Vec3_tpl Structure

C++

```
template <typename F>
struct Vec3_tpl {
  F x, y, z;
  template <typename F>
struct Ang3_tpl {
    F x, y, z;
    template <typename F>
struct AngleAxis_tpl {
      F angle;
      Vec3_tpl<F> axis;
    };
    template <typename F>
struct Plane_tpl {
      Vec3_tpl<F> n;
      F d;
    };
  };
};
```

File

Cry_Vector3.h

Description

class Vec3_tpl

Vec3_tpl::Ang3_tpl Structure

C++

```
template <typename F>
struct Ang3_tpl {
  F x, y, z;
  template <typename F>
struct AngleAxis_tpl {
    F angle;
    Vec3_tpl<F> axis;
  };
  template <typename F>
struct Plane_tpl {
    Vec3_tpl<F> n;
    F d;
  };
};
```

File

Cry_Vector3.h

Description

struct Ang3_tpl

Vec3_tpl::Ang3_tpl::AngleAxis_tpl Structure

C++

```
template <typename F>
struct AngleAxis_tpl {
  F angle;
  Vec3_tpl<F> axis;
};
```

File

Cry_Vector3.h

Description

struct CAngleAxis

Vec3_tpl::Ang3_tpl::AngleAxis_tpl::angle Data Member

C++

```
F angle;
```

Description

! storage for the Angle&Axis coordinates.

Vec3_tpl::Ang3_tpl::AngleAxis_tpl::axis Data Member

C++

```
Vec3_tpl<F> axis;
```

Vec3_tpl::Ang3_tpl::AngleAxis_tpl::() Operator

C++

```
void operator ()(F a, const Vec3_tpl<F> & n);
```

Vec3_tpl::Ang3_tpl::AngleAxis_tpl::* Operator

C++

```
const Vec3_tpl<F> operator *(const Vec3_tpl<F>& v) const;
```

Vec3_tpl::Ang3_tpl::AngleAxis_tpl::AngleAxis_tpl Constructor ()

C++

```
AngleAxis_tpl();
AngleAxis_tpl(F a, F ax, F ay, F az);
AngleAxis_tpl(F a, Vec3_tpl<F> & n);
AngleAxis_tpl(const Quat_tpl<F>& q);
```

Description

default quaternion constructor

Vec3_tpl::Ang3_tpl::AngleAxis_tpl::AngleAxis_tpl Constructor (AngleAxis_tpl<F>&)

```
C++
AngleAxis_tpl(const AngleAxis_tpl<F>& aa);
```

Description

CAngleAxis aa=angleaxis

Vec3_tpl::Ang3_tpl::Plane_tpl Structure

```
C++
template <typename F>
struct Plane_tpl {
  Vec3_tpl<F> n;
  F d;
};
```

File

Cry_Vector3.h

Vec3_tpl::Ang3_tpl::Plane_tpl::d Data Member

```
C++
F d;
```

Description

!< distance

Vec3_tpl::Ang3_tpl::Plane_tpl::n Data Member

```
C++
Vec3_tpl<F> n;
```

Description

!< normal

Vec3_tpl::Ang3_tpl::Plane_tpl::- Operator ()

C++

```
Plane_tpl<F> operator -() const;
```

Vec3_tpl::Ang3_tpl::Plane_tpl::- Operator (Plane_tpl<F> &)

C++

```
Plane_tpl<F> operator -(const Plane_tpl<F> & p) const;
```

Vec3_tpl::Ang3_tpl::Plane_tpl::* Operator

C++

```
Plane_tpl<F> operator *(F s) const;
```

Vec3_tpl::Ang3_tpl::Plane_tpl::/ Operator

C++

```
Plane_tpl<F> operator /(F s) const;
```

Vec3_tpl::Ang3_tpl::Plane_tpl::| Operator

C++

```
F operator |(const Vec3_tpl<F> & point) const;
```

Description

!

- Computes signed distance from point to plane.
- This is the standard plane-equation: d=Ax*By*Cz+D.
- The normal-vector is assumed to be normalized.

*

- Example:
- Vec3 v(1,2,3);
- Plane_tplplane=CalculatePlane(v0,v1,v2);
- f32 distance = plane|v;

Vec3_tpl::Ang3_tpl::Plane_tpl::+ Operator

C++

```
Plane_tpl<F> operator +(const Plane_tpl<F> & p) const;
```

Vec3_tpl::Ang3_tpl::Plane_tpl::-= Operator

C++

```
void operator -=(const Plane_tpl<F> & p);
```

Vec3_tpl::Ang3_tpl::Plane_tpl::CreatePlane Method (Vec3_tpl<F> &, Vec3_tpl<F> &)

C++

```
static Plane_tpl<F> CreatePlane(const Vec3_tpl<F> & normal, const Vec3_tpl<F> & point);
```

Vec3_tpl::Ang3_tpl::Plane_tpl::CreatePlane Method (Vec3_tpl<F> &, Vec3_tpl<F> &, Vec3_tpl<F> &)

```
C++
static Plane_tpl<F> CreatePlane(const Vec3_tpl<F> & v0, const Vec3_tpl<F> & v1, const Vec3
```

Vec3_tpl::Ang3_tpl::Plane_tpl::DistFromPlane Method

```
C++
F DistFromPlane(const Vec3_tpl<F> & vPoint) const;
```

Vec3_tpl::Ang3_tpl::Plane_tpl::Plane_tpl Constructor ()

```
C++
Plane_tpl();
```

Vec3_tpl::Ang3_tpl::Plane_tpl::Plane_tpl Constructor (Plane_tpl<F> &)

```
C++
Plane_tpl(const Plane_tpl<F> & p);
```

Vec3_tpl::Ang3_tpl::Plane_tpl::Plane_tpl Constructor (Vec3_tpl<F> &, F &)

```
C++
Plane_tpl(const Vec3_tpl<F> & normal, const F & distance);
```

Vec3_tpl::Ang3_tpl::Plane_tpl::Set Method

```
C++
void Set(const Vec3_tpl<F> & vNormal, const F fDist);
```

Description

! set normal and dist for this plane and then calculate plane type

Vec3_tpl::Ang3_tpl::Plane_tpl::SetPlane Method (Vec3_tpl<F> &, Vec3_tpl<F> &)

```
C++
void SetPlane(const Vec3_tpl<F> & normal, const Vec3_tpl<F> & point);
```

Vec3_tpl::Ang3_tpl::Plane_tpl::SetPlane Method (Vec3_tpl<F> &, Vec3_tpl<F> &, Vec3_tpl<F> &)

```
C++
void SetPlane(const Vec3_tpl<F> & v0, const Vec3_tpl<F> & v1, const Vec3_tpl<F> & v2);
```

Description

!

- Constructs the plane by tree given Vec3s (=triangle) with a right-hand (anti-clockwise) winding

*

- Example 1:
- Vec3 v0(1,2,3),v1(4,5,6),v2(6,5,6);
- Plane_tplplane;
- plane.SetPlane(v0,v1,v2);

*

- Example 2:
- Vec3 v0(1,2,3),v1(4,5,6),v2(6,5,6);
- Plane_tplplane=Plane_tpl::CreatePlane(v0,v1,v2);

Vec3_tpl::Ang3_tpl::Plane_tpl::Vec3Constants<T>::fVec3_One Method

```
C++
template <typename T> const Vec3_tpl<T> Vec3Constants<T>::fVec3_One(1, 1, 1);
```

Vec3_tpl::Ang3_tpl::Plane_tpl::Vec3Constants<T>::fVec3_OneX Method

```
C++
template <typename T> const Vec3_tpl<T> Vec3Constants<T>::fVec3_OneX(1, 0, 0);
```

Vec3_tpl::Ang3_tpl::Plane_tpl::Vec3Constants<T>::fVec3_OneY Method

```
C++
template <typename T> const Vec3_tpl<T> Vec3Constants<T>::fVec3_OneY(0, 1, 0);
```

Vec3_tpl::Ang3_tpl::Plane_tpl::Vec3Constants<T>::fVec3_OneZ Method

```
C++
template <typename T> const Vec3_tpl<T> Vec3Constants<T>::fVec3_OneZ(0, 0, 1);
```

Vec3_tpl::Ang3_tpl::Plane_tpl::Vec3Constants<T>::fVec3_Zero Method

```
C++
template <typename T> const Vec3_tpl<T> Vec3Constants<T>::fVec3_Zero(0, 0, 0);
```

Description

define the constants

friend bool operator ==(const Plane_tpl<F> &p1, const Plane_tpl<F> &p2) { if (fabsf(p1.n.x-p2.n.x)>0.0001f) return (false); if (fabsf(p1.n.y-p2.n.y)>0.0001f)

return (false); if (fabsf(p1.n.z-p2.n.z)>0.0001f) return (false); if (fabsf(p1.d-p2.d) <0.01f) return(true); return (false); } Vec3_tpl<F> MirrorVector(const Vec3_tpl<F>& i) { return n*(2* (n|i))-i; } Vec3_tpl<F> MirrorPosition(const Vec3_tpl<F>& i) { return i - n*(2* ((n|i)+d)); } AUTO_STRUCT_INFO } Friend

```
C++
friend bool operator ==(const Plane_tpl<F> &p1, const Plane_tpl<F> &p2) { if (fabsf(p1.n.x
```

Description

! check for equality between two planes

Vec3_tpl::Ang3_tpl::Plane_tpl::Plane Nested Type

```
C++
typedef Plane_tpl<f32> Plane;
```

Description

always 32 bit

Vec3_tpl::Ang3_tpl::Plane_tpl::Planed Nested Type

```
C++
typedef Plane_tpl<f64> Planed;
```

Description

always 64 bit

Vec3_tpl::Ang3_tpl::Plane_tpl::Planer Nested Type

```
C++
typedef Plane_tpl<real> Planer;
```

Description

variable float precision. depending on the target system it can be between 32, 64 or 80 bit

Vec3_tpl::Ang3_tpl::x Data Member

```
C++
F x;
```

Vec3_tpl::Ang3_tpl::y Data Member

```
C++
```

```
F y;
```

Vec3_tpl::Ang3_tpl::z Data Member

C++

```
F z;
```

Vec3_tpl::Ang3_tpl::- Operator ()

C++

```
Ang3_tpl<F> operator -() const;
```

Description

explicit ILINE Ang3_tpl& operator = (const Vec3_tpl& v) { x=v.x; y=v.y; z=v.z; return *this;

Vec3_tpl::Ang3_tpl::- Operator (Ang3_tpl<F1> &, Ang3_tpl<F2> &)

C++

```
template <class F1,class F2> Ang3_tpl<F1> operator -(const Ang3_tpl<F1> & v0, const Ang3_t
```

Description

vector subtraction

Vec3_tpl::Ang3_tpl::!= Operator

C++

```
bool operator !=(const Ang3_tpl<F> & vec);
```

Vec3_tpl::Ang3_tpl::() Operator

C++

```
void operator ()(F vx, F vy, F vz);
```

Vec3_tpl::Ang3_tpl::* Operator

C++

```
Ang3_tpl<F> operator *(F k) const;
```

Vec3_tpl::Ang3_tpl::*= Operator

C++

```
Ang3_tpl<F>& operator *=(F k);
```

Vec3_tpl::Ang3_tpl::/ Operator

```
C++
Ang3_tpl<F> operator /(F k) const;
```

Vec3_tpl::Ang3_tpl::+ Operator

```
C++
template <class F1,class F2> Ang3_tpl<F1> operator +(const Ang3_tpl<F1> & v0, const Ang3_t
```

Description

vector addition

Vec3_tpl::Ang3_tpl::+= Operator

```
C++
template <class F1,class F2> Ang3_tpl<F1>& operator +=(Ang3_tpl<F1> & v0, const Ang3_tpl<F
```

Description

vector self-addition

Vec3_tpl::Ang3_tpl::-= Operator

```
C++
template <class F1,class F2> Ang3_tpl<F1>& operator -=(Ang3_tpl<F1> & v0, const Ang3_tpl<F
```

Description

vector self-subtraction

Vec3_tpl::Ang3_tpl::== Operator

```
C++
bool operator ==(const Ang3_tpl<F> & vec);
```

Vec3_tpl::Ang3_tpl::Ang3_tpl Constructor ()

```
C++
Ang3_tpl();
```

Vec3_tpl::Ang3_tpl::Ang3_tpl Constructor (F, F, F)

```
C++
Ang3_tpl<F>(F vx, F vy, F vz);
```

Vec3_tpl::Ang3_tpl::Ang3_tpl Constructor (Vec3_tpl<F>&)

```
C++
explicit Ang3_tpl(const Vec3_tpl<F>& v);
```

Vec3_tpl::Ang3_tpl::Ang3_tpl Constructor (type_zero)

```
C++
Ang3_tpl(type_zero);
```

Vec3_tpl::Ang3_tpl::AngleAxis_tpl<F>::* Method

```
C++
template <typename F> const Vec3_tpl<F> AngleAxis_tpl<F>::*(const Vec3_tpl<F> & v) const;
```

friend bool operator ==(const Ang3_tpl<F> &v0, const Ang3_tpl<F> &v1) { return ((v0.x==v1.x) && (v0.y==v1.y) && (v0.z==v1.z)); } void Set(F xval,F yval, F zval) { x=xval; y=yval; z=zval; } bool IsEquivalent(const Ang3_tpl<F>& v1, F epsilon=VEC_EPSILON) const { return ((fabs_tpl(x-v1.x) <= epsilon) && (fabs_tpl(y-v1.y) <= epsilon)&& (fabs_tpl(z-v1.z) <= epsilon)); } bool IsInRangePI() const { F pi=(F)(gf_PI+0.001); return ((x>-pi)&&(x<pi) && (y>-pi) &&(y<pi) && (z>-pi)&&(z<pi)); } void RangePI() { const F modX = fmod (x+gf_PI, gf_PI2); x = if_neg_else(modX, modX+gf_PI, modX - gf_PI); const F modY = fmod(y+gf_PI, gf_PI2); y = if_neg_else(modY, modY+gf_PI, modY - gf_PI); const F modZ = fmod(z+gf_PI, gf_PI2); z = if_neg_else(modZ, modZ+gf_PI, modZ - gf_PI); } template<class F1> explicit Ang3_tpl(const Quat_tpl<F1>& q) { assert(q.IsValid()); y = F(asin_tpl(max((F)-1.0,min((F)1.0,-(q.v.x*q.v.z-q.w*q.v.y)*2)))); if (fabs_tpl(fabs_tpl(y)-(F)((F)g_PI*(F)0.5))<(F) 0.01) { x = F(0); z = F(atan2_tpl(-2*(q.v.x*q.v.y-q.w*q.v.z),1-(q.v.x*q.v.x+q.v.z*q. v.z)*2)); } else { x = F(atan2_tpl((q.v.y*q.v.z+q.w*q.v.x)*2, 1-(q.v.x*q.v.x+q.v. y*q.v.y)*2)); z = F(atan2_tpl((q.v.x*q.v.y+q.w*q.v.z)*2, 1-(q.v.z*q.v.z+q.v.y*q.v. y)*2)); } } template<class F1> explicit Ang3_tpl(const Matrix33_tpl<F1>& m) { assert(m.IsOrthonormalRH(0.001f)); y = (F)asin_tpl(max((F)-1.0,min((F)1.0,-m.m20))); if (fabs_tpl(fabs_tpl(y)-(F)((F)g_PI*(F)0.5))<(F)0.01) { x = F(0); z = F (atan2_tpl(-m.m01,m.m11)); } else { x = F(atan2_tpl(m.m21, m.m22)); z = F (atan2_tpl(m.m10, m.m00)); } } template<class F1> explicit Ang3_tpl(const Matrix34_tpl<F1>& m) { assert(m.IsOrthonormalRH(0.001f)); y = F(asin_tpl (max((F)-1.0,min((F)1.0,-m.m20)))); if (fabs_tpl(fabs_tpl(y)-(F)((F)g_PI*(F)0.5)) <(F)0.01) { x = F(0); z = F(atan2_tpl(-m.m01,m.m11)); } else { x = F(atan2_tpl (m.m21, m.m22)); z = F(atan2_tpl(m.m10, m.m00)); } } template<class F1> explicit Ang3_tpl(const Matrix44_tpl<F1>& m) { assert(Matrix33(m). IsOrthonormalRH(0.001f)); y = F(asin_tpl(max((F)-1.0,min((F)1.0,-m.m20)))); if (fabs_tpl(fabs_tpl(y)-(F)((F)g_PI*(F)0.5))<(F)0.01) { x = F(0); z = F(atan2_tpl(-m.m01,m.m11)); } else { x = F(atan2_tpl(m.m21, m.m22)); z = F(atan2_tpl(m. m10, m.m00)); } } template<typename F1> static F CreateRadZ(const

Vec2_tpl<F1>& v0, const Vec2_tpl<F1>& v1) { F cz = v0.x*v1.y-v0.y*v1.x; F c = v0.x*v1.x+v0.y*v1.y; return F(atan2_tpl(cz,c)); } template<typename F1> static F CreateRadZ(const Vec3_tpl<F1>& v0, const Vec3_tpl<F1>& v1) { F cz = v0.x*v1.y-v0.y*v1.x; F c = v0.x*v1.x+v0.y*v1.y; return F(atan2_tpl(cz,c)); } template<typename F1> static Ang3_tpl<F> GetAnglesXYZ(const Quat_tpl<F1>& q) { return Ang3_tpl<F>(q); } template<typename F1> void SetAnglesXYZ(const Quat_tpl<F1>& q) { *this=Ang3_tpl<F>(q); } template<typename F1> static Ang3_tpl<F> GetAnglesXYZ(const Matrix33_tpl<F1>& m) { return Ang3_tpl<F>(m); } template<typename F1> void SetAnglesXYZ(const Matrix33_tpl<F1>& m) { *this=Ang3_tpl<F>(m); } template<typename F1> static Ang3_tpl<F> GetAnglesXYZ(const Matrix34_tpl<F1>& m) { return Ang3_tpl<F>(m); } template<typename F1> void SetAnglesXYZ(const Matrix34_tpl<F1>& m) { *this=Ang3_tpl<F>(m); } F &operator [] (int index) { assert(index>=0 && index<=2); return ((F*)this)[index]; } F operator [] (int index) const { assert(index>=0 && index<=2); return ((F*)this)[index]; } bool IsValid() const { if (!NumberValid(x)) return false; if (!NumberValid(y)) return false; if (!NumberValid(z)) return false; return true; } AUTO_STRUCT_INFO } Friend

```
C++
friend bool operator ==(const Ang3_tpl<F> &v0, const Ang3_tpl<F> &v1) { return ((v0.x==v1.
```

Vec3_tpl::Ang3_tpl::Ang3 Nested Type

```
C++
typedef Ang3_tpl<f32> Ang3;
```

Vec3_tpl::Ang3_tpl::Ang3_f64 Nested Type

```
C++
typedef Ang3_tpl<f64> Ang3_f64;
```

Vec3_tpl::Ang3_tpl::Ang3r Nested Type

```
C++
typedef Ang3_tpl<real> Ang3r;
```

Vec3_tpl::Ang3_tpl::AngleAxis Nested Type

```
C++
typedef AngleAxis_tpl<f32> AngleAxis;
```

Vec3_tpl::Ang3_tpl::AngleAxis_f64 Nested Type

```
C++
typedef AngleAxis_tpl<f64> AngleAxis_f64;
```

Vec3_tpl::x Data Member

```
C++
F x;
```

Vec3_tpl::y Data Member

```
C++
F y;
```

Vec3_tpl::z Data Member

```
C++
F z;
```

Vec3_tpl::- Operator (Vec2_tpl<F1> &, Vec3_tpl<F2> &)

```
C++
template <class F1,class F2> Vec3_tpl<F1> operator -(const Vec2_tpl<F1> & v0, const Vec3_t
```

Vec3_tpl::- Operator (Vec3_tpl<F1> &, Vec2_tpl<F2> &)

```
C++
template <class F1,class F2> Vec3_tpl<F1> operator -(const Vec3_tpl<F1> & v0, const Vec2_t
```

Vec3_tpl::- Operator (Vec3_tpl<F1> &, Vec3_tpl<F2> &)

```
C++
template <class F1,class F2> Vec3_tpl<F1> operator -(const Vec3_tpl<F1> & v0, const Vec3_t
```

Description

vector subtraction

Vec3_tpl::% Operator

```
C++
template <class F1,class F2> Vec3_tpl<F1> operator %(const Vec3_tpl<F1> & v0, const Vec3_t
```

Vec3_tpl::() Operator

```
C++
void operator ()(F vx, F vy, F vz);
```

Vec3_tpl::* Operator (F)

```
C++
Vec3_tpl<F> operator *(F k) const;
```

Description

!

- overloaded arithmetic operator

*

- Example:
- Vec3 v0=v1*4;

Vec3_tpl::* Operator (Vec3_tpl<F1> &, Vec3_tpl<F2> &)

```
C++
template <class F1,class F2> F1 operator *(const Vec3_tpl<F1> & v0, const Vec3_tpl<F2> & v
```

Description

dot product (2 versions)

Vec3_tpl::/ Operator (F)

```
C++
Vec3_tpl<F> operator /(F k) const;
```

Vec3_tpl::/ Operator (Vec3_tpl<F1> &, Vec3_tpl<F2> &)

```
C++
template <class F1,class F2> Vec3_tpl<F1> operator /(const Vec3_tpl<F1> & v0, const Vec3_t
```

Vec3_tpl::^ Operator

```
C++
template <class F1,class F2> Vec3_tpl<F1> operator ^(const Vec3_tpl<F1> & v0, const Vec3_t
```

Description

cross product (2 versions)

Vec3_tpl::| Operator

```
C++
template <class F1,class F2> F1 operator |(const Vec3_tpl<F1> & v0, const Vec3_tpl<F2> & v
```

Vec3_tpl::+ Operator (Vec2_tpl<F1> &, Vec3_tpl<F2> &)

```
C++
```

```
template <class F1,class F2> Vec3_tpl<F1> operator +(const Vec2_tpl<F1> & v0, const Vec3_t
```

Description

vector addition

Vec3_tpl::+ Operator (Vec3_tpl<F1> &, Vec2_tpl<F2> &)

C++

```
template <class F1,class F2> Vec3_tpl<F1> operator +(const Vec3_tpl<F1> & v0, const Vec2_t
```

Description

vector addition

Vec3_tpl::+ Operator (Vec3_tpl<F1> &, Vec3_tpl<F2> &)

C++

```
template <class F1,class F2> Vec3_tpl<F1> operator +(const Vec3_tpl<F1> & v0, const Vec3_t
```

Description

vector addition

Vec3_tpl::+= Operator

C++

```
template <class F1,class F2> Vec3_tpl<F1>& operator +=(Vec3_tpl<F1> & v0, const Vec3_tpl<F
```

Description

vector self-addition

Vec3_tpl::-= Operator

C++

```
template <class F1,class F2> Vec3_tpl<F1>& operator -=(Vec3_tpl<F1> & v0, const Vec3_tpl<F
```

Description

vector self-subtraction

Vec3_tpl::BiRandom Method

C++

```
Vec3 BiRandom(const Vec3& vRange);
```

Vec3_tpl::IsEquivalent Method

```
C++
template <class F> bool IsEquivalent(const Vec3_tpl<F> & v0, const Vec3_tpl<F> & v1, f32 e
```

Vec3_tpl::Random Method (Vec3&)

```
C++
Vec3 Random(const Vec3& v);
```

Description

Random vector functions.

Vec3_tpl::Random Method (Vec3&, Vec3&)

```
C++
Vec3 Random(const Vec3& a, const Vec3& b);
```

Vec3_tpl::Set Method

```
C++
Vec3_tpl<F>& Set(const F xval, const F yval, const F zval);
```

Vec3_tpl::SphereRandom Method

```
C++
Vec3 SphereRandom(float fRadius);
```

Description

Random point in sphere.

Vec3_tpl::Vec3_tpl Constructor ()

```
C++
Vec3_tpl();
```

Vec3_tpl::Vec3_tpl Constructor (Ang3_tpl<F>&)

```
C++
explicit Vec3_tpl<F>(const Ang3_tpl<F>& v);
```

Vec3_tpl::Vec3_tpl Constructor (Ang3_tpl<T>&)

```
C++
```

```
template <class T>  explicit Vec3_tpl<F>(const Ang3_tpl<T>& v);
```

Vec3_tpl::Vec3_tpl Constructor (F)

```
C++
explicit Vec3_tpl(F f);
```

Vec3_tpl::Vec3_tpl Constructor (F, F, F)

```
C++
Vec3_tpl(F vx, F vy, F vz);
```

Description

!

- constructors and bracket-operator to initialize a vector

*

- Example:
- Vec3 v0=Vec3(1,2,3);
- Vec3 v1(1,2,3);
- v2.Set(1,2,3);

Vec3_tpl::Vec3_tpl Constructor (Vec2_tpl<F>&)

```
C++
Vec3_tpl<F>(const Vec2_tpl<F>& v);
```

Vec3_tpl::Vec3_tpl Constructor (Vec2_tpl<T>&)

```
C++
template <class T> Vec3_tpl<F>(const Vec2_tpl<T>& v);
```

Vec3_tpl::Vec3_tpl Constructor (Vec3_tpl&)

```
C++
Vec3_tpl(const Vec3_tpl& v);
```

Description

!

- the copy/casting/assignement constructor

*

- Example:
- Vec3 v0=v1;
- Vec3 v0=Vec3(angle);
- Vec3 v0=Vec3(vector4);

Vec3_tpl::Vec3_tpl Constructor (Vec3_tpl<F1>&)

C++

```
template <class F1> Vec3_tpl<F>(const Vec3_tpl<F1>& v);
```

Vec3_tpl::Vec3_tpl Constructor (Vec4_tpl<F> &)

C++

```
explicit Vec3_tpl<F>(const Vec4_tpl<F> & v);
```

Vec3_tpl::Vec3_tpl Constructor (Vec4_tpl<T> &)

C++

```
template <class T>  explicit Vec3_tpl<F>(const Vec4_tpl<T> & v);
```

Vec3_tpl::Vec3_tpl Constructor (type_max)

C++

```
Vec3_tpl(type_max);
```

Vec3_tpl::Vec3_tpl Constructor (type_min)

C++

```
Vec3_tpl(type_min);
```

Vec3_tpl::Vec3_tpl Constructor (type_zero)

C++

```
Vec3_tpl(type_zero);
```

Description

!

- template specialization to initialize a vector

*

- Example:
- Vec3 v0=Vec3(ZERO);
- Vec3 v1=Vec3(MIN);
- Vec3 v2=Vec3(MAX);

Vec3_tpl::Vec3_tpl<f32>::Vec3_tpl Method (type_max)

C++

```
template <> inline Vec3_tpl<f32>::Vec3_tpl(type_max);
```

Vec3_tpl::Vec3_tpl<f32>::Vec3_tpl Method (type_min)

C++

```
template <> inline Vec3_tpl<f32>::Vec3_tpl(type_min);
```

Vec3_tpl::Vec3_tpl<f64>::Vec3_tpl Method (type_max)

```
C++
template <> inline Vec3_tpl<f64>::Vec3_tpl(type_max);
```

Vec3_tpl::Vec3_tpl<f64>::Vec3_tpl Method (type_min)

```
C++
template <> inline Vec3_tpl<f64>::Vec3_tpl(type_min);
```

friend Vec3_tpl<F> operator * (f32 f, const Vec3_tpl &vec) { return Vec3_tpl((F)(f*vec.x), (F)(f*vec.y), (F)(f*vec.z)); } Vec3_tpl<F>& operator *= (F k) { x*=k; y*=k;z*=k; return *this; } Vec3_tpl<F>& operator /= (F k) { k=(F)1.0/k; x*=k;y*=k; z*=k; return *this; } Vec3_tpl<F> operator - (void) const { return Vec3_tpl<F>(-x,-y,-z); } Vec3_tpl<F>& Flip() { x=-x; y=-y; z=-z; return *this; } F &operator [] (int32 index) { assert(index>=0 && index<=2); return ((F*)this)[index]; } F operator [] (int32 index) const { assert(index>=0 && index<=2); return ((F*)this)[index]; } bool operator==(const Vec3_tpl<F> &vec) { return x == vec.x && y == vec.y && z == vec.z; } bool operator!=(const Vec3_tpl<F> &vec) { return !(*this == vec); } friend bool operator ==(const Vec3_tpl<F> &v0, const Vec3_tpl<F> &v1) { return ((v0.x==v1.x) && (v0.y==v1.y) && (v0.z==v1.z)); } friend bool operator !=(const Vec3_tpl<F> &v0, const Vec3_tpl<F> &v1) { return !(v0==v1); } bool IsZero(F e = (F)0.0) const { return (fabs_tpl(x) <= e) && (fabs_tpl(y) <= e) && (fabs_tpl(z) <= e); } bool IsZeroFast(F e = (F)0.0003) const { return (fabs_tpl(x) + fabs_tpl(y) + fabs_tpl(z)) <= e; } bool IsEquivalent (const Vec3_tpl<F> &v1, F epsilon=VEC_EPSILON) const { assert(v1.IsValid()); assert(this->IsValid()); return ((fabs_tpl(x-v1.x) <= epsilon) && (fabs_tpl(y-v1.y) <= epsilon)&& (fabs_tpl(z-v1.z) <= epsilon)); } static bool IsEquivalent(const Vec3_tpl<F>& v0, const Vec3_tpl<F>& v1, F epsilon=VEC_EPSILON) { assert(v0.IsValid()); assert(v1.IsValid()); return ((fabs_tpl(v0.x-v1.x) <= epsilon) && (fabs_tpl(v0.y-v1.y) <= epsilon)&& (fabs_tpl(v0.z-v1.z) <= epsilon)); } bool IsUnit(F epsilon=VEC_EPSILON) const { return (fabs_tpl(1 - GetLengthSquared()) <= epsilon); } bool IsValid() const { if (!NumberValid(x)) return false; if (!NumberValid(y)) return false; if (!NumberValid(z)) return false; return true; } void SetLength(F fLen) { F fLenMe = GetLengthSquared(); if(fLenMe<0.00001f*0.00001f) return; fLenMe = fLen * isqrt_tpl(fLenMe); x*=fLenMe; y*=fLenMe; z*=fLenMe; } void ClampLength(F maxLength) { F sqrLength = GetLengthSquared(); if (sqrLength > (maxLength * maxLength)) { F scale = maxLength * isqrt_tpl(sqrLength); x *= scale; y *= scale; z *= scale; } } F GetLength() const { return sqrt_tpl(x*x+y*y+z*z); } F GetLengthFloat() const { return GetLength(); } F GetLengthFast() const { return

sqrt_fast_tpl(x*x+y*y+z*z); } F GetLengthSquared() const { return x*x+y*y+z*z; } F GetLengthSquaredFloat() const { return GetLengthSquared(); } F GetLength2D() const { return sqrt_tpl(x*x+y*y); } F GetLengthSquared2D() const { return x*x+y*y; } F GetDistance(const Vec3_tpl<F> &vec1) const { return sqrt_tpl((x-vec1.x)*(x-vec1.x)+(y-vec1.y)*(y-vec1.y)+(z-vec1.z)*(z-vec1.z)); } F GetSquaredDistance (const Vec3_tpl<F> &v) const { return (x-v.x)*(x-v.x) + (y-v.y)*(y-v.y) + (z-v.z)*(z-v.z); } F GetSquaredDistance2D (const Vec3_tpl<F> &v) const { return (x-v.x)*(x-v.x) + (y-v.y)*(y-v.y); } void Normalize() { assert(this->IsValid()); F fInvLen = isqrt_safe_tpl(x*x+y*y+z*z); x*=fInvLen; y*=fInvLen; z*=fInvLen; } void NormalizeFast() { assert(this->IsValid()); F fInvLen = isqrt_fast_tpl(x*x+y*y+z*z); x*=fInvLen; y*=fInvLen; z*=fInvLen; } F NormalizeSafe(const struct Vec3_tpl<F>& safe = Vec3Constants<F>::fVec3_Zero) { assert(this->IsValid()); F fLen2 = x*x+y*y+z*z; IF (VecPrecisionValues<F>::CheckGreater(fLen2), 1) { F fInvLen = isqrt_tpl(fLen2); x*=fInvLen; y*=fInvLen; z*=fInvLen; return F(1) / fInvLen; } else { *this = safe; return F(0); } } Vec3_tpl GetNormalizedFloat() const { return GetNormalized(); } Vec3_tpl GetNormalized() const { F fInvLen = isqrt_safe_tpl (x*x+y*y+z*z); return *this * fInvLen; } Vec3_tpl GetNormalizedFast() const { F fInvLen = isqrt_fast_tpl(x*x+y*y+z*z); return *this * fInvLen; } Vec3_tpl GetNormalizedSafe(const struct Vec3_tpl<F>& safe = Vec3Constants<F>::fVec3_OneX) const { F fLen2 = x*x+y*y+z*z; IF (VecPrecisionValues<F>::CheckGreater(fLen2), 1) { F fInvLen = isqrt_tpl(fLen2); return *this * fInvLen; } else { return safe; } } Vec3_tpl GetNormalizedSafeFloat(const struct Vec3_tpl<F>& safe = Vec3Constants<F>::fVec3_OneX) const { return GetNormalizedSafe(safe); } Vec3_tpl GetPermutated(int new_z) const { return Vec3_tpl(*(&x+inc_mod3[new_z]), *(&x+dec_mod3[new_z]), *(&x+new_z)); } F GetVolume() const { return x*y*z; } Vec3_tpl<F> abs() const { return Vec3_tpl(fabs_tpl(x),fabs_tpl(y),fabs_tpl(z)); } void CheckMin(const Vec3_tpl<F> other) { x = min(other.x,x); y = min(other.y,y); z = min(other.z,z); } void CheckMax (const Vec3_tpl<F> other) { x = max(other.x,x); y = max(other.y,y); z = max (other.z,z); } void SetOrthogonal(const Vec3_tpl<F>& v) { sqr(F(0.9))*(v|v)-v.x*v.x<0 ? (x=-v.z,y=0,z=v.x) : (x=0,y=v.z,z=-v.y); } Vec3_tpl GetOrthogonal() const { return sqr(F(0.9))*(x*x+y*y+z*z)-x*x<0 ? Vec3_tpl<F>(-z,0,x) : Vec3_tpl<F>(0,z,-y); } void SetProjection(const Vec3_tpl& i, const Vec3_tpl& n) { *this = i-n*(n|i); } static Vec3_tpl<F> CreateProjection(const Vec3_tpl& i, const Vec3_tpl& n) { return i-n*(n|i); } void SetReflection(const Vec3_tpl<F>& i, const Vec3_tpl<F>& n) { *this=(n*(i|n)*2)-i; } static Vec3_tpl<F> CreateReflection(const Vec3_tpl<F>& i, const Vec3_tpl<F>& n) { return (n*(i|n)*2)-i; } void SetLerp(const Vec3_tpl<F>& p, const Vec3_tpl<F>& q, F t) { const Vec3_tpl<F> diff = q-p; *this = p + (diff*t); } static Vec3_tpl<F> CreateLerp(const Vec3_tpl<F>& p, const Vec3_tpl<F>& q, F t) { const Vec3_tpl<F> diff = q-p; return p+(diff*t); } void SetSlerp(const Vec3_tpl<F>& p,

```
const Vec3_tpl<F>& q, F t ) { assert(p.IsUnit(0.005f)); assert(q.IsUnit(0.005f)); F cosine = clamp_tpl((p|q), F(-1), F(1)); if(cosine>=(F)0.99) { SetLerp(p,q,t); Normalize(); } else { F rad = acos_tpl(cosine); F scale_0 = sin_tpl((1-t)*rad); F scale_1 = sin_tpl(t*rad); *this=(p*scale_0 + q*scale_1) / sin_tpl(rad); Normalize(); } } static Vec3_tpl<F> CreateSlerp( const Vec3_tpl<F>& p, const Vec3_tpl<F>& q, F t ) { Vec3_tpl<F> v; v.SetSlerp(p,q,t); return v; } void SetQuadraticCurve(const Vec3_tpl<F>& v0, const Vec3_tpl<F>& v1, const Vec3_tpl<F>& v2, F t1) { F t0=1.0f-t1; *this = t0*t0*v0 + t0*t1*2.0f*v1 + t1*t1*v2; } static Vec3_tpl<F> CreateQuadraticCurve(const Vec3_tpl<F>& v0, const Vec3_tpl<F>& v1, const Vec3_tpl<F>& v2, F t) { Vec3_tpl<F> ip; ip.SetQuadraticCurve(v0,v1,v2,t); return ip; } void SetCubicCurve(const Vec3_tpl<F>& v0, const Vec3_tpl<F>& v1, const Vec3_tpl<F>& v2, const Vec3_tpl<F>& v3, F t1) { F t0=1.0f-t1; *this=t0*t0*t0*v0 + 3*t0*t0*t1*v1 + 3*t0*t1*t1*v2 + t1*t1*t1*v3; } static Vec3_tpl<F> CreateCubicCurve(const Vec3_tpl<F>& v0, const Vec3_tpl<F>& v1, const Vec3_tpl<F>& v2, const Vec3_tpl<F>& v3, F t) { Vec3_tpl<F> ip; ip.SetCubicCurve(v0,v1,v2,v3,t); return ip; } void SetQuadraticSpline(const Vec3_tpl<F>& v0, const Vec3_tpl<F>& v1, const Vec3_tpl<F>& v2, F t) { SetQuadraticCurve(v0,v1-(v0*0.5f+v1+v2*0.5f-v1*2.0f),v2,t); } static Vec3_tpl<F> CreateQuadraticSpline(const Vec3_tpl<F>& v0, const Vec3_tpl<F>& v1, const Vec3_tpl<F>& v2, F t) { Vec3_tpl<F> ip; ip.SetQuadraticSpline(v0,v1,v2,t); return ip; } void SetRandomDirection( void ) { int nMax = 5; F Length2; do { x = 1.0f - 2.0 f*cry_frand(); y = 1.0f - 2.0f*cry_frand(); z = 1.0f - 2.0f*cry_frand(); Length2 = len2(); nMax--; } while((Length2>1.0f || Length2<0.0001f) && nMax > 0); F InvScale = isqrt_tpl(Length2); x *= InvScale; y *= InvScale; z *= InvScale; } Vec3_tpl<F> GetRotated(const Vec3_tpl<F>& axis, F angle) const { return GetRotated(axis,cos_tpl(angle),sin_tpl(angle)); } Vec3_tpl<F> GetRotated (const Vec3_tpl<F>& axis, F cosa,F sina) const { Vec3_tpl<F> zax = axis* (*this|axis); Vec3_tpl<F> xax = *this-zax; Vec3_tpl<F> yax = axis%xax; return xax*cosa + yax*sina + zax; } Vec3_tpl<F> GetRotated(const Vec3_tpl& center, const Vec3_tpl<F>& axis, F angle) const { return center+(*this-center). GetRotated(axis,angle); } Vec3_tpl<F> GetRotated(const Vec3_tpl<F>& center, const Vec3_tpl<F>& axis, F cosa,F sina) const { return center+(*this-center). GetRotated(axis,cosa,sina); } Vec3_tpl CompMul( const Vec3_tpl<F> rhs ) const { return( Vec3_tpl( x * rhs.x, y * rhs.y, z * rhs.z ) ); } F Dot (const Vec3_tpl<F> v) const { return x*v.x + y*v.y + z*v.z; } Vec3_tpl<F> Cross (const Vec3_tpl<F> vec2) const { return Vec3_tpl<F>( y*vec2.z - z*vec2.y, z*vec2.x - x*vec2.z, x*vec2.y - y*vec2.x); } DEPRICATED operator F* () { return (F*)this; } template <class T> explicit DEPRICATED Vec3_tpl(const T *src) { x=src[0]; y=src[1]; z=src[2]; } Vec3_tpl& zero() { x=y=z=0; return *this; } F len() const { return sqrt_tpl(x*x+y*y+z*z); } F len2() const { return x*x +y*y + z*z; } Vec3_tpl& normalize() { F len2 = x*x+y*y+z*z; if (len2>(F)1e-20f) { F rlen =
```

isqrt_tpl(len2); x*=rlen; y*=rlen; z*=rlen; } else Set(0,0,1); return *this; } Vec3_tpl normalized() const { F len2 = x*x+y*y+z*z; if (len2>(F)1e-20f) { F rlen = isqrt_tpl(len2); return Vec3_tpl(x*rlen,y*rlen,z*rlen); } else return Vec3_tpl (0,0,1); } template<class F1> Vec3_tpl<F1> sub(const Vec3_tpl<F1>& v) const { return Vec3_tpl<F1>(x-v.x, y-v.y, z-v.z); } template<class F1> Vec3_tpl<F1> scale(const F1 k) const { return Vec3_tpl<F>(x*k,y*k,z*k); } template<class F1> F1 dot(const Vec3_tpl<F1>& v) const { return (F1)(x*v.x+y*v.y+z*v.z); } template<class F1> Vec3_tpl<F1> cross(const Vec3_tpl<F1> &v) const { return Vec3_tpl<F1>(y*v.z-z*v.y, z*v.x-x*v.z, x*v.y-y*v.x); } AUTO_STRUCT_INFO } Friend

```
C++
friend Vec3_tpl<F> operator * (f32 f, const Vec3_tpl &vec) { return Vec3_tpl((F)(f*vec.x),
```

Vec3_tpl::value_type Nested Type

```
C++
typedef F value_type;
```

Vec3_tpl::Vec3 Nested Type

```
C++
typedef Vec3_tpl<f32> Vec3;
```

Description

always 32 bit

Vec3_tpl::Vec3d Nested Type

```
C++
typedef Vec3_tpl<f64> Vec3d;
```

Description

always 64 bit

Vec3_tpl::Vec3i Nested Type

```
C++
typedef Vec3_tpl<int> Vec3i;
```

Vec3_tpl::Vec3r Nested Type

```
C++
typedef Vec3_tpl<real> Vec3r;
```

Description

variable float precision. depending on the target system it can be 32, 64 or 80 bit

In This Topic