2.1. Coordinates System

geometry. ✱

(1) $\forall p \in \mathbb{R}^n$ $\quad p = p_0 + s_1 v_1 + \dots + s_n v_n$

p_0: origin

$v_1 \cdots v_n$ basis

$s_1 \cdots s_n$ scalar $\in \mathbb{R}$

mathematical entities and are not freely interchangeable.

This definition of points and vectors in terms of coordinate systems reveals a paradox: to define a frame we need a point and a set of vectors, but we can only meaningfully talk about points and vectors with respect to a particular frame. Therefore, in three dimensions we need a *standard frame* with origin $(0, 0, 0)$ and basis vectors $(1, 0, 0)$, $(0, 1, 0)$, and $(0, 0, 1)$. All other frames will be defined with respect to this canonical coordinate system, which we call *world space*.

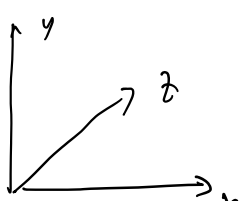
(2) frame → point, vectors → frame

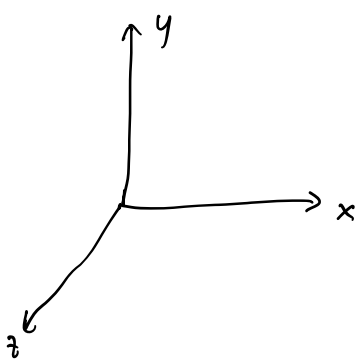
paradox !

(3) standard frame

(1,0,0) (0,1,0) (0,0,1)

(3). left hand coord





2.2. Vectors

(1) Basic structure

```
<<Vector Declarations>>=
template <typename T> class Vector2 {
public:
    <<Vector2 Public Methods>>
    <<Vector2 Public Data>>
       T x, y;

};

<<Vector Declarations>>+=
template <typename T> class Vector3 {
public:
    <<Vector3 Public Methods>>
    <<Vector3 Public Data>>
       T x, y, z;

};
```

(2). v.x (not transparent!) ⇒ v[0]

```
<<Vector3 Public Methods>>=
T operator[](int i) const {
    Assert(i >= 0 && i <= 2);
    if (i == 0) return x;
    if (i == 1) return y;
    return z;
}
T &operator[](int i) {
    Assert(i >= 0 && i <= 2);
    if (i == 0) return x;
    if (i == 1) return y;
    return z;
}
```

if we dont want to change:
xxx = v[0]

if we want to change:
v[0] = xxx ;

Tip: 1. const (so that const object can visit this function)

2. You can change the buffer pointer (point to) not single bit of the buffer

(3)

```
<<Vector Declarations>>+=
typedef Vector2<Float> Vector2f;
typedef Vector2<int>   Vector2i;
typedef Vector3<Float> Vector3f;
typedef Vector3<int>   Vector3i;
```

14). why we made it public (readable!)

15). assert

By default, the (x, y, z) values are set to zero, although the user of the class can optionally supply values for each of the components. If the user does supply values, we check that none of them has the floating-point "not a number" (NaN) value using the `Assert()` macro. When compiled in optimized mode, this macro disappears from the compiled code, saving the expense of verifying this case. NaNs almost certainly indicate a bug in the system; if a NaN is generated by some computation, we'd like to catch it as soon as possible in order to make isolating its source easier. (See Section 3.9.1 for more discussion of NaN values.)

<<Vector3 Public Methods>>+=
```
Vector3() { x = y = z = 0; }
Vector3(T x, T y, T z)
    : x(x), y(y), z(z) {
    Assert(!HasNaNs());
}
```

The code to check for NaNs just calls the `std::isnan()` function on each of the x, y, and z components.

<<Vector3 Public Methods>>+=
```
bool HasNaNs() const {
    return std::isnan(x) || std::isnan(y) || std::isnan(z);
}
```

(b) addition

<<Vector3 Public Methods>>+=
```
Vector3<T> operator+(const Vector3<T> &v) const {
    return Vector3(x + v.x, y + v.y, z + v.z);
}
Vector3<T>& operator+=(const Vector3<T> &v) {
    x += v.x; y += v.y; z += v.z;
    return *this;
}
```

scalar multiplication

A vector can be multiplied component-wise by a scalar, thereby changing its length. Three functions are needed in order to cover all of the different ways that this operation may be written in source code (i.e. `v*s`, `s*v`, and `v *= s`):

<<Vector3 Public Methods>>+=
```
Vector3<T> operator*(T s) const { return Vector3<T>(s*x, s*y, s*z); }
Vector3<T> &operator*=(T s) {
    x *= s; y *= s; z *= s;
    return *this;
}
```

<<Geometry Inline Functions>>=
```
template <typename T> inline Vector3<T>
operator*(T s, const Vector3<T> &v) { return v * s; }
```

(7.) scalar division

scalar multiplication, although division of a scalar by a vector is not w

In the implementation of these methods, we use a single division t then perform three component-wise multiplications. This is a useful which are generally much slower than multiplies on modern CPUs.[†]

We use the `Assert()` macro to make sure that the provided divisor and would indicate a bug elsewhere in the system.

It is a common misconception that these sorts of optimizations are unnecessary because the compiler will perform the necessary analysis. Compilers are generally restricted from performing many transformations of this type. For division, the IEEE floating-point standard requires that x/x = 1 for all x, but if we compute 1/x and store it in a variable and then multiply x by that value, it is not guaranteed that 1 will be the result. In this case, we are willing to lose that guarantee in exchange for higher performance. See Section 3.9 for more discussion of these issues.

<<Vector3 Public Methods>>+=

```
Vector3<T> operator/(T f) const {
    Assert(f != 0);
    Float inv = (Float)1 / f;
    return Vector3<T>(x * inv, y * inv, z * inv);
}

Vector3<T> &operator/=(T f) {
    Assert(f != 0);
    Float inv = (Float)1 / f;
    x *= inv; y *= inv; z *= inv;
    return *this;
}
```

(8.) Dot and Cross Product

$$\mathbf{v}_x\mathbf{w}_x + \mathbf{v}_y\mathbf{w}_y + \mathbf{v}_z\mathbf{w}_z.$$

<<Geometry Inline Functions>>+=

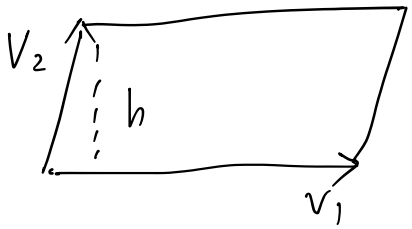
```
template <typename T> inline T
Dot(const Vector3<T> &v1, const Vector3<T> &v2) {
    return v1.x * v2.x + v1.y * v2.y + v1.z * v2.z;
}
```

In the implementation here, the vector elements are converted to double-precision (regardless of the type of `Float`) before the subtractions in the `Cross()` function. Using extra precision for 32-bit floating-point values here protects against error from catastrophic cancellation, a type of floating-point error that can happen when subtracting two values that are very close together. This isn't a theoretical concern: this change was necessary to fix bugs that came up from this issue previously. See Section 3.9 for more information on floating-point rounding error.

<<Geometry Inline Functions>>+=

```
template <typename T> inline Vector3<T>
Cross(const Vector3<T> &v1, const Vector3<T> &v2) {
    double v1x = v1.x, v1y = v1.y, v1z = v1.z;
    double v2x = v2.x, v2y = v2.y, v2z = v2.z;
    return Vector3<T>((v1y * v2z) - (v1z * v2y),
                      (v1z * v2x) - (v1x * v2z),
                      (v1x * v2y) - (v1y * v2x));
}
```

$S = \|v_1 \times v_2\|$



(9)

2.2.2 Normalization

It is often necessary to *normalize* a vector—that is, to compute a new vector pointing in the same direction but with unit length. A normalized vector is often called a *unit vector*. The notation used in this book for normalized vectors is that $\hat{\mathbf{v}}$ is the normalized version of $\mathbf{v}$. To normalize a vector, it's first useful to be able to compute its length.

<<Vector3 Public Methods>>+=
```
Float LengthSquared() const { return x * x + y * y + z * z; }
Float Length() const { return std::sqrt(LengthSquared()); }
```

`Normalize()` normalizes a vector. It divides each component by the length of the vector, $\|\mathbf{v}\|$. It returns a new vector; it does *not* normalize the vector in place:

<<Geometry Inline Functions>>+=
```
template <typename T> inline Vector3<T>
Normalize(const Vector3<T> &v) { return v / v.Length(); }
```

(10). get a coordinate from V_1

<<Geometry Inline Functions>>+=
```
template <typename T> inline void
CoordinateSystem(const Vector3<T> &v1, Vector3<T> *v2, Vector3<T> *v3) {
    if (std::abs(v1.x) > std::abs(v1.y))
        *v2 = Vector3<T>(-v1.z, 0, v1.x) /
              std::sqrt(v1.x * v1.x + v1.z * v1.z);
    else
        *v2 = Vector3<T>(0, v1.z, -v1.y) /
              std::sqrt(v1.y * v1.y + v1.z * v1.z);
    *v3 = Cross(v1, *v2);
}
```

$V_1 (x, y, z)$

$V_2 (-z, 0, x)$

$V_3 = V_1 \times V_2$

2.3. Point.

$P_1 + V_1 \longrightarrow P_2$

<<Point3 Public Methods>>+=

```
Point3<T> operator+(const Vector3<T> &v) const {
    return Point3<T>(x + v.x, y + v.y, z + v.z);
}
Point3<T> &operator+=(const Vector3<T> &v) {
    x += v.x; y += v.y; z += v.z;
    return *this;
}
```

$P_1 - P_2 \rightarrow V_1$

<<Point3 Public Methods>>+=

```
Vector3<T> operator-(const Point3<T> &p) const {
    return Vector3<T>(x - p.x, y - p.y, z - p.z);
}
```

Subtracting a vector from a point gives a new point.

<<Point3 Public Methods>>+=

```
Point3<T> operator-(const Vector3<T> &v) const {
    return Point3<T>(x - v.x, y - v.y, z - v.z);
}
Point3<T> &operator-=(const Vector3<T> &v) {
    x -= v.x; y -= v.y; z -= v.z;
    return *this;
}
```

$P_1 - V_1 \rightarrow P_2$

Although in general it doesn't make sense mathematically to weight points by a scalar or add two points together, the point classes still allow these operations in order to be able to compute weighted sums of points, which is mathematically meaningful as long as the weights used all sum to one. The code for scalar multiplication and addition with points is identical to the corresponding code for vectors, so it is not shown here.

On a related note, it's useful to be able to linearly interpolate between two points. `Lerp()` returns `p0` at `t==0`, `p1` at `t==1`, and linearly interpolates between them at other values of `t`. For `t<0` or `t>1`, `Lerp()` extrapolates.

<<Geometry Inline Functions>>+=

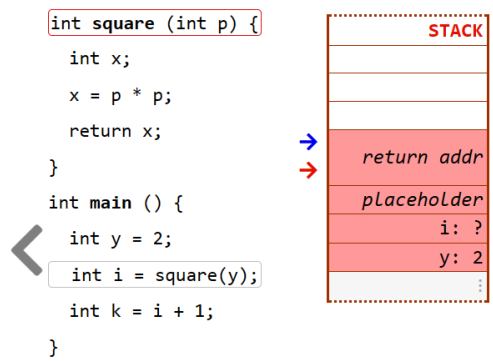
```
template <typename T> Point3<T>
Lerp(Float t, const Point3<T> &p0, const Point3<T> &p1) {
    return (1 - t) * p0 + t * p1;
}
```

Tip: why we use inline function:

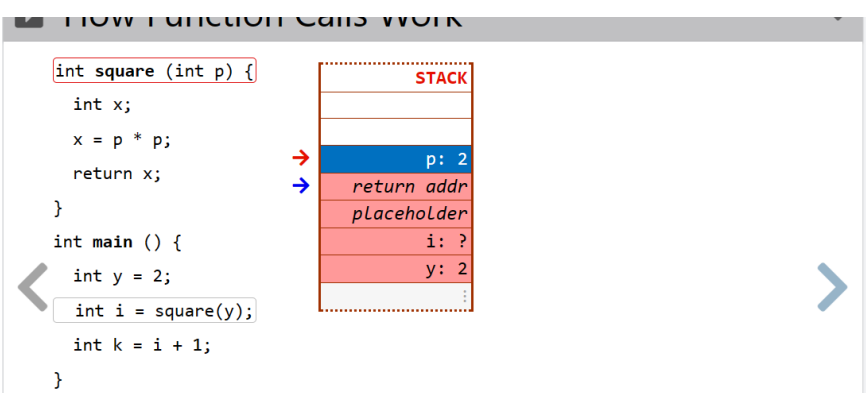
When the program executes the function call instruction the CPU stores the memory address of the instruction following the function call, copies the arguments of the function on the stack and finally transfers control to the specified function. The CPU then executes the function code, stores the function return value in a predefined memory location/register and returns control to the calling function. This can become overhead if the execution time of function is less than the switching time from the caller function to called function (callee). For functions that are large and/or perform complex tasks, the overhead of the function call is usually insignificant compared to the amount of time the function takes to run. However, for small, commonly-used functions, the time needed to make the function call is often a lot more than the time needed to actually execute the function's code. This overhead occurs for small functions because execution time of small function is less than the switching time.

fcall 77
trun

How Function Calls Work



Execution jumps to the memory address of the function `square`.



Function parameter `p` is put on the stack; its value is determined by the call argument (= the value of `y`).

ℹ The order in which return address, placeholder, local parameters, etc. are put on the stack depends on the calling convention of the platform (CPU architecture + OS + compiler)

2.4. Normals.

A *surface normal* (or just *normal*) is a vector that is perpendicular to a surface at a particular position. It can be defined as the cross product of any two nonparallel vectors that are tangent to the surface at a point. Although normals are superficially similar to vectors, it is important to distinguish between the two of them: because normals are defined in terms of their relationship to a particular surface, they behave

The implementations of `Normal3`s and `Vector3`s are very similar. Like vectors, normals are represented by three components x, y, and z; they can be added and subtracted to compute new normals; and they can be scaled and normalized. However, a normal cannot be added to a point, and one cannot take the cross product of two normals. Note that, in an unfortunate turn of terminology, normals are *not* necessarily normalized.

(n).

`Normal3` provides an extra constructor that initializes a `Normal3` from a `Vector3`. Because `Normal3`s and `Vector3`s are different in subtle ways, we want to make sure that this conversion doesn't happen when we don't intend it to, so the C++ `explicit` keyword is again used here. `Vector3` also provides a constructor that converts the other way. Thus, given the declarations `Vector3f v;` and `Normal3f n;`, then the assignment `n = v` is illegal, so it is necessary to explicitly convert the vector, as in `n = Normal3f(v)`.

<<Normal3 Public Methods>>=

```
explicit Normal3<T>(const Vector3<T> &v) : x(v.x), y(v.y), z(v.z) {
    Assert(!v.HasNaNs());
}
```

<<Geometry Inline Functions>>+=

```
template <typename T> inline
Vector3<T>::Vector3(const Normal3<T> &n) : x(n.x), y(n.y), z(n.z) {
    Assert(!n.HasNaNs());
}
```

The `Dot()` and `AbsDot()` functions are also overloaded to compute dot products between the various possible combinations of normals and vectors. This code won't be included in the text here. We also won't include implementations of all of the various other `Normal3` methods here, since they are similar to those for vectors.

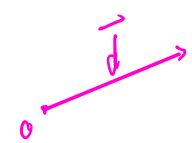
(b)

One new operation to implement comes from the fact it's often necessary to flip a surface normal so that it lies in the same hemisphere as a given vector—for example, the surface normal that lies in the same hemisphere as an outgoing ray is frequently needed. The `Faceforward()` utility function encapsulates this small computation. (`pbrt` also provides variants of this function for the other three combinations of `Vector3`s and `Normal3`s as parameters.) Be careful when using the other instances, though: when using the version that takes two `Vector3`s, for example, ensure that the first parameter is the one that should be returned (possibly flipped) and the second is the one to test against. Reversing the two parameters will give unexpected results.

<<Geometry Inline Functions>>+=

```
template <typename T> inline Normal3<T>
Faceforward(const Normal3<T> &n, const Vector3<T> &v) {
    return (Dot(n, v) < 0.f) ? -n : n;
}
```

2.5. Rays.



$r(t) = o + td \quad o \le t < \infty$

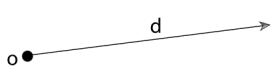


Figure 2.7: A ray is a semi-infinite line defined by its origin o and its direction vector **d**.

<<Ray Declarations>>=

```
class Ray {
public:
    <<Ray Public Methods>>
    <<Ray Public Data>>
        Point3f o;
        Vector3f d;
        mutable Float tMax;
        Float time;
        const Medium *medium;
};
```

animation

transpose

Notice that this field is declared as `mutable`, meaning that it can be changed even if the `Ray` that contains it is `const`—thus, when a ray is passed to a method that takes a `const Ray &`, that method is not allowed to modify its origin or direction but can modify its extent. This convention fits one of the most common uses of rays in the system, as parameters to ray–object intersection testing routines, which will record the offsets to the closest intersection in `tMax`.

<<Ray Public Data>>+=
```
mutable Float tMax;
```

Finally, each ray records the medium containing its origin. The `Medium` class, introduced in Section 11.3, encapsulates the (potentially spatially varying) properties of media such as a foggy atmosphere, smoke, or scattering liquids like milk or shampoo. Associating this information with rays makes it possible for other parts of the system to account correctly for the effect of rays passing from one medium to another.

<<Ray Public Data>>+=
```
const Medium *medium;
```

<<Ray Public Methods>>=
```
Ray() : tMax(Infinity), time(0.f), medium(nullptr) { }
Ray(const Point3f &o, const Vector3f &d, Float tMax = Infinity,
    Float time = 0.f, const Medium *medium = nullptr)
    : o(o), d(d), tMax(tMax), time(time), medium(medium) { }
```

Because position along a ray can be thought of as a function of a single parameter t, the `Ray` class overloads the function application operator for rays. This way, when we need to find the point at a particular position along a ray, we can write code like:

```
Ray r(Point3f(0, 0, 0), Vector3f(1, 2, 3));
Point3f p = r(1.7);
```

<<Ray Public Methods>>+=
```
Point3f operator()(Float t) const { return o + d * t; }
```

2.5.1 Ray differential : Ray

(TODO: go back to this page after futher reading)

Because the RayDifferential class inherits from Ray, geometric interfaces in the system can be written to take const Ray & parameters, so that either a Ray or RayDifferential can be passed to them. Only the routines that need to account for antialiasing and texturing require RayDifferential parameters.

```
<<Ray Declarations>>+=
class RayDifferential : public Ray {
public:
    <<RayDifferential Public Methods>> ⊞
    <<RayDifferential Public Data>> ⊞
};
```

The RayDifferential constructors mirror the Ray's constructors.

```
<<RayDifferential Public Methods>>=
RayDifferential() { hasDifferentials = false; }
RayDifferential(const Point3f &o, const Vector3f &d,
        Float tMax = Infinity, Float time = 0.f,
        const Medium *medium = nullptr)
    : Ray(o, d, tMax, time, medium) {
    hasDifferentials = false;
}
```

```
<<RayDifferential Public Data>>=
bool hasDifferentials;
Point3f rxOrigin, ryOrigin;
Vector3f rxDirection, ryDirection;
```

There is a constructor to create RayDifferentials from Rays. The constructor sets hasDifferentials to false initially because the neighboring rays, if any, are not known.

```
<<RayDifferential Public Methods>>+=
RayDifferential(const Ray &ray) : Ray(ray) {
    hasDifferentials = false;
}
```

Camera implementations in pbrt compute differentials for rays leaving the camera under the assumption that camera rays are spaced one pixel apart. Integrators such as the SamplerIntegrator can generate multiple camera rays per pixel, in which case the actual distance between samples is lower. The fragment <<*Generate camera ray for current sample*>> encountered in Chapter 1 called the ScaleDifferentials() method defined below to update differential rays for an estimated sample spacing of s.

```
<<RayDifferential Public Methods>>+=
void ScaleDifferentials(Float s) {
    rxOrigin = o + (rxOrigin - o) * s;
    ryOrigin = o + (ryOrigin - o) * s;
    rxDirection = d + (rxDirection - d) * s;
    ryDirection = d + (ryDirection - d) * s;
}
```

2.6. Bounding Boxes.

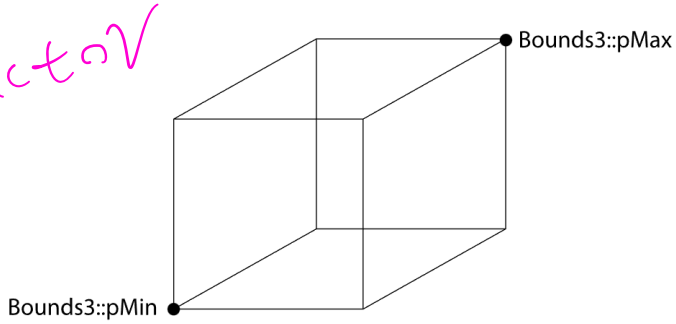


Figure 2.8: An Axis-Aligned Bounding Box. The Bounds2 and Bounds3 classes store only the coordinates of the minimum and maximum points of the box; the other box corners are implicit in this representation.

<<Bounds3 Public Methods>>=
```
Bounds3() {
    T minNum = std::numeric_limits<T>::lowest();
    T maxNum = std::numeric_limits<T>::max();
    pMin = Point3<T>(maxNum, maxNum, maxNum);
    pMax = Point3<T>(minNum, minNum, minNum);
}
```

<<Bounds3 Public Data>>=
```
Point3<T> pMin, pMax;
```

It is also useful to be able to initialize a Bounds3 to enclose a single point:

<<Bounds3 Public Methods>>+=
```
Bounds3(const Point3<T> &p) : pMin(p), pMax(p) { }
```

If the caller passes two corner points (p1 and p2) to define the box, the constructor needs to find their component-wise minimum and maximum values since p1 and p2 are not necessarily chosen so that p1.x <= p2.x, and so on.

<<Bounds3 Public Methods>>+=
```
Bounds3(const Point3<T> &p1, const Point3<T> &p2)
    : pMin(std::min(p1.x, p2.x), std::min(p1.y, p2.y),
           std::min(p1.z, p2.z)),
      pMax(std::max(p1.x, p2.x), std::max(p1.y, p2.y),
           std::max(p1.z, p2.z)) {
}
```

In some cases, it's also useful to use array indexing to select between the two points at the corners of the

In some cases, it's also useful to use array indexing to select between the two points at the corners of the box. The implementations of these methods select between `pMin` and `pMax` based on the value of `i`.

index

<<Bounds3 Public Methods>>+=

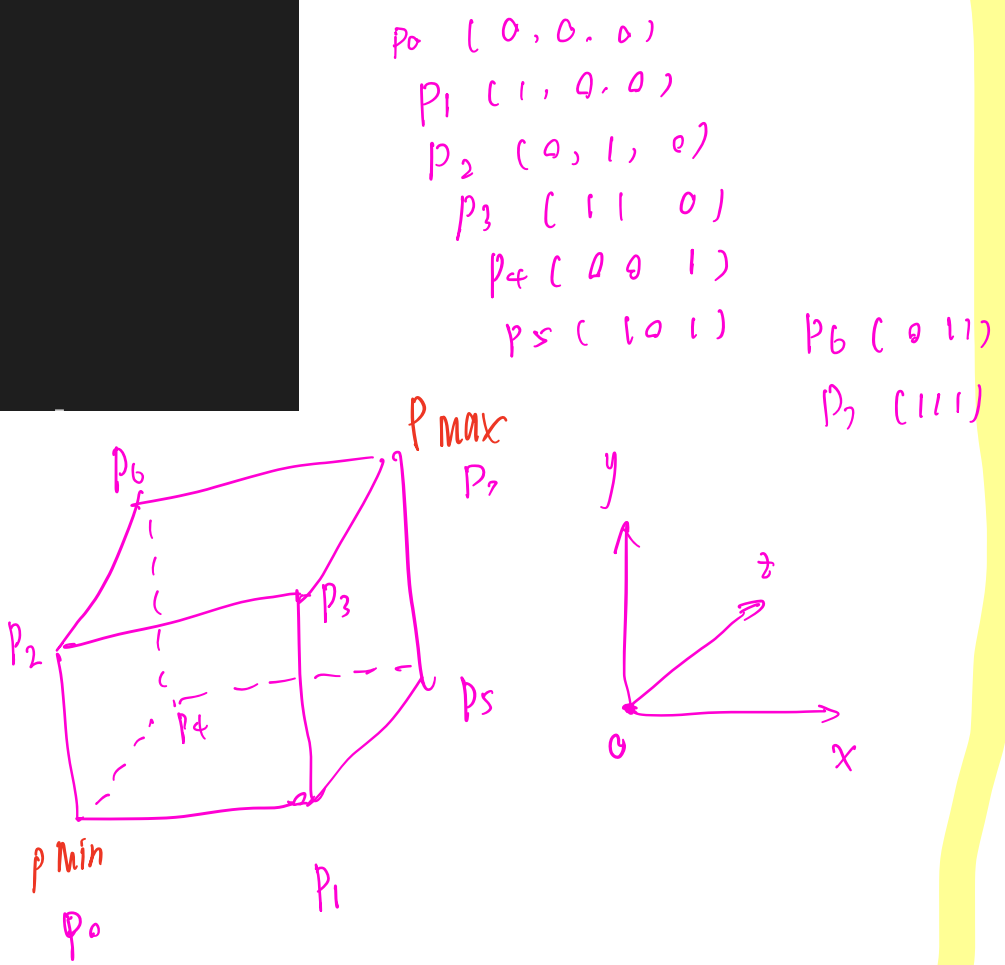
```
const Point3<T> &operator[](int i) const;
Point3<T> &operator[](int i);
```

The `Corner()` method returns the coordinates of one of the eight corners of the bounding box.

<<Bounds3 Public Methods>>+=

```
Point3<T> Corner(int corner) const {
    return Point3<T>((*this)[(corner & 1)].x,
                     (*this)[(corner & 2) ? 1 : 0].y,
                     (*this)[(corner & 4) ? 1 : 0].z);
}
```

```
0&1 = 0
0&2 = 0
0&4 = 0
1&1 = 1
1&2 = 0
1&4 = 0
2&1 = 0
2&2 = 2
2&4 = 0
3&1 = 1
3&2 = 2
3&4 = 0
4&1 = 0
4&2 = 0
4&4 = 4
5&1 = 1
5&2 = 0
5&4 = 4
6&1 = 0
6&2 = 2
6&4 = 4
7&1 = 1
7&2 = 2
7&4 = 4
```



```
}
```

(Merge) (Intersection : check the box)

Given a bounding box and a point, the `Union()` function returns a new bounding box that encompasses that point as well as the original box.

```
<<Geometry Inline Functions>>+=
template <typename T> Bounds3 <T>
Union(const Bounds3<T> &b, const Point3<T> &p) {
    return Bounds3<T>(Point3<T>(std::min(b.pMin.x, p.x),
                                std::min(b.pMin.y, p.y),
                                std::min(b.pMin.z, p.z)),
                      Point3<T>(std::max(b.pMax.x, p.x),
                                std::max(b.pMax.y, p.y),
                                std::max(b.pMax.z, p.z)));
}
```

It is similarly possible to construct a new box that bounds the space encompassed by two other bounding boxes. The definition of this function is similar to the earlier `Union()` method that takes a `Point3f`; the difference is that the `pMin` and `pMax` of the second box are used for the `std::min()` and `std::max()` tests, respectively.

```
<<Geometry Inline Functions>>+=
template <typename T> Bounds3<T>
Union(const Bounds3<T> &b1, const Bounds3<T> &b2) {
    return Bounds3<T>(Point3<T>(std::min(b1.pMin.x, b2.pMin.x),
                                std::min(b1.pMin.y, b2.pMin.y),
                                std::min(b1.pMin.z, b2.pMin.z)),
                      Point3<T>(std::max(b1.pMax.x, b2.pMax.x),
                                std::max(b1.pMax.y, b2.pMax.y),
                                std::max(b1.pMax.z, b2.pMax.z)));
}
```

Transformation

(1) for convenience, combine Inverse inside

```
<<Transform Public Methods>>+=
Transform(const Matrix4x4 &m, const Matrix4x4 &mInv)
   : m(m), mInv(mInv) {
}
```

accordingly, write Transpose and Inverse function like this

```
<<Transform Public Methods>>+=
friend Transform Inverse(const Transform &t) {
    return Transform(t.mInv, t.m);
}
```

```
<<Transform Public Methods>>+=
friend Transform Transpose(const Transform &t) {
    return Transform(Transpose(t.m), Transpose(t.mInv));
}
```

(II). Scalor Trans

It's useful to be able to test if a transformation has a scaling term in it; an easy way to do this is to transform the three coordinate axes and see if any of their lengths are appreciably different from one.

```
<<Transform Public Methods>>+=
bool HasScale() const {
    Float la2 = (*this)(Vector3f(1, 0, 0)).LengthSquared();
    Float lb2 = (*this)(Vector3f(0, 1, 0)).LengthSquared();
    Float lc2 = (*this)(Vector3f(0, 0, 1)).LengthSquared();
#define NOT_ONE(x) ((x) < .999f || (x) > 1.001f)
    return (NOT_ONE(la2) || NOT_ONE(lb2) || NOT_ONE(lc2));
#undef NOT_ONE
}
```

Rotation around Arbitrary Axis

Derivation of this part is wrong

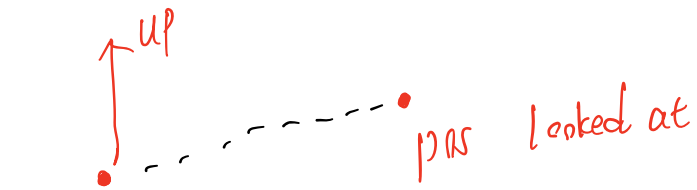
$V' = V_c + V_1 \cos\theta + V_2 \sin\theta$

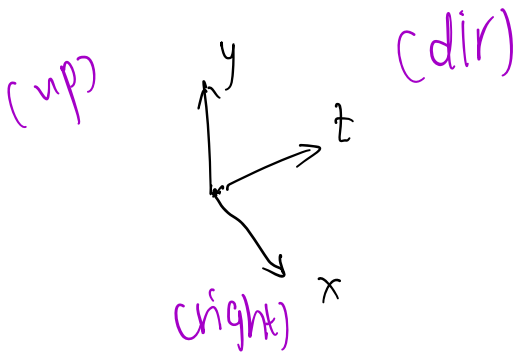
Todo: learn quaternion

look At Transformation

camera space → world space

- known





$$\begin{bmatrix} & M & \end{bmatrix}_{4\times4} \begin{bmatrix} x_{camera} \\ y_{camera} \\ z_{camera} \\ 1 \end{bmatrix}_{4\times1} = \begin{bmatrix} x_{world} \\ y_{world} \\ z_{world} \\ 1 \end{bmatrix}_{4\times1}$$

$$\begin{bmatrix} & ? & \end{bmatrix} \begin{bmatrix} x_{cameraPos} \\ y_{cameraPos} \\ z_{cameraPos} \\ 1 \end{bmatrix} \begin{bmatrix} x_{camera} \\ y_{camera} \\ z_{camera} \\ 1 \end{bmatrix} = \begin{bmatrix} x_{world} \\ y_{world} \\ z_{world} \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \qquad \begin{bmatrix} x_{cameraPos} \\ y_{cameraPos} \\ z_{cameraPos} \end{bmatrix}$$

LookAt() → axis that z [0 0 1 0] should map to

should be normalized

$$\begin{bmatrix} dir.x & x_{camera\,pos} \\ dir.y & y_{camera\,pos} \\ dir.z & z_{camera\,pos} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_{camera} \\ y_{camera} \\ z_{camera} \\ 1 \end{bmatrix} = \begin{bmatrix} x_{world} \\ y_{world} \\ z_{world} \\ 1 \end{bmatrix}$$

4x4 4x1 4x1

$$\begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \quad \begin{bmatrix} dir.x \\ dir.y \\ dir.z \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \quad \begin{bmatrix} newUp.x \\ .y \\ .z \end{bmatrix}$$

$$\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad \begin{bmatrix} right.x \\ right.y \\ right.z \end{bmatrix}$$