Math and Container Libraries in the C4 Game Engine Architecture

Eric Lengyel, Ph.D.

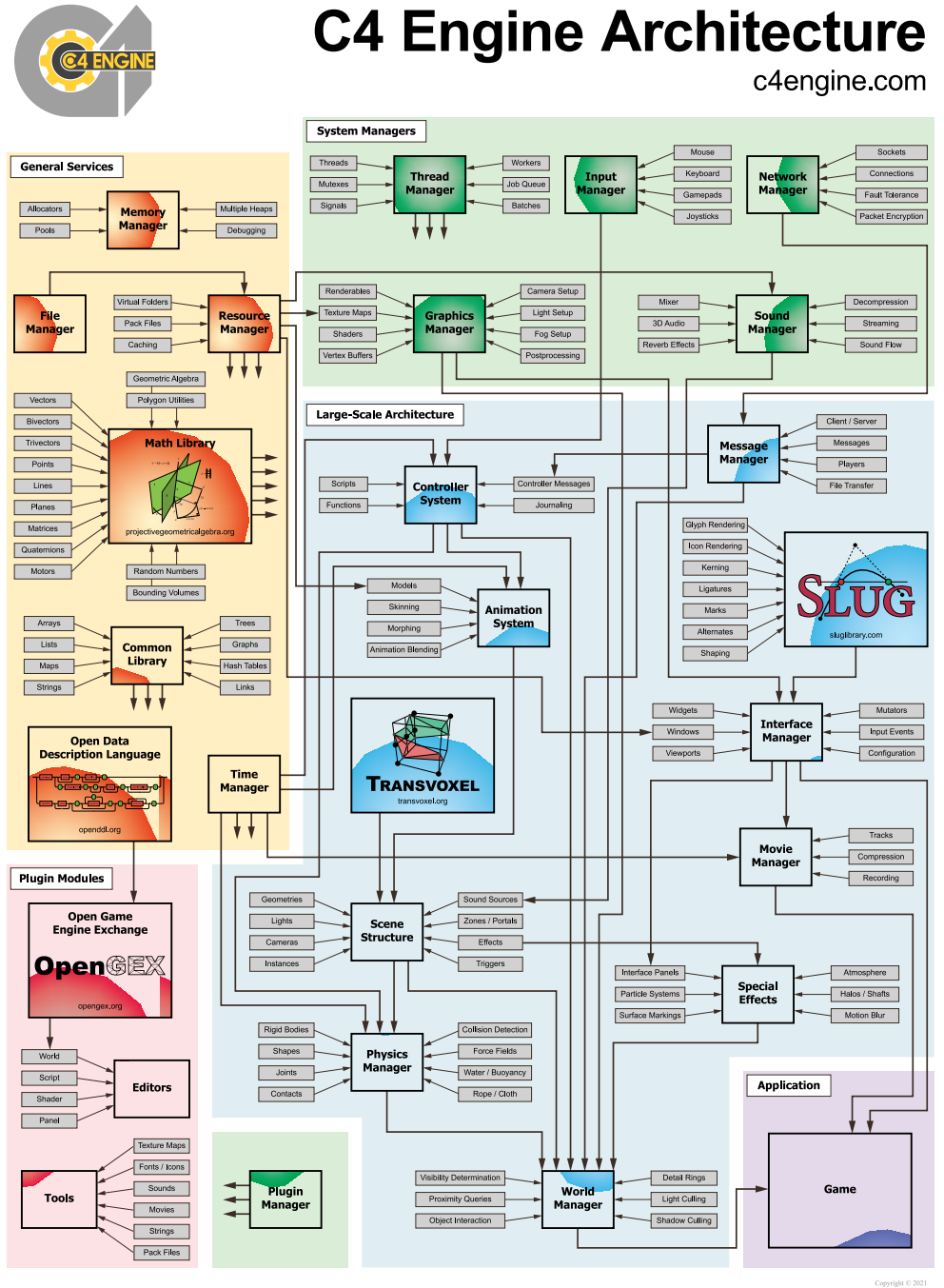
Better Software Conference 2026

About This Talk

- Math and container libraries developed over 25+ years
- How they’re used in C4 Engine and Radical Pie
- Code available on GitHub:
  - https://github.com/EricLengyel/Terathon-Math-Library
  - https://github.com/EricLengyel/Terathon-Container-Library

C4 Engine

- Started in 1999
- Commercial in 2005



Math Library

- Vector2D
- Vector3D
- Bivector3D
- Vector4D
- Point2D
- Point3D
- Matrix2D
- Matrix3D
- Matrix4D
- Quaternion
- Motor2D
- Motor3D
- Line2D
- Line3D
- Plane3D
- FlatPoint2D
- RoundPoint2D
- Dipole2D
- Circle2D
- FlatPoint3D
- RoundPoint3D
- Dipole3D
- Circle3D
- Sphere3D

Basic Vector

```
class Vector2D
{
    public:

    float x, y;

    ...
};
```

```
class Vector3D
{
    public:

    float x, y, z;

    ...
};
```

```
class Vector4D
{
    public:

    float x, y, z, w;

    ...
};
```

Constructors

```
Vector3D() = default;

Vector3D(float a, float b, float c)
{
    x = a;
    y = b;
    z = c;
}

Vector3D(const Vector3D& v)
{
    x = v.x;
    y = v.y;
    z = v.z;
}
```

Promotions

```
Vector3D(const Vector2D& v)
{
    x = v.x;
    y = v.y;
    z = 0.0F;
}
```

```
Vector4D(const Vector3D& v)
{
    x = v.x;
    y = v.y;
    z = v.z;
    w = 0.0F;
}
```

Overloaded Operators

- Obvious addition / subtraction
- Multiplication by scalar
- Division by scalar
- Multiplication of two vectors componentwise
  - Consistent with shading languages

Division by Scalar

```
Vector3D& operator /=(float s)
{
    float t = 1.0F / s;
    x *= t;
    y *= t;
    z *= t;
    return (*this);
}
```

Dot and Cross Product

- Recommend `Dot()` and `Cross()` functions

- Could overload * and % (or others)
  - But this can get confusing
  - Makes code hard to read

Wedge and Antiwedge Product

- Use `Wedge()` and `Antiwedge()` functions

- Also overload ^ operator
  - Have to deal with low operator precedence
  - Just means using parentheses a lot

- Overloaded ^ used for both wedge and antiwedge product

Wedge Product

- Vector3D ^ Vector3D = Bivector3D
  - Like cross product

- Vector3D ^ Bivector3D = scalar
  - Like dot product

Points

- Same components as vector
- But behaves differently
- $w = 1$ when promoted to 4D
- Translation included when multiplied by 4×4 matrix

Points

- Often, we just want point to be a vector
- Other times, we want to enforce point used instead of vector
- Special case math for points

Points

- What works:

- Make point a subclass of vector

Point3D Class

```
class Point3D : public Vector3D
{
    public:

    Point3D() = default;
    Point3D(float a, float b, float c) : Vector3D(a, b, c) {}
    Point3D(const Point3D& p) : Vector3D(p) {}
};
```

Point3D Class

- Point type can always be implicitly converted to vector type at no cost
- Can pass point to function accepting vector
- Can mix vectors and points in calculations
- Overload functions/operators where special behavior is needed

Point as Subclass

- Point is specific type of vector
- Vector can’t be implicitly converted to point
- Function accepting a point must always have an actual point passed to it

Point Operations

- Point + Point = Point
- Point − Point = Vector
- Point * Scalar = Point

Wedge Product

- Point3D ^ Point3D = Line3D
  - Two points make a line
  - 3D line is 4D bivector

- Point3D ^ Point3D ^ Point3D = Plane3D
  - Three points make a plane
  - 3D plane is 4D trivector

Conversion of Vector to Point

- Sometimes want a vector to be a point
  - Rare for me, only 19 uses in 600K+ lines of engine code

- Create a “origin” type to act as point at origin
  - Add vector to origin to turn it into a point
  - No casting, zero cost

Origin Type

```
class Origin3D {};

static const Origin3D Point3D::origin;

const Point3D& operator +(const Origin3D&, const Vector3D& v)
{
    return (reinterpret_cast<const Point3D&>(v));
}


Point3D p = Point3D::origin + (expression producing 3D vector);
```

Promotion of Points

```
Vector4D(const Point2D& p)
{
    x = p.x;
    y = p.y;
    z = 0.0F;
    w = 1.0F;
}
```

```
Vector4D(const Point3D& p)
{
    x = p.x;
    y = p.y;
    z = p.z;
    w = 1.0F;
}
```

Basic Matrix

```
class Matrix3D
{
    public:

    float n[3][3];

    ...
};
```

```
class Matrix4D
{
    public:

    float n[4][4];

    ...
};
```

Matrices

- Think of matrix as array of vectors
- We want this storage order

- Column vectors → matrix is array of columns
- Row vectors → matrix is array of rows

Vector Array

- Object-to-world transform is array of vectors
  - World *x* = 1st vector
  - World *y* = 2nd vector
  - World *z* = 3rd vector

Operator []

```
Vector3D& operator [](int index)
{
    return (*reinterpret_cast<Vector3D *>(n[index]);
}

const Vector3D& operator [](int index) const
{
    return (*reinterpret_cast<const Vector3D *>(n[index]);
}
```

Row or Column Vectors

- Vectors can be thought of as $1 \times n$ row matrices
- Or vectors can be thought of as $n \times 1$ column matrices
- Neither is more mathematically correct than the other

Row or Column Vectors

- Consistency is what matters
- Pick one convention and stick to it
- Everything works out the same
- But matrix-vector products will have operands in opposite orders

Matrix Storage

- Row vectors → matrix is array of rows
  - “Row-major” storage order
- Column vectors → matrix is array of columns
  - “Column-major” storage order

Operand Order

- Row vectors transformed by matrix on right

$$\begin{bmatrix} v'_x & v'_y & v'_z \end{bmatrix} = \begin{bmatrix} v_x & v_y & v_z \end{bmatrix} \mathbf{m}$$

- Column vectors transformed by matrix on left

$$\begin{bmatrix} v'_x \\ v'_y \\ v'_z \end{bmatrix} = \mathbf{m} \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix}$$

Column Vectors

- I prefer column vectors
- Remainder of this talk uses column vectors
- Convention used by scientists and engineers
- Matrix composition and quaternion composition have same ordering

Vectors and Antivectors

- Vectors are $n \times 1$ column matrices
- Antivectors are $1 \times n$ row matrices
- (Whichever you choose for vectors, it's the opposite for antivectors.)

Vectors and Antivectors

- Vectors are ordinary directions
  - Tangent, bitangent, velocity, force, etc.

- Antivectors are formed by cross products
  - Normal, angular velocity, torque, etc.
  - Bivectors in 3D, with planar orientation

Planes

- Planes are 4D antivectors

- Wedge product of 3 homogeneous points

Transformation

- Vector **v** (column) is transformed as

$$\mathbf{v}' = \mathbf{m}\mathbf{v}$$

- Antivector **n** (row) is transformed as

$$\mathbf{n}' = \mathbf{n}\,\mathrm{adj}(\mathbf{m}) = \mathbf{n}\det(\mathbf{m})\mathbf{m}^{-1}$$

Transformation

- Can enforce transformation rules by using different types for vectors / antivectors
- Implement operators only for valid products
- Can’t accidentally multiply normal or plane with matrix on the left

Transformation

```
Vector3D operator *(const Matrix3D& m, const Vector3D& v);

Vector4D operator *(const Matrix4D& m, const Vector4D& v);

Bivector3D operator *(const Bivector3D& b, const Matrix3D& m);

Line3D operator *(const Line3D& t, const Matrix4D& m);
```

Normal Transformation

- If 3×3 matrix $\mathbf{m}$ is orthogonal, $\mathbf{m}^{-1} = \mathbf{m}^{\mathrm{T}}$

- So don't need inverse to transform normal:

$$\mathbf{n}' = \mathbf{n}\mathbf{m}^{-1} = \mathbf{n}\mathbf{m}^{\mathrm{T}}$$

- In this case, can treat normal like ordinary vector:

$$(\mathbf{n}')^{\mathrm{T}} = \mathbf{m}\mathbf{n}^{\mathrm{T}}$$

Plane Transformation

- Can’t do same thing for plane $\mathbf{g}$
- 4×4 matrix $\mathbf{m}$ generally not orthogonal
- Always need adjugate for correct transform:

$$\mathbf{g}' = \mathbf{g}\,\mathrm{adj}(\mathbf{m}) = \mathbf{g}\det(\mathbf{m})\mathbf{m}^{-1}$$

Swizzling

- Shading languages have swizzles
- Can rearrange components
- Can extract subvectors
- All we can do in C++ with our basic vector class is `.x`, `.y`, `.z`, ... to get individual components

Swizzling Examples

```
Vector3D    v;

v.xyz
v.zyx

v.xy
v.zx

...
```

Swizzling

- Making this work requires some abstraction
- And some C++ templates
- But it can be done cleanly with zero perf cost
- And it’s worth it, IMO

Type Structures

- One for each mathematical entity
  - `Vector2D`, `Vector3D`, `Bivector3D`, `Matrix3D`, etc.
- Holds type info about components and subparts
- Used by templates

Type Structures

```
struct TypeVector3D
{
    typedef float component_type;
    typedef Vector2D vector2D_type;
    typedef Vector3D vector3D_type;
};
```

Component Template

- Abstraction of vector component
- Type struct is a template parameter
- Important part is that the component index is also a template parameter

Component Template

```
template <typename type_struct, int count, int index>
class Component
{
    public:

    typedef typename type_struct::component_type component_type;

    component_type       data[count];

    operator component_type&(void) {return (data[index]);}
    operator const component_type&(void) const {return (data[index]);}

    ...
```

2D Subvector Template

- Abstraction of two components of *n*-D vector
- Type struct is again a template parameter
- New parameter: boolean value indicating whether subvector is a vector or an antivector

2D Subvector Template

```
template <typename type_struct, int count, int index_x, int index_y>
class Subvec2D
{
    public:

    typedef typename type_struct::component_type component_type;
    typedef typename type_struct::vector2D_type vector2D_type;

    component_type       data[count];
    ...
```

3D and 4D Subvectors

- Similar to 2D subvector
- With more index template parameters
- Also with an “anti” template parameter to distinguish between vectors and antivectors

Conversion to Vector

- Subvectors are generic set of components
- Need to be able to implicitly convert to vectors of same dimension
- When components are consecutive in memory, don’t want any copying

Converter Templates

- Turns an *n*-D subvector into an *n*-D vector
- Used by conversion operators in subvector class templates
- Explicit specializations handle cases when components are consecutive

ConverterVector2D Template

```
template <typename type_struct, int index_x, int index_y>
struct ConverterVector2D
{
    typedef typename type_struct::component_type component_type;
    typedef typename type_struct::vector2D_type vector2D_type;
    typedef typename type_struct::vector2D_type const_vector2D_type;

    static vector2D_type Convert(component_type *data)
    {
        return (vector2D_type(data[index_x], data[index_y]));
    }
};
```

Explicit Specializations

```
template <typename type_struct>
struct ConverterVector2D<type_struct, 0, 1>
{
    typedef typename type_struct::component_type component_type;
    typedef typename type_struct::vector2D_type& vector2D_type;
    typedef const typename type_struct::vector2D_type& const_vector2D_type;

    static vector2D_type Convert(component_type *data)
    {
        return (reinterpret_cast<vector2D_type>(data[0]));
    }
};
```

Explicit Specializations

```
template <typename type_struct>
struct ConverterVector2D<type_struct, 1, 2>
{
    typedef typename type_struct::component_type component_type;
    typedef typename type_struct::vector2D_type& vector2D_type;
    typedef const typename type_struct::vector2D_type& const_vector2D_type;

    static vector2D_type Convert(component_type *data)
    {
        return (reinterpret_cast<vector2D_type>(data[1]));
    }
};
```

Explicit Specializations

```
template <typename type_struct>
struct ConverterVector2D<type_struct, 2, 3>
{
    typedef typename type_struct::component_type component_type;
    typedef typename type_struct::vector2D_type& vector2D_type;
    typedef const typename type_struct::vector2D_type& const_vector2D_type;

    static vector2D_type Convert(component_type *data)
    {
        return (reinterpret_cast<vector2D_type>(data[2]));
    }
};
```

Conversion to Vector

- Notice that generic converter constructs a new object
- But explicit specializations return references
- Difference captured in typedefs inside converter
- Used by conversion operator

Conversion of Subvec2D

```
template <typename type_struct, int count, int index_x, int index_y>
class Subvec2D
{
    ...

    operator typename
    ConverterVector2D<type_struct, index_x, index_y>::vector2D_type(void)
    {
        return (ConverterVector2D<type_struct, index_x, index_y>
            ::Convert(data));
    }
```

Overloaded Operators

- Arithmetic done with Subvec2D, Subvec3D, Subvec4D
- Allows general swizzling of both operands
- Compiler automatically generates code for all combinations actually used in the program

Assignment Operator

```
template <typename type_struct, int count, int index_x, int index_y>
class Subvec2D
{
    ...

    template <typename type, int cnt, int ind_x, int ind_y>
    Subvec2D& operator =(const Subvec2D<type, cnt, ind_x, ind_y>& value)
    {
        data[index_x] = value.data[ind_x];
        data[index_y] = value.data[ind_y];
        return (*this);
    }
```

Vector / Antivector Safeguard

- Note that `anti` template parameter must be same for both operands in 3D/4D

- Can’t accidentally mix vectors and antivectors

Operators for Full Vectors

- Also overload operators with full vector operands
- Subvectors will be result of swizzles
- Otherwise, would have to write swizzles all the time, even if components not reordered

Assignment of Full Vector

```
template <typename type_struct, int count, int index_x, int index_y>
class Subvec2D
{
    ...

    Subvec2D& operator =(const vector2D_type& value)
    {
        data[index_x] = value.x;
        data[index_y] = value.y;
        return (*this);
    }
```

Overloaded Operators

- Componentwise +, −, * work similarly
- Scalar *, / affect components identified by `index` template parameters
- Nothing fancy going on

Unions

- Swizzle members implemented with union
- Holds all individual components
- Holds all possible subvectors
- May choose to exclude subvectors with repeated components
  - Enable with a conditional compile `#define`

Vec2D Class Template

```
template <typename type_struct>
class Vec2D
{
    public:

    union
    {
        Component<type_struct, 2, 0>        x;
        Component<type_struct, 2, 1>        y;
        Subvec2D<type_struct, 2, 0, 1>      xy;
        Subvec2D<type_struct, 2, 1, 0>      yx;
    };
```

Vec3D Union (15 members)

```
Component<type_struct, 3, 0>                    x;
Component<type_struct, 3, 1>                    y;
Component<type_struct, 3, 2>                    z;
Subvec2D<type_struct, 3, 0, 1>                  xy;
Subvec2D<type_struct, 3, 0, 2>                  xz;
Subvec2D<type_struct, 3, 1, 0>                  yx;
Subvec2D<type_struct, 3, 1, 2>                  yz;
Subvec2D<type_struct, 3, 2, 0>                  zx;
Subvec2D<type_struct, 3, 2, 1>                  zy;
Subvec3D<type_struct, anti, 3, 0, 1, 2>         xyz;
Subvec3D<type_struct, anti, 3, 0, 2, 1>         xzy;
Subvec3D<type_struct, anti, 3, 1, 0, 2>         yxz;
Subvec3D<type_struct, anti, 3, 1, 2, 0>         yzx;
Subvec3D<type_struct, anti, 3, 2, 0, 1>         zxy;
Subvec3D<type_struct, anti, 3, 2, 1, 0>         zyx;
```

Vec4D Union (64 members)

```
Component<type_struct, 4, 0>                x;      Subvec3D<type_struct, anti, 4, 1, 0, 3>     yxw;     Subvec4D<type_struct, anti, 4, 1, 2, 0, 3>  yzxw;
Component<type_struct, 4, 1>                y;      Subvec3D<type_struct, anti, 4, 1, 3, 0>     ywx;     Subvec4D<type_struct, anti, 4, 1, 2, 3, 0>  yzwx;
Component<type_struct, 4, 2>                z;      Subvec3D<type_struct, anti, 4, 1, 2, 3>     yzw;     Subvec4D<type_struct, anti, 4, 1, 3, 0, 2>  ywxz;
Component<type_struct, 4, 3>                w;      Subvec3D<type_struct, anti, 4, 1, 3, 2>     ywz;     Subvec4D<type_struct, anti, 4, 1, 3, 2, 0>  ywzx;
Subvec2D<type_struct, 4, 0, 1>              xy;     Subvec3D<type_struct, anti, 4, 2, 0, 1>     zxy;     Subvec4D<type_struct, anti, 4, 2, 0, 1, 3>  zxyw;
Subvec2D<type_struct, 4, 0, 2>              xz;     Subvec3D<type_struct, anti, 4, 2, 1, 0>     zyx;     Subvec4D<type_struct, anti, 4, 2, 0, 3, 1>  zxwy;
Subvec2D<type_struct, 4, 0, 3>              xw;     Subvec3D<type_struct, anti, 4, 2, 0, 3>     zxw;     Subvec4D<type_struct, anti, 4, 2, 1, 0, 3>  zyxw;
Subvec2D<type_struct, 4, 1, 0>              yx;     Subvec3D<type_struct, anti, 4, 2, 3, 0>     zwx;     Subvec4D<type_struct, anti, 4, 2, 1, 3, 0>  zywx;
Subvec2D<type_struct, 4, 1, 2>              yz;     Subvec3D<type_struct, anti, 4, 2, 1, 3>     zyw;     Subvec4D<type_struct, anti, 4, 2, 3, 0, 1>  zwxy;
Subvec2D<type_struct, 4, 1, 3>              yw;     Subvec3D<type_struct, anti, 4, 2, 3, 1>     zwy;     Subvec4D<type_struct, anti, 4, 2, 3, 1, 0>  zwyx;
Subvec2D<type_struct, 4, 2, 0>              zx;     Subvec3D<type_struct, anti, 4, 3, 0, 1>     wxy;     Subvec4D<type_struct, anti, 4, 3, 0, 1, 2>  wxyz;
Subvec2D<type_struct, 4, 2, 1>              zy;     Subvec3D<type_struct, anti, 4, 3, 1, 0>     wyx;     Subvec4D<type_struct, anti, 4, 3, 0, 2, 1>  wxzy;
Subvec2D<type_struct, 4, 2, 3>              zw;     Subvec3D<type_struct, anti, 4, 3, 0, 2>     wxz;     Subvec4D<type_struct, anti, 4, 3, 1, 0, 2>  wyxz;
Subvec2D<type_struct, 4, 3, 0>              wx;     Subvec3D<type_struct, anti, 4, 3, 2, 0>     wzx;     Subvec4D<type_struct, anti, 4, 3, 1, 2, 0>  wyzx;
Subvec2D<type_struct, 4, 3, 1>              wy;     Subvec3D<type_struct, anti, 4, 3, 1, 2>     wyz;     Subvec4D<type_struct, anti, 4, 3, 2, 0, 1>  wzxy;
Subvec2D<type_struct, 4, 3, 2>              wz;     Subvec3D<type_struct, anti, 4, 3, 2, 1>     wzy;     Subvec4D<type_struct, anti, 4, 3, 2, 1, 0>  wzyx;
Subvec3D<type_struct, anti, 4, 0, 1, 2>     xyz;    Subvec4D<type_struct, anti, 4, 0, 1, 2, 3>  xyzw;
Subvec3D<type_struct, anti, 4, 0, 2, 1>     xzy;    Subvec4D<type_struct, anti, 4, 0, 1, 3, 2>  xywz;
Subvec3D<type_struct, anti, 4, 0, 1, 3>     xyw;    Subvec4D<type_struct, anti, 4, 0, 2, 1, 3>  xzyw;
Subvec3D<type_struct, anti, 4, 0, 3, 1>     xwy;    Subvec4D<type_struct, anti, 4, 0, 2, 3, 1>  xzwy;
Subvec3D<type_struct, anti, 4, 0, 2, 3>     xzw;    Subvec4D<type_struct, anti, 4, 0, 3, 1, 2>  xwyz;
Subvec3D<type_struct, anti, 4, 0, 3, 2>     xwz;    Subvec4D<type_struct, anti, 4, 0, 3, 2, 1>  xwzy;
Subvec3D<type_struct, anti, 4, 1, 0, 2>     yxz;    Subvec4D<type_struct, anti, 4, 1, 0, 2, 3>  yxzw;
Subvec3D<type_struct, anti, 4, 1, 2, 0>     yzx;    Subvec4D<type_struct, anti, 4, 1, 0, 3, 2>  yxwz;
```

Repeated Components

- Rarely useful (I’ve never needed them)
- Adds a lot more members to union, increases compilation time
- Don’t want them to be assignable
- Declare them `const` in the union

Final Types

- Components and subvectors are internal implementation
- Rest of code doesn’t need to know about them
- Use high-level classes for specific mathematical types

Final Types

```
class Vector2D : public Vec2D<TypeVector2D>
class Vector3D : public Vec3D<TypeVector3D, false>
class Vector4D : public Vec4D<TypeVector4D, false>

class Point3D : public Vector3D

class Bivector3D : public Vec3D<TypeBivector3D, true>
class Plane3D : public Vec4D<TypePlane3D, true>

class Integer2D : public Vec2D<TypeInteger2D>
class Integer3D : public Vec2D<TypeInteger3D>
```

Antivector Type Structure Example

```
struct TypePlane3D
{
    typedef float component_type;
    typedef Vector2D vector2D_type;
    typedef Bivector3D vector3D_type;
    typedef Plane3D vector4D_type;
};
```

Matrices

- We can extend same concepts to matrices
- Subvectors can be used to extract rows and columns
- Then they can be used in expressions without any copying
- Compiler generates new functions to access components in the right places

Matrices

- Submatrices of lower dimension can be extracted
- Only one I've ever used is 3×3 upper-left part of a 4×4 matrix
- Submatrix just indexes components inside larger matrix, so no copying

Matrices

- We can “swizzle” a matrix to turn it into its transpose
- Again, no copying
- And compiler generates new functions to perform operations on transposed matrix

3D Submatrix Template

- Abstraction of 3×3 submatrix
- Type struct and entry count make two template parameters
- Entry locations defined by 9 more template parameters

Submat3D Template

```
template <typename type_struct, int count, int index_00, int index_01,
    int index_02, int index_10, int index_11, int index_12,
    int index_20, int index_21, int index_22>
class Submat3D
{
    public:

    typedef typename type_struct::component_type component_type;
    typedef typename type_struct::matrix3D_type matrix3D_type;

    component_type    data[count];
    ...
```

Submat4D Template

```
template <typename type_struct, int count, int index_00, int index_01, int index_02,
    int index_03, int index_10, int index_11, int index_12, int index_13,
    int index_20, int index_21, int index_22, int index_23, int index_30,
    int index_31, int index_32, int index_33>
class Submat4D
{
    public:

    typedef typename type_struct::component_type component_type;
    typedef typename type_struct::matrix4D_type matrix4D_type;

    component_type    data[count];
    ...
```

Unions

- Same concept as with vectors
- Holds all individual entries
- Holds sets of columns and rows
- Holds submatrices, if needed
- Holds transpose

Columns and Rows

- Columns are vectors
- Rows are antivectors
- This is reflected in subvector types

Matrix Rows

- Noncontiguous components
- Still behaves like ordinary antivectors in code doing operations with them
- Rows of 4×4 matrix are planes

Mat3D Union

```
Component<type_struct, 9, 0>                                m00;
Component<type_struct, 9, 1>                                m10;
Component<type_struct, 9, 2>                                m20;
Component<type_struct, 9, 3>                                m01;
Component<type_struct, 9, 4>                                m11;
Component<type_struct, 9, 5>                                m21;
Component<type_struct, 9, 6>                                m02;
Component<type_struct, 9, 7>                                m12;
Component<type_struct, 9, 8>                                m22;

Subvec3D<column_type_struct, false, 9, 0, 1, 2>             col0;
Subvec3D<column_type_struct, false, 9, 3, 4, 5>             col1;
Subvec3D<column_type_struct, false, 9, 6, 7, 8>             col2;

Subvec3D<row_type_struct, true, 9, 0, 3, 6>                 row0;
Subvec3D<row_type_struct, true, 9, 1, 4, 7>                 row1;
Subvec3D<row_type_struct, true, 9, 2, 5, 8>                 row2;

Submat3D<type_struct, 9, 0, 3, 6, 1, 4, 7, 2, 5, 8>         matrix;
Submat3D<type_struct, 9, 0, 1, 2, 3, 4, 5, 6, 7, 8>         transpose;
```

Mat4D Union

```
Component<type_struct, 16, 0>    m00;    Subvec4D<column_type_struct, false, 16, 0, 1, 2, 3>      col0;
Component<type_struct, 16, 1>    m10;    Subvec4D<column_type_struct, false, 16, 4, 5, 6, 7>      col1;
Component<type_struct, 16, 2>    m20;    Subvec4D<column_type_struct, false, 16, 8, 9, 10, 11>    col2;
Component<type_struct, 16, 3>    m30;    Subvec4D<column_type_struct, false, 16, 12, 13, 14, 15>  col3;
Component<type_struct, 16, 4>    m01;
Component<type_struct, 16, 5>    m11;    Subvec4D<row_type_struct, true, 16, 0, 4, 8, 12>         row0;
Component<type_struct, 16, 6>    m21;    Subvec4D<row_type_struct, true, 16, 1, 5, 9, 13>         row1;
Component<type_struct, 16, 7>    m31;    Subvec4D<row_type_struct, true, 16, 2, 6, 10, 14>        row2;
Component<type_struct, 16, 8>    m02;    Subvec4D<row_type_struct, true, 16, 3, 7, 11, 15>        row3;
Component<type_struct, 16, 9>    m12;
Component<type_struct, 16, 10>   m22;    Submat3D<type_struct, 16, 0, 4, 8, 1, 5, 9, 2, 6, 10>    matrix3D;
Component<type_struct, 16, 11>   m32;    Submat3D<type_struct, 16, 0, 1, 2, 4, 5, 6, 8, 9, 10>    transpose3D;
Component<type_struct, 16, 12>   m03;
Component<type_struct, 16, 13>   m13;    Submat4D<type_struct, 16, 0, 4, 8, 12, 1, 5, 9, 13,
Component<type_struct, 16, 14>   m23;        2, 6, 10, 14, 3, 7, 11, 15>                          matrix;
Component<type_struct, 16, 15>   m33;    Submat4D<type_struct, 16, 0, 1, 2, 3, 4, 5, 6, 7,
                                             8, 9, 10, 11, 12, 13, 14, 15>                        transpose;
```

Final Types

- As with vector types, details of submatrices and subvectors are internal

- Application works with high-level classes

Final Types

```
class Matrix3D : public Mat3D<TypeMatrix3D>
class Matrix4D : public Mat4D<TypeMatrix4D>

class Transform4D : public Matrix4D
```

Matrix Type Struct

- Contains component type
- Contains high-level matrix type for conversions
- Contains type structs for columns and rows
  - Used by subvector representations

Matrix Type Structs

```
struct TypeMatrix3D
{
    typedef float component_type;
    typedef Matrix3D matrix3D_type;
    typedef TypeVector3D column_type_struct;
    typedef TypeBivector3D row_type_struct;
};
```

Matrix Type Structs

```
struct TypeMatrix4D
{
    typedef float component_type;
    typedef Matrix3D matrix3D_type;
    typedef Matrix4D matrix4D_type;
    typedef TypeVector4D column_type_struct;
    typedef TypePlane3D row_type_struct;
};
```

Matrix Operations

- Matrix-matrix products and matrix-vector products defined in terms of submatrices and subvectors

- Compiler can generate specialized functions for any swizzled combinations

Normal Transformation

- Remember when $\mathbf{m}$ is orthogonal

$$\mathbf{n}' = \mathbf{n}\mathbf{m}^{-1} = \mathbf{n}\mathbf{m}^{\mathrm{T}}$$

- We can do this without making copy:

```
n2 = n1 * m.transpose;
```

Plane Extraction

- Rows of 4x4 matrix are *y*-*z*, *x*-*z*, and *x*-*y* planes in dest space
- To calculate distance *d* from point **p** to *x*-*y* plane:

```
float d = m.row2 ^ p;
```

- This automatically generates a function that takes entries 2, 6, 10, and 14 from matrix **m**

Matrix Operations

- Could just declare matrix-matrix product in header file
  - 36 template parameters for 4x4 matrices!
- Then define product in .cpp file
- Explicitly instantiate for unswizzled matrices and transposes

Container Library

- Array<type>
- List<type>
- ListElement<type>
- Map<type>
- MapElement<type>
- HashTable<type>
- HashTableElement<type>
- Tree<type>
- Graph<elementType>
- GraphElement<elementType, relationType>
- GraphRelation<elementType, relationType>

Array Container

- Basic contiguous array of objects with same type
  - Like `std::vector`, but without the bloat
- Grows as necessary to accommodate new elements
  - Reserved space grows by 50%, rounded up to next multiple of 4 elements, whenever array needs to be reallocated
- Can optionally include internal storage for small array

Array Container

- `Array<type>`
  - Minimal local storage
  - All elements in single heap-allocated buffer
- `Array<type, n>`
  - Includes local storage for n elements
  - Array moved to heap allocation when size exceeds n elements

Array Container

- `Array<type, n>` has `ImmutableArray<type>` base class so functions can accept arrays with different values of n

- ImmutableArray object is 16 bytes:

```
template <typename type> class ImmutableArray
{
    int32    elementCount;
    int32    reservedCount;
    type     *arrayPointer;
};
```

Array Container

- Constructor invoked with placement new when element actually added to array

```
template <typename type, int32 n> type *Array<type, n>::AppendArrayElement(void)
{
    if (elementCount >= reservedCount)
    {
        SetReservedCount(elementCount + 1);
    }

    type *pointer = arrayPointer + elementCount;
    new(pointer) type;

    elementCount++;
    return (pointer);
}
```

Array Container

- Move construction is supported:

```
template <typename type, int32 n>
template <typename T>
type *Array<type, baseCount>::AppendArrayElement(T&& element)
{
    if (elementCount >= reservedCount)
    {
        SetReservedCount(elementCount + 1);
    }

    type *pointer = arrayPointer + elementCount;
    new(pointer) type(static_cast<T&&>(element));

    elementCount++;
    return (pointer);
}
```

Array Container

- Range for is supported:

```
template <typename type> class ImmutableArray
{
    ...

    type *begin(void) const
    {
        return (arrayPointer);
    }

    type *end(void) const
    {
        return (arrayPointer + elementCount);
    }
```

Array Container

- Using range `for`:

```
Array<type>   objectArray;

for (const type& object : objectArray)
{
    ...
}
```

List Container

- Holds doubly-linked list
- Can contain polymorphic object types
- Links are intrusive, stored inside objects
  - Not like `std::list` at all
- Typically used to hold sizeable objects
  - Perhaps 100s of bytes

List Container

- `List<type>` template declares a list container
- Object inherits from `ListElement<type>` template to get links
  - CRTP -- template parameter is type of derived class
- Each of these classes has a non-template base class
  - Handles everything that doesn’t depend on derived type
  - Prevents compiler from generating same functions over and over

List Container

- `ListBase` is 32 bytes:

```
class ListBase
{
    friend class ListElementBase;

    private:

        ListElementBase    *firstListElement;
        ListElementBase    *lastListElement;
};

template <class type> List : public ListBase
{
    ...
};
```

List Container

- `ListElementBase` is 32 bytes:

```
class ListElementBase
{
    friend class ListBase;

    private:

    ListElementBase    *prevListElement;
    ListElementBase    *nextListElement;
    ListBase           *owningList;
};

template <class type> ListElement : public ListElementBase
{
    ...
};
```

List Container

- Object inserted or removed from list:

```
List<Object>   myList;

Object *myObject = new Object;
myList.AppendListElement(myObject);

...

myList.RemoveListElement(myObject);
```

List Iteration

- Iterating through list:

```
List<Object>   myList;

Object *myObject = myList.GetFirstListElement();
while (myObject)
{
    ....
    myObject = myObject->GetNextListElement();
}
```

- Range for supported:

```
for (Object *myObject : myList)
{
}
```

List Element Deletion

- An object knows what list it's in, if any

- `ListElementBase` destructor removes itself:

```
ListElementBase::~ListElementBase()
{
    if (owningList)
    {
        owningList->RemoveListElement(this);
    }
}
```

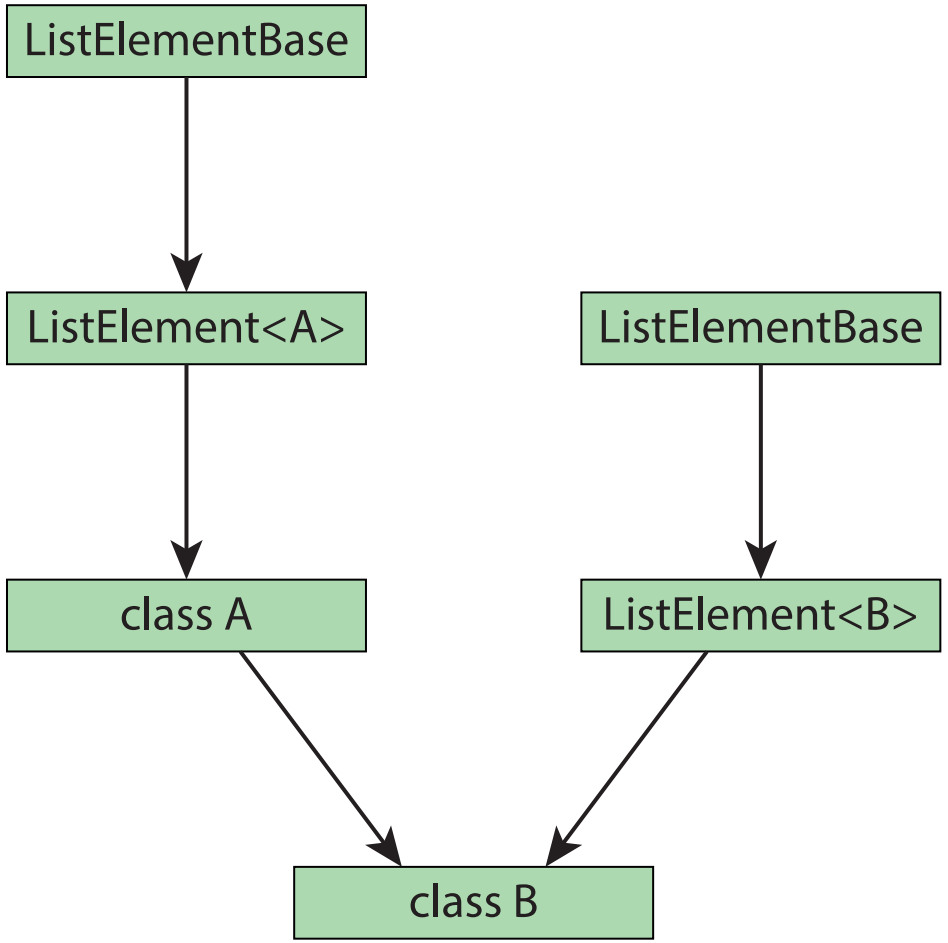
- Never have to worry about dangling pointers on deletion

Object in Multiple Lists

- IME, rare that object needs to be two lists *holding the same type* simultaneously

- However, it's natural for a base class to reside one list while a derived class resides in another list

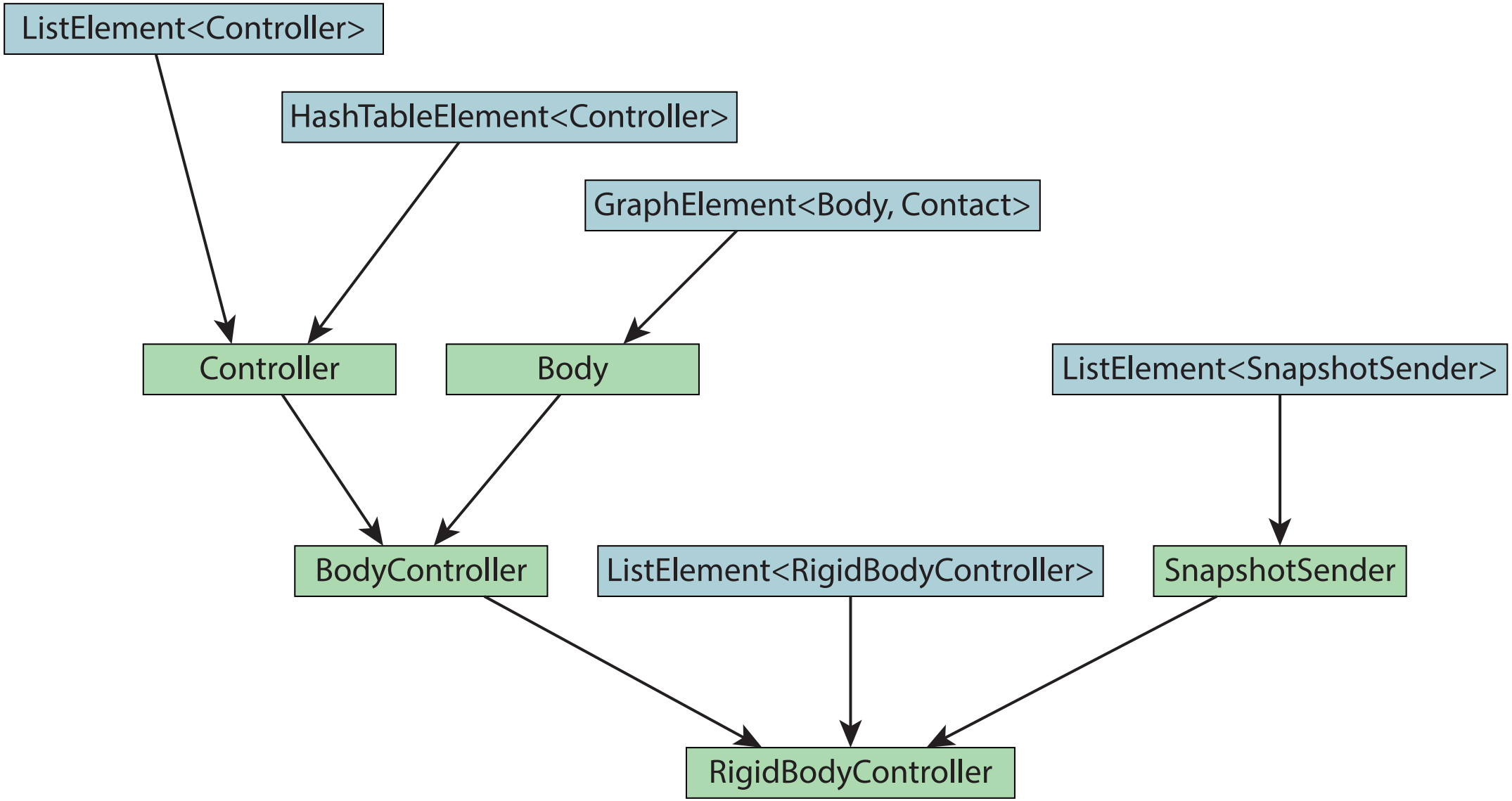
Object in Multiple Lists



Object in Multiple Lists

- Multiple lists means multiple `ListElementBase` base objects
- Links are generic pointers to `ListElementBase`
- `ListElement<type>` subclass translates to pointer to `type`

```
template <class type> class ListElement : public ListElementBase
{
    type *GetNextListElement(void) const
    {
        return (static_cast<type *>(static_cast<ListElement<type> *>
            (ListElementBase::GetNextListElement())));
    }
};
```


ListElement<Controller>
HashTableElement<Controller>
GraphElement<Body, Contact>
Controller
Body
ListElement<SnapshotSender>
BodyController
ListElement<RigidBodyController>
SnapshotSender
RigidBodyController

Map Container

- Associative key-object container
- Stored internally as an AVL tree
  - Balanced such that heights of left and right subtrees at every node never differ by more than one
- Has intrusive links, like list
- Keys are provided by objects themselves

Map Container

- MapBase class is only 8 bytes:

```
class MapBase
{
    friend class MapElementBase;

    private:

        MapElementBase    *rootElement;
};

template <class type> Map : public MapBase
{
    ...
};
```

Map Container

- `MapElementBase` is 48 bytes:

```
class MapElementBase
{
    friend class MapBase;

    private:

    MapElementBase    *superNode;
    MapElementBase    *leftSubnode;
    MapElementBase    *rightSubnode;
    MapBase           *owningMap;
    int32             balance;
};

template <class type> MapElement : public MapElementBase
{
};
```

Map Container

- Iteration visits elements in order of keys
- `Find(key)` function runs in $O(\log n)$ time

HashTable Container

- Essentially a table of a power-of-two number of “buckets”
- Each bucket is just list containing all objects with same hash value modulo 2^n
- Buckets are reallocated as necessary to enforce maximum average size (like 4 elements)
- Objects themselves provide hash values

Tree Container

- A collection of objects arranged in a tree
- Each object is doubly-linked to its siblings and also owns a list of its children
- The `Tree<type>` class serves as the base class for both the container and the objects in the container
- Each object is the root of a subtree

Tree Container

- `GetNextSubnode()` and `GetPreviousSubnode()` functions iterate forward and backward among siblings
- `GetFirstSubnode()` and `GetLastSubnode()` functions provide access to children
- Easy to do pre-order and post-order traversals
- More sophisticated functions provide depth-first traversals

Tree Container

- Fundamental container for lots of high-level architecture:
  - Equations in Radical Pie
  - Node hierarchy in C4 Engine
  - Structures in OpenDDL
  - Widget tree
  - Animator tree
  - Visibility region tree
  - Expression parse tree
  - Input device control tree

Graph Container

- Generic directed graph container
  - `Graph<elementType, relationType>`

- Involves two types:
  - Type of element / vertex
  - Type of relation / edge

- Two base classes:
  - `GraphElement<elementType, relationType>`
  - `GraphRelation<elementType, relationType>`

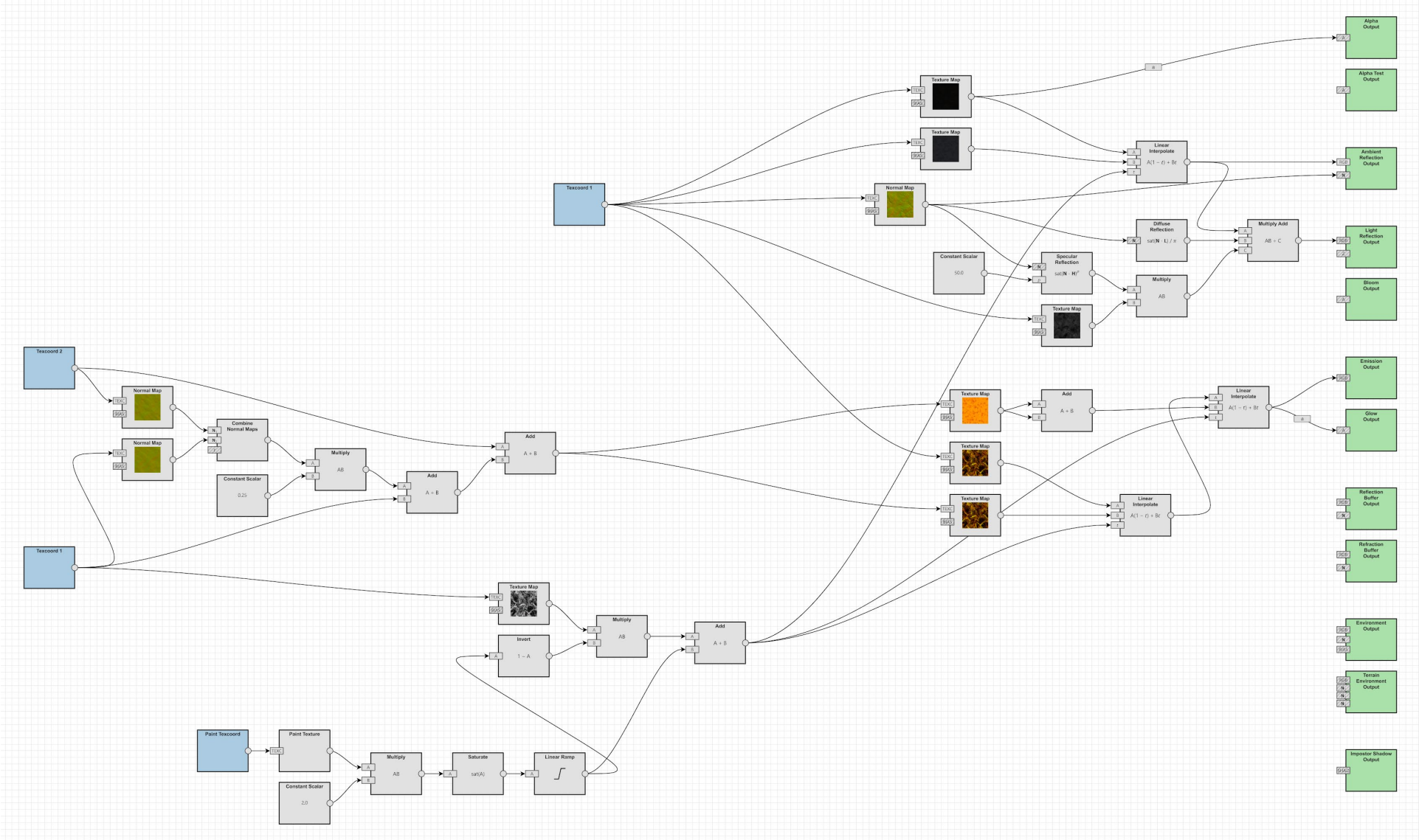
Graph Container

- IME, uses for graph container arise surprisingly often:

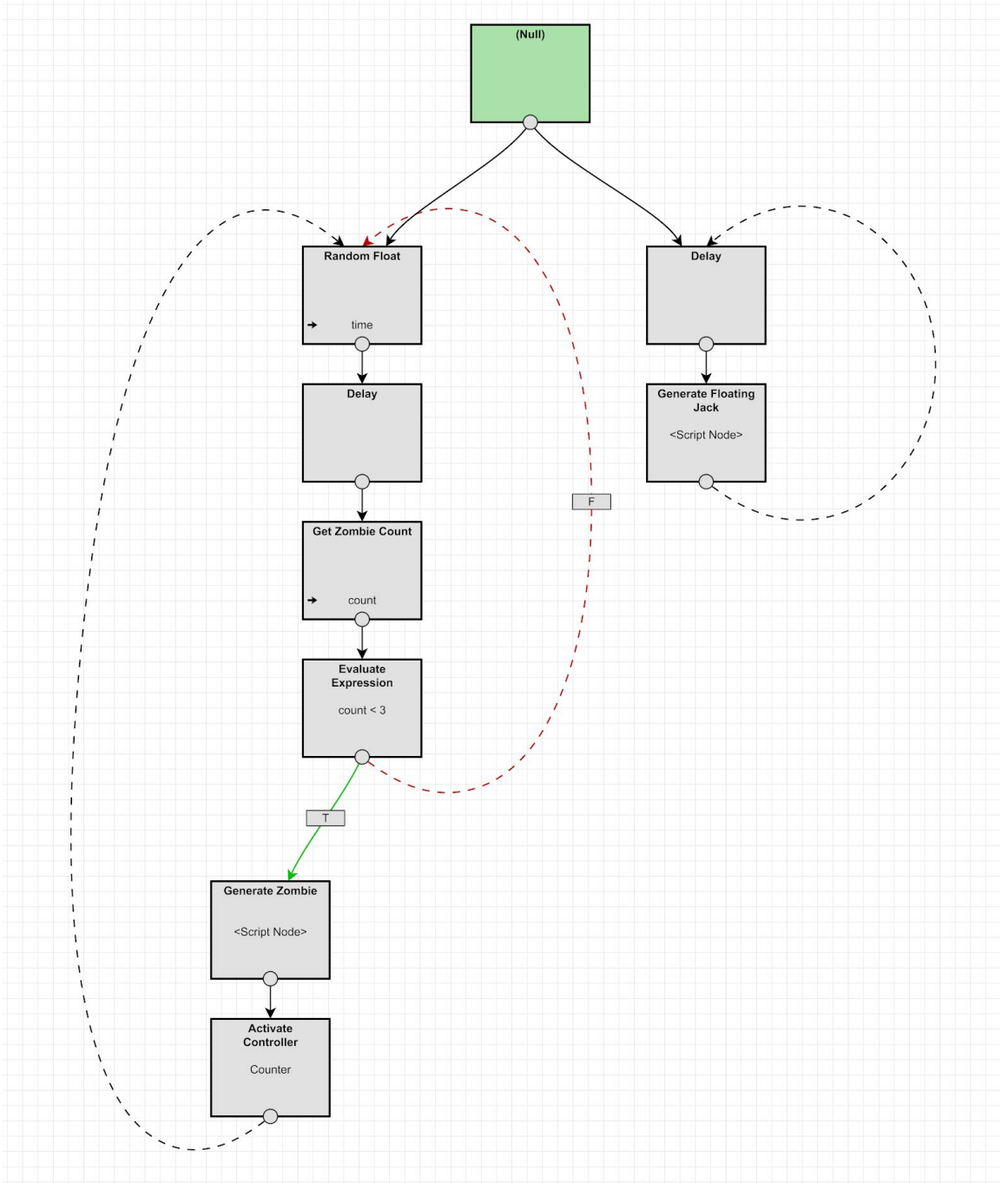
- In Radical Pie:
  - Anchor attachments for drawings and annotations

- Major occurrences in C4 Engine:
  - Graphical shader editor
  - Graphical scripting system
  - General connector network
  - Physics contact graph
  - Large-scale visibility determination

Shader Graph



Script Graph



Graph Element

- Graph element contains lists of outgoing and incoming relations

```
class GraphElementBase
{
    private:

    List<GraphRelationStart>    outgoingRelationList;
    List<GraphRelationFinish>   incomingRelationList;

    ...
```

- Since each relation must connect one element to another, it's always in two lists simultaneously

Graph Relation

- Two base classes hold list links

```
class GraphRelationStart : public ListElement<GraphRelationStart>
{
    GraphElementBase    *startElement;
};

class GraphRelationFinish : public GraphRelationStart,
                            public ListElement<GraphRelationFinish>
{
    GraphElementBase    *finishElement;
};

template <class elementType, class relationType>
class GraphRelation : public GraphRelationFinish
{
};
```

Relation Iteration

- `GraphElement` provides access to first or last outgoing or incoming relations

- `GraphRelation` provides access to next or previous relation of same kind

```
Contact *contact = rigidBody->GetFirstOutgoingRelation();
while (contact)
{
    ...
    contact = contact->GetNextOutgoingRelation();
}
```

Relation Management

- Relation constructor requires start and finish elements
  - Automatically inserts itself into outgoing / incoming lists

```
new Fiber(method1, method2);
```

- Deleting a graph element automatically deletes all outgoing and incoming relations
  - This also removes them from lists on the other ends of the relations

Cell Graph in C4 Engine

- Each zone divided into octree of cells
  - Axis-aligned boxes
- Cells have fixed sizes
  - Half size of parent in each dimension
- Each cell contained within parent cell
  - If parent not visible, nothing succeeding it is either

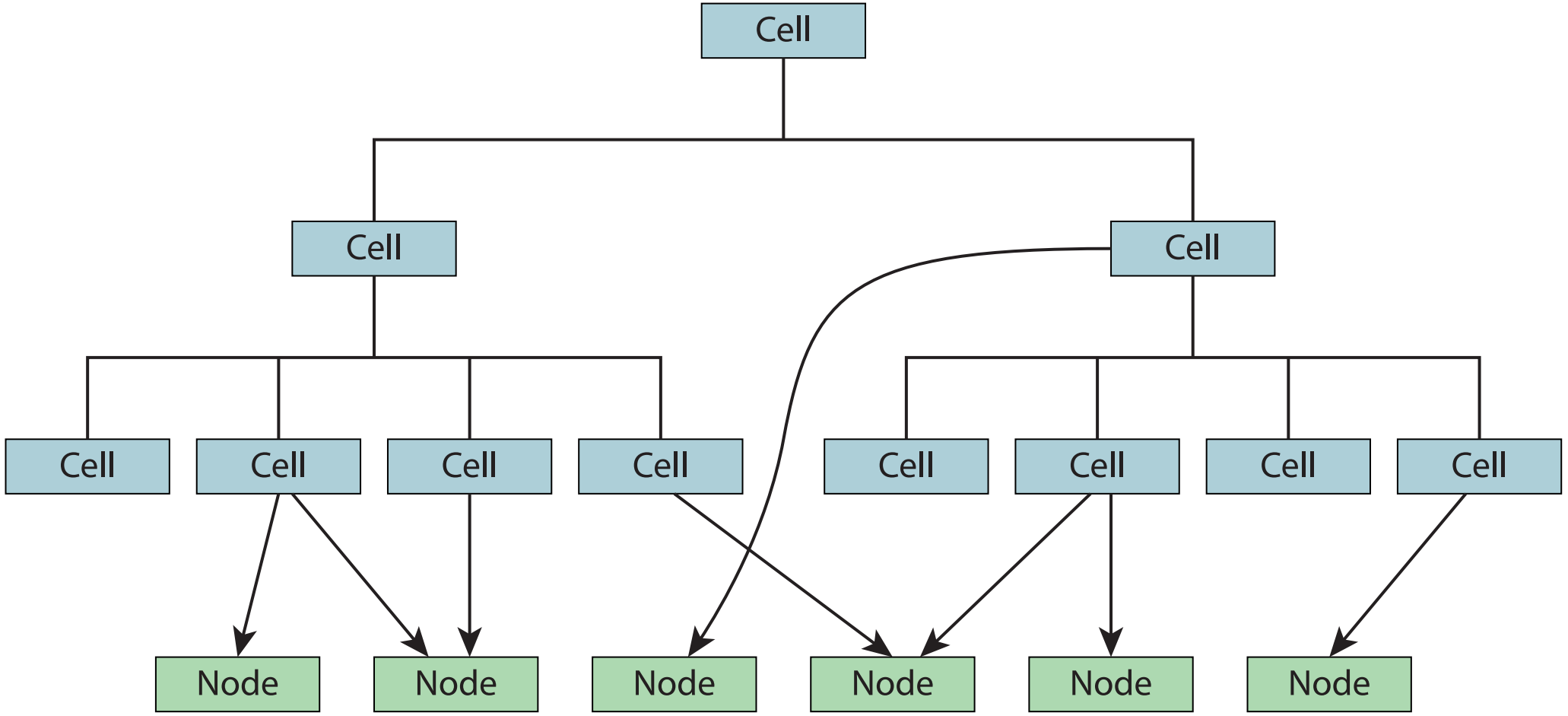
Cell Graph in C4 Engine

- There are multiple cell graphs for different “detail rings”
- Node inserted into graph corresponding to its detail ring
  - Ring 0 = always visible
  - Ring 1 = fades out far away
  - Ring 2 = fades out closer
  - ...
- Nodes often get inserted into multiple cells because they straddle boundaries

Cell Graph in C4 Engine

- Traversals aim to eliminate largest cells possible to lop off huge parts of world
- Nodes can be visited through multiple paths in cell graph
- Concurrent traversals can happen in multiple threads
  - E.g., for different shadow cascades
- Frame stamping prevents double-processing
  - Atomics handle access from multiple threads
  - Frame stamp + cascade bit

Cell Graph in C4 Engine



Cell Graph in C4 Engine

- Nodes that move are re-inserted into cell graph
  - Old relations deleted
  - New relations created
  - Dedicated memory pool for fixed-size relations very fast

- Extremely convenient that graph container classes automatically clean up relations when a node is deleted

Contact

- lengyel@terathon.com
- Twitter: @EricLengyel
- Bluesky: @ericlengyel.bsky.social
- LinkedIn: www.linkedin.com/in/eric-lengyel