

# Math for Game Programmers: Dual Numbers

**Gino van den Bergen**  
gino@dtecta.com

# Introduction

- Dual numbers extend real numbers, similar to complex numbers.
- Complex numbers adjoin an element  $i$ , for which  $i^2 = -1$ .
- Dual numbers adjoin an element  $\epsilon$ , for which  $\epsilon^2 = 0$ .

# Complex Numbers

- Complex numbers have the form

$$z = a + b i$$

where  $a$  and  $b$  are real numbers.

- $a = \text{real}(z)$  is the real part, and
- $b = \text{imag}(z)$  is the imaginary part.

# Complex Numbers (cont'd)

- Complex operations pretty much follow rules for real operators:

- Addition:

$$(a + b i) + (c + d i) = (a + c) + (b + d) i$$

- Subtraction:

$$(a + b i) - (c + d i) = (a - c) + (b - d) i$$

# Complex Numbers (cont'd)

- Multiplication:

$$(a + b i) (c + d i) = (ac - bd) + (ad + bc) i$$

- Products of imaginary parts feed back into real parts.

# Dual Numbers

- Dual numbers have the form

$$z = a + b \varepsilon$$

similar to complex numbers.

- $a = \text{real}(z)$  is the real part, and
- $b = \text{dual}(z)$  is the dual part.

# Dual Numbers (cont'd)

- Operations are similar to complex numbers, however since  $\varepsilon^2 = 0$ , we have:

$$(a + b \varepsilon) (c + d \varepsilon) = (ac + 0) + (ad + bc)\varepsilon$$

- Dual parts do not feed back into real parts!

# Dual Numbers (cont'd)

- The real part of a dual calculation is independent of the dual parts of the inputs.
- The dual part of a multiplication is a “cross” product of real and dual parts.

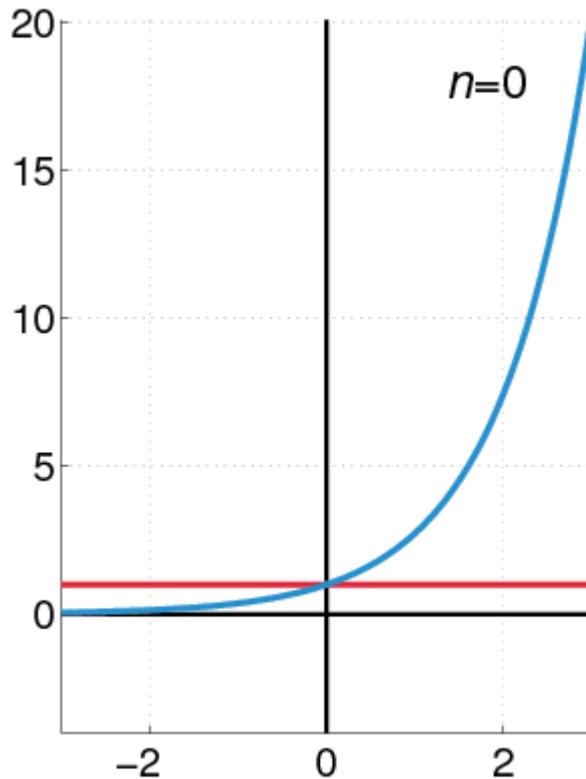
# Taylor Series

- Any value  $f(a + h)$  of a smooth function  $f$  can be expressed as an infinite sum:

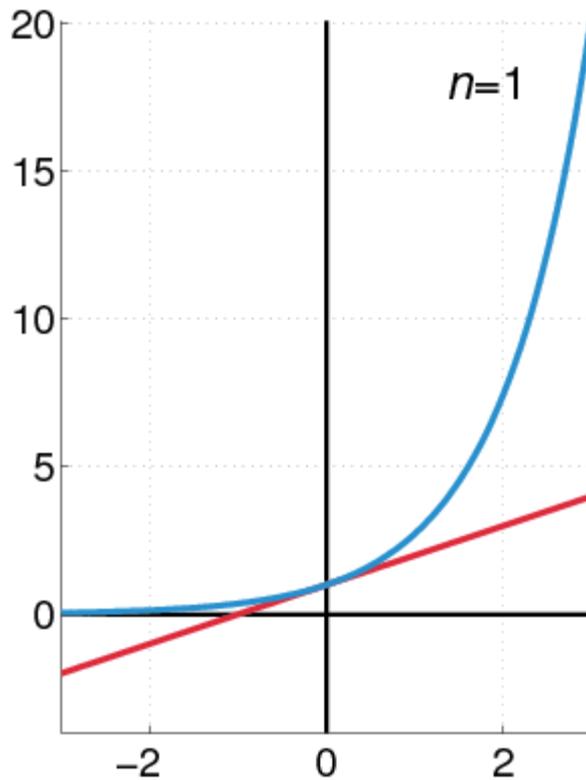
$$f(a + h) = f(a) + \frac{f'(a)}{1!} h + \frac{f''(a)}{2!} h^2 + \dots$$

where  $f'$ ,  $f''$ , ...,  $f^{(n)}$  are the first, second, ...,  $n$ -th derivative of  $f$ .

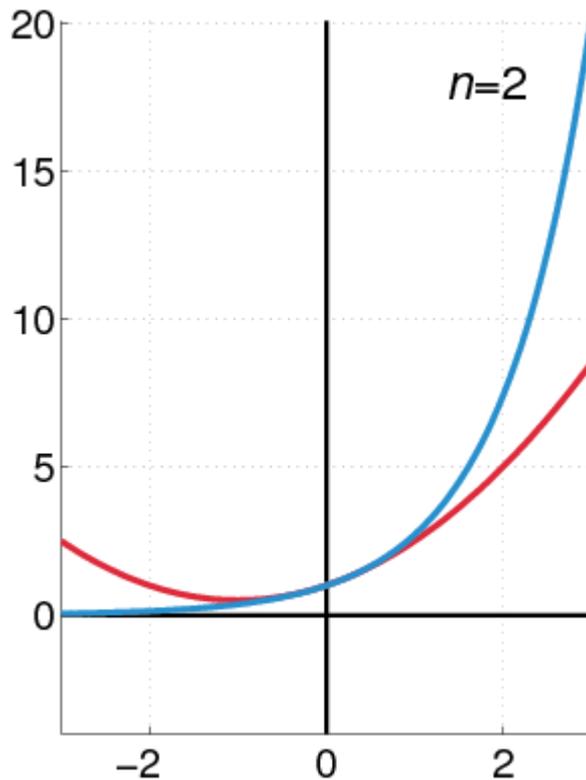
# Taylor Series Example



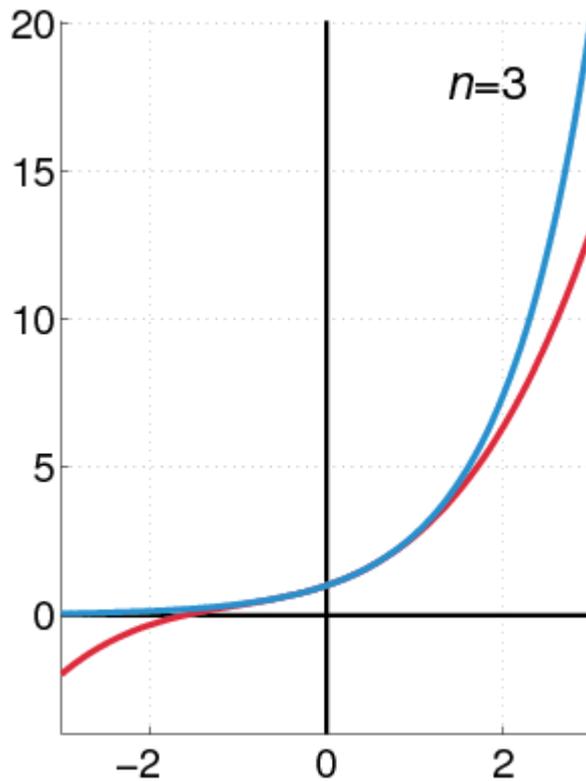
# Taylor Series Example



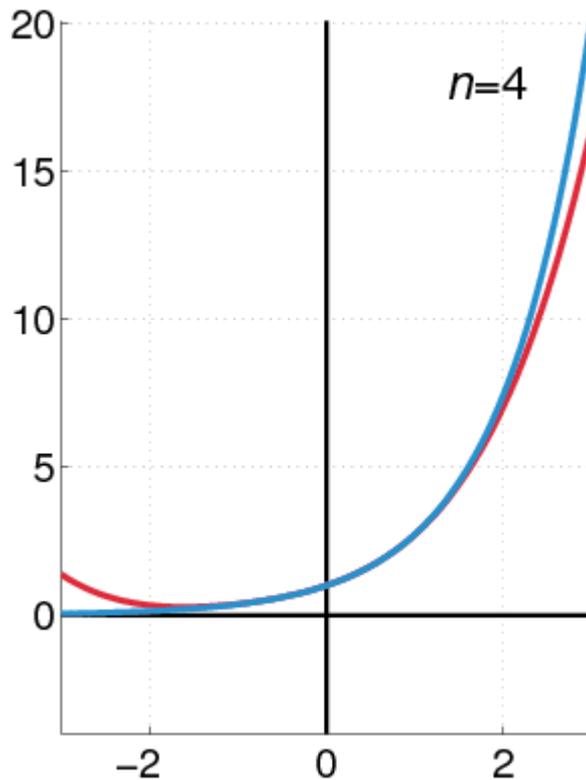
# Taylor Series Example



# Taylor Series Example



# Taylor Series Example



# Taylor Series and Dual Numbers

- For  $f(a + b \varepsilon)$ , the Taylor series is:

$$f(a + b\varepsilon) = f(a) + \frac{f'(a)}{1!} b\varepsilon + \dots 0$$

- All second- and higher-order terms vanish!
- We have a closed-form expression that holds the function and its derivative.

# Real Functions on Dual Numbers

- Any differentiable real function  $f$  can be extended to dual numbers, as:

$$f(a + b \varepsilon) = f(a) + b f'(a) \varepsilon$$

- For example,

$$\sin(a + b \varepsilon) = \sin(a) + b \cos(a) \varepsilon$$

# Automatic Differentiation

- Add a unit dual part to the input value of a real function.
- Evaluate function using dual arithmetic.
- The output has the function value as real part and the derivate's value as dual part:

$$f(a + \varepsilon) = f(a) + f'(a) \varepsilon$$

# How does it work?

- Check out the product rule of differentiation:  $(f \cdot g)' = f \cdot g' + f' \cdot g$
- Notice the “cross” product of functions and their derivatives.

- Recall that

$$(a + a'\varepsilon)(b + b'\varepsilon) = ab + (ab' + a'b)\varepsilon$$

# Automatic Differentiation in C++

- We need some easy way of extending functions on floating-point types to dual numbers...
- ...and we need a type that holds dual numbers and offers operators for performing dual arithmetic.

# Extension by Abstraction

- C++ allows you to abstract from the numerical type through:
  - Typedefs
  - Function templates
  - Constructors and conversion operators
  - Overloading
  - Traits class templates

# Abstract Scalar Type

- Never use built-in floating-point types, such as `float` or `double`, explicitly.
- Instead use a type name, e.g. `Scalar`, either as template parameter or as typedef,

```
typedef float Scalar;
```

# Constructors

- Built-in types have constructors as well:
  - Default: `float() == 0.0f`
  - Conversion: `float(2) == 2.0f`
- Use constructors for defining constants, e.g. use `Scalar(2)`, rather than `2.0f` or `(Scalar)2`.

# Overloading

- Operators and functions on built-in types can be overloaded in numerical classes, such as `std::complex`.
- Built-in types support operators: `+`, `-`, `*`, `/`
- ...and functions: `sqrt`, `pow`, `sin`, ...
- NB: Use `<cmath>` rather than `<math.h>`.  
That is, use `sqrt` NOT `sqrtf` on floats.

# Traits Class Templates

- Type-dependent constants, such as the machine epsilon, are obtained through a traits class defined in `<limits>`.
- Use `std::numeric_limits<Scalar>::epsilon()` rather than `FLT_EPSILON` in C++.
- Either specialize `std::numeric_limits` for your numerical classes or write your own traits class.

# Example Code (before)

```
float smoothstep(float x)
{
    if (x < 0.0f)
        x = 0.0f;
    else if (x > 1.0f)
        x = 1.0f;
    return (3.0f - 2.0f * x) * x * x;
}
```

# Example Code (after)

```
template <typename T>
T smoothstep(T x)
{
    if (x < T())
        x = T();
    else if (x > T(1))
        x = T(1);
    return (T(3) - T(2) * x) * x * x;
}
```

# Dual Numbers in C++

- C++ has a standard class template `std::complex<T>` for complex numbers.
- We create a similar class template `Dual<T>` for dual numbers.
- `Dual<T>` defines constructors, accessors, operators, and standard math functions.

# Dual<T>

```
template <typename T>
class Dual
{
...
private:
    T mReal;
    T mDual;
};
```

# Dual<T>: Constructor

```
template <typename T>
Dual<T>::Dual(T real = T(), T dual = T())
    : mReal(real)
    , mDual(dual)
{}

```

```
...
Dual<Scalar> z1; // zero initialized
Dual<Scalar> z2(2); // zero dual part
Dual<Scalar> z3(2, 1);

```

# Dual<T>: operators

```
template <typename T>
Dual<T> operator* (Dual<T> a, Dual<T> b)
{
    return Dual<T>(
        a.real() * b.real(),
        a.real() * b.dual() +
        a.dual() * b.real()
    );
}
```

# Dual<T>: Standard Math

```
template <typename T>
Dual<T> sqrt(Dual<T> z)
{
    T tmp = sqrt(z.real());
    return Dual<T>(
        tmp,
        z.dual() * T(0.5) / tmp
    );
}
```

# Curve Tangent

- For a 3D curve

$$\mathbf{p}(t) = (x(t), y(t), z(t)), \quad \text{where } t \in [a, b]$$

The tangent is

$$\frac{\mathbf{p}'(t)}{\|\mathbf{p}'(t)\|}, \quad \text{where } \mathbf{p}'(t) = (x'(t), y'(t), z'(t))$$

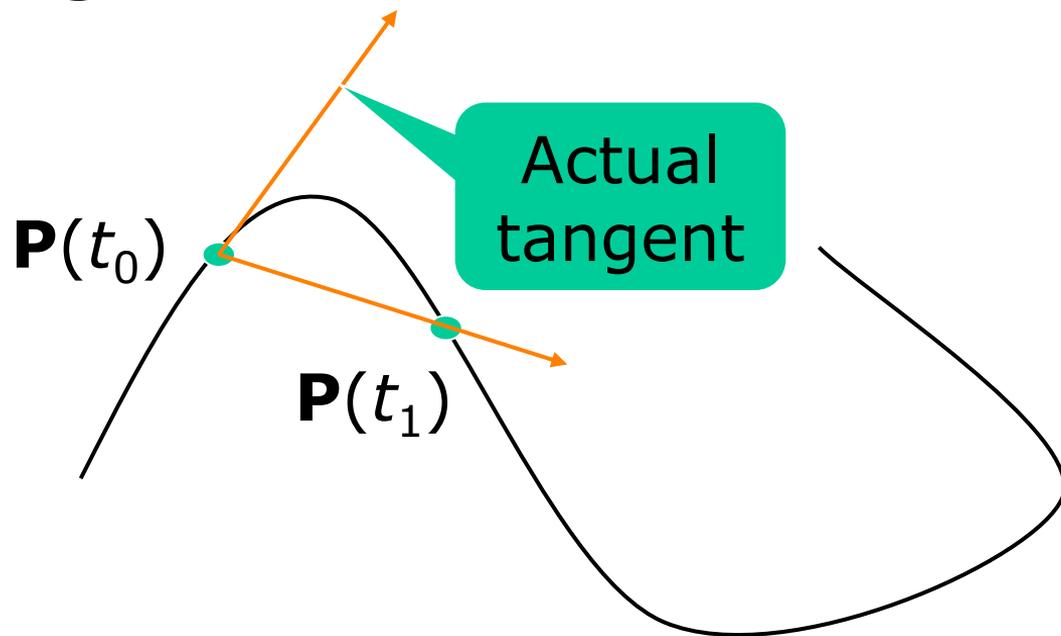
# Curve Tangent

- Curve tangents are often computed by approximation:

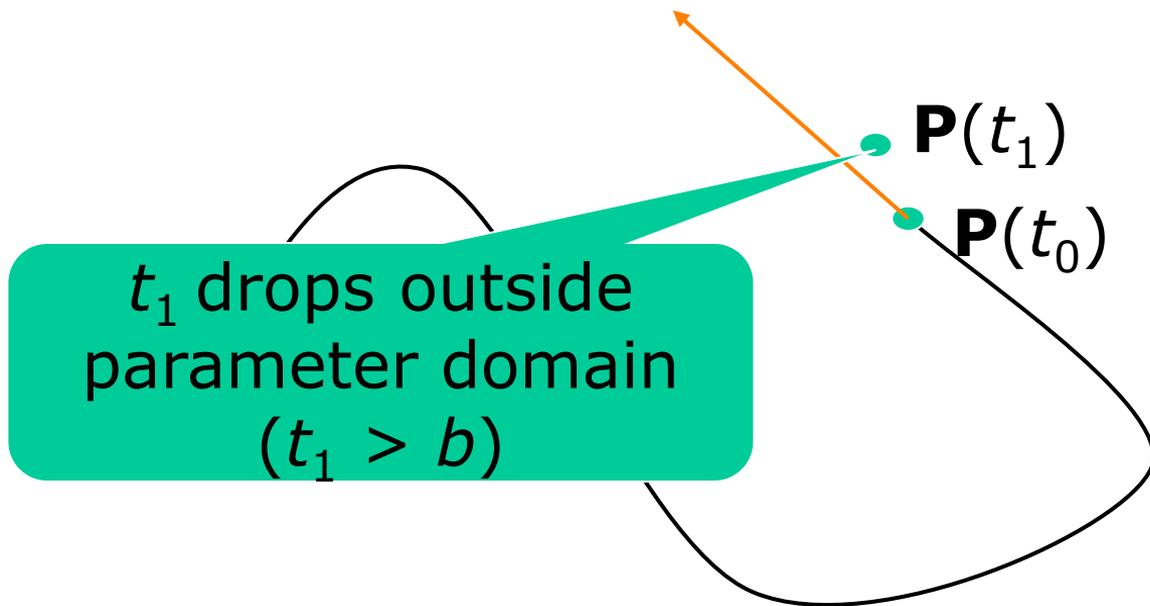
$$\frac{\mathbf{p}(t_1) - \mathbf{p}(t_0)}{\|\mathbf{p}(t_1) - \mathbf{p}(t_0)\|}, \quad \text{where } t_1 = t_0 + h$$

for tiny values of  $h$ .

# Curve Tangent: Bad #1



# Curve Tangent: Bad #2



# Curve Tangent: Duals

- Make a curve function template using a class template for 3D vectors:

```
template <typename T>  
Vector3<T> curveFunc (T x) ;
```

# Curve Tangent: Duals (cont'd)

- Call the curve function using a dual number  $x = \text{Dual}\langle\text{Scalar}\rangle(t, 1)$ , (add  $\varepsilon$  to *parameter*  $t$ ):

```
Vector3<Dual<Scalar> > y =  
    curveFunc(Dual<Scalar>(t, 1));
```

# Curve Tangent: Duals (cont'd)

- The real part is the evaluated position:

```
Vector3<Scalar> position = real(y);
```

- The normalized dual part is the tangent at this position:

```
Vector3<Scalar> tangent =  
    normalize(dual(y));
```

# Line Geometry

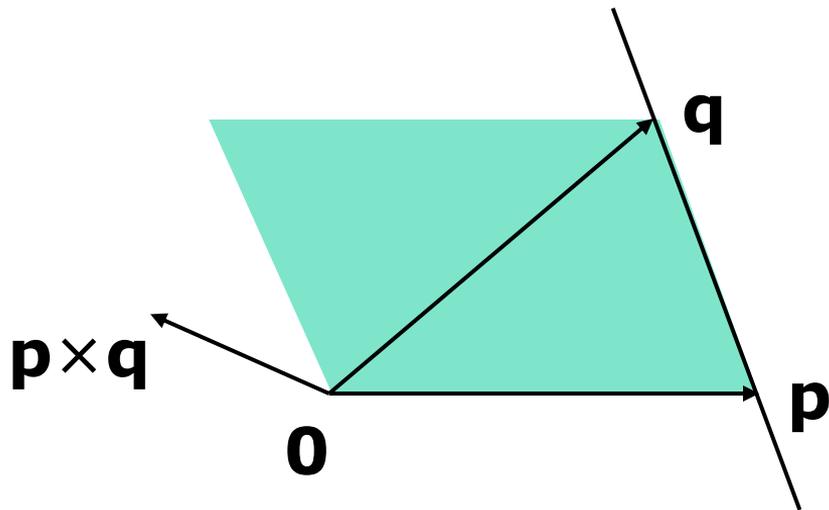
- The line through points  $\mathbf{p}$  and  $\mathbf{q}$  can be expressed explicitly as:

$$\mathbf{x}(t) = \mathbf{p} + (\mathbf{q} - \mathbf{p})t, \text{ and}$$

- Implicitly, as a set of points  $\mathbf{x}$  for which:

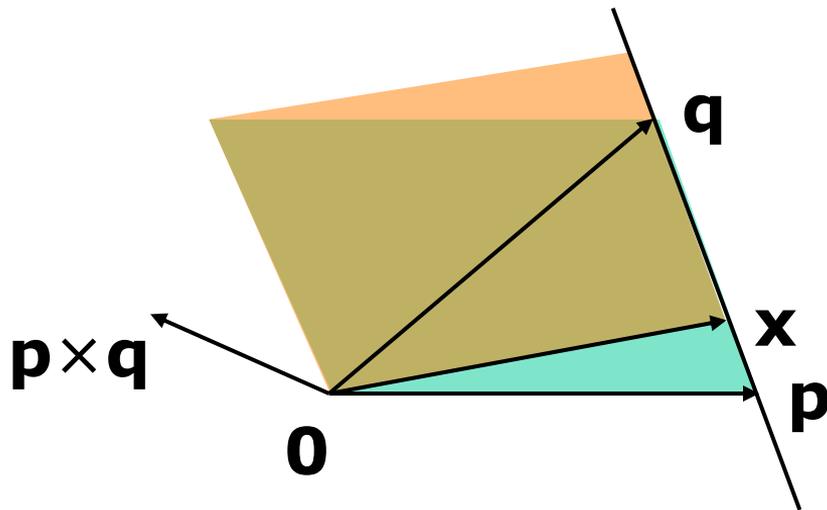
$$(\mathbf{q} - \mathbf{p}) \times \mathbf{x} + \mathbf{p} \times \mathbf{q} = \mathbf{0}$$

# Line Geometry



$\mathbf{p} \times \mathbf{q}$  is orthogonal to the plane  $\mathbf{opq}$ , and its length is equal to the area of the parallelogram spanned by  $\mathbf{p}$  and  $\mathbf{q}$

# Line Geometry



All points  $\mathbf{x}$  on the line  $\mathbf{pq}$  span with  $\mathbf{q} - \mathbf{p}$  a parallelogram that has the same area and orientation as the one spanned by  $\mathbf{p}$  and  $\mathbf{q}$ .

# Plücker Coordinates

- Plücker coordinates are 6-tuples of the form  $(u_x, u_y, u_z, v_x, v_y, v_z)$ , where

$$\mathbf{u} = (u_x, u_y, u_z) = \mathbf{q} - \mathbf{p}, \text{ and}$$

$$\mathbf{v} = (v_x, v_y, v_z) = \mathbf{p} \times \mathbf{q}$$

# Plücker Coordinates (cont'd)

- For  $(\mathbf{u}_1:\mathbf{v}_1)$  and  $(\mathbf{u}_2:\mathbf{v}_2)$  directed lines, if

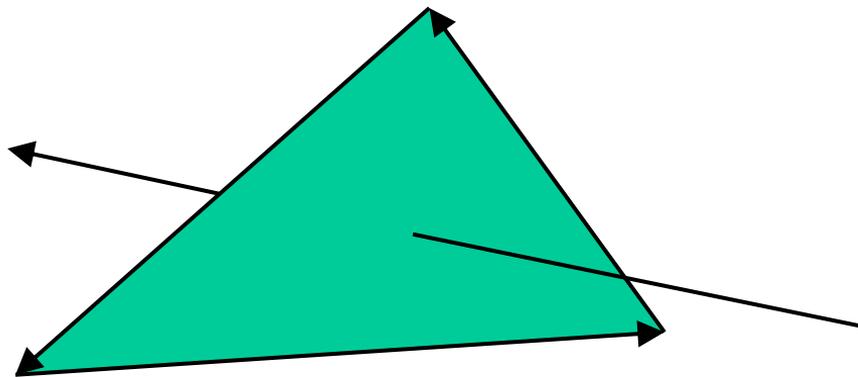
$$\mathbf{u}_1 \bullet \mathbf{v}_2 + \mathbf{v}_1 \bullet \mathbf{u}_2 \quad \text{is}$$

zero:           the lines intersect

positive:       the lines cross right-handed

negative:       the lines cross left-handed

# Triangle vs. Ray



If the signs of permuted dot products of the ray and edges are all equal, then the ray intersects the triangle.

# Plücker Coordinates and Duals

- Dual 3D vectors conveniently represent Plücker coordinates:

`Vector3<Dual<Scalar> >`

- For a line ( $\mathbf{u}:\mathbf{v}$ ),  $\mathbf{u}$  is the real part and  $\mathbf{v}$  is the dual part.

# Dot Product of Dual Vectors

- The dot product of dual vectors  $\mathbf{u}_1 + \mathbf{v}_1\varepsilon$  and  $\mathbf{u}_2 + \mathbf{v}_2\varepsilon$  is a dual number  $z$ , for which

$$\text{real}(z) = \mathbf{u}_1 \bullet \mathbf{u}_2, \text{ and}$$

$$\text{dual}(z) = \mathbf{u}_1 \bullet \mathbf{v}_2 + \mathbf{v}_1 \bullet \mathbf{u}_2$$

- The dual part is the permuted dot product

# Angle of Dual Vectors

- For **a** and **b** dual vectors, we have

$$\theta + d\varepsilon = \arccos\left(\frac{\mathbf{a} \bullet \mathbf{b}}{\|\mathbf{a}\| \|\mathbf{b}\|}\right)$$

where  $\theta$  is the angle and  $d$  is the signed distance between the lines **a** and **b**.

# Translation

- Translation of lines only affects the dual part. Translation of line **pq** over **c** gives:
  - Real:  $(\mathbf{q} + \mathbf{c}) - (\mathbf{p} + \mathbf{c}) = \mathbf{q} - \mathbf{p}$
  - Dual:  $(\mathbf{p} + \mathbf{c}) \times (\mathbf{q} + \mathbf{c})$   
 $= \mathbf{p} \times \mathbf{q} + \mathbf{c} \times (\mathbf{q} - \mathbf{p})$
  - **q - p** pops up in the dual part!

# Rotation

- Real and dual parts are rotated in the same way. For a rotation matrix **R**:
- Real:  **$\mathbf{Rq} - \mathbf{Rp} = \mathbf{R}(\mathbf{q} - \mathbf{p})$**
- Dual:  **$\mathbf{Rp} \times \mathbf{Rq} = \mathbf{R}(\mathbf{p} \times \mathbf{q})$**
- The latter holds for rotations only! That is, **R** performs no scaling or reflection.

# Rigid-Body Transform

- For rotation matrix  $\mathbf{R}$  and translation vector  $\mathbf{c}$ , the dual  $3 \times 3$  matrix  $\mathbf{M}$  with

$\text{real}(\mathbf{M}) = \mathbf{R}$ , and

$$\text{dual}(\mathbf{M}) = [\mathbf{c}]_{\times} \mathbf{R} = \begin{bmatrix} 0 & -c_z & c_y \\ c_z & 0 & -c_x \\ -c_y & c_x & 0 \end{bmatrix} \mathbf{R}$$

maps Plücker coordinates to the new reference frame.

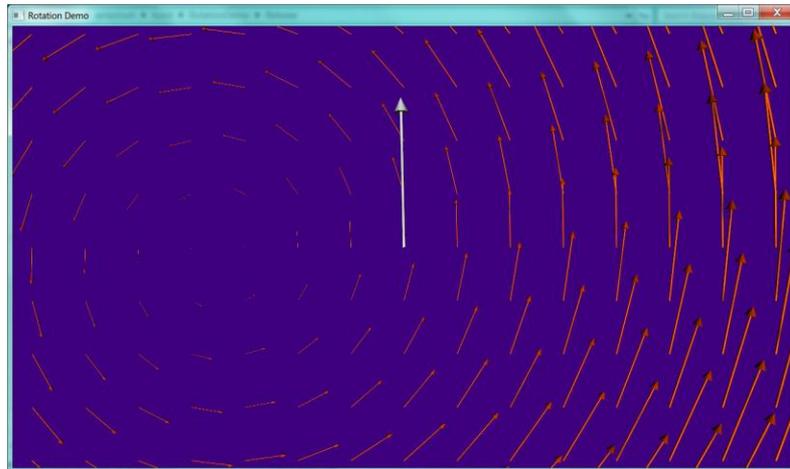
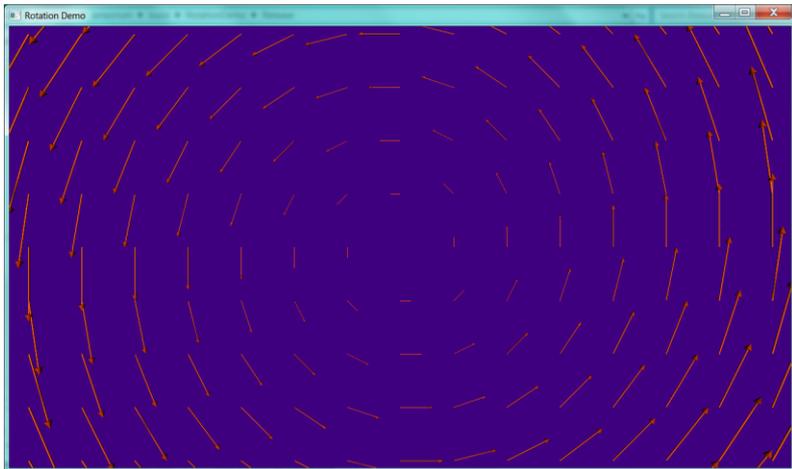
# Screw Theory

- A screw motion is a rotation about a line and a translation along the same line.
- *"Any rigid body displacement can be defined by a screw motion."* (Chasles)

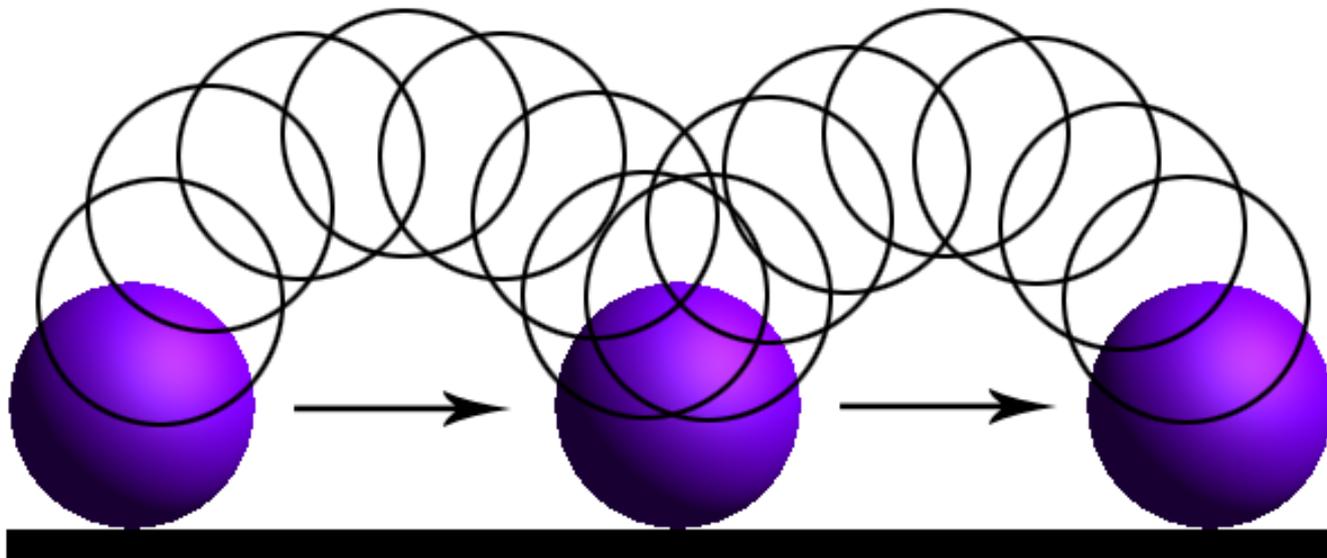
# Chasles' Theorem (Sketchy Proof)

- Decompose translation into a term along the line and a term orthogonal to the line.
- Translation orthogonal to the axis of rotation offsets the axis.
- Translation along the axis does not care about the position of the axis.

# Translations Orthogonal to Axis



# Example: Rolling Ball



# Dual Quaternions

- Unit dual quaternions represent screw motions.
- The rigid body transform over a unit quaternion  $\mathbf{q}$  and vector  $\mathbf{t}$  is:

$$\mathbf{q} + \frac{1}{2} \mathbf{t} \mathbf{q} \varepsilon$$

Here,  $\mathbf{t}$  is a quaternion with zero scalar part.

# Where is the Screw?

- A unit dual quaternion can be written as

$$\cos\left(\frac{\theta + d\varepsilon}{2}\right) + \sin\left(\frac{\theta + d\varepsilon}{2}\right)(\mathbf{u} + \mathbf{v}\varepsilon)$$

where  $\theta$  is the rotation angle,  $d$ , the translation distance, and  $\mathbf{u} + \mathbf{v}\varepsilon$ , the line given in Plücker coordinates.

# Rigid-Body Transform Revisited

- Similar to 3D vectors, Plücker coordinates can be transformed using dual quaternions.
- The mapping of a dual vector  $\mathbf{v}$  according to a screw motion  $\mathbf{q}$  is

$$\mathbf{v}' = \mathbf{q} \mathbf{v} \mathbf{q}^*$$

# Traditional Skinning

- Bones are defined by transformation matrices  $\mathbf{T}_i$  relative to the rest pose.
- Each vertex is transformed as

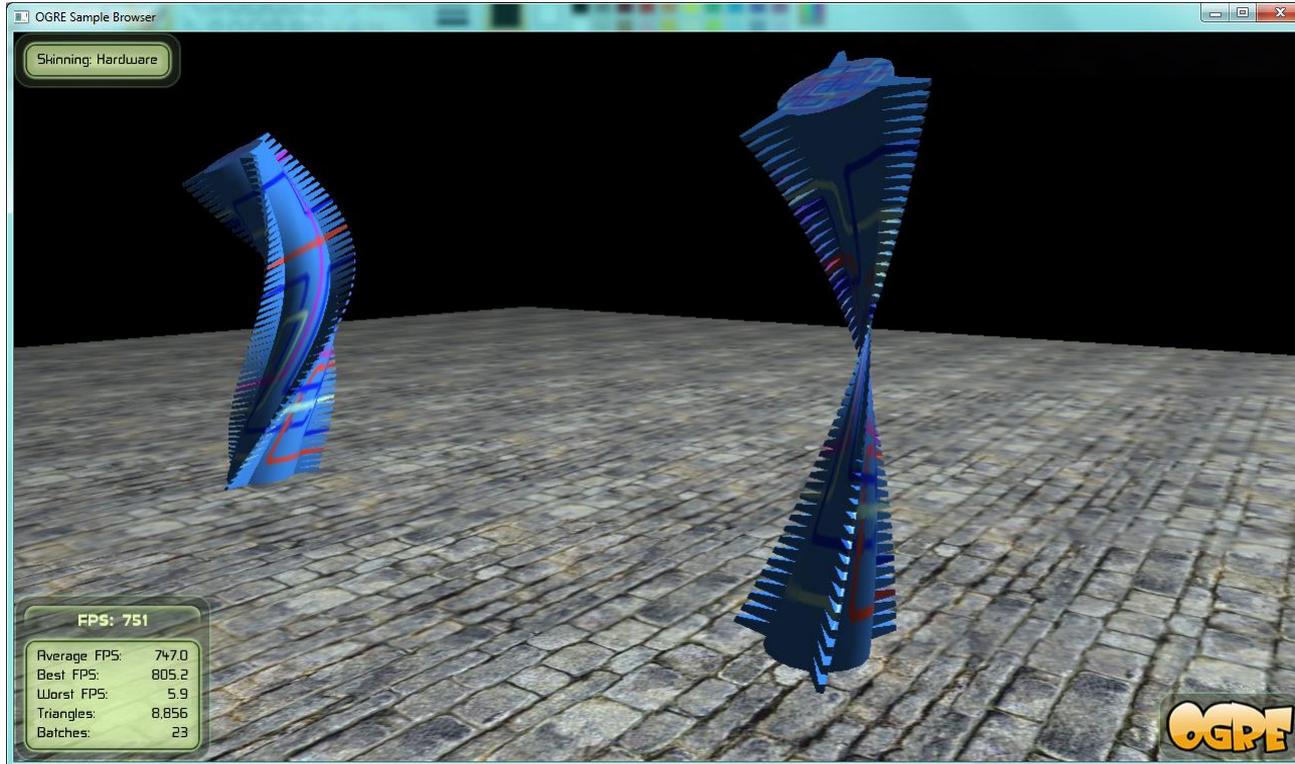
$$\mathbf{p}' = \lambda_1 \mathbf{T}_1 \mathbf{p} + \dots + \lambda_n \mathbf{T}_n \mathbf{p} = (\lambda_1 \mathbf{T}_1 + \dots + \lambda_n \mathbf{T}_n) \mathbf{p}$$

Here,  $\lambda_i$  are blend weights.

# Traditional Skinning (cont'd)

- A weighted sum of matrices is not necessarily a rigid-body transformation.
- Most notable artifact is “candy wrapper”:  
The skin collapses while transiting from one bone to the other.

# Candy Wrapper



# Dual Quaternion Skinning

- Use a blend operation that always returns a rigid-body transformation.
- Several options exist. The simplest one is a normalized lerp of dual quaternions:

$$\mathbf{q} = \frac{\lambda_1 \mathbf{q}_1 + \dots + \lambda_n \mathbf{q}_n}{\|\lambda_1 \mathbf{q}_1 + \dots + \lambda_n \mathbf{q}_n\|}$$

# Dual Quaternion Skinning (cont'd)

- Can the weighted sum of dual quaternions ever get zero?
- Not if all dual quaternions lie in the same hemisphere.
- Observe that  $\mathbf{q}$  and  $-\mathbf{q}$  are the same pose. If necessary, negate each  $\mathbf{q}_i$  to dot positively with  $\mathbf{q}_0$ .

# Further Uses

- **Motor Algebra:** Linear and angular velocity of a rigid body combined in a dual 3D vector.
- **Spatial Vector Algebra:** Featherstone uses 6D vectors for representing velocities and forces in robot dynamics.

# Conclusions

- Abstract from numerical types in your C++ code.
- Differentiation is easy, fast, and exact with dual numbers.
- Dual numbers have other uses as well. Explore yourself!

# References

- D. Vandevoorde and N. M. Josuttis. *C++ Templates: The Complete Guide*. Addison-Wesley, 2003.
- K. Shoemake. *Plücker Coordinate Tutorial*. [Ray Tracing News, Vol. 11, No. 1](#)
- R. Featherstone. *Robot Dynamics Algorithms*. Kluwer Academic Publishers, 1987.
- L. Kavan et al. Skinning with dual quaternions. *Proc. ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, 2007

# Thank You!

- For sample code, check out free\* MoTo C++ template library on:

`https://code.google.com/p/motion-toolkit/`

(\* ) gratis (as in "free beer") and libre (as in "free speech")