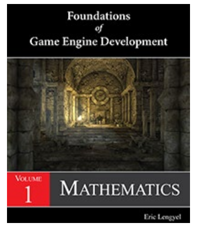
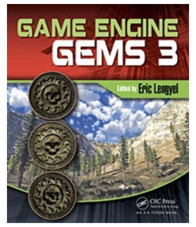
Linear Algebra Upgraded

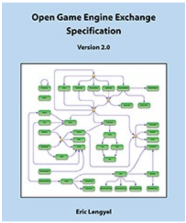
Eric Lengyel, Ph.D.
Terathon Software

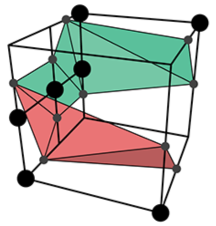
About the speaker

- Working in game/graphics dev since 1994
  - Previously at Sierra, Apple, Naughty Dog
- Current projects:
  - Slug Library, C4 Engine, The 31st, FGED









About this talk

- Vector / matrix implementation in C++
  - Vectors, points, planes, lines, antivectors
  - Swizzling (like shading languages)
  - Matrix manipulation
  - 2D, 3D, 4D

About this talk

- Promoting mathematical correctness
- Providing zero-cost conveniences
  - Vector swizzling: v1 = v2.zyx
  - Row/column extraction: m.row1, m.col0
  - Free transpose: v1 = m.transpose * v2

About this talk

- Many ways to implement math library
- Many ways equally correct
- Purpose of this talk is to share my experiences with code I've developed and refined over many years and give advice
- You are free to implement what you like

About this talk

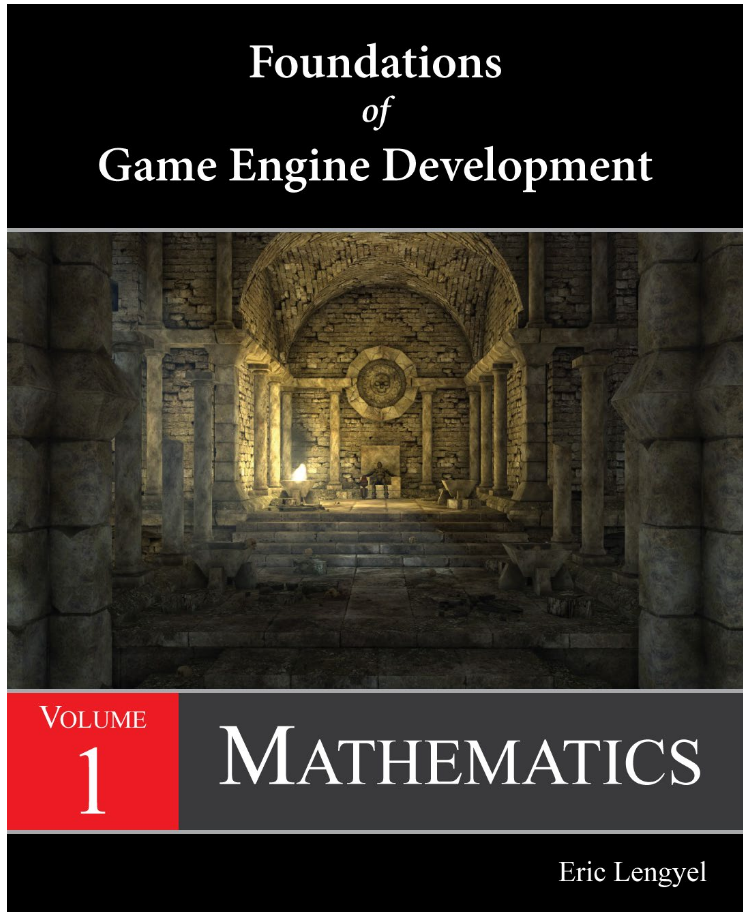
- Skipping obvious stuff
  - Like overloading operator + for two vectors
- Focusing on things that are not common

Class Names

- Vector2D, Vector3D, Vector4D, Point3D
- Matrix3D, Matrix4D, Transform4D
- Bivector3D (normals)
- Bivector4D (lines)
- Trivector4D (planes)

Grassmann Algebra

- *Foundations of Game Engine Development, Volume 1*
- “Fundamentals of Grassmann Algebra” (GDC 2012)
- “Grassmann Algebra in Game Development” (GDC 2014)



Grassmann Algebra

- Thorough understanding not necessary
- Vector/antivector distinction most important
- Various facts stated throughout this talk

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

- Could use wedge() function
- I prefer overloading ^ operator
  - Have to deal with low operator precedence
  - Just means using parentheses a lot
- 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 a point passed to it

Point Operations

- Point + Point = Point
- Point – Point = Vector
- Point * Scalar = Point

Wedge Product

- Point3D ^ Point3D = Bivector4D
  - Two points make a line

- Point3D ^ Point3D ^ Point3D = Trivector4D
  - Three points make a plane

Conversion of Vector to Point

- Sometimes want a vector to be a point
  - Rare for me, only 19 uses in 600K+ lines of code
- Create a “Zero” type to act as point at origin
  - Add vector to origin to turn it into a point
  - No casting, zero cost

Zero Type

```
class Zero3DType {};

extern const Zero3DType Zero3D;

const Point3D& operator +(const Zero3DType&, const Vector3D& v)
{
    return (reinterpret_cast<const Point3D&>(v));
}


Point3D p = Zero3D + (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 use column vectors
- Remainder of this talk uses column vectors
- Convention used by scientists and engineers
- Matrix composition and quaternion composition have same ordering
  - Consistency is king

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 (column) is transformed as

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

- Antivector (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);

Trivector4D operator *(const Trivector4D& 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 vector:

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

Plane Transformation

- Can't do same thing for plane **f**
- 4×4 matrix **H** generally not orthogonal
- Always need adjugate for correct transform:

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

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, etc.

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
- 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 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 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

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

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;
Component<type_struct, 4, 1>              y;
Component<type_struct, 4, 2>              z;
Component<type_struct, 4, 3>              w;
Subvec2D<type_struct, 4, 0, 1>            xy;
Subvec2D<type_struct, 4, 0, 2>            xz;
Subvec2D<type_struct, 4, 0, 3>            xw;
Subvec2D<type_struct, 4, 1, 0>            yx;
Subvec2D<type_struct, 4, 1, 2>            yz;
Subvec2D<type_struct, 4, 1, 3>            yw;
Subvec2D<type_struct, 4, 2, 0>            zx;
Subvec2D<type_struct, 4, 2, 1>            zy;
Subvec2D<type_struct, 4, 2, 3>            zw;
Subvec2D<type_struct, 4, 3, 0>            wx;
Subvec2D<type_struct, 4, 3, 1>            wy;
Subvec2D<type_struct, 4, 3, 2>            wz;
Subvec3D<type_struct, anti, 4, 0, 1, 2>  xyz;
Subvec3D<type_struct, anti, 4, 0, 2, 1>  xzy;
Subvec3D<type_struct, anti, 4, 0, 1, 3>  xyw;
Subvec3D<type_struct, anti, 4, 0, 3, 1>  xwy;
Subvec3D<type_struct, anti, 4, 0, 2, 3>  xzw;
Subvec3D<type_struct, anti, 4, 0, 3, 2>  xwz;
Subvec3D<type_struct, anti, 4, 1, 0, 2>  yxz;
Subvec3D<type_struct, anti, 4, 1, 2, 0>  yzx;

Subvec3D<type_struct, anti, 4, 1, 0, 3>     yxw;
Subvec3D<type_struct, anti, 4, 1, 3, 0>     ywx;
Subvec3D<type_struct, anti, 4, 1, 2, 3>     yzw;
Subvec3D<type_struct, anti, 4, 1, 3, 2>     ywz;
Subvec3D<type_struct, anti, 4, 2, 0, 1>     zxy;
Subvec3D<type_struct, anti, 4, 2, 1, 0>     zyx;
Subvec3D<type_struct, anti, 4, 2, 0, 3>     zxw;
Subvec3D<type_struct, anti, 4, 2, 3, 0>     zwx;
Subvec3D<type_struct, anti, 4, 2, 1, 3>     zyw;
Subvec3D<type_struct, anti, 4, 2, 3, 1>     zwy;
Subvec3D<type_struct, anti, 4, 3, 0, 1>     wxy;
Subvec3D<type_struct, anti, 4, 3, 1, 0>     wyx;
Subvec3D<type_struct, anti, 4, 3, 0, 2>     wxz;
Subvec3D<type_struct, anti, 4, 3, 2, 0>     wzx;
Subvec3D<type_struct, anti, 4, 3, 1, 2>     wyz;
Subvec3D<type_struct, anti, 4, 3, 2, 1>     wzy;
Subvec4D<type_struct, anti, 4, 0, 1, 2, 3>  xyzw;
Subvec4D<type_struct, anti, 4, 0, 1, 3, 2>  xywz;
Subvec4D<type_struct, anti, 4, 0, 2, 1, 3>  xzyw;
Subvec4D<type_struct, anti, 4, 0, 2, 3, 1>  xzwy;
Subvec4D<type_struct, anti, 4, 0, 3, 1, 2>  xwyz;
Subvec4D<type_struct, anti, 4, 0, 3, 2, 1>  xwzy;
Subvec4D<type_struct, anti, 4, 1, 0, 2, 3>  yxzw;
Subvec4D<type_struct, anti, 4, 1, 0, 3, 2>  yxwz;

Subvec4D<type_struct, anti, 4, 1, 2, 0, 3>  yzxw;
Subvec4D<type_struct, anti, 4, 1, 2, 3, 0>  yzwx;
Subvec4D<type_struct, anti, 4, 1, 3, 0, 2>  ywxz;
Subvec4D<type_struct, anti, 4, 1, 3, 2, 0>  ywzx;
Subvec4D<type_struct, anti, 4, 2, 0, 1, 3>  zxyw;
Subvec4D<type_struct, anti, 4, 2, 0, 3, 1>  zxwy;
Subvec4D<type_struct, anti, 4, 2, 1, 0, 3>  zyxw;
Subvec4D<type_struct, anti, 4, 2, 1, 3, 0>  zywx;
Subvec4D<type_struct, anti, 4, 2, 3, 0, 1>  zwxy;
Subvec4D<type_struct, anti, 4, 2, 3, 1, 0>  zwyx;
Subvec4D<type_struct, anti, 4, 3, 0, 1, 2>  wxyz;
Subvec4D<type_struct, anti, 4, 3, 0, 2, 1>  wxzy;
Subvec4D<type_struct, anti, 4, 3, 1, 0, 2>  wyxz;
Subvec4D<type_struct, anti, 4, 3, 1, 2, 0>  wyzx;
Subvec4D<type_struct, anti, 4, 3, 2, 0, 1>  wzxy;
Subvec4D<type_struct, anti, 4, 3, 2, 1, 0>  wzyx;
```

Repeated Components

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

Final Types

- Components and subvectors are generally 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 Bivector3D : public Vec3D<TypeBivector3D, true>
class Trivector4D : public Vec4D<TypeTrivector4D, true>

class Point3D : public Vector3D

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

Antivector Type Structure Example

```
struct TypeTrivector4D
{
	typedef float component_type;
	typedef Vector2D vector2D_type;
	typedef Bivector3D vector3D_type;
	typedef Trivector4D 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 going on
  - 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 is one template parameter
- 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 set 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 to 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;    Subvec4D<row_type_struct, true, 16, 0, 4, 8, 12>         row0;
Component<type_struct, 16, 5>    m11;    Subvec4D<row_type_struct, true, 16, 1, 5, 9, 13>         row1;
Component<type_struct, 16, 6>    m21;    Subvec4D<row_type_struct, true, 16, 2, 6, 10, 14>        row2;
Component<type_struct, 16, 7>    m31;    Subvec4D<row_type_struct, true, 16, 3, 7, 11, 15>        row3;
Component<type_struct, 16, 8>    m02;    Submat3D<type_struct, 16, 0, 4, 8, 1, 5, 9, 2, 6, 10>    matrix3D;
Component<type_struct, 16, 9>    m12;    Submat3D<type_struct, 16, 0, 1, 2, 4, 5, 6, 8, 9, 10>    transpose3D;
Component<type_struct, 16, 10>   m22;
Component<type_struct, 16, 11>   m32;    Submat4D<type_struct, 16, 0, 4, 8, 12, 1, 5, 9, 13,
Component<type_struct, 16, 12>   m03;        2, 6, 10, 14, 3, 7, 11, 15>                          matrix;
Component<type_struct, 16, 13>   m13;    Submat4D<type_struct, 16, 0, 1, 2, 3, 4, 5, 6, 7,
Component<type_struct, 16, 14>   m23;        8, 9, 10, 11, 12, 13, 14, 15>                        transpose;
Component<type_struct, 16, 15>   m33;
```

Final Types

- As with vectors, 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 TypeTrivector4D 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 **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;
```

Matrix Operations

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

Contact

- lengyel@terathon.com
- @EricLengyel on Twitter
- Expo floor, booth #2204

- Slides posted here after conference:

  https://terathon.com/lengyel/