp.x = ( y * rhs.z ) - ( rhs.y * z );
  temp.y = ( rhs.x * z ) - ( x * rhs.z );
  temp.z = ( x * rhs.y ) - ( rhs.x * y );
  return temp;
}

inline float Vec3::Dot( const Vec3 & rhs ) const {
  float temp = ( x * rhs.x ) + ( y * rhs.y ) + ( z * rhs.z );
  return temp;
}

inline const Vec3 & Vec3::Normalize() {
  float mag = GetMagnitude();
  float invMag = 1.0f / mag;
  if ( 0.0f * invMag == 0.0f * invMag ) {
    x *= invMag;
    y *= invMag;
    z *= invMag;
  }
    return *this;
}

inline float Vec3::GetMagnitude() const {
  float mag;

  mag = x * x + y * y + z * z;
  mag = sqrtf( mag );

  return mag;
}

inline bool Vec3::IsValid() const {
  if ( x * 0.0f != x * 0.0f ) {
    return false;
  }

  if ( y * 0.0f != y * 0.0f ) {
    return false;
  }

  if ( z * 0.0f != z * 0.0f ) {
    return false;
  }
```

```
  return true;
}

inline void Vec3::GetOrtho( Vec3 & u, Vec3 & v ) const {
  Vec3 n = *this;
  n.Normalize();

  const Vec3 w = ( n.z * n.z > 0.9f * 0.9f ) ? Vec3( 1, 0, 0 ) : Vec3( 0, 0, 1 );
  u = w.Cross( n );
  u.Normalize();

  v = n.Cross( u );
  v.Normalize();
  u = v.Cross( n );
  u.Normalize();
}

/*
 ================================
 Vec4
 ================================
 */
class Vec4 {
public:
  Vec4();
  Vec4( const float value );
  Vec4( const Vec4 & rhs );
  Vec4( float X, float Y, float Z, float W );
  Vec4( const float * rhs );
  Vec4 & operator = ( const Vec4 & rhs );

  bool      operator == ( const Vec4 & rhs ) const;
  bool      operator != ( const Vec4 & rhs ) const;
  Vec4      operator + ( const Vec4 & rhs ) const;
  const Vec4 &  operator += ( const Vec4 & rhs );
  const Vec4 &  operator -= ( const Vec4 & rhs );
    const Vec4 &  operator *= ( const Vec4 & rhs );
  const Vec4 &  operator /= ( const Vec4 & rhs );
  Vec4      operator - ( const Vec4 & rhs ) const;
  Vec4      operator * ( const float rhs ) const;
  float     operator [] ( const int idx ) const;
  float &     operator [] ( const int idx );

  float Dot( const Vec4 & rhs ) const;
  const Vec4 & Normalize();
  float GetMagnitude() const;
  bool IsValid() const;
  void Zero() { x = 0; y = 0; z = 0; w = 0; }

    const float *   ToPtr() const   { return &x; }
  float *         ToPtr()         { return &x; }

public:
  float x;
  float y;
  float z;
  float w;
};

inline Vec4::Vec4() :
x( 0 ),
y( 0 ),
z( 0 ),
w( 0 ) {
}

inline Vec4::Vec4( const float value ) :
x( value ),
y( value ),
z( value ),
w( value ) {
```

```
}

inline Vec4::Vec4( const Vec4 & rhs ) :
x( rhs.x ),
y( rhs.y ),
z( rhs.z ),
w( rhs.w ) {
}

inline Vec4::Vec4( float X, float Y, float Z, float W ) :
x( X ),
y( Y ),
z( Z ),
w( W ) {
}

inline Vec4::Vec4( const float * rhs ) {
  x = rhs[ 0 ];
  y = rhs[ 1 ];
  z = rhs[ 2 ];
  w = rhs[ 3 ];
}

inline Vec4 & Vec4::operator = ( const Vec4 & rhs ) {
  x = rhs.x;
  y = rhs.y;
  z = rhs.z;
  w = rhs.w;
  return *this;
}

inline bool Vec4::operator == ( const Vec4 & rhs ) const {
  if ( x != rhs.x ) {
    return false;
  }

  if ( y != rhs.y ) {
    return false;
  }

  if ( z != rhs.z ) {
    return false;
  }

  if ( w != rhs.w ) {
    return false;
  }

  return true;
}

inline bool Vec4::operator != ( const Vec4 & rhs ) const {
  if ( *this == rhs ) {
    return false;
  }

  return true;
}

inline Vec4 Vec4::operator + ( const Vec4 & rhs ) const {
  Vec4 temp;
  temp.x = x + rhs.x;
  temp.y = y + rhs.y;
  temp.z = z + rhs.z;
  temp.w = w + rhs.w;
  return temp;
}

inline const Vec4 & Vec4::operator += ( const Vec4 & rhs ) {
  x += rhs.x;
  y += rhs.y;
```

```
  z += rhs.z;
  w += rhs.w;
  return *this;
}

inline const Vec4 & Vec4::operator -= ( const Vec4 & rhs ) {
  x -= rhs.x;
  y -= rhs.y;
  z -= rhs.z;
  w -= rhs.w;
  return *this;
}

inline const Vec4 & Vec4::operator *= ( const Vec4 & rhs ) {
  x *= rhs.x;
  y *= rhs.y;
  z *= rhs.z;
  w *= rhs.w;
  return *this;
}

inline const Vec4 & Vec4::operator /= ( const Vec4 & rhs ) {
  x /= rhs.x;
  y /= rhs.y;
  z /= rhs.z;
  w /= rhs.w;
  return *this;
}

inline Vec4 Vec4::operator - ( const Vec4 & rhs ) const {
  Vec4 temp;
  temp.x = x - rhs.x;
  temp.y = y - rhs.y;
  temp.z = z - rhs.z;
  temp.w = w - rhs.w;
  return temp;
}

inline Vec4 Vec4::operator * ( const float rhs ) const {
  Vec4 temp;
  temp.x = x * rhs;
  temp.y = y * rhs;
  temp.z = z * rhs;
  temp.w = w * rhs;
  return temp;
}

inline float Vec4::operator [] ( const int idx ) const {
  assert( idx >= 0 && idx < 4 );
  return ( &x )[ idx ];
}

inline float& Vec4::operator [] ( const int idx ) {
  assert( idx >= 0 && idx < 4 );
  return ( &x )[ idx ];
}

inline float Vec4::Dot( const Vec4 & rhs ) const {
  float xx = x * rhs.x;
  float yy = y * rhs.y;
  float zz = z * rhs.z;
  float ww = w * rhs.w;
  return ( xx + yy + zz + ww );
}

inline const Vec4 & Vec4::Normalize() {
  float mag = GetMagnitude();
  float invMag = 1.0f / mag;
  if ( 0.0f * invMag == 0.0f * invMag ) {
    x *= invMag;
    y *= invMag;
```

```
    z *= invMag;
    w *= invMag;
  }

    return *this;
}

inline float Vec4::GetMagnitude() const {
  float mag;

  mag = x * x + y * y + z * z + w * w;
  mag = sqrtf( mag );

  return mag;
}

inline bool Vec4::IsValid() const {
  if ( x * 0.0f != x * 0.0f ) {
    return false;
  }

  if ( y * 0.0f != y * 0.0f ) {
    return false;
  }

  if ( z * 0.0f != z * 0.0f ) {
    return false;
  }

  if ( w * 0.0f != w * 0.0f ) {
    return false;
  }

  return true;
}
```

3 The Matrix Classes

It would be nearly impossible to render 3D geometry without linear algebra. And this is also true of physics simulations. We will definitely need a class to represent the inertia tensor (which can be thought of as a 3x3 matrix). And for this will use the Mat2, Mat3, and Mat4 class.

The Mat2 class is used to represent 2x2 matrices, and is only used here as a support class to the Mat3 class. The Mat3 class is for 3x3 matrices, which will be the matrix class most often used. The Mat4 class is for 4x4 matrices, which is only used by the renderer for constructing the projection matrix.

Again, if you are using the "renderer" from the github page, these classes are already included.

```
/*
================================
Mat2
================================
*/
class Mat2 {
public:
  Mat2() {}
  Mat2( const Mat2 & rhs );
  Mat2( const float * mat );
  Mat2( const Vec2 & row0, const Vec2 & row1 );
  Mat2 & operator = ( const Mat2 & rhs );

  const Mat2 & operator *= ( const float rhs );
  const Mat2 & operator += ( const Mat2 & rhs );

  float Determinant() const { return rows[ 0 ].x * rows[ 1 ].y - rows[ 0 ].y * rows[ 1 ].x; }

public:
  Vec2 rows[ 2 ];
};

inline Mat2::Mat2( const Mat2 & rhs ) {
  rows[ 0 ] = rhs.rows[ 0 ];
  rows[ 1 ] = rhs.rows[ 1 ];
}

inline Mat2::Mat2( const float * mat ) {
  rows[ 0 ] = mat + 0;
  rows[ 1 ] = mat + 2;
}

inline Mat2::Mat2( const Vec2 & row0, const Vec2 & row1 ) {
  rows[ 0 ] = row0;
  rows[ 1 ] = row1;
}

inline Mat2 & Mat2::operator = ( const Mat2 & rhs ) {
  rows[ 0 ] = rhs.rows[ 0 ];
  rows[ 1 ] = rhs.rows[ 1 ];
  return *this;
}

inline const Mat2 & Mat2::operator *= ( const float rhs ) {
  rows[ 0 ] *= rhs;
  rows[ 1 ] *= rhs;
  return *this;
}

inline const Mat2 & Mat2::operator += ( const Mat2 & rhs ) {
  rows[ 0 ] += rhs.rows[ 0 ];
  rows[ 1 ] += rhs.rows[ 1 ];
  return *this;
}

/*
================================
Mat3
================================
```

```
*/
class Mat3 {
public:
  Mat3() {}
  Mat3( const Mat3 & rhs );
  Mat3( const float * mat );
  Mat3( const Vec3 & row0, const Vec3 & row1, const Vec3 & row2 );
  Mat3 & operator = ( const Mat3 & rhs );

  void Zero();
  void Identity();

  float Trace() const;
  float Determinant() const;
  Mat3 Transpose() const;
  Mat3 Inverse() const;
  Mat2 Minor( const int i, const int j ) const;
  float Cofactor( const int i, const int j ) const;

  Vec3 operator * ( const Vec3 & rhs ) const;
  Mat3 operator * ( const float rhs ) const;
  Mat3 operator * ( const Mat3 & rhs ) const;
  Mat3 operator + ( const Mat3 & rhs ) const;
  const Mat3 & operator *= ( const float rhs );
  const Mat3 & operator += ( const Mat3 & rhs );

public:
  Vec3 rows[ 3 ];
};

inline Mat3::Mat3( const Mat3 & rhs ) {
  rows[ 0 ] = rhs.rows[ 0 ];
  rows[ 1 ] = rhs.rows[ 1 ];
  rows[ 2 ] = rhs.rows[ 2 ];
}

inline Mat3::Mat3( const float * mat ) {
  rows[ 0 ] = mat + 0;
  rows[ 1 ] = mat + 3;
  rows[ 2 ] = mat + 6;
}

inline Mat3::Mat3( const Vec3 & row0, const Vec3 & row1, const Vec3 & row2 ) {
  rows[ 0 ] = row0;
  rows[ 1 ] = row1;
  rows[ 2 ] = row2;
}

inline Mat3 & Mat3::operator = ( const Mat3 & rhs ) {
  rows[ 0 ] = rhs.rows[ 0 ];
  rows[ 1 ] = rhs.rows[ 1 ];
  rows[ 2 ] = rhs.rows[ 2 ];
  return *this;
}

inline const Mat3 & Mat3::operator *= ( const float rhs ) {
  rows[ 0 ] *= rhs;
  rows[ 1 ] *= rhs;
  rows[ 2 ] *= rhs;
  return *this;
}

inline const Mat3 & Mat3::operator += ( const Mat3 & rhs ) {
  rows[ 0 ] += rhs.rows[ 0 ];
  rows[ 1 ] += rhs.rows[ 1 ];
  rows[ 2 ] += rhs.rows[ 2 ];
  return *this;
}

inline void Mat3::Zero() {
  rows[ 0 ].Zero();
```

```
  rows[ 1 ].Zero();
  rows[ 2 ].Zero();
}

inline void Mat3::Identity() {
  rows[ 0 ] = Vec3( 1, 0, 0 );
  rows[ 1 ] = Vec3( 0, 1, 0 );
  rows[ 2 ] = Vec3( 0, 0, 1 );
}

inline float Mat3::Trace() const {
  const float xx = rows[ 0 ][ 0 ] * rows[ 0 ][ 0 ];
  const float yy = rows[ 1 ][ 1 ] * rows[ 1 ][ 1 ];
  const float zz = rows[ 2 ][ 2 ] * rows[ 2 ][ 2 ];
  return ( xx + yy + zz );
}

inline float Mat3::Determinant() const {
  const float i = rows[ 0 ][ 0 ] * ( rows[ 1 ][ 1 ] * rows[ 2 ][ 2 ] - rows[ 1 ][ 2 ] * rows[ 2 ][
     1 ] );
  const float j = rows[ 0 ][ 1 ] * ( rows[ 1 ][ 0 ] * rows[ 2 ][ 2 ] - rows[ 1 ][ 2 ] * rows[ 2 ][
     0 ] );
  const float k = rows[ 0 ][ 2 ] * ( rows[ 1 ][ 0 ] * rows[ 2 ][ 1 ] - rows[ 1 ][ 1 ] * rows[ 2 ][
     0 ] );
  return ( i - j + k );
}

inline Mat3 Mat3::Transpose() const {
  Mat3 transpose;
  for ( int i = 0; i < 3; i++ ) {
    for ( int j = 0; j < 3; j++ ) {
      transpose.rows[ i ][ j ] = rows[ j ][ i ];
    }
  }
  return transpose;
}

inline Mat3 Mat3::Inverse() const {
  Mat3 inv;
  for ( int i = 0; i < 3; i++ ) {
    for ( int j = 0; j < 3; j++ ) {
      inv.rows[ j ][ i ] = Cofactor( i, j );  // Perform the transpose with the cofactors
    }
  }
  float det = Determinant();
  float invDet = 1.0f / det;
  inv *= invDet;
  return inv;
}

inline Mat2 Mat3::Minor( const int i, const int j ) const {
  Mat2 minor;

  int yy = 0;
  for ( int y = 0; y < 3; y++ ) {
    if ( y == j ) {
      continue;
    }

    int xx = 0;
    for ( int x = 0; x < 3; x++ ) {
      if ( x == i ) {
        continue;
      }

      minor.rows[ xx ][ yy ] = rows[ x ][ y ];
      xx++;
    }

    yy++;
  }
```

```
  return minor;
}

inline float Mat3::Cofactor( const int i, const int j ) const {
  const Mat2 minor = Minor( i, j );
  const float C = float( pow( -1, i + 1 + j + 1 ) ) * minor.Determinant();
  return C;
}

inline Vec3 Mat3::operator * ( const Vec3 & rhs ) const {
  Vec3 tmp;
  tmp[ 0 ] = rows[ 0 ].Dot( rhs );
  tmp[ 1 ] = rows[ 1 ].Dot( rhs );
  tmp[ 2 ] = rows[ 2 ].Dot( rhs );
  return tmp;
}

inline Mat3 Mat3::operator * ( const float rhs ) const {
  Mat3 tmp;
  tmp.rows[ 0 ] = rows[ 0 ] * rhs;
  tmp.rows[ 1 ] = rows[ 1 ] * rhs;
  tmp.rows[ 2 ] = rows[ 2 ] * rhs;
  return tmp;
}

inline Mat3 Mat3::operator * ( const Mat3 & rhs ) const {
  Mat3 tmp;
  for ( int i = 0; i < 3; i++ ) {
    tmp.rows[ i ].x = rows[ i ].x * rhs.rows[ 0 ].x + rows[ i ].y * rhs.rows[ 1 ].x + rows[ i ].z
    * rhs.rows[ 2 ].x;
    tmp.rows[ i ].y = rows[ i ].x * rhs.rows[ 0 ].y + rows[ i ].y * rhs.rows[ 1 ].y + rows[ i ].z
    * rhs.rows[ 2 ].y;
    tmp.rows[ i ].z = rows[ i ].x * rhs.rows[ 0 ].z + rows[ i ].y * rhs.rows[ 1 ].z + rows[ i ].z
    * rhs.rows[ 2 ].z;
  }
  return tmp;
}

inline Mat3 Mat3::operator + ( const Mat3 & rhs ) const {
  Mat3 tmp;
  for ( int i = 0; i < 3; i++ ) {
    tmp.rows[ i ] = rows[ i ] + rhs.rows[ i ];
  }
  return tmp;
}

/*
================================
Mat4
================================
*/
class Mat4 {
public:
  Mat4() {}
  Mat4( const Mat4 & rhs );
  Mat4( const float * mat );
  Mat4( const Vec4 & row0, const Vec4 & row1, const Vec4 & row2, const Vec4 & row3 );
  Mat4 & operator = ( const Mat4 & rhs );
  ~Mat4() {}

  void Zero();
  void Identity();

  float Trace() const;
  float Determinant() const;
  Mat4 Transpose() const;
  Mat4 Inverse() const;
  Mat3 Minor( const int i, const int j ) const;
  float Cofactor( const int i, const int j ) const;

  void Orient( Vec3 pos, Vec3 fwd, Vec3 up );
```

```
  void LookAt( Vec3 pos, Vec3 lookAt, Vec3 up );
  void PerspectiveOpenGL( float fovy, float aspect_ratio, float near, float far );
  void PerspectiveVulkan( float fovy, float aspect_ratio, float near, float far );
  void OrthoOpenGL( float xmin, float xmax, float ymin, float ymax, float znear, float zfar );
  void OrthoVulkan( float xmin, float xmax, float ymin, float ymax, float znear, float zfar );

  const float * ToPtr() const { return rows[ 0 ].ToPtr(); }
  float * ToPtr() { return rows[ 0 ].ToPtr(); }

  Vec4 operator * ( const Vec4 & rhs ) const;
  Mat4 operator * ( const float rhs ) const;
  Mat4 operator * ( const Mat4 & rhs ) const;
  const Mat4 & operator *= ( const float rhs );

public:
  Vec4 rows[ 4 ];
};

inline Mat4::Mat4( const Mat4 & rhs ) {
  rows[ 0 ] = rhs.rows[ 0 ];
  rows[ 1 ] = rhs.rows[ 1 ];
  rows[ 2 ] = rhs.rows[ 2 ];
  rows[ 3 ] = rhs.rows[ 3 ];
}

inline Mat4::Mat4( const float * mat ) {
  rows[ 0 ] = mat + 0;
  rows[ 1 ] = mat + 4;
  rows[ 2 ] = mat + 8;
  rows[ 3 ] = mat + 12;
}

inline Mat4::Mat4( const Vec4 & row0, const Vec4 & row1, const Vec4 & row2, const Vec4 & row3 ) {
  rows[ 0 ] = row0;
  rows[ 1 ] = row1;
  rows[ 2 ] = row2;
  rows[ 3 ] = row3;
}

inline Mat4 & Mat4::operator = ( const Mat4 & rhs ) {
  rows[ 0 ] = rhs.rows[ 0 ];
  rows[ 1 ] = rhs.rows[ 1 ];
  rows[ 2 ] = rhs.rows[ 2 ];
  rows[ 3 ] = rhs.rows[ 3 ];
  return *this;
}

inline const Mat4 & Mat4::operator *= ( const float rhs ) {
  rows[ 0 ] *= rhs;
  rows[ 1 ] *= rhs;
  rows[ 2 ] *= rhs;
  rows[ 3 ] *= rhs;
  return *this;
}

inline void Mat4::Zero() {
  rows[ 0 ].Zero();
  rows[ 1 ].Zero();
  rows[ 2 ].Zero();
  rows[ 3 ].Zero();
}

inline void Mat4::Identity() {
  rows[ 0 ] = Vec4( 1, 0, 0, 0 );
  rows[ 1 ] = Vec4( 0, 1, 0, 0 );
  rows[ 2 ] = Vec4( 0, 0, 1, 0 );
  rows[ 3 ] = Vec4( 0, 0, 0, 1 );
}

inline float Mat4::Trace() const {
  const float xx = rows[ 0 ][ 0 ] * rows[ 0 ][ 0 ];
```

```
  const float yy = rows[ 1 ][ 1 ] * rows[ 1 ][ 1 ];
  const float zz = rows[ 2 ][ 2 ] * rows[ 2 ][ 2 ];
  const float ww = rows[ 3 ][ 3 ] * rows[ 3 ][ 3 ];
  return ( xx + yy + zz + ww );
}

inline float Mat4::Determinant() const {
  float det = 0.0f;
  float sign = 1.0f;
  for ( int j = 0; j < 4; j++ ) {
    Mat3 minor = Minor( 0, j );

    det += rows[ 0 ][ j ] * minor.Determinant() * sign;
    sign = sign * -1.0f;
  }
  return det;
}

inline Mat4 Mat4::Transpose() const {
  Mat4 transpose;
  for ( int i = 0; i < 4; i++ ) {
    for ( int j = 0; j < 4; j++ ) {
      transpose.rows[ i ][ j ] = rows[ j ][ i ];
    }
  }
  return transpose;
}

inline Mat4 Mat4::Inverse() const {
  Mat4 inv;
  for ( int i = 0; i < 4; i++ ) {
    for ( int j = 0; j < 4; j++ ) {
      inv.rows[ j ][ i ] = Cofactor( i, j );  // Perform the transpose with the cofactors
    }
  }
  float det = Determinant();
  float invDet = 1.0f / det;
  inv *= invDet;
  return inv;
}

inline Mat3 Mat4::Minor( const int i, const int j ) const {
  Mat3 minor;

  int yy = 0;
  for ( int y = 0; y < 4; y++ ) {
    if ( y == j ) {
      continue;
    }

    int xx = 0;
    for ( int x = 0; x < 4; x++ ) {
      if ( x == i ) {
        continue;
      }

      minor.rows[ xx ][ yy ] = rows[ x ][ y ];
      xx++;
    }

    yy++;
  }
  return minor;
}

inline float Mat4::Cofactor( const int i, const int j ) const {
  const Mat3 minor = Minor( i, j );
  const float C = float( pow( -1, i + 1 + j + 1 ) ) * minor.Determinant();
  return C;
}
```

```
inline void Mat4::Orient( Vec3 pos, Vec3 fwd, Vec3 up ) {
  Vec3 left = up.Cross( fwd );

  // For our coordinate system where:
  // +x-axis = fwd
  // +y-axis = left
  // +z-axis = up
  rows[ 0 ] = Vec4( fwd.x, left.x, up.x, pos.x );
  rows[ 1 ] = Vec4( fwd.y, left.y, up.y, pos.y );
  rows[ 2 ] = Vec4( fwd.z, left.z, up.z, pos.z );
  rows[ 3 ] = Vec4( 0, 0, 0, 1 );
}

inline void Mat4::LookAt( Vec3 pos, Vec3 lookAt, Vec3 up ) {
  Vec3 fwd = pos - lookAt;
  fwd.Normalize();

  Vec3 right = up.Cross( fwd );
  right.Normalize();

  up = fwd.Cross( right );
  up.Normalize();

  // For NDC coordinate system where:
  // +x-axis = right
  // +y-axis = up
  // +z-axis = fwd
  rows[ 0 ] = Vec4( right.x, right.y, right.z, -pos.Dot( right ) );
  rows[ 1 ] = Vec4( up.x, up.y, up.z, -pos.Dot( up ) );
  rows[ 2 ] = Vec4( fwd.x, fwd.y, fwd.z, -pos.Dot( fwd ) );
  rows[ 3 ] = Vec4( 0, 0, 0, 1 );
}

inline void Mat4::PerspectiveOpenGL( float fovy, float aspect_ratio, float near, float far ) {
  const float pi = acosf( -1.0f );
  const float fovy_radians = fovy * pi / 180.0f;
  const float f = 1.0f / tanf( fovy_radians * 0.5f );
  const float xscale = f;
  const float yscale = f / aspect_ratio;

  rows[ 0 ] = Vec4( xscale, 0, 0, 0 );
  rows[ 1 ] = Vec4( 0, yscale, 0, 0 );
  rows[ 2 ] = Vec4( 0, 0, ( far + near ) / ( near - far ), ( 2.0f * far * near ) / ( near - far )
    );
  rows[ 3 ] = Vec4( 0, 0, -1, 0 );
}

inline void Mat4::PerspectiveVulkan( float fovy, float aspect_ratio, float near, float far ) {
  // Vulkan changed its NDC.  It switch from a left handed
  // coordinate system to a right handed one.
  // +x points to the right,
  // +z points into the screen,
  // +y points down (it used to point up, in opengl).
  // It also changed the range from [-1,1] to [0,1] for the z.
  // Clip space remains [-1,1] for x and y.
  // Check section 23 of the specification.
  Mat4 matVulkan;
  matVulkan.rows[ 0 ] = Vec4( 1, 0, 0, 0 );
  matVulkan.rows[ 1 ] = Vec4( 0, -1, 0, 0 );
  matVulkan.rows[ 2 ] = Vec4( 0, 0, 0.5f, 0.5f );
  matVulkan.rows[ 3 ] = Vec4( 0, 0, 0, 1 );

  Mat4 matOpenGL;
  matOpenGL.PerspectiveOpenGL( fovy, aspect_ratio, near, far );

  *this = matVulkan * matOpenGL;
}

inline void Mat4::OrthoOpenGL( float xmin, float xmax, float ymin, float ymax, float znear, float
    zfar ) {
  const float width = xmax - xmin;
```

```
  const float height  = ymax - ymin;
  const float depth = zfar - znear;

  const float tx = -( xmax + xmin ) / width;
  const float ty = -( ymax + ymin ) / height;
  const float tz = -( zfar + znear ) / depth;

  rows[ 0 ] = Vec4( 2.0f / width, 0, 0, tx );
  rows[ 1 ] = Vec4( 0, 2.0f / height, 0, ty );
  rows[ 2 ] = Vec4( 0, 0, -2.0f / depth, tz );
  rows[ 3 ] = Vec4( 0, 0, 0, 1 );
}

inline void Mat4::OrthoVulkan( float xmin, float xmax, float ymin, float ymax, float znear, float
    zfar ) {
  // Vulkan changed its NDC.  It switch from a left handed
  // coordinate system to a right handed one.
  // +x points to the right,
  // +z points into the screen,
  // +y points down (it used to point up, in opengl).
  // It also changed the range from [-1,1] to [0,1] for the z.
  // Clip space remains [-1,1] for x and y.
  // Check section 23 of the specification.
  Mat4 matVulkan;
  matVulkan.rows[ 0 ] = Vec4( 1, 0, 0, 0 );
  matVulkan.rows[ 1 ] = Vec4( 0, -1, 0, 0 );
  matVulkan.rows[ 2 ] = Vec4( 0, 0, 0.5f, 0.5f );
  matVulkan.rows[ 3 ] = Vec4( 0, 0, 0, 1 );

  Mat4 matOpenGL;
  matOpenGL.OrthoOpenGL( xmin, xmax, ymin, ymax, znear, zfar );

  *this = matVulkan * matOpenGL;
}

inline Vec4 Mat4::operator * ( const Vec4 & rhs ) const {
  Vec4 tmp;
  tmp[ 0 ] = rows[ 0 ].Dot( rhs );
  tmp[ 1 ] = rows[ 1 ].Dot( rhs );
  tmp[ 2 ] = rows[ 2 ].Dot( rhs );
  tmp[ 3 ] = rows[ 3 ].Dot( rhs );
  return tmp;
}

inline Mat4 Mat4::operator * ( const float rhs ) const {
  Mat4 tmp;
  tmp.rows[ 0 ] = rows[ 0 ] * rhs;
  tmp.rows[ 1 ] = rows[ 1 ] * rhs;
  tmp.rows[ 2 ] = rows[ 2 ] * rhs;
  tmp.rows[ 3 ] = rows[ 3 ] * rhs;
  return tmp;
}

inline Mat4 Mat4::operator * ( const Mat4 & rhs ) const {
  Mat4 tmp;
  for ( int i = 0; i < 4; i++ ) {
    tmp.rows[ i ].x = rows[ i ].x * rhs.rows[ 0 ].x + rows[ i ].y * rhs.rows[ 1 ].x + rows[ i ].z
    * rhs.rows[ 2 ].x + rows[ i ].w * rhs.rows[ 3 ].x;
    tmp.rows[ i ].y = rows[ i ].x * rhs.rows[ 0 ].y + rows[ i ].y * rhs.rows[ 1 ].y + rows[ i ].z
    * rhs.rows[ 2 ].y + rows[ i ].w * rhs.rows[ 3 ].y;
    tmp.rows[ i ].z = rows[ i ].x * rhs.rows[ 0 ].z + rows[ i ].y * rhs.rows[ 1 ].z + rows[ i ].z
    * rhs.rows[ 2 ].z + rows[ i ].w * rhs.rows[ 3 ].z;
    tmp.rows[ i ].w = rows[ i ].x * rhs.rows[ 0 ].w + rows[ i ].y * rhs.rows[ 1 ].w + rows[ i ].z
    * rhs.rows[ 2 ].w + rows[ i ].w * rhs.rows[ 3 ].w;
  }
  return tmp;
}
```

4 The Quat Class

While we could use the Mat3 class to represent orientations, we will use quaternions instead.

If you're not familiar with quaternions that's okay, you won't actually need to fully understand them to work through this book. Just know that we will use them for rotations in 3D. If, however, you want a career as a programmer in physics, animation, or robotics, then you'll want to do a deep dive and become comfortable with quaternion math.

And once again, this class comes pre-packaged with the project available on github.

```
/*
 ================================
 Quat
 ================================
 */
class Quat {
public:
  Quat();
  Quat( const Quat & rhs );
  Quat( float X, float Y, float Z, float W );
  Quat( Vec3 n, const float angleRadians );
  const Quat & operator = ( const Quat & rhs );

  Quat &  operator *= ( const float & rhs );
  Quat &  operator *= ( const Quat & rhs );
  Quat  operator * ( const Quat & rhs ) const;

  void  Normalize();
  void  Invert();
  Quat  Inverse() const;
  float MagnitudeSquared() const;
  float GetMagnitude() const;
  Vec3  RotatePoint( const Vec3 & rhs ) const;
  Mat3  RotateMatrix( const Mat3 & rhs ) const;
  Vec3  xyz() const { return Vec3( x, y, z ); }
  bool  IsValid() const;

  Mat3  ToMat3() const;
  Vec4  ToVec4() const { return Vec4( w, x, y, z ); }

public:
  float w;
  float x;
  float y;
  float z;
};

inline Quat::Quat() :
x( 0 ),
y( 0 ),
z( 0 ),
w( 1 ) {
}

inline Quat::Quat( const Quat &rhs ) :
x( rhs.x ),
y( rhs.y ),
z( rhs.z ),
w( rhs.w ) {
}

inline Quat::Quat( float X, float Y, float Z, float W ) :
x( X ),
y( Y ),
z( Z ),
w( W ) {
}

inline Quat::Quat( Vec3 n, const float angleRadians ) {
  const float halfAngleRadians = 0.5f * angleRadians;
```

```
  w = cosf( halfAngleRadians );

  const float halfSine = sinf( halfAngleRadians );
  n.Normalize();
  x = n.x * halfSine;
  y = n.y * halfSine;
  z = n.z * halfSine;
}

inline const Quat & Quat::operator = ( const Quat & rhs ) {
  x = rhs.x;
  y = rhs.y;
  z = rhs.z;
  w = rhs.w;
  return *this;
}

inline Quat & Quat::operator *= ( const float & rhs ) {
    x *= rhs;
    y *= rhs;
    z *= rhs;
    w *= rhs;
    return *this;
}

inline Quat & Quat::operator *= ( const Quat & rhs ) {
  Quat temp = *this * rhs;
  w = temp.w;
  x = temp.x;
  y = temp.y;
  z = temp.z;
  return *this;
}

inline Quat Quat::operator * ( const Quat & rhs ) const {
  Quat temp;
  temp.w = ( w * rhs.w ) - ( x * rhs.x ) - ( y * rhs.y ) - ( z * rhs.z );
  temp.x = ( x * rhs.w ) + ( w * rhs.x ) + ( y * rhs.z ) - ( z * rhs.y );
  temp.y = ( y * rhs.w ) + ( w * rhs.y ) + ( z * rhs.x ) - ( x * rhs.z );
  temp.z = ( z * rhs.w ) + ( w * rhs.z ) + ( x * rhs.y ) - ( y * rhs.x );
  return temp;
}

inline void Quat::Normalize() {
  float invMag = 1.0f / GetMagnitude();

  if ( 0.0f * invMag == 0.0f * invMag ) {
    x = x * invMag;
    y = y * invMag;
    z = z * invMag;
    w = w * invMag;
  }
}

inline void Quat::Invert() {
    *this *= 1.0f / MagnitudeSquared();
    x = -x;
    y = -y;
    z = -z;
}

inline Quat Quat::Inverse() const {
    Quat val( *this );
    val.Invert();
    return val;
}

inline float Quat::MagnitudeSquared() const {
    return ( ( x * x ) + ( y * y ) + ( z * z ) + ( w * w ) );
}
```

```
inline float Quat::GetMagnitude() const {
  return sqrtf( MagnitudeSquared() );
}

inline Vec3 Quat::RotatePoint( const Vec3 & rhs ) const {
  Quat vector( rhs.x, rhs.y, rhs.z, 0.0f );
  Quat final = *this * vector * Inverse();
  return Vec3( final.x, final.y, final.z );
}

inline bool Quat::IsValid() const {
  if ( x * 0 != x * 0 ) {
    return false;
  }
  if ( y * 0 != y * 0 ) {
    return false;
  }
  if ( z * 0 != z * 0 ) {
    return false;
  }
  if ( w * 0 != w * 0 ) {
    return false;
  }
  return true;
}

inline Mat3 Quat::RotateMatrix( const Mat3 & rhs ) const {
  Mat3 mat;
  mat.rows[ 0 ] = RotatePoint( rhs.rows[ 0 ] );
  mat.rows[ 1 ] = RotatePoint( rhs.rows[ 1 ] );
  mat.rows[ 2 ] = RotatePoint( rhs.rows[ 2 ] );
  return mat;
}

inline Mat3 Quat::ToMat3() const {
  Mat3 mat;
  mat.Identity();

  mat.rows[ 0 ] = RotatePoint( mat.rows[ 0 ] );
  mat.rows[ 1 ] = RotatePoint( mat.rows[ 1 ] );
  mat.rows[ 2 ] = RotatePoint( mat.rows[ 2 ] );
  return mat;
}
```

5 Bodies

We all have to start somewhere and we're going to start with bodies. A typical simulation is just a collection of bodies that collide. And before we can simulate collisions, we need to discuss how to represent the bodies.

Every body has a position in space. This can be represented with a vector in cartesian coordinates. We will use the Vec3 class that was outlined in the previous chapters.

Also, bodies have orientations. We're going to use quaternions to represent the orientation of our bodies.

The final property that a body must have to be represented in our simulation is a shape. The shape defines the actual geometry of the body. There's all kinds of shapes that we may want to use to represent a body, but in this book, we're only going to bother with spheres. However, since we will simulate more general shapes later, we're going to create a base shape class as well as a sphere shape class that inherits from it.

```
class Shape {
public:
  enum shapeType_t {
    SHAPE_SPHERE,
  };
  virtual shapeType_t GetType() const = 0;
};

class ShapeSphere : public Shape {
public:
  ShapeSphere( float radius ) : m_radius( radius ) {}
  shapeType_t GetType() const override { return SHAPE_SPHERE; }

  float m_radius;
};
```

Now we can start to implement the body class. The simplest version of this class is:

```
class Body {
public:
  Vec3     m_position;
  Quat     m_orientation;
  Shape * m_shape;
};
```

And if you're using the meager renderer provided on the companion website, then we can add this body to the scene like so:

```
void Scene::Initialize() {
  Body body;
  body.m_position = Vec3( 0, 0, 0 );
  body.m_orientation = Quat( 0, 0, 0, 1 );
  body.m_shape = new ShapeSphere( 1.0f );
  m_bodies.push_back( body );
}
```

And running this code should result in something like in Figure 1

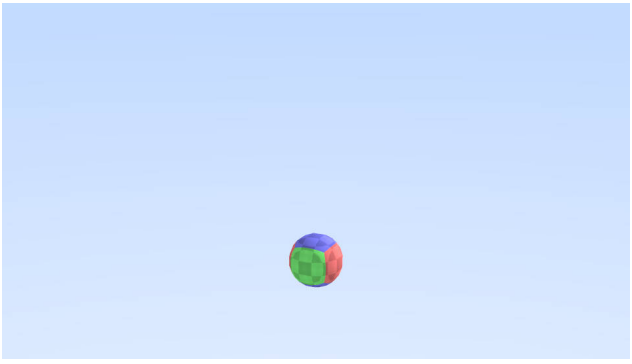


Figure 1: Running sample

And with that you've effectively completed the Hello World of physics simulation. You've managed to create a body and a shape and have them added to a scene that is then rendered to the screen. Congratulations!

5.1 Scene Class

Before moving on, we should probably discuss the scene class a little. It's just a simple way of storing, initializing, and updating all the bodies in the scene. So it's very minimal. The bare bones skeleton of the class looks like this:

```
class Scene {
public:
  void Initialize();
  void Update( const float dt_sec );

  std::vector< Body > m_bodies;
};
```

The initialize function is called once, at application startup to create the bodies in their initial state. And the update function is called at least once per frame. I'm also expecting the application to run at 60+fps. And therefore the value for dt_sec is less than or equal to 0.016 seconds.

5.2 Body Space and World Space

We should probably quickly cover the difference between world space and body space. When things collide, we're going to have information about where in world space that collision occured. And sometimes, we're going to want to know where specifically on the body that hit is located. In order to do this, we'll need a function that can transform a world space point to local space. And we'll also want a function that can take us from local space to world space.

We also need to consider center of mass. We'll get deeper into this concept later in this book. This is important to note because while body space is similar to model space, they are different. Model space is the space centered about the shape geometry's origin. However, body space is centered about the center of mass. Fortunately for our sphere shapes, their center of mass coincides with their geometric center; which makes things easier for this book.

So, let's go ahead and add the center of mass to the shape class.

```
class Shape {
public:
  enum shapeType_t {
    SHAPE_SPHERE,
  };
  virtual shapeType_t GetType() const = 0;

  virtual Vec3 GetCenterOfMass() const { return m_centerOfMass; }

protected:
  Vec3 m_centerOfMass;
};

class ShapeSphere : public Shape {
public:
  ShapeSphere( float radius ) : m_radius( radius ) {
    m_centerOfMass.Zero();
  }
  shapeType_t GetType() const override { return SHAPE_SPHERE; }

  float m_radius;
};
```

And now the world space and body space functions to our body class.

```
class Body {
public:
  Vec3     m_position;
  Quat     m_orientation;
```

```
  Shape * m_shape;

  Vec3 GetCenterOfMassWorldSpace() const;
  Vec3 GetCenterOfMassModelSpace() const;

  Vec3 WorldSpaceToBodySpace( const Vec3 & pt ) const;
  Vec3 BodySpaceToWorldSpace( const Vec3 & pt ) const;
};

Vec3 Body::GetCenterOfMassWorldSpace() const {
  const Vec3 centerOfMass = m_shape->GetCenterOfMass();
  const Vec3 pos = m_position + m_orientation.RotatePoint( centerOfMass );
  return pos;
}

Vec3 Body::GetCenterOfMassModelSpace() const {
  const Vec3 centerOfMass = m_shape->GetCenterOfMass();
  return centerOfMass;
}

Vec3 Body::WorldSpaceToBodySpace( const Vec3 & worldPt ) const {
  Vec3 tmp           = worldPt - GetCenterOfMassWorldSpace();
  Quat inverseOrient = m_orientation.Inverse();
  Vec3 bodySpace     = inverseOrient.RotatePoint( tmp );
  return bodySpace;
}

Vec3 Body::BodySpaceToWorldSpace( const Vec3 & worldPt ) const {
  Vec3 worldSpace = GetCenterOfMassWorldSpace() + m_orientation.RotatePoint( worldPt );
  return worldSpace;
}
```

6 Gravity

Now that we have a body added to the scene and drawn to screen, it would be nice if it also moved. You may have noticed that objects move when an external force is applied to them. And you may have also noticed that all objects are affected by gravity. Therefore, a simple way to get objects moving in our little physics experiment is to apply gravity.

Positions change when they have velocity and the equation relating the change of position with the velocity is:

$$dx = v * dt \tag{1}$$

Velocities change when there is an acceleration. The equation relating the change of velocity with the acceleration is:

$$dv = a * dt \tag{2}$$

The acceleration due to gravity, applies to all objects equally, and near the surface of the Earth is approximately $9.8m/s^2$. We, however, are going to approximate this further to a nice round $10m/s^2$.

Since an object's velocity is typically unique to itself, this is a new property that we should add to the body class:

```
class Body {
public:
  Vec3     m_position;
  Quat     m_orientation;
  Vec3     m_linearVelocity;
  Shape * m_shape;

  Vec3 GetCenterOfMassWorldSpace() const;
  Vec3 GetCenterOfMassModelSpace() const;

  Vec3 WorldSpaceToBodySpace( const Vec3 & pt ) const;
  Vec3 BodySpaceToWorldSpace( const Vec3 & pt ) const;
};
```

We call this linear velocity because we are currently only concerned with translational velocity. We will get to rotational velocity later.

Now, we can add gravity to the velocity of each body and then update the positions of the bodies by modifying the update function of the scene class like so:

```
void Scene::Update( const float dt_sec ) {
  for ( int i = 0; i < m_bodies.size(); i++ ) {
    // Acceleration due to gravity
    m_bodies[ i ].m_linearVelocity += Vec3( 0, 0, -10 ) * dt_sec;
  }

  for ( int i = 0; i < m_bodies.size(); i++ ) {
    // Position update
    m_bodies[ i ].m_position += m_bodies[ i ].m_linearVelocity * dt_sec;
  }
}
```

Running this will now make the body fall at a rate of $10m/s^2$.

7 Gravity as an Impulse - Masses and Impulses

While directly adding the acceleration, due to gravity, to the velocity of a body works, it's not exactly best for us in the long run. Gravity, is a force, and forces act on bodies that have mass. Force is defined as:

$$F = m * a \tag{3}$$

And the force of gravity (near the surface of the earth) is defined as:

$$F = m * g \tag{4}$$

where g is $9.8m/s^2$ (or for us $10m/s^2$). By setting these two equations equal to each other and solving for acceleration, we quickly find that the acceleration due to gravity is g.

Now, it's perfectly valid to deal with gravity this way on paper, but we are going to try and make a robust physics simulation. And for the sake of the simulation it would be better if we applied all forces in a uniform way. In this sense, we would like to be agnostic about gravity and just treat it as we would any other force.

The physics "engine" that we're going to build will handle forces indirectly by applying impulses. This means we need to know what impulses are and how they affect momentum. Momentum is defined as:

$$p = m * v \tag{5}$$

You can think of momentum as the amount a moving object would like to keep moving in a particular direction. So, the more momentum an object has, the more force is required to change its momentum. The change in the momentum is related to the applied force by:

$$dp = F * dt \tag{6}$$

Now the definition of impulse, J, is the change of momentum, so this means

$$J \equiv dp = F * dt \tag{7}$$

And from the relation (5), then:

$$J = m * dv \tag{8}$$

Which, solving for dv yields:

$$dv = J/m \tag{9}$$

This means that we can easily calculate the impulse due to gravity by multiplying the force of gravity by the time delta between frames, dt. And from that we can find the change in velocity by simply dividing by the mass of the body. Now we can go ahead and modify the body class to apply impulses.

```
class Body {
public:
  Body();

  Vec3     m_position;
  Quat     m_orientation;
  Vec3     m_linearVelocity;
  float    m_invMass;
  Shape * m_shape;

  Vec3 GetCenterOfMassWorldSpace() const;
  Vec3 GetCenterOfMassModelSpace() const;

  Vec3 WorldSpaceToBodySpace( const Vec3 & pt ) const;
  Vec3 BodySpaceToWorldSpace( const Vec3 & pt ) const;

  void ApplyImpulseLinear( const Vec3 & impulse );
};

void Body::ApplyImpulseLinear( const Vec3 & impulse ) {
  if ( 0.0f == m_invMass ) {
    return;
```

```
  }

  // p = mv
  // dp = m dv = J
  // => dv = J / m
  m_linearVelocity += impulse * m_invMass;
}
```

It's important to note that we're actually storing the inverse mass, not the mass itself. There's a couple of reasons for this. The first is that we need a way to describe objects with infinite mass. Nothing in the real world actually has infinite mass, but for the kinds of forces that we're going to simulate, we can approximate certain objects as having infinite mass. For instance, a tennis ball thrown against the sidewalk. The tennis ball imparts a force upon the ground and the ground in turn imparts a force upon the ball, and the Earth itself will move a little. But the amount the Earth moves is so small that we can easily approximate this collision as the Earth having an infinite mass.

And now we can adjust our update loop to apply gravity as an impulse, instead of directly using it as an acceleration.

```
void Scene::Update( const float dt_sec ) {
  for ( int i = 0; i < m_bodies.size(); i++ ) {
    Body * body = &m_bodies[ i ];

    // Gravity needs to be an impulse
    // I = dp, F = dp/dt => dp = F * dt => I = F * dt
    // F = mgs
    float mass = 1.0f / body->m_invMass;
    Vec3 impulseGravity = Vec3( 0, 0, -10 ) * mass * dt_sec;
    body->ApplyImpulseLinear( impulseGravity );
  }

  for ( int i = 0; i < m_bodies.size(); i++ ) {
    // Position update
    m_bodies[ i ].m_position += m_bodies[ i ].m_linearVelocity * dt_sec;
  }
}
```

Finally, we can also add a ground body, that won't fall under the influence of gravity either:

```
void Scene::Initialize() {
  Body body;
  body.m_position = Vec3( 0, 0, 10 );
  body.m_orientation = Quat( 0, 0, 0, 1 );
  body.m_invMass = 1.0f;
  body.m_shape = new ShapeSphere( 1.0f );
  m_bodies.push_back( body );

  // Add a "ground" sphere that won't fall under the influence of gravity
  body.m_position = Vec3( 0, 0, -1000 );
  body.m_orientation = Quat( 0, 0, 0, 1 );
  body.m_invMass = 0.0f;
  body.m_shape = new ShapeSphere( 1000.0f );
  m_bodies.push_back( body );
}
```

Now, our simulation has two spheres, one very large "ground" sphere that doesn't move and another normal sized sphere that'll fall.

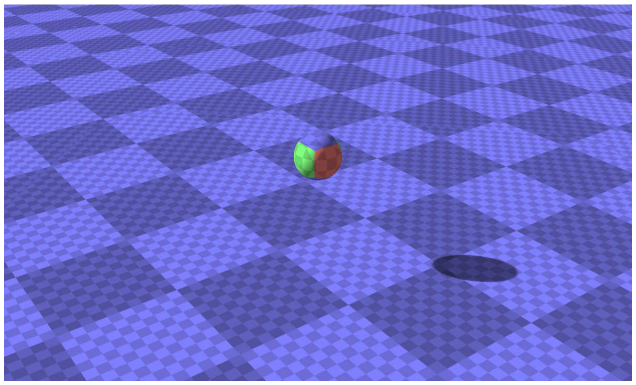


Figure 2: Gravity

8 Collision

Sadly, we still don't have any means of colliding two bodies. Which means running our code so far, will cause the small sphere to pass through the large sphere. This clearly won't work for us.

Fortunately, sphere sphere collisions are the simplest to test. And we can easily add this to our code.

Each sphere has a position, and this position is the center of the sphere in world space. A simple way to check for overlap is to compute the distance between these two points and then compare them with the sum of the two radii. If the distance is less than the two radii, then there's an intersection.

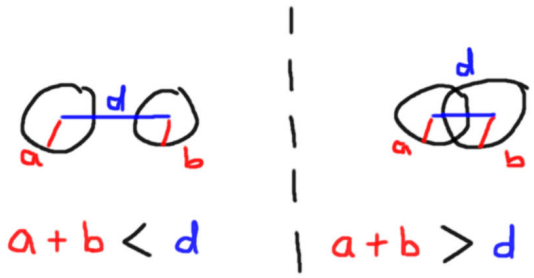


Figure 3: Collision

And to convert this from math to code:

```
bool Intersect( const Body * bodyA, const Body * bodyB ) {
  const Vec3 ab = bodyB->m_position - bodyA->m_position;

  const ShapeSphere * sphereA = (const ShapeSphere *)bodyA->m_shape;
  const ShapeSphere * sphereB = (const ShapeSphere *)bodyB->m_shape;

  const float radiusAB = sphereA->m_radius + sphereB->m_radius;
  const float lengthSquare = ab.GetLengthSqr();
  if ( lengthSquare <= ( radiusAB * radiusAB ) ) {
    return true;
  }

  return false;
}
```

A quick note, a common trick in coding is to compare squares. Taking square roots takes a few more instructions. So, we typically just skip the square root and compare the squared version.

Next we need to add this check to the update function.

Often times people will try to do something clever right out of the gate. But this is a mistake because they, myself included, typically get stuck second guessing themselves. Instead we're going to do the dumb brute force thing to just get it working.

This means we're going to compare every body against every other body for an intersection. And since we need to do something when things intersect, we're going to just set their velocity to zero for now.

```
void Scene::Update( const float dt_sec ) {
  for ( int i = 0; i < m_bodies.size(); i++ ) {
    Body * body = &m_bodies[ i ];

    float mass = 1.0f / body->m_invMass;
    Vec3 impulseGravity = Vec3( 0, 0, -10 ) * mass * dt_sec;
    body->ApplyImpulseLinear( impulseGravity );
  }

  // Check for collisions with other bodies
  for ( int i = 0; i < m_bodies.size(); i++ ) {
    for ( int j = i + 1; j < m_bodies.size(); j++ ) {
      Body * bodyA = &m_bodies[ i ];
      Body * bodyB = &m_bodies[ j ];
```

```
      // Skip body pairs with infinite mass
      if ( 0.0f == bodyA->m_invMass && 0.0f == bodyB->m_invMass ) {
        continue;
      }

      if ( Intersect( bodyA, bodyB ) ) {
        bodyA->m_linearVelocity.Zero();
        bodyB->m_linearVelocity.Zero();
      }
    }
  }

  for ( int i = 0; i < m_bodies.size(); i++ ) {
    m_bodies[ i ].m_position += m_bodies[ i ].m_linearVelocity * dt_sec;
  }
}
```

A couple of notes here. There's no need to bother testing for intersections among pairs of bodies that both have infinite mass, since those bodies are never going to move anyway. So, we just skip those.

Running this new code, the body in the air falls until it hits the ground, then it stops. We now have our first glimpse into collision detection and we're headed in the right direction.

Now, it's great that it stops. But if you look at it closely, you'll notice that the two spheres are actually interpenetrating. What can we do about that?

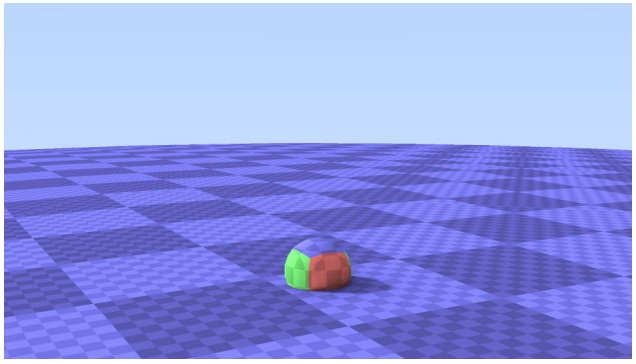


Figure 4: Collision Demo

9 Contacts and Projection method

We would like to work towards a physically based collision response for pairs of bodies, but first we need to solve this interpenetration. One way of solving it, is to use what's known as the projection method. This uses the contacts points on the surface of each body and their masses to separate them appropriately.

Let's start by defining a data structure for the contacts:

```
struct contact_t {
  Vec3 ptOnA_WorldSpace;
  Vec3 ptOnB_WorldSpace;
  Vec3 ptOnA_LocalSpace;
  Vec3 ptOnB_LocalSpace;

  Vec3 normal;  // In World Space coordinates
  float separationDistance; // positive when non-penetrating, negative when penetrating
  float timeOfImpact;

  Body * bodyA;
  Body * bodyB;
};
```

This structure contains all the data that we'll need to properly resolve a collision. For now, we're only going to focus on the ptOnA_WorldSpace, ptOnB_WorldSpace, and the normal. These are the collision points on body A, body B, and the normalized direction from point A to point B.

Now, let's modify the intersect function:

```
bool Intersect( Body * bodyA, Body * bodyB, contact_t & contact ) {
  contact.bodyA = bodyA;
  contact.bodyB = bodyB;

  const Vec3 ab = bodyB->m_position - bodyA->m_position;
  contact.normal = ab;
  contact.normal.Normalize();

  const ShapeSphere * sphereA = (const ShapeSphere *)bodyA->m_shape;
  const ShapeSphere * sphereB = (const ShapeSphere *)bodyB->m_shape;

  contact.ptOnA_WorldSpace = bodyA->m_position + contact.normal * sphereA->m_radius;
  contact.ptOnB_WorldSpace = bodyB->m_position - contact.normal * sphereB->m_radius;

  const float radiusAB = sphereA->m_radius + sphereB->m_radius;
  const float lengthSquare = ab.GetLengthSqr();
  if ( lengthSquare <= ( radiusAB * radiusAB ) ) {
    return true;
  }

  return false;
}
```

This new intersection function fills out the contact struct with all the collision info that we'll need to properly resolve the collisions in the rest of this book. And we're going to celebrate that milestone by changing the scene update function to call a new function:

```
void ResolveContact( contact_t & contact ) {
  Body * bodyA = contact.bodyA;
  Body * bodyB = contact.bodyB;

  bodyA->m_linearVelocity.Zero();
  bodyB->m_linearVelocity.Zero();
}

void Scene::Update( const float dt_sec ) {
  for ( int i = 0; i < m_bodies.size(); i++ ) {
    Body * body = &m_bodies[ i ];

    float mass = 1.0f / body->m_invMass;
    Vec3 impulseGravity = Vec3( 0, 0, -10 ) * mass * dt_sec;
    body->ApplyImpulseLinear( impulseGravity );
  }
```

```
  // Check for collisions with other bodies
  for ( int i = 0; i < m_bodies.size(); i++ ) {
    for ( int j = i + 1; j < m_bodies.size(); j++ ) {
      Body * bodyA = &m_bodies[ i ];
      Body * bodyB = &m_bodies[ j ];

      // Skip body pairs with infinite mass
      if ( 0.0f == bodyA->m_invMass && 0.0f == bodyB->m_invMass ) {
        continue;
      }

      contact_t contact;
      if ( Intersect( bodyA, bodyB, contact ) ) {
        ResolveContact( contact );
      }
    }
  }

  for ( int i = 0; i < m_bodies.size(); i++ ) {
    m_bodies[ i ].m_position += m_bodies[ i ].m_linearVelocity * dt_sec;
  }
}
```

Moving the contact resolution out of the update function and into its own function, is going to let us just focus on the physics of impulses. This is just an organizational change, running the code now won't change any behaviors. But, conveniently, we won't need to look at the update function for a while now.

So now we return to the problem of interpenetration and how best to separate two bodies. For our little program, we have one object with infinite mass and another of finite mass. Since the infinite mass shouldn't move, we could just move the finite mass along the direction of the contact normal by the penetration distance.

While this would work, in this case, it wouldn't be a good idea to simply move the body with the lower mass. After all, if we had two bodies with equal mass, then both bodies should move equally.

A useful concept that we can exploit is the center of mass of the system. The center of mass of N massive particles is defined as:

$$x_{cm} = \frac{\sum_i x_i * m_i}{\sum_i m_i} \tag{10}$$

What we're going to want to do is to separate the colliding bodies such that the center of mass of the two bodies doesn't change. So, for two masses the equation for the center of mass is:

$$x_{cm} = \frac{x_1 * m_1 + x_2 * m_2}{m_1 + m_2} \tag{11}$$

or using inverse mass:

$$x_{cm} = \frac{x_1 * m_2^{-1} + x_2 * m_1^{-1}}{m_2^{-1} + m_1^{-1}} \tag{12}$$

The current separation distance, d, is defined as:

$$d \equiv x_2 - x_1 \tag{13}$$

And we want to find x_1' and x_2' such that d' is zero. Knowing that x_{cm} is constant and d' is zero, we have enough knowledge to solve for x_1' and x_2':

$$d' \equiv x_2' - x_1' = 0 \tag{14}$$

$$
\begin{aligned}
x_{cm} &= \frac{x_1 * m_2^{-1} + x_2 * m_1^{-1}}{m_2^{-1} + m_1^{-1}} \\
&= \frac{x_1' * m_2^{-1} + x_2' * m_1^{-1}}{m_2^{-1} + m_1^{-1}} \\
\implies x_1 * m_2^{-1} + x_2 * m_1^{-1} &= x_1' * m_2^{-1} + x_2' * m_1^{-1}
\end{aligned}
$$

Solving for x_1' and x_2' yields:

$$x_1' = x_1 + \frac{d * m_1^{-1}}{m_1^{-1} + m_2^{-1}} \tag{15}$$

$$x_2' = x_2 - \frac{d * m_2^{-1}}{m_1^{-1} + m_2^{-1}} \tag{16}$$

And translating that into code we get:

```
void ResolveContact( contact_t & contact ) {
  Body * bodyA = contact.bodyA;
  Body * bodyB = contact.bodyB;

  bodyA->m_linearVelocity.Zero();
  bodyB->m_linearVelocity.Zero();

  // Let's also move our colliding objects to just outside of each other
  const float tA = bodyA->m_invMass / ( bodyA->m_invMass + bodyB->m_invMass );
  const float tB = bodyB->m_invMass / ( bodyA->m_invMass + bodyB->m_invMass );

  const Vec3 ds = contact.ptOnB_WorldSpace - contact.ptOnA_WorldSpace;
  bodyA->m_position += ds * tA;
  bodyB->m_position -= ds * tB;
}
```

Now, when we run the program, there's no more interpenetration! Check it out in figure 5

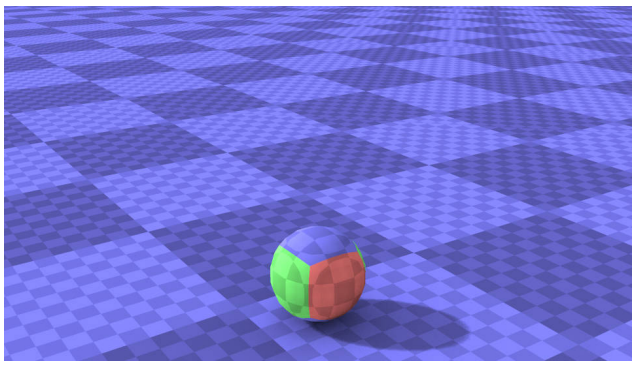


Figure 5: Projection Method

10 Conservation of Momentum

Now that the penetration problem is solved, we need to solve the velocity problem. We can't just set the velocities of our bodies to zero anytime they collide, since that's not very realistic and also boring to watch.

Fortunately, we have the physical laws of conservation of momentum and conservation of energy to exploit. Recall that momentum is defined as:

$$p = m * v \tag{17}$$

And kinetic energy, T, is defined as:

$$T = ½ * m * v^2 \tag{18}$$

Knowing that these quantities are conserved, implies:

$$m_1 * v_1 + m_2 * v_2 = m_1 * v_1' + m_2 * v_2'$$
$$m_1 * v_1^2 + m_2 * v_2^2 = m_1 * v_1'^2 + m_2 * v_2'^2$$

reorganizing these so all the v_1's are on the left and all the v_2's are on the right gives for momentum:

$$m_1 * (v_1 - v_1') = -m_2 * (v_2 - v_2')$$

and for energy:

$$m_1 * (v_1^2 - v_1'^2) = -m_2 * (v_2^2 - v_2'^2)$$
$$\implies m_1 * (v_1 - v_1') * (v_1 + v_1') = -m_2 * (v_2 - v_2') * (v_2 + v_2')$$

from the momentum equation, we can see that some terms cancel, yielding:

$$v_1 + v_1' = v_2 + v_2'$$

solving for v_2':

$$v_2' = v_1 + v_1' - v_2$$

and plugging back into momentum and solving for v_1':

$$m_1 * v_1' = m_1 * v_1 + m_2 * v_2 - m_2 * v_1 - m_2 * v_1' + m_2 * v_2$$
$$\implies (m_1 + m_2) * v_1' = (m_1 - m_2) * v_1 + 2 * m_2 * v_2$$

plugging this into the equation for v_2' gives:

$$(m_1 + m_2) * v_2' = (m_1 + m_2) * v_1 + (m_1 - m_2) * v_1 + 2 * m_2 * v_2 - (m_1 + m_2) * v_2$$
$$= 2 * m_1 * v_1 - m_1 * v_2 + m_2 * v_2$$
$$\implies (m_1 + m_2) * v_2' = (m_2 - m_1) * v_2 + 2 * m_1 * v_1$$

Recall that the definition of impulse is the change in momentum:

$$J_1 = m_1 * (v_1' - v_1)$$
$$J_2 = m_2 * (v_2' - v_2)$$

Then plugging in our solutions for v1' and v2':

$$
\begin{aligned}
J_1 &= m_1 * v_1' - m_1 * v_1 \\
&= \frac{m_1}{m_1 + m_2} * [(m_1 - m_2) * v_1 + 2 * m_2 * v_2] - m_1 * v_1 \\
\implies (m_1 + m_2) * J_1 &= m_1 * (m_1 - m_2) * v_1 + 2 * m_1 * m_2 * v_2 - m_1 * (m_1 + m_2) * v_1 \\
&= -m_1 * m_2 * v_1 + 2 * m_1 * m_2 * v_2 - m_1 * m_2 * v_1 \\
&= 2 * m_1 * m_2 * v_2 - 2 * m_1 * m_2 * v_1 \\
&= 2 * m_1 * m_2 * (v_2 - v_1) \\
\implies J_1 &= \frac{2 * m_1 * m_2 * (v_2 - v_1)}{m_1 + m_2}
\end{aligned}
$$

And likewise for J_2:

$$
J_2 = \frac{2 * m_1 * m_2 (v_1 - v_2)}{m_1 + m_2} = -J_1
$$

Now that was for the one dimensional case. Fortunately translating this into 3D is pretty easy. However, I'm just going to use an informal argument to formulate it. Basically, we only care about the velocities that are normal to the collision. Anything tangential, can be ignored, and the impulse response of the collision should only be along the normal of the contact.

So, to get the component of the relative velocity to the normal, we only need to perform the dot product with the normal.

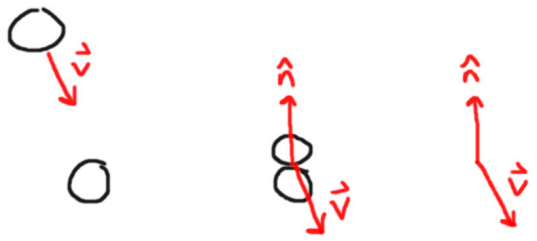


Figure 6: collision normal

And one last thing is that since we're storing the inverse masses of our bodies, it would be nice if the equation was written in terms of the inverse masses. This is pretty simple too, we only need to multiply the numerator and denominator by the inverse masses, which gives:

$$
\begin{aligned}
J_1 &= \frac{2 * (v_2 - v_1)}{m_1^{-1} + m_2^{-1}} \\
J_2 &= \frac{2 * (v_1 - v_2)}{m_1^{-1} + m_2^{-1}}
\end{aligned}
$$

And translating this into code gives:

```
void ResolveContact( contact_t & contact ) {
  Body * bodyA = contact.bodyA;
  Body * bodyB = contact.bodyB;

  const float invMassA = bodyA->m_invMass;
  const float invMassB = bodyB->m_invMass;
```

```
    // Calculate the collision impulse
    const Vec3 & n = contact.normal;
    const Vec3 vab = bodyA->m_linearVelocity - bodyB->m_linearVelocity;
    const float ImpulseJ = -2.0f * vab.Dot( n ) / ( invMassA + invMassB );
    const Vec3 vectorImpulseJ = n * ImpulseJ;

    bodyA->ApplyImpulseLinear( vectorImpulseJ * 1.0f );
    bodyB->ApplyImpulseLinear( vectorImpulseJ * -1.0f );

    //
    // Let's also move our colliding objects to just outside of each other
    //
    const float tA = bodyA->m_invMass / ( bodyA->m_invMass + bodyB->m_invMass );
    const float tB = bodyB->m_invMass / ( bodyA->m_invMass + bodyB->m_invMass );

    const Vec3 ds = contact.ptOnB_WorldSpace - contact.ptOnA_WorldSpace;
    bodyA->m_position += ds * tA;
    bodyB->m_position -= ds * tB;
}
```

Now, running this code, we see that the two bodies bounce appropriately off each other.

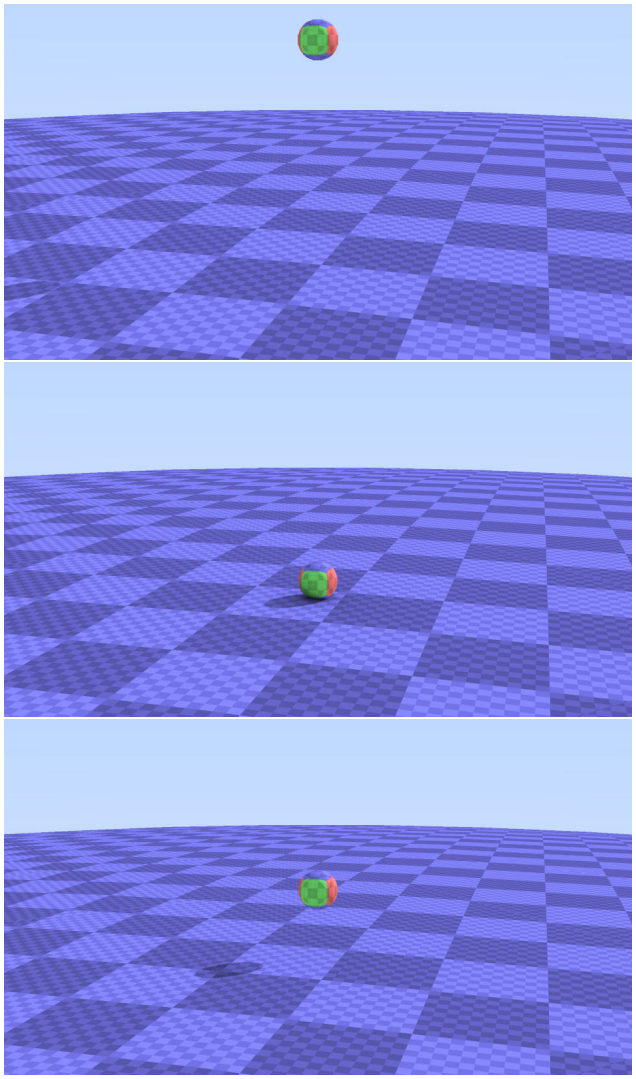


Figure 7: Bounce

11 Elasticity

The collision response in the last chapter is physically accurate. However, it only simulates a collision that perfectly conserves kinetic energy. Which is very closely how billiard balls bounce off each other, but what if we wanted to simulate something a little softer? Suppose instead of having two very hard billiard balls, we have two balls made of clay. Such a collision would actually lose a lot of kinetic energy. Of course, it should go without saying, that in the real world such a collision doesn't lose energy, it's simply converted from kinetic energy to thermal energy. If you ever have the opportunity, you may notice that a large chunk of metal that's been twisted from a high impact collision is very hot.

Let's quickly define some terms. A collision that perfectly conserves kinetic energy is called elastic. And a collision that loses some or all of its kinetic energy is called inelastic.

Since our simulation already simulates an elastic collision, the question becomes "how do we simulate an inelastic collision?"

Well, we can introduce a new variable, ϵ, that represents elasticity. This is also known as the constitution of restitution. And sometimes it's also referred to as "bounciness". ϵ is a number in the range $[0, 1]$ and it relates to the initial and final velocties by the equation:

$$\epsilon = -\frac{v_2' - v_1'}{v_2 - v_1}$$

And if we recall from the last chapter, the definition of impulse and conservation of momentum gave us these equations:

$$v_1' = v_1 + \frac{J}{m_1}$$
$$v_2' = v_2 - \frac{J}{m_2}$$

Subtracting these two equations gives us:

$$v_2' - v_1' = v_2 - v_1 - \frac{J}{m_2} - \frac{J}{m_1}$$

And using the equation for the coefficient of restitution we get:

$$\begin{aligned} -\epsilon * (v_2 - v_1) &= v_2 - v_1 - \frac{J}{m_2} - \frac{J}{m_1} \\ \implies \epsilon * (v_2 - v_1) + (v_2 - v_1) &= \frac{J}{m_2} + \frac{J}{m_1} \\ \implies (1 + \epsilon) * (v_2 - v_1) &= \frac{J}{m_2} + \frac{J}{m_1} \\ \implies J &= (1 + \epsilon) * \frac{v_2 - v_1}{m_1^{-1} + m_2^{-1}} \end{aligned}$$

As we can see, when ϵ is set to 1, we get back the original equation for impulse from the last chapter.

```
class Body {
public:
  Body();

  Vec3     m_position;
  Quat     m_orientation;
  Vec3     m_linearVelocity;

  float    m_invMass;
  float    m_elasticity;
  Shape * m_shape;
```

```
  Vec3 GetCenterOfMassWorldSpace() const;
  Vec3 GetCenterOfMassModelSpace() const;

  Vec3 WorldSpaceToBodySpace( const Vec3 & pt ) const;
  Vec3 BodySpaceToWorldSpace( const Vec3 & pt ) const;

  void ApplyImpulseLinear( const Vec3 & impulse );
};
```

And in the ResolveContact function, we need to determine which body's elasticity to use. In a professional engine for games, you may have some more parameters exposed to allow designers to control how the elasticity is combined for two bodies. However, I like to use the simple pattern of multiplying the two elasticities together. This way, the elasticity that's used reflects the physical properties of both, while remaining in the $[0, 1]$ range. The argument I use for this is if a hard billiard ball hits another ball that's made of clay, then they'll tend to stick. But if you had two balls made of clay, then they should stick even more. So, this should be a pretty good approximation of what you'd expect from the real world.

```
void ResolveContact( contact_t & contact ) {
  Body * bodyA = contact.bodyA;
  Body * bodyB = contact.bodyB;

  const float invMassA = bodyA->m_invMass;
  const float invMassB = bodyB->m_invMass;

  const float elasticityA = bodyA->m_elasticity;
  const float elasticityB = bodyB->m_elasticity;
  const float elasticity = elasticityA * elasticityB;

  // Calculate the collision impulse
  const Vec3 & n = contact.normal;
  const Vec3 vab = bodyA->m_linearVelocity - bodyB->m_linearVelocity;
  const float ImpulseJ = -( 1.0f + elasticity ) * vab.Dot( n ) / ( invMassA + invMassB );
  const Vec3 vectorImpulseJ = n * ImpulseJ;

  bodyA->ApplyImpulseLinear( vectorImpulseJ * 1.0f );
  bodyB->ApplyImpulseLinear( vectorImpulseJ * -1.0f );

  //
  // Let's also move our colliding objects to just outside of each other
  //
  const float tA = bodyA->m_invMass / ( bodyA->m_invMass + bodyB->m_invMass );
  const float tB = bodyB->m_invMass / ( bodyA->m_invMass + bodyB->m_invMass );

  const Vec3 ds = contact.ptOnB_WorldSpace - contact.ptOnA_WorldSpace;
  bodyA->m_position += ds * tA;
  bodyB->m_position -= ds * tB;
}
```

Let's test it. We can set the bodies to be inelastic, then the bouncing ball will never bounce back to its original height:

```
void Scene::Initialize() {
  Body body;
  body.m_position = Vec3( 0, 0, 10 );
  body.m_orientation = Quat( 0, 0, 0, 1 );
  body.m_invMass = 1.0f;
  body.m_elasticity = 0.5f;
  body.m_shape = new ShapeSphere( 1.0f );
  m_bodies.push_back( body );

  // Add a "ground" sphere that won't fall under the influence of gravity
  body.m_position = Vec3( 0, 0, -1000 );
  body.m_orientation = Quat( 0, 0, 0, 1 );
  body.m_invMass = 0.0f;
  body.m_elasticity = 1.0f;
  body.m_shape = new ShapeSphere( 1000.0f );
  m_bodies.push_back( body );
}
```

As you can see, if you run this demo. The ball will bounce less and less, until eventually coming to rest.

12 Angular velocity

You may have noticed that we're still missing rotations. Fortunately, dealing with rotation is almost as simple as handling translation. To do this we must introduce the concepts of torque and angular momentum. These quantities are defined as:

$$\vec{\tau} = \mathbf{I} \cdot \vec{\alpha} = \vec{r} \times \vec{F}$$
$$\vec{L} = \mathbf{I} \cdot \vec{\omega} = \vec{r} \times \vec{p}$$

α is angular acceleration, ω is angular velocity. And these quantities are related to orientation by:

$$d\theta = \omega * dt$$
$$d\omega = \alpha * dt$$

angular momentum and torque are related to each other in a similar way that force and momentum are related:

$$d\vec{L} = \vec{\tau} \cdot dt = \vec{J},$$
$$d\vec{L} = \mathbf{I} \cdot d\vec{\omega},$$
$$\implies d\vec{\omega} = \mathbf{I}^{-1} \cdot \vec{J}$$

Now something that does make this a little more complicated is $\mathbf{I}$. $\mathbf{I}$ is called the inertia tensor. It is similar to mass, where it is a measure of how much an object "resists" change to its angular momentum. We will go into more detail about this in the next book, but for this book it is only important to note that it is a 3x3 matrix.

The inertia tensor is calculated from the mass distribution of the body. Since our bodies have a uniform density and the shapes of the bodies are spheres, the inertia tensor is defined as:

$$\begin{pmatrix} \frac{2}{5}MR^2 & 0 & 0 \\ 0 & \frac{2}{5}MR^2 & 0 \\ 0 & 0 & \frac{2}{5}MR^2 \end{pmatrix} \tag{19}$$

Now, typically in a physics simulation, you only need the inverse inertia tensor; similar to how you really only need to store the inverse mass, as opposed to the mass of a body. However, I thought it would be a little easier to digest this concept if we stored the proper inertia tensor instead of its inverse.

In order to access this in code, we're going to add a function to get it from the shape class:

```
class Shape {
public:
  enum shapeType_t {
    SHAPE_SPHERE,
  };
  virtual shapeType_t GetType() const = 0;
  virtual Mat3 InertiaTensor() const = 0;

  virtual Vec3 GetCenterOfMass() const { return m_centerOfMass; }

protected:
  Vec3 m_centerOfMass;
};



class ShapeSphere : public Shape {
public:
  ShapeSphere( const float radius ) : m_radius( radius ) {
    m_centerOfMass.Zero();
  }
  shapeType_t GetType() const override { return SHAPE_SPHERE; }
```

```
  Mat3 InertiaTensor() const override;

  float m_radius;
};

Mat3 ShapeSphere::InertiaTensor() const {
  Mat3 tensor;
  tensor.Zero();
  tensor.rows[ 0 ][ 0 ] = 2.0f * m_radius * m_radius / 5.0f;
  tensor.rows[ 1 ][ 1 ] = 2.0f * m_radius * m_radius / 5.0f;
  tensor.rows[ 2 ][ 2 ] = 2.0f * m_radius * m_radius / 5.0f;
  return tensor;
}
```

As you've probably noticed, the shapes don't have mass. So to get the full inertia tensor, we're going to also add code to the body class. And we'll want to be able to access the inertia tensor in both world space and local body space.

```
class Body {
public:
  Body();

  Vec3     m_position;
  Quat     m_orientation;
  Vec3     m_linearVelocity;

  float    m_invMass;
  float    m_elasticity;
  Shape * m_shape;

  Vec3 GetCenterOfMassWorldSpace() const;
  Vec3 GetCenterOfMassModelSpace() const;

  Vec3 WorldSpaceToBodySpace( const Vec3 & pt ) const;
  Vec3 BodySpaceToWorldSpace( const Vec3 & pt ) const;

  Mat3 GetInverseInertiaTensorBodySpace() const;
  Mat3 GetInverseInertiaTensorWorldSpace() const;

  void ApplyImpulseLinear( const Vec3 & impulse );
};

Mat3 Body::GetInverseInertiaTensorBodySpace() const {
  Mat3 inertiaTensor    = m_shape->InertiaTensor();
  Mat3 invInertiaTensor = inertiaTensor.Inverse() * m_invMass;
  return invInertiaTensor;
}

Mat3 Body::GetInverseInertiaTensorWorldSpace() const {
  Mat3 inertiaTensor    = m_shape->InertiaTensor();
  Mat3 invInertiaTensor = inertiaTensor.Inverse() * m_invMass;
  Mat3 orient           = m_orientation.ToMat3();
  invInertiaTensor      = orient * invInertiaTensor * orient.Transpose();
  return invInertiaTensor;
}
```

Now that we've got the inertia tensor sorted out, we can get back to the angular impulse. Recall that we've already defined it to be:

$$
\begin{aligned}
d\vec{L} &= \vec{\tau} \cdot dt = \vec{J}, \\
d\vec{L} &= \mathbf{I} \cdot d\vec{\omega}, \\
\implies d\vec{\omega} &= \mathbf{I}^{-1} \cdot \vec{J}
\end{aligned}
$$

It's important to note that these vector quantities are normal to the plane of rotation. For instance, a rotating bicycle wheel. Its angular momentum is a vector quantity that is parallel to the axis of the wheel.

And when you apply a torque to the wheel by peddling harder or applying the break, the torque is a vector quantity that is also parallel to the axis of the wheel.

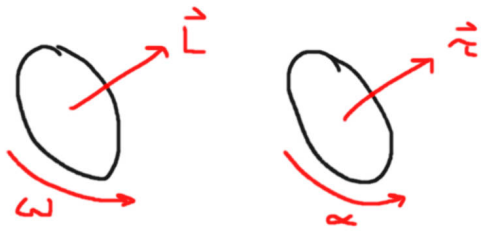


Figure 8: torque angular momentum

With this, we have enough knowledge to implement the angular impulse function:

```
class Body {
public:
  Body();

  Vec3     m_position;
  Quat     m_orientation;
  Vec3     m_linearVelocity;
  Vec3     m_angularVelocity;

  float    m_invMass;
  float    m_elasticity;
  Shape * m_shape;

  Vec3 GetCenterOfMassWorldSpace() const;
  Vec3 GetCenterOfMassModelSpace() const;

  Vec3 WorldSpaceToBodySpace( const Vec3 & pt ) const;
  Vec3 BodySpaceToWorldSpace( const Vec3 & pt ) const;

  Mat3 GetInverseInertiaTensorBodySpace() const;
  Mat3 GetInverseInertiaTensorWorldSpace() const;

  void ApplyImpulseLinear( const Vec3 & impulse );
  void ApplyImpulseAngular( const Vec3 & impulse );
};

void Body::ApplyImpulseAngular( const Vec3 & impulse ) {
  if ( 0.0f == m_invMass ) {
    return;
  }

  // L = I w = r x p
  // dL = I dw = r x J
  // => dw = I^-1 * ( r x J )
  m_angularVelocity += GetInverseInertiaTensorWorldSpace() * impulse;

  const float maxAngularSpeed = 30.0f; // 30 rad/s is fast enough for us. But feel free to adjust.
  if ( m_angularVelocity.GetLengthSqr() > maxAngularSpeed * maxAngularSpeed ) {
    m_angularVelocity.Normalize();
    m_angularVelocity *= maxAngularSpeed;
  }
}
```

Let's take a break for a moment to discuss two curious chunks of code.

The first is the angular velocity clamp in the ApplyImpulseAngular function. It is common practice to clamp both the linear and angular velocity of bodies in game simulations.

This is mainly for performance. It'll become more obvious why this is an issue when we get to the broad-phase, and continuous collision detection for non-spherical bodies.

Now, we haven't bothered with clamping the linear velocity in this book, mainly because we don't need it. None of the setups that we'll explore in this series will have objects that move fast enough to cause performance issues. There will be a teleportation bug that we discuss later, but this will get fixed with continous collision detection.

The next line of code I'd like to discuss is from the GetInverseInertiaTensorWorldSpace function:

```
orient * invInertiaTensor * orient.Transpose();
```

This is the world space inverse inertia tensor. You see, the inertia tensor that we had calculated before is in the local or model space of the body. But, the impulse that we're applying is in world space. So, we need to transform the local space inverse inertia tensor into world space. And one way to do that, is by multiplying by the inverse orientation matrix and the orientation matrix.

One way to think of this is an order of operations. We have a world space angular impulse that we'll dub $\vec{J}_w$. And the inverse inertia tensor is in local space, so we need to transform $\vec{J}_w$ into local space, and we can do that with:

$$\vec{J}_{local} = \mathbf{R}^{-1} \cdot \vec{J}_{world}$$

Now we can apply the angular impulse in local space to get the local space change in angular velocity:

$$d\vec{\omega}_{local} = \mathbf{I} \cdot \vec{J}_{local}$$

And then in order to get the change in angular velocity back into world space, we simply use the orientation matrix:

$$d\vec{\omega}_{world} = \mathbf{R} \cdot d\vec{\omega}_{local}$$

So, there's two ways to think of this. Either we're transforming the inverse inertia tensor from local space to world space:

$$\mathbf{I}_{world} = \mathbf{R} \cdot \mathbf{I}_{local} \cdot \mathbf{R}^{-1}$$

Or we're transforming the impulse into local space, applying it, then transforming the change of velocity from local space to world space.

I hope that wasn't too confusing.

And on one last note. The rotation matrix has a determinant of one, because it doesn't scale any vertices, it simlpy rotates them. So that means the inverse of the matrix and the transpose of the matrix are the same thing. A matrix with this special property is known as an orthogonal matrix. So we can also write:

$$\mathbf{I}_{world} = \mathbf{R} \cdot \mathbf{I}_{local} \cdot \mathbf{R}^{T}$$

13 General Impulses

We have now covered both linear impulses and angular impulses. One can think of linear impulses as a general impulse that's applied through the center of mass of the body. And the angular impulses we're applying are also through the center of mass of the body.

However, in practice, very few impulses that are applied to a body will ever be exactly applied through the center of mass. And this means that we'll need to account for that. Which means we need to start with the center of mass.

We've already touched on center of mass when we discussed the projection method for separating bodies. If we recall equation (10) it is defined by:

$$x_{cm} = \frac{\sum_i x_i * m_i}{\sum_i m_i} \tag{20}$$

Now, since we are only going to simulate bodies with uniform mass densities, the center of mass of each body will be based solely upon the body's shape. So let's go ahead and review the shape class:

```
class Shape {
public:
  enum shapeType_t {
    SHAPE_SPHERE,
  };
  virtual shapeType_t GetType() const = 0;
  virtual Mat3 InertiaTensor() const = 0;

  virtual Vec3 GetCenterOfMass() const { return m_centerOfMass; }

protected:
  Vec3 m_centerOfMass;
};



class ShapeSphere : public Shape {
public:
  ShapeSphere( const float radius ) : m_radius( radius ) {
    m_centerOfMass.Zero();
  }
  shapeType_t GetType() const override { return SHAPE_SPHERE; }

  Mat3 InertiaTensor() const override;

  float m_radius;
};
```

Fortunately for us, a sphere's center of mass is its geometric center. So we've been able to get away with using the body's position as equivalent to the center of mass. But it's probably worthwhile for us now to address it, especially since we won't be restricted to spheres in the next book.

If we have a shape where we can't assume its center of mass is also its geometric center, then how do we use the body's position and orientation to get the center of mass in world space?

Well, to do that, it is the same thing we do to transform the vertices of a model from model space to world space. The first thing we need to do is rotate the center of mass by the body's orientation, and then follow that up with a translation by the body's position. And just as a reminder, this is a review of what we've done in an earlier chapter:

```
class Body {
public:
  Body();

  Vec3    m_position;
  Quat    m_orientation;
  Vec3    m_linearVelocity;
  Vec3    m_angularVelocity;

  float   m_invMass;
  float   m_elasticity;
```

```
  Shape * m_shape;

  Vec3 GetCenterOfMassWorldSpace() const;
  Vec3 GetCenterOfMassModelSpace() const;

  Vec3 WorldSpaceToBodySpace( const Vec3 & pt ) const;
  Vec3 BodySpaceToWorldSpace( const Vec3 & pt ) const;

  Mat3 GetInverseInertiaTensorBodySpace() const;
  Mat3 GetInverseInertiaTensorWorldSpace() const;

  void ApplyImpulseLinear( const Vec3 & impulse );
  void ApplyImpulseAngular( const Vec3 & impulse );
};

Vec3 Body::GetCenterOfMassWorldSpace() const {
  const Vec3 centerOfMass = m_shape->GetCenterOfMass();
  const Vec3 pos = m_position + m_orientation.RotatePoint( centerOfMass );
  return pos;
}

Vec3 Body::GetCenterOfMassModelSpace() const {
  const Vec3 centerOfMass = m_shape->GetCenterOfMass();
  return centerOfMass;
}
```

So, since most impulses will be applied to a point on the surface of the body, we can presume we have both the position as well as the impulse itself. Now, how do we figure out what the linear impulse and angular impulses are from the position and impulse itself?

Well, as it turns out the linear impulse will just be the impulse itself. But we need the position of the impulse to figure out the angular impulse. Fortunately, we can just use the definition of the angular impulse to do that.

Recall that:

$$\vec{L} = \mathbf{I} \cdot \vec{w} = \vec{r} \times \vec{p}$$
$$\implies d\vec{L} = \mathbf{I} \cdot d\vec{w} = \vec{r} \times \vec{J}_{linear}$$
$$\implies \vec{J}_{angular} = \vec{r} \times \vec{J}_{linear}$$

Figure 9: angular impulse crossproduct

And translating this into code:

```
class Body {
public:
  Body();

  Vec3     m_position;
  Quat     m_orientation;
  Vec3     m_linearVelocity;
  Vec3     m_angularVelocity;
```

```
  float    m_invMass;
  float    m_elasticity;
  Shape * m_shape;

  Vec3 GetCenterOfMassWorldSpace() const;
  Vec3 GetCenterOfMassModelSpace() const;

  Vec3 WorldSpaceToBodySpace( const Vec3 & pt ) const;
  Vec3 BodySpaceToWorldSpace( const Vec3 & pt ) const;

  Mat3 GetInverseInertiaTensorBodySpace() const;
  Mat3 GetInverseInertiaTensorWorldSpace() const;

  void ApplyImpulse( const Vec3 & impulsePoint, const Vec3 & impulse );
  void ApplyImpulseLinear( const Vec3 & impulse );
  void ApplyImpulseAngular( const Vec3 & impulse );
};

void Body::ApplyImpulse( const Vec3 & impulsePoint, const Vec3 & impulse ) {
  if ( 0.0f == m_invMass ) {
    return;
  }

  // impulsePoint is the world space location of the application of the impulse
  // impulse is the world space direction and magnitude of the impulse
  ApplyImpulseLinear( impulse );

  Vec3 position = GetCenterOfMassWorldSpace();  // applying impulses must produce torques through
    the center of mass
  Vec3 r = impulsePoint - position;
  Vec3 dL = r.Cross( impulse ); // this is in world space
  ApplyImpulseAngular( dL );
}
```

The next thing that we'll need to do is change the way we update the body's position. Currently, we're updating the body's position by the linear velocity in the Scene::Update function in the loop of this code snippet:

```
for ( int i = 0; i < m_bodies.size(); i++ ) {
  m_bodies[ i ].m_position += m_bodies[ i ].m_linearVelocity * dt_sec;
}
```

Instead of doing this, let's give the Body class an update function and then call that in the Scene::Update function. So that'll change the loop to:

```
for ( int i = 0; i < m_bodies.size(); i++ ) {
  m_bodies[ i ].Update( dt_sec );
}
```

And obviously the Body::Update function is defined as:

```
class Body {
public:
  Body();

  Vec3     m_position;
  Quat     m_orientation;
  Vec3     m_linearVelocity;
  Vec3     m_angularVelocity;

  float    m_invMass;
  float    m_elasticity;
  Shape * m_shape;

  Vec3 GetCenterOfMassWorldSpace() const;
  Vec3 GetCenterOfMassModelSpace() const;

  Vec3 WorldSpaceToBodySpace( const Vec3 & pt ) const;
  Vec3 BodySpaceToWorldSpace( const Vec3 & pt ) const;
```

```
  Mat3 GetInverseInertiaTensorBodySpace() const;
  Mat3 GetInverseInertiaTensorWorldSpace() const;

  void ApplyImpulse( const Vec3 & impulsePoint, const Vec3 & impulse );
  void ApplyImpulseLinear( const Vec3 & impulse );
  void ApplyImpulseAngular( const Vec3 & impulse );

  void Update( const float dt_sec );
};

void Body::Update( const float dt_sec ) {
  m_position += m_linearVelocity * dt_sec;
}
```

Now we just need to figure out how to update the orientation from the angular velocity and then implement it.

Updating the orientation from the angular velocity is slightly more complicated than updating position. If the body's shape is asymmetric, then the object will precess and cause an internal torque on itself. This may be counter intuitive, but you can search for videos on the internet of astronauts demonstrating this with a T-handle in orbit. Some names of this effect are "The Tennis Racket Theorem", "Dzhanibekov effect", and "Intermediate Axis Theorem".

$$\vec{\tau} = \vec{\omega} \times \mathbf{I} \cdot \vec{\omega}$$
$$\vec{\tau} = \mathbf{I} \cdot \vec{\alpha}$$
$$\vec{\alpha} = \mathbf{I}^{-1} \cdot (\vec{\omega} \times \mathbf{I} \cdot \vec{\omega})$$
$$d\vec{\omega} = \vec{\alpha} \cdot dt$$
$$d\vec{\theta} = \vec{\omega} \cdot dt$$

And now, in order to update the orientation, instead of using addition, we update it with multiplication:

$$q' = dq * q$$

```
void Body::Update( const float dt_sec ) {
  m_position += m_linearVelocity * dt_sec;

  // okay, we have an angular velocity around the center of mass, this needs to be
  // converted somehow to relative to model position.  This way we can properly update
  // the orientation of the model.
  Vec3 positionCM = GetCenterOfMassWorldSpace();
  Vec3 cmToPos = m_position - positionCM;

  // Total Torque is equal to external applied torques + internal torque (precession)
  // T = T_external + omega x I * omega
  // T_external = 0 because it was applied in the collision response function
  // T = Ia = w x I * w
  // a = I^-1 ( w x I * w )
  Mat3 orientation = m_orientation.ToMat3();
  Mat3 inertiaTensor = orientation * m_shape->InertiaTensor() * orientation.Transpose();
  Vec3 alpha = inertiaTensor.Inverse() * ( m_angularVelocity.Cross( inertiaTensor *
    m_angularVelocity ) );
  m_angularVelocity += alpha * dt_sec;

  // Update orientation
  Vec3 dAngle = m_angularVelocity * dt_sec;
  Quat dq = Quat( dAngle, dAngle.GetMagnitude() );
  m_orientation = dq * m_orientation;
  m_orientation.Normalize();

  // Now get the new model position
  m_position = positionCM + dq.RotatePoint( cmToPos );
}
```

Now, the next thing we need to do is update how we collide our bodies.

14 Add Angular Collision Impulse

You might've thought that we had finished the contact resolution function. But unfortunately, the impulse calculated for the impact did not take into account angular velocity. And that is the goal of this chapter.

Since angular velocity doesn't really matter for the impact impulse of spherical shapes, we won't be fully testing it yet. It'll matter much more when we investigate other shapes in the next book. But, I want to introduce it for two reasons.

One reason is for completeness. Since this book is all about resolving ballistic contacts, it is important to include the angular velocity in that contact response.

The other reason is so that when we begin the next book (assuming you continue the series), we no longer have to worry about the ResolveContact function. It'll be finished and we can just focus on the new tasks.

So let's get on with it. Assume we have two rotating bodies that collide with each other like in figure 10:

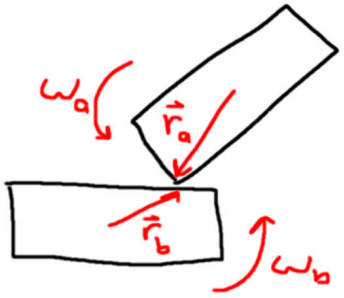


Figure 10: angular collision

Now, if you recall from before we had the following equations from the linear conservation of momentum:

$$v_1' = v_1 + \frac{J}{m_1}$$
$$v_2' = v_2 - \frac{J}{m_2}$$

But now we also need to consider the conservation of angular momentum as well:

$$\vec{\omega}_1' = \vec{\omega}_1 + \mathbf{I}^{-1} \cdot (\vec{r}_1 \times \vec{n}) \cdot \vec{J}$$
$$\vec{\omega}_2' = \vec{\omega}_2 + \mathbf{I}^{-1} \cdot (\vec{r}_2 \times \vec{n}) \cdot \vec{J}$$

And we also had from elasticity that:

$$v_{12}' = -\epsilon * v_{12}$$

Only now the total linear velocity at the point of impact is:

$$\vec{v}_{1_{total}} = \vec{v}_1 + \vec{r}_1 \times \vec{\omega}_1$$
$$\vec{v}_{2_{total}} = \vec{v}_2 + \vec{r}_2 \times \vec{\omega}_2$$

Using these equations to solve for the impulse gives:

$$\vec{J} = \frac{(1+\epsilon) * (\vec{v}_2 - \vec{v}_1)}{m_1^{-1} + m_2^{-1} + (\mathbf{I_1}^{-1}(\vec{r_1} \times \vec{n}) \times \vec{r_1} + \mathbf{I_2}^{-1}(\vec{r_2} \times \vec{n}) \times \vec{r_2}) \cdot \vec{n}} \tag{21}$$

And translating this into code:

```
void ResolveContact( contact_t & contact ) {
  Body * bodyA = contact.bodyA;
  Body * bodyB = contact.bodyB;

  const Vec3 ptOnA = contact.ptOnA_WorldSpace;
  const Vec3 ptOnB = contact.ptOnB_WorldSpace;

  const float elasticityA = bodyA->m_elasticity;
  const float elasticityB = bodyB->m_elasticity;
  const float elasticity = elasticityA * elasticityB;

  const float invMassA = bodyA->m_invMass;
  const float invMassB = bodyB->m_invMass;

  const Mat3 invWorldInertiaA = bodyA->GetInverseInertiaTensorWorldSpace();
  const Mat3 invWorldInertiaB = bodyB->GetInverseInertiaTensorWorldSpace();

  const Vec3 n = contact.normal;

  const Vec3 ra = ptOnA - bodyA->GetCenterOfMassWorldSpace();
  const Vec3 rb = ptOnB - bodyB->GetCenterOfMassWorldSpace();

  const Vec3 angularJA = ( invWorldInertiaA * ra.Cross( n ) ).Cross( ra );
  const Vec3 angularJB = ( invWorldInertiaB * rb.Cross( n ) ).Cross( rb );
  const float angularFactor = ( angularJA + angularJB ).Dot( n );

  // Get the world space velocity of the motion and rotation
  const Vec3 velA = bodyA->m_linearVelocity + bodyA->m_angularVelocity.Cross( ra );
  const Vec3 velB = bodyB->m_linearVelocity + bodyB->m_angularVelocity.Cross( rb );

  // Calculate the collision impulse
  const Vec3 vab = velA - velB;
  const float ImpulseJ = ( 1.0f + elasticity ) * vab.Dot( n ) / ( invMassA + invMassB +
    angularFactor );
  const Vec3 vectorImpulseJ = n * ImpulseJ;

  bodyA->ApplyImpulse( ptOnA, vectorImpulseJ * -1.0f );
  bodyB->ApplyImpulse( ptOnB, vectorImpulseJ * 1.0f );

  //
  // Let's also move our colliding objects to just outside of each other
  //
  const float tA = bodyA->m_invMass / ( bodyA->m_invMass + bodyB->m_invMass );
  const float tB = bodyB->m_invMass / ( bodyA->m_invMass + bodyB->m_invMass );

  const Vec3 ds = contact.ptOnB_WorldSpace - contact.ptOnA_WorldSpace;
  bodyA->m_position += ds * tA;
  bodyB->m_position -= ds * tB;
}
```

15 Friction

Now that we've included rotation into our simulation. We can introduce friction.

Sliding friction (also known as Coulomb friction) is defined as:

$$F_f = \mu * N \tag{22}$$

F_f is the force of friction. N is the normal force. And μ is the coefficient of friction.

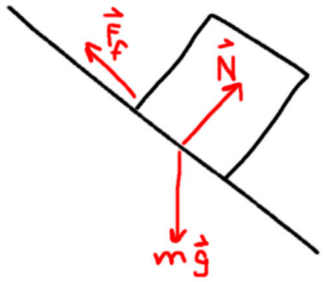


Figure 11: friction free body diagram

In the real world, we tend to make a distinction between static coefficients of friction and kinetic coefficients of friction. However, we're not going to bother with this, and we will only have a singular value for μ.

Another important note is that the force of friction is always in the opposite direction of the velocity. And it always removes energy from the system, it never makes a sliding object slide faster. So it would be more appropriate to write:

$$F_f \leq \mu * N \tag{23}$$

So how do we calculate the normal force? Well, since our impulse response is already along the direction of the normal, we can treat that as our normal force.

And then if we calculate the tangential velocity of bodyA and bodyB at the contact point, then we can use that as the direction that the force of friction is applied.

However, a surprisingly effective approach to friction is to treat it as a tangential collision that removes a portion of the energy each frame. As long as the values for friction that we use are between $[0, 1)$ then we can use all the previous mathematics we've already derived. Only instead of using the normal to calculate the impulses, we use the tangential velocity for the direction. This yields:

$$\vec{J} = \frac{\mu * \vec{v_t}}{m_1^{-1} + m_2^{-1} + (\mathbf{I_1}^{-1}(\vec{r_1} \times \hat{v}_t) \times \vec{r_1} + \mathbf{I_2}^{-1}(\vec{r_2} \times \hat{v}_t) \times \vec{r_2}) \cdot \hat{v}_t} \tag{24}$$

And translating this into code:

```
void ResolveContact( contact_t & contact ) {
  Body * bodyA = contact.bodyA;
  Body * bodyB = contact.bodyB;

  const Vec3 ptOnA = contact.ptOnA_WorldSpace;
  const Vec3 ptOnB = contact.ptOnB_WorldSpace;

  const float elasticityA = bodyA->m_elasticity;
  const float elasticityB = bodyB->m_elasticity;
  const float elasticity = elasticityA * elasticityB;

  const float invMassA = bodyA->m_invMass;
  const float invMassB = bodyB->m_invMass;

  const Mat3 invWorldInertiaA = bodyA->GetInverseInertiaTensorWorldSpace();
  const Mat3 invWorldInertiaB = bodyB->GetInverseInertiaTensorWorldSpace();
```

```
  const Vec3 n = contact.normal;

  const Vec3 ra = ptOnA - bodyA->GetCenterOfMassWorldSpace();
  const Vec3 rb = ptOnB - bodyB->GetCenterOfMassWorldSpace();

  const Vec3 angularJA = ( invWorldInertiaA * ra.Cross( n ) ).Cross( ra );
  const Vec3 angularJB = ( invWorldInertiaB * rb.Cross( n ) ).Cross( rb );
  const float angularFactor = ( angularJA + angularJB ).Dot( n );

  // Get the world space velocity of the motion and rotation
  const Vec3 velA = bodyA->m_linearVelocity + bodyA->m_angularVelocity.Cross( ra );
  const Vec3 velB = bodyB->m_linearVelocity + bodyB->m_angularVelocity.Cross( rb );

  // Calculate the collision impulse
  const Vec3 vab = velA - velB;
  const float ImpulseJ = ( 1.0f + elasticity ) * vab.Dot( n ) / ( invMassA + invMassB +
    angularFactor );
  const Vec3 vectorImpulseJ = n * ImpulseJ;

  bodyA->ApplyImpulse( ptOnA, vectorImpulseJ * -1.0f );
  bodyB->ApplyImpulse( ptOnB, vectorImpulseJ * 1.0f );

  //
  // Calculate the impulse caused by friction
  //

  const float frictionA = bodyA->m_friction;
  const float frictionB = bodyB->m_friction;
  const float friction = frictionA * frictionB;

  // Find the normal direction of the velocity with respect to the normal of the collision
  const Vec3 velNorm = n * n.Dot( vab );

  // Find the tangent direction of the velocity with respect to the normal of the collision
  const Vec3 velTang = vab - velNorm;

  // Get the tangential velocities relative to the other body
  Vec3 relativeVelTang = velTang;
  relativeVelTang.Normalize();

  const Vec3 inertiaA = ( invWorldInertiaA * ra.Cross( relativeVelTang ) ).Cross( ra );
  const Vec3 inertiaB = ( invWorldInertiaB * rb.Cross( relativeVelTang ) ).Cross( rb );
  const float invInertia = ( inertiaA + inertiaB ).Dot( relativeVelTang );

  // Calculate the tangential impulse for friction
  const float reducedMass = 1.0f / ( bodyA->m_invMass + bodyB->m_invMass + invInertia );
  const Vec3 impulseFriction = velTang * reducedMass * friction;

  // Apply kinetic friction
  bodyA->ApplyImpulse( ptOnA, impulseFriction * -1.0f );
  bodyB->ApplyImpulse( ptOnB, impulseFriction * 1.0f );

  //
  // Let's also move our colliding objects to just outside of each other (projection method)
  //
  const Vec3 ds = ptOnB - ptOnA;

  const float tA = invMassA / ( invMassA + invMassB );
  const float tB = invMassB / ( invMassA + invMassB );

  bodyA->m_position += ds * tA;
  bodyB->m_position -= ds * tB;
}
```

And of course we need to add a friction member to the body class:

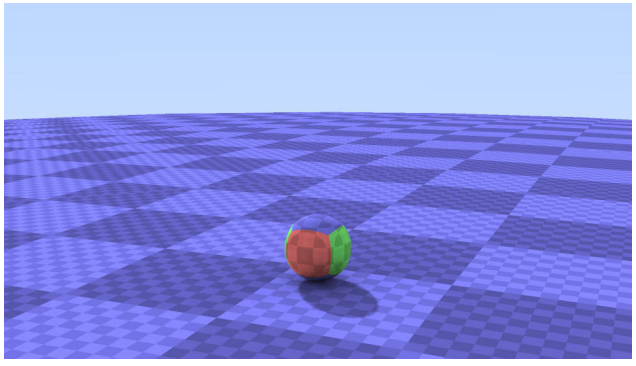
```
class Body {
public:
  Body();
```

```
  Vec3     m_position;
  Quat     m_orientation;
  Vec3     m_linearVelocity;
  Vec3     m_angularVelocity;

  float    m_invMass;
  float    m_elasticity;
  float    m_friction;
  Shape *    m_shape;

  Vec3 GetCenterOfMassWorldSpace() const;
  Vec3 GetCenterOfMassModelSpace() const;

  Vec3 WorldSpaceToBodySpace( const Vec3 & pt ) const;
  Vec3 BodySpaceToWorldSpace( const Vec3 & pt ) const;

  Mat3 GetInverseInertiaTensorBodySpace() const;
  Mat3 GetInverseInertiaTensorWorldSpace() const;

  void ApplyImpulse( const Vec3 & impulsePoint, const Vec3 & impulse );
  void ApplyImpulseLinear( const Vec3 & impulse );
  void ApplyImpulseAngular( const Vec3 & impulse );

  void Update( const float dt_sec );
};
```

And finally if we set the elasticity of our dynamic body to zero so that it won't bounce. And give it a velocity that's tangential to the surface of the ground, then we can test our friction.

```
void Scene::Initialize() {
  Body body;
  body.m_position = Vec3( 0, 0, 1 );
  body.m_orientation = Quat( 0, 0, 0, 1 );
  body.m_linearVelocity = Vec3( 1, 0, 0 );
  body.m_invMass = 1.0f;
  body.m_elasticity = 0.0f;
  body.m_friction = 0.5f;
  body.m_shape = new ShapeSphere( 1.0f );
  m_bodies.push_back( body );

  // Add a "ground" sphere that won't fall under the influence of gravity
  body.m_position = Vec3( 0, 0, -1000 );
  body.m_orientation = Quat( 0, 0, 0, 1 );
  body.m_linearVelocity = Vec3( 0, 0, 0 );
  body.m_invMass = 0.0f;
  body.m_elasticity = 1.0f;
  body.m_friction = 0.5f;
  body.m_shape = new ShapeSphere( 1000.0f );
  m_bodies.push_back( body );
}
```

As we can see, running this code with friction, the dynamic body slides and then starts rotating. And everything is finally behaving the way we would expect if we dropped a bunch of balls or marbles.



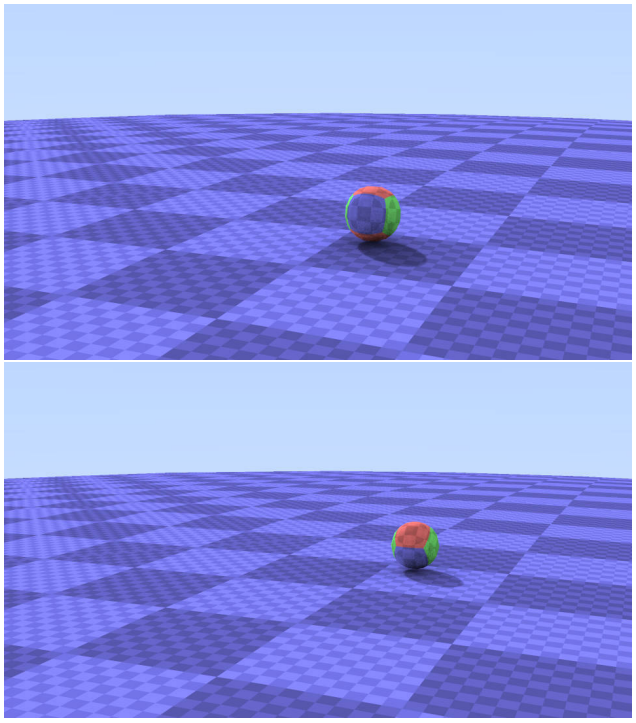


Figure 12: Friction

16 Continuous Collision Detection

Now, that we've fully formed our contact resolution. It's time to discuss the limitations of checking for collisions at finite time steps.

The most glaring problem that we have right now, is that we don't consider the velocities of objects when we check for collision. This means, that if we have some very thin geometry and an object that moves very fast (a classic example is a bullet vs a paper thin wall or a fast moving sphere as in Figure 13) then the object may end up passing through the geometry. This is known as teleportation.

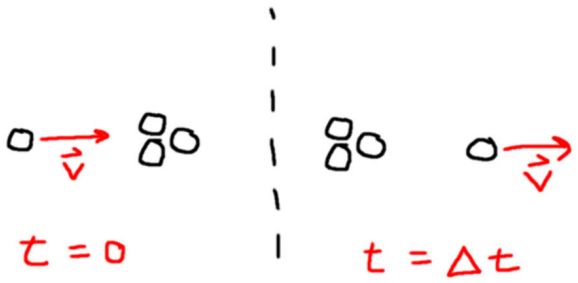


Figure 13: Teleportation Bug

We can fix this problem with what's known as continuous collision detection. All we really need to do is take the velocities of the bodies, and the time step of the frame, into account when we check for intersections. Then, if the two bodies collide, we also need to return the time of impact.

To illustrate the basics of this, let's look at a sphere-sphere collision. We can imagine that in the frame's time step, the two spheres move a certain distance, and their trajectories sweep out a capsule in space:

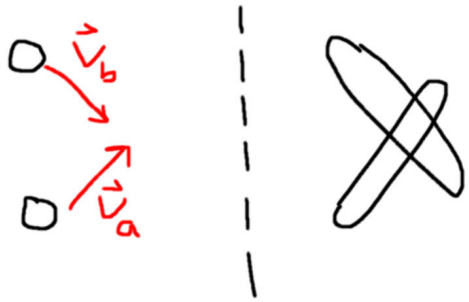


Figure 14: sphere sphere collision

Of course, we could also simplify the situation by using the relative velocities of the two spheres. In this case, one body will appear to be at rest and the other body will move more.

Figure 15: sphere sphere collision relative

Now we only have to collide between a capsule and a sphere. This in turn can be calculated as a line segment versus a sphere with the combined radius of both sphere A and sphere B.

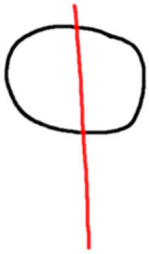


Figure 16: sphere sphere collision ray

This is even simpler, it's a line segment vs a sphere. And solving for a ray trace vs a sphere is pretty common in computer graphics. So it should be pretty simple to solve the equations for intersection.

$$\vec{d} = \vec{b} - \vec{a}$$
$$\vec{r}(t) = \vec{a} + \vec{d} * t$$
$$\vec{s}(t) = \vec{c} - \vec{r}(t)$$

We just need to solve for t such that $\vec{s} \cdot \vec{s} = R^2$

$$\begin{aligned} 0 &= R^2 - \vec{s} \cdot \vec{s} \\ &= R^2 - c^2 - r^2 + 2 \cdot \vec{c} \cdot \vec{r} \\ &= R^2 - c^2 - a^2 - \vec{a} \cdot \vec{d} * t - d^2 * t^2 \end{aligned}$$

Which can be solved with the quadratic equation. And the code for such a simple ray trace is:

```
bool RaySphere( const Vec3 & rayStart, const Vec3 & rayDir, const Vec3 & sphereCenter, const float
    sphereRadius, float & t1, float & t2 ) {
  const Vec3 m = sphereCenter - rayStart;
  const float a = rayDir.Dot( rayDir );
  const float b = m.Dot( rayDir );
  const float c = m.Dot( m ) - sphereRadius * sphereRadius;

  const float delta = b * b - a * c;
```

```
  const float invA = 1.0f / a;

  if ( delta < 0 ) {
    // no real solutions exist
    return false;
  }

  const float deltaRoot = sqrtf( delta );
  t1 = invA * ( b - deltaRoot );
  t2 = invA * ( b + deltaRoot );

  return true;
}
```

What's really great about this is that the t-parameters become the times of impact:

```
bool SphereSphereDynamic( const ShapeSphere * shapeA, const ShapeSphere * shapeB, const Vec3 &
    posA, const Vec3 & posB, const Vec3 & velA, const Vec3 & velB, const float dt, Vec3 & ptOnA,
    Vec3 & ptOnB, float & toi ) {
  const Vec3 relativeVelocity = velA - velB;

  const Vec3 startPtA = posA;
  const Vec3 endPtA = posA + relativeVelocity * dt;
  const Vec3 rayDir = endPtA - startPtA;

  float t0 = 0;
  float t1 = 0;
  if ( rayDir.GetLengthSqr() < 0.001f * 0.001f ) {
    // Ray is too short, just check if already intersecting
    Vec3 ab = posB - posA;
    float radius = shapeA->m_radius + shapeB->m_radius + 0.001f;
    if ( ab.GetLengthSqr() > radius * radius ) {
      return false;
    }
  } else if ( !RaySphere( posA, rayDir, posB, shapeA->m_radius + shapeB->m_radius, t0, t1 ) ) {
    return false;
  }

  // Change from [0,1] range to [0,dt] range
  t0 *= dt;
  t1 *= dt;

  // If the collision is only in the past, then there's not future collision this frame
  if ( t1 < 0.0f ) {
    return false;
  }

  // Get the earliest positive time of impact
  toi = ( t0 < 0.0f ) ? 0.0f : t0;

  // If the earliest collision is too far in the future, then there's no collision this frame
  if ( toi > dt ) {
    return false;
  }

  // Get the points on the respective points of collision and return true
  Vec3 newPosA = posA + velA * toi;
  Vec3 newPosB = posB + velB * toi;
  Vec3 ab = newPosB - newPosA;
  ab.Normalize();

  ptOnA = newPosA + ab * shapeA->m_radius;
  ptOnB = newPosB - ab * shapeB->m_radius;
  return true;
}
```

And now we can update the intersection function to use the dynamic function:

```
bool Intersect( Body * bodyA, Body * bodyB, const float dt, contact_t & contact ) {
  contact.bodyA = bodyA;
  contact.bodyB = bodyB;
```

```
  if ( bodyA->m_shape->GetType() == Shape::SHAPE_SPHERE && bodyB->m_shape->GetType() == Shape::
    SHAPE_SPHERE ) {
    const ShapeSphere * sphereA = (const ShapeSphere *)bodyA->m_shape;
    const ShapeSphere * sphereB = (const ShapeSphere *)bodyB->m_shape;

    Vec3 posA = bodyA->m_position;
    Vec3 posB = bodyB->m_position;

    Vec3 velA = bodyA->m_linearVelocity;
    Vec3 velB = bodyB->m_linearVelocity;

    if ( SphereSphereDynamic( sphereA, sphereB, posA, posB, velA, velB, dt, contact.
    ptOnA_WorldSpace, contact.ptOnB_WorldSpace, contact.timeOfImpact ) ) {
      // Step bodies forward to get local space collision points
      bodyA->Update( contact.timeOfImpact );
      bodyB->Update( contact.timeOfImpact );

      // Convert world space contacts to local space
      contact.ptOnA_LocalSpace = bodyA->WorldSpaceToBodySpace( contact.ptOnA_WorldSpace );
      contact.ptOnB_LocalSpace = bodyB->WorldSpaceToBodySpace( contact.ptOnB_WorldSpace );

      contact.normal = bodyA->m_position - bodyB->m_position;
      contact.normal.Normalize();

      // Unwind time step
      bodyA->Update( -contact.timeOfImpact );
      bodyB->Update( -contact.timeOfImpact );

      // Calculate the separation distance
      Vec3 ab = bodyB->m_position - bodyA->m_position;
      float r = ab.GetMagnitude() - ( sphereA->m_radius + sphereB->m_radius );
      contact.separationDistance = r;
      return true;
    }
  }
  return false;
}
```

Take your time to look at the new Intersect function. We made a lot of changes to it.

Okay, so this is great. We should be able to detect collisions that happen between the frames. However, what's this time of impact parameter and how do we use it for properly resolving the collision?

17 Time of Impact (TOI)

Let's imagine a situtation where there's three objects and they might all potentially collide. But two collide before the other, and then the change in velocities from that collision might prevent the third object from colliding with either.

So, not only do we need to check for collisions, but we need to sort our collisions by TOI and perform our contact resolution in proper temporal order.

Obviously, when bodies collide, their velocities change. So, if a body has two potential collisions, but the first collision changes its velocity enough, to where the second collision's toi changes or doesn't happen at all... then we need to account for that. Something we could do is update all bodies by the TOI of the first collision, resolve that contact, then brute force recalculate all possible collisions for the simulation.

If we worked in aerospace and were designing a jet engine, then sure, we could dedicate 100+hours to simulate 5 minutes of an eagle being sucked into the engine. And we'd be right to do that since we want to make sure the engine doesn't explode and kill everyone on the plane.

However, we're making a simulation intended for games. We don't need super ultra accuracy as much as we need efficiency. For a high performance game we only have 16ms for each frame, and probably less than 2ms to service physics.

So it's okay for us to have some errors in our simulation. As long as they are not so blatantly obvious as to offend any would be users.

Now what are we to do? We can't recalculate every single ricochet because fast moving objects might cause so many secondary and tertiary collisions that our performance tanks and the simulation is no longer "interactive".

One thing we could do is add an "isDynamic" member to the body class. Only update a body's position when it's set to dynamic. And a body is dynamic until it has a collision. This would solve the teleportation bug, and not tank performance.

But it's a solution that has its own problems. If from frame to frame an object has a collision very early in the frame, then its update will appear to stutter. How do we solve that?

Well, instead of setting "isDynamic" to false when it has a single collision, we could wait until its second or third collision to set it to false. That would make a happy medium of avoiding stutter and keeping performance up.

However, it won't be what we do here. Instead, we won't bother recalculating the secondary collision for bodies. This means that it's possible for two bodies to still teleport through each other, but it should now be hidden from the user. This way if there's a cluster of bodies, and a fast moving projectile, then it will still hit the proper first collision. But it might teleport through some other bodies or even have some phantom collisions (collisions that shouldn't have occured). However, the player probably won't notice this error, since the projectile still hit the first proper collision.

```
int CompareContacts( const void * p1, const void * p2 ) {
  contact_t a = *(contact_t*)p1;
  contact_t b = *(contact_t*)p2;

  if ( a.timeOfImpact < b.timeOfImpact ) {
    return -1;
  }

  if ( a.timeOfImpact == b.timeOfImpact ) {
    return 0;
  }

  return 1;
}

void Scene::Update( const float dt_sec ) {
  // Gravity impulse
  for ( int i = 0; i < m_bodies.size(); i++ ) {
    Body * body = &m_bodies[ i ];
    float mass = 1.0f / body->m_invMass;
    Vec3 impulseGravity = Vec3( 0, 0, -10 ) * mass * dt_sec;
    body->ApplyImpulseLinear( impulseGravity );
  }

  int numContacts = 0;
```

```
  const int maxContacts = m_bodies.size() * m_bodies.size();
  contact_t * contacts = (contact_t *)alloca( sizeof( contact_t ) * maxContacts );
  for ( int i = 0; i < m_bodies.size(); i++ ) {
    for ( int j = i + 1; j < m_bodies.size(); j++ ) {
      Body * bodyA = &m_bodies[ i ];
      Body * bodyB = &m_bodies[ j ];

      // Skip body pairs with infinite mass
      if ( 0.0f == bodyA->m_invMass && 0.0f == bodyB->m_invMass ) {
        continue;
      }

      contact_t contact;
      if ( Intersect( bodyA, bodyB, dt_sec, contact ) ) {
        contacts[ numContacts ] = contact;
        numContacts++;
      }
    }
  }

  // Sort the times of impact from earliest to latest
  if ( numContacts > 1 ) {
    qsort( contacts, numContacts, sizeof( contact_t ), CompareContacts );
  }

  float accumulatedTime = 0.0f;
  for ( int i = 0; i < numContacts; i++ ) {
    contact_t & contact = contacts[ i ];
    const float dt = contact.timeOfImpact - accumulatedTime;

    Body * bodyA = contact.bodyA;
    Body * bodyB = contact.bodyB;

    // Skip body pairs with infinite mass
    if ( 0.0f == bodyA->m_invMass && 0.0f == bodyB->m_invMass ) {
      continue;
    }

    // Position update
    for ( int j = 0; j < m_bodies.size(); j++ ) {
      m_bodies[ j ].Update( dt );
    }

    ResolveContact( contact );
    accumulatedTime += dt;
  }

  // Update the positions for the rest of this frame's time
  const float timeRemaining = dt_sec - accumulatedTime;
  if ( timeRemaining > 0.0f ) {
    for ( int i = 0; i < m_bodies.size(); i++ ) {
      m_bodies[ i ].Update( timeRemaining );
    }
  }
}
```

As you can see, we now have the collision loop broken up a little. We first collect all the contacts, and then sort them based upon their timeOfImpact. Then we resolve each contact in temporal order. As we do that, we then update the bodies. It's almost as if we've broken the entire update loop into smaller slices of time.

Something else that I think is important is a modification to the ResolveContact function. In theory, if we resolve a collision at the time the collision happens, then we don't need to manually push the two objects apart. So, now we can change the ResolveContact function to only perform the projection method when the time of impact is zero:

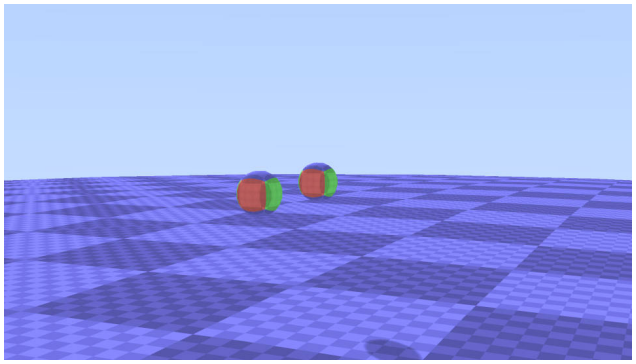
```
void ResolveContact( contact_t & contact ) {
  Body * bodyA = contact.bodyA;
  Body * bodyB = contact.bodyB;
```

```
const Vec3 ptOnA = bodyA->BodySpaceToWorldSpace( contact.ptOnA_LocalSpace );
const Vec3 ptOnB = bodyB->BodySpaceToWorldSpace( contact.ptOnB_LocalSpace );

const float elasticityA = bodyA->m_elasticity;
const float elasticityB = bodyB->m_elasticity;
const float elasticity = elasticityA * elasticityB;

const float invMassA = bodyA->m_invMass;
const float invMassB = bodyB->m_invMass;

const Mat3 invWorldInertiaA = bodyA->GetInverseInertiaTensorWorldSpace();
const Mat3 invWorldInertiaB = bodyB->GetInverseInertiaTensorWorldSpace();

const Vec3 n = contact.normal;

const Vec3 ra = ptOnA - bodyA->GetCenterOfMassWorldSpace();
const Vec3 rb = ptOnB - bodyB->GetCenterOfMassWorldSpace();

const Vec3 angularJA = ( invWorldInertiaA * ra.Cross( n ) ).Cross( ra );
const Vec3 angularJB = ( invWorldInertiaB * rb.Cross( n ) ).Cross( rb );
const float angularFactor = ( angularJA + angularJB ).Dot( n );

// Get the world space velocity of the motion and rotation
const Vec3 velA = bodyA->m_linearVelocity + bodyA->m_angularVelocity.Cross( ra );
const Vec3 velB = bodyB->m_linearVelocity + bodyB->m_angularVelocity.Cross( rb );

// Calculate the collision impulse
const Vec3 vab = velA - velB;
const float ImpulseJ = ( 1.0f + elasticity ) * vab.Dot( n ) / ( invMassA + invMassB +
  angularFactor );
const Vec3 vectorImpulseJ = n * ImpulseJ;

bodyA->ApplyImpulse( ptOnA, vectorImpulseJ * -1.0f );
bodyB->ApplyImpulse( ptOnB, vectorImpulseJ * 1.0f );

//
// Calculate the impulse caused by friction
//

const float frictionA = bodyA->m_friction;
const float frictionB = bodyB->m_friction;
const float friction = frictionA * frictionB;

// Find the normal direction of the velocity with respect to the normal of the collision
const Vec3 velNorm = n * n.Dot( vab );

// Find the tangent direction of the velocity with respect to the normal of the collision
const Vec3 velTang = vab - velNorm;

// Get the tangential velocities relative to the other body
Vec3 relativeVelTang = velTang;
relativeVelTang.Normalize();

const Vec3 inertiaA = ( invWorldInertiaA * ra.Cross( relativeVelTang ) ).Cross( ra );
const Vec3 inertiaB = ( invWorldInertiaB * rb.Cross( relativeVelTang ) ).Cross( rb );
const float invInertia = ( inertiaA + inertiaB ).Dot( relativeVelTang );

// Calculate the tangential impulse for friction
const float reducedMass = 1.0f / ( bodyA->m_invMass + bodyB->m_invMass + invInertia );
const Vec3 impulseFriction = velTang * reducedMass * friction;

// Apply kinetic friction
bodyA->ApplyImpulse( ptOnA, impulseFriction * -1.0f );
bodyB->ApplyImpulse( ptOnB, impulseFriction * 1.0f );

//
// Let's also move our colliding objects to just outside of each other (projection method)
//
if ( 0.0f == contact.timeOfImpact ) {
  const Vec3 ds = ptOnB - ptOnA;
```

```
    const float tA = invMassA / ( invMassA + invMassB );
    const float tB = invMassB / ( invMassA + invMassB );

    bodyA->m_position += ds * tA;
    bodyB->m_position -= ds * tB;
  }
}
```

Let's go ahead and setup a scene to test this. We'll add a floating body with infinite mass, and another body with high velocity.

```
void Scene::Initialize() {
  Body body;

  body.m_position = Vec3( -3, 0, 3 );
  body.m_orientation = Quat( 0, 0, 0, 1 );
  body.m_linearVelocity = Vec3( 1000, 0, 0 );
  body.m_invMass = 1.0f;
  body.m_elasticity = 0.0f;
  body.m_friction = 0.5f;
  body.m_shape = new ShapeSphere( 0.5f );
  m_bodies.push_back( body );

  body.m_position = Vec3( 0, 0, 3 );
  body.m_orientation = Quat( 0, 0, 0, 1 );
  body.m_linearVelocity = Vec3( 0, 0, 0 );
  body.m_invMass = 0.0f;
  body.m_elasticity = 0.0f;
  body.m_friction = 0.5f;
  body.m_shape = new ShapeSphere( 0.5f );
  m_bodies.push_back( body );

  // Add a "ground" sphere that won't fall under the influence of gravity
  body.m_position = Vec3( 0, 0, -1000 );
  body.m_orientation = Quat( 0, 0, 0, 1 );
  body.m_linearVelocity = Vec3( 0, 0, 0 );
  body.m_invMass = 0.0f;
  body.m_elasticity = 1.0f;
  body.m_friction = 0.0f;
  body.m_shape = new ShapeSphere( 1000.0f );
  m_bodies.push_back( body );
}
```



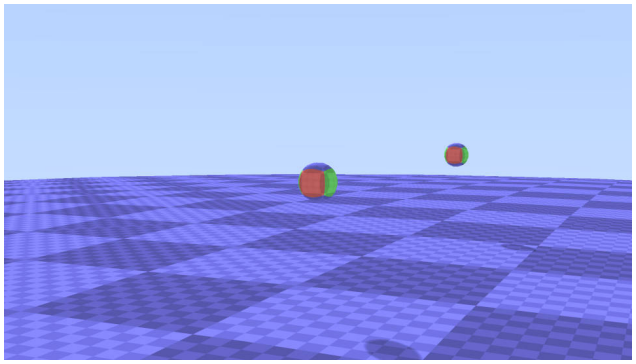
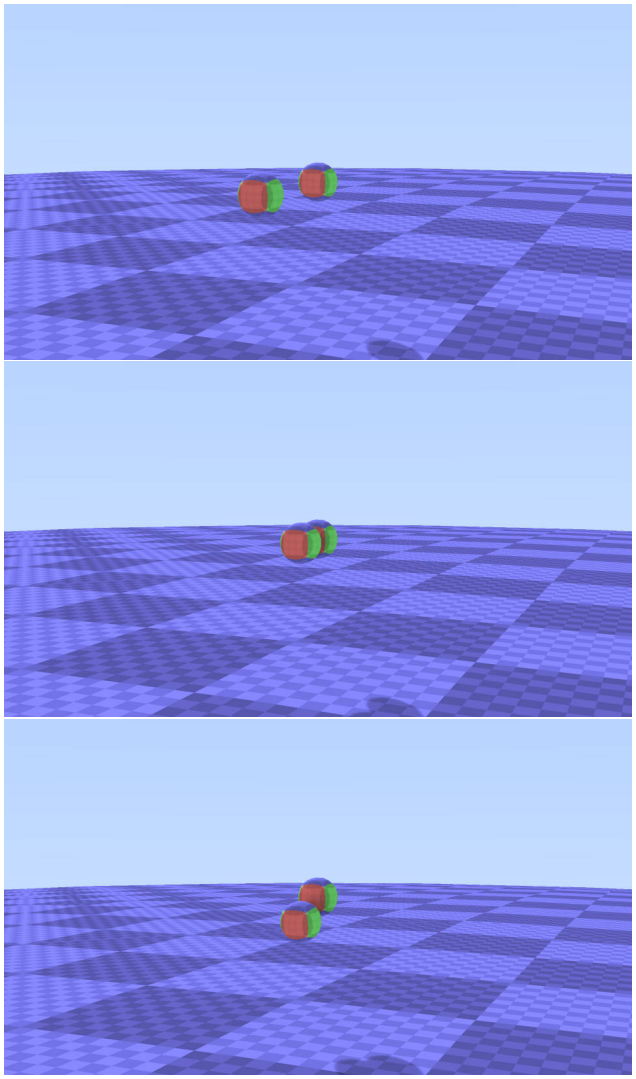


Figure 17: Teleportation Bug

As can be seen above, without properly handling the TOI, fast moving objects will just simply teleport through each other.

Now let's check on what happens when we do properly handle the time of impact.



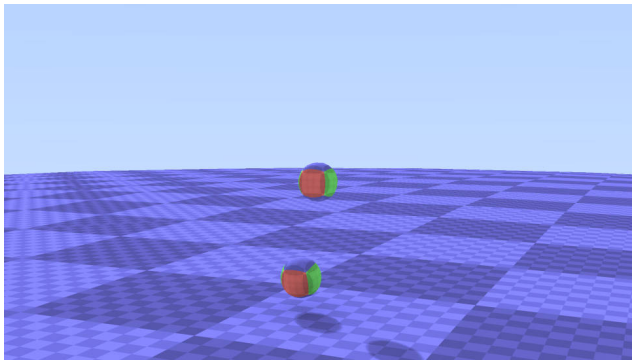


Figure 18: Teleportation Fixed

18 The Bounds Class

Here's a question for you. Do you think there's anything we could do to speed up our collision checking? I'll let you think about that for a minute.

Well, checking the bounds of two shapes is significantly faster than performing an actual collision check. So an easy way to speed up these collision checks is by determining if their bounds overlap. Odds are good the bounds will not overlap and then we've determined there's no collision, without the hard work of actually checking the collision.

So let's have a look at a typical implementation of a bounds class:

```
class Bounds {
public:
  Bounds() { Clear(); }
  Bounds( const Bounds & rhs ) : mins( rhs.mins ), maxs( rhs.maxs ) {}
  const Bounds & operator = ( const Bounds & rhs );
  ~Bounds() {}

  void Clear() { mins = Vec3( 1e6 ); maxs = Vec3( -1e6 ); }
  bool DoesIntersect( const Bounds & rhs ) const;
  void Expand( const Vec3 * pts, const int num );
  void Expand( const Vec3 & rhs );
  void Expand( const Bounds & rhs );

  float WidthX() const { return maxs.x - mins.x; }
  float WidthY() const { return maxs.y - mins.y; }
  float WidthZ() const { return maxs.z - mins.z; }

public:
  Vec3 mins;
  Vec3 maxs;
};

const Bounds & Bounds::operator = ( const Bounds & rhs ) {
  mins = rhs.mins;
  maxs = rhs.maxs;
  return *this;
}

bool Bounds::DoesIntersect( const Bounds & rhs ) const {
  if ( maxs.x < rhs.mins.x || maxs.y < rhs.mins.y || maxs.z < rhs.mins.z ) {
    return false;
  }
  if ( rhs.maxs.x < mins.x || rhs.maxs.y < mins.y || rhs.maxs.z < mins.z ) {
    return false;
  }
  return true;
}

void Bounds::Expand( const Vec3 * pts, const int num ) {
  for ( int i = 0; i < num; i++ ) {
    Expand( pts[ i ] );
  }
}

void Bounds::Expand( const Vec3 & rhs ) {
  if ( rhs.x < mins.x ) {
    mins.x = rhs.x;
  }
  if ( rhs.y < mins.y ) {
    mins.y = rhs.y;
  }
  if ( rhs.z < mins.z ) {
    mins.z = rhs.z;
  }

  if ( rhs.x > maxs.x ) {
    maxs.x = rhs.x;
  }
  if ( rhs.y > maxs.y ) {
```

```
    maxs.y = rhs.y;
  }
  if ( rhs.z > maxs.z ) {
    maxs.z = rhs.z;
  }
}

void Bounds::Expand( const Bounds & rhs ) {
  Expand( rhs.mins );
  Expand( rhs.maxs );
}
```

As you can see the bounds class only stores the minimum and maximum points. And looking at the DoesIntersect function, you can see why this check is so fast. It takes almost no work to check for overlap between bounds.

Now, we need to modify our shape classes too:

```
class Shape {
public:
  virtual Mat3 InertiaTensor() const = 0;

  virtual Bounds GetBounds( const Vec3 & pos, const Quat & orient ) const = 0;
  virtual Bounds GetBounds() const = 0;

  virtual Vec3 GetCenterOfMass() const { return m_centerOfMass; }

  enum shapeType_t {
    SHAPE_SPHERE,
  };
  virtual shapeType_t GetType() const = 0;

protected:
  Vec3 m_centerOfMass;
};

class ShapeSphere : public Shape {
public:
  explicit ShapeSphere( const float radius ) : m_radius( radius ) {
    m_centerOfMass.Zero();
  }

  Mat3 InertiaTensor() const override;

  Bounds GetBounds( const Vec3 & pos, const Quat & orient ) const override;
  Bounds GetBounds() const override;

  shapeType_t GetType() const override { return SHAPE_SPHERE; }

public:
  float m_radius;
};

Bounds ShapeSphere::GetBounds( const Vec3 & pos, const Quat & orient ) const {
  Bounds tmp;
  tmp.mins = Vec3( -m_radius ) + pos;
  tmp.maxs = Vec3( m_radius ) + pos;
  return tmp;
}

Bounds ShapeSphere::GetBounds() const {
  Bounds tmp;
  tmp.mins = Vec3( -m_radius );
  tmp.maxs = Vec3( m_radius );
  return tmp;
}
```

Notice that the GetBounds function takes in a position and orientation. This is mostly for our added convenience. When doing bounds checks on bodies, we need to take into account the position and orientation (for non-spherical bodies). So, I think it's easier to add that utility to the shape class itself.

Now, how do we actually use these bounds? That's for the next chapter!

19 Broadphase & Narrowphase

Doing things brute force is pretty effective for getting started. But it's not very efficient when the number of bodies starts increasing. This means that we'll need to come up with a clever means of efficiently culling out potential collision pairs. In order to do this we will introduce the concept of a broadphase and a narrowphase.

The narrowphase is pretty much what we've been doing this whole time. It's where we calculate contacts between bodies and then resolve those contacts.

The broadphase is where we calculate potential collisions. Basically, we sort the bodies into a data structure that gives us possible collisions pairs. It's a divide and conquer algorithm where we do a little extra work, at the beginning, to avoid doing wasteful work later.

The data structures that could potentially be used in a broadphase are numerous enough to fill their own book (octree, kd-tree, bvh, etc). So, we'll just be using the basic, but effective, sort and sweep algorithm (aka sweep and prune).

In the one dimensional case, we take the bounds of each body, and then sort them into an array of ascending order. Then we simply create collision pairs from the overlapping bodies.

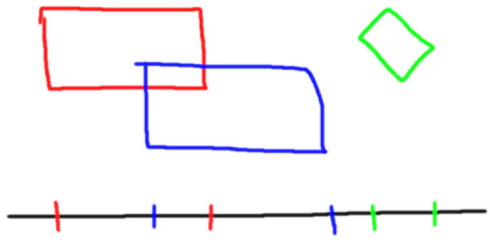


Figure 19: sort and sweep

You might be asking "which axis should we sort on?" Well, sorting on the z-axis can be problematic when all the bodies have fallen onto the same floor height. Then the algorithm ends up in the worst case scenario and we're back to poor performance.

Should we then prefer the x or y axis? I'd say we shouldn't take preference over any of them. And instead we should project the bounds onto the (1, 1, 1) axis. That should lead to decent enough performance for any scenario that we'll encounter in this book series.

And another question you may have is "do we need to account for the velocities of the objects during the broadphase?" The answer to that is yes. We will need to expand the bounds of each body by the distance they would travel in a single timestep. Otherwise all of our work put into continuous collision detection will be for nothing.

The code for this is actually pretty straightforward and it looks a little something like this:

```
struct psuedoBody_t {
  int id;
  float value;
  bool ismin;
};

int CompareSAP( const void * a, const void * b ) {
  const psuedoBody_t * ea = (const psuedoBody_t *)a;
  const psuedoBody_t * eb = (const psuedoBody_t *)b;

  if ( ea->value < eb->value ) {
    return -1;
  }
  return 1;
}

void SortBodiesBounds( const Body * bodies, const int num, psuedoBody_t * sortedArray, const float
     dt_sec ) {
  Vec3 axis = Vec3( 1, 1, 1 );
```

```
  axis.Normalize();

  for ( int i = 0; i < num; i++ ) {
    const Body & body = bodies[ i ];
    Bounds bounds = body.m_shape->GetBounds( body.m_position, body.m_orientation );

    // Expand the bounds by the linear velocity
    bounds.Expand( bounds.mins + body.m_linearVelocity * dt_sec );
    bounds.Expand( bounds.maxs + body.m_linearVelocity * dt_sec );

    const float epsilon = 0.01f;
    bounds.Expand( bounds.mins + Vec3(-1,-1,-1 ) * epsilon );
    bounds.Expand( bounds.maxs + Vec3( 1, 1, 1 ) * epsilon );

    sortedArray[ i * 2 + 0 ].id = i;
    sortedArray[ i * 2 + 0 ].value = axis.Dot( bounds.mins );
    sortedArray[ i * 2 + 0 ].ismin = true;

    sortedArray[ i * 2 + 1 ].id = i;
    sortedArray[ i * 2 + 1 ].value = axis.Dot( bounds.maxs );
    sortedArray[ i * 2 + 1 ].ismin = false;
  }

  qsort( sortedArray, num * 2, sizeof( psuedoBody_t ), CompareSAP );
}

struct collisionPair_t {
  int a;
  int b;

  bool operator == ( const collisionPair_t & rhs ) const {
    return ( ( ( a == rhs.a ) && ( b == rhs.b ) ) || ( ( a == rhs.b ) && ( b == rhs.a ) ) );
  }
  bool operator != ( const collisionPair_t & rhs ) const {
    return !( *this == rhs );
  }
};

void BuildPairs( std::vector< collisionPair_t > & collisionPairs, const psuedoBody_t *
    sortedBodies, const int num ) {
  collisionPairs.clear();

  // Now that the bodies are sorted, build the collision pairs
  for ( int i = 0; i < num * 2; i++ ) {
    const psuedoBody_t & a = sortedBodies[ i ];
    if ( !a.ismin ) {
      continue;
    }

    collisionPair_t pair;
    pair.a = a.id;

    for ( int j = i + 1; j < num * 2; j++ ) {
      const psuedoBody_t & b = sortedBodies[ j ];
      // if we've hit the end of the a element, then we're done creating pairs with a
      if ( b.id == a.id ) {
        break;
      }

      if ( !b.ismin ) {
        continue;
      }

      pair.b = b.id;
      collisionPairs.push_back( pair );
    }
  }
}

void SweepAndPrune1D( const Body * bodies, const int num, std::vector< collisionPair_t > &
    finalPairs, const float dt_sec ) {
```

```
  psuedoBody_t * sortedBodies = (psuedoBody_t *)alloca( sizeof( psuedoBody_t ) * num * 2 );

  SortBodiesBounds( bodies, num, sortedBodies, dt_sec );
  BuildPairs( finalPairs, sortedBodies, num );
}

void BroadPhase( const Body * bodies, const int num, std::vector< collisionPair_t > & finalPairs,
    const float dt_sec ) {
  finalPairs.clear();

  SweepAndPrune1D( bodies, num, finalPairs, dt_sec );
}
```

This straightforward little algorithm may not seem like much. But this does in fact provide a large performance improvement, especially when we get into more complicated shapes in the next book.

And of course, we also need to have a look at how this modifies our core physics loop:

```
void Scene::Update( const float dt_sec ) {
  // Gravity impulse
  for ( int i = 0; i < m_bodies.size(); i++ ) {
    Body * body = &m_bodies[ i ];
    float mass = 1.0f / body->m_invMass;
    Vec3 impulseGravity = Vec3( 0, 0, -10 ) * mass * dt_sec;
    body->ApplyImpulseLinear( impulseGravity );
  }

  //
  // Broadphase
  //
  std::vector< collisionPair_t > collisionPairs;
  BroadPhase( m_bodies.data(), (int)m_bodies.size(), collisionPairs, dt_sec );

  //
  //  NarrowPhase (perform actual collision detection)
  //
  int numContacts = 0;
  const int maxContacts = m_bodies.size() * m_bodies.size();
  contact_t * contacts = (contact_t *)alloca( sizeof( contact_t ) * maxContacts );
  for ( int i = 0; i < collisionPairs.size(); i++ ) {
    const collisionPair_t & pair = collisionPairs[ i ];
    Body * bodyA = &m_bodies[ pair.a ];
    Body * bodyB = &m_bodies[ pair.b ];

    // Skip body pairs with infinite mass
    if ( 0.0f == bodyA->m_invMass && 0.0f == bodyB->m_invMass ) {
      continue;
    }

    contact_t contact;
    if ( Intersect( bodyA, bodyB, dt_sec, contact ) ) {
      contacts[ numContacts ] = contact;
      numContacts++;
    }
  }

  // Sort the times of impact from first to last
  if ( numContacts > 1 ) {
    qsort( contacts, numContacts, sizeof( contact_t ), CompareContacts );
  }

  //
  // Apply ballistic impulses
  //
  float accumulatedTime = 0.0f;
  for ( int i = 0; i < numContacts; i++ ) {
    contact_t & contact = contacts[ i ];
    const float dt = contact.timeOfImpact - accumulatedTime;

    // Position update
    for ( int j = 0; j < m_bodies.size(); j++ ) {
      m_bodies[ j ].Update( dt );
```

```
    }

    ResolveContact( contact );
    accumulatedTime += dt;
  }

  // Update the positions for the rest of this frame's time
  const float timeRemaining = dt_sec - accumulatedTime;
  if ( timeRemaining > 0.0f ) {
    for ( int i = 0; i < m_bodies.size(); i++ ) {
      m_bodies[ i ].Update( timeRemaining );
    }
  }
}
```

Let's go ahead and test this by modifying the number of bodies we are generating:

```
void Scene::Initialize() {
  Body body;

  // Dynamic Bodies
  for ( int x = 0; x < 6; x++ ) {
    for ( int y = 0; y < 6; y++ ) {
      float radius = 0.5f;
      float xx = float( x - 1 ) * radius * 1.5f;
      float yy = float( y - 1 ) * radius * 1.5f;
      body.m_position = Vec3( xx, yy, 10.0f );
      body.m_orientation = Quat( 0, 0, 0, 1 );
      body.m_linearVelocity.Zero();
      body.m_invMass = 1.0f;
      body.m_elasticity = 0.5f;
      body.m_friction = 0.5f;
      body.m_shape = new ShapeSphere( radius );
      m_bodies.push_back( body );
    }
  }

  // Static "floor"
  for ( int x = 0; x < 3; x++ ) {
    for ( int y = 0; y < 3; y++ ) {
      float radius = 80.0f;
      float xx = float( x - 1 ) * radius * 0.25f;
      float yy = float( y - 1 ) * radius * 0.25f;
      body.m_position = Vec3( xx, yy, -radius );
      body.m_orientation = Quat( 0, 0, 0, 1 );
      body.m_linearVelocity.Zero();
      body.m_invMass = 0.0f;
      body.m_elasticity = 0.99f;
      body.m_friction = 0.5f;
      body.m_shape = new ShapeSphere( radius );
      m_bodies.push_back( body );
    }
  }
}
```

With the broadphase in place, instead of doing 2,025 brute force collision checks; I was getting an average of 500 collision checks. Which is a significant savings, especially when we start colliding with more complex shapes.

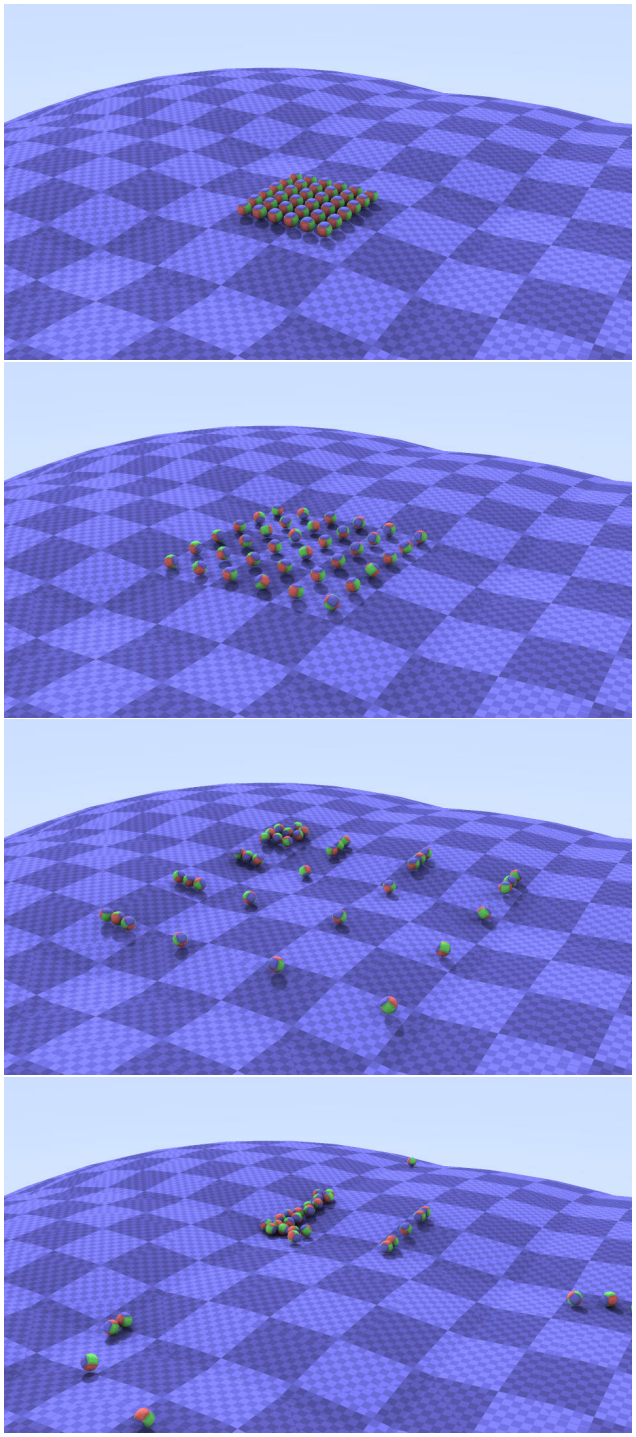


Figure 20: Final

20 Conclusion

And that's it for this book. If you've worked through the whole thing, you now have a very basic physics simulation of bouncing balls with friction. In the next book we're going to expand this simulation to include general convex shapes.

21 Further Reading & References

"Classical Mechanics" - Goldstein, Poole, Safko

"Impulse-based Dynamic Simulation of Rigid Body Systems" - Mirtich 1996