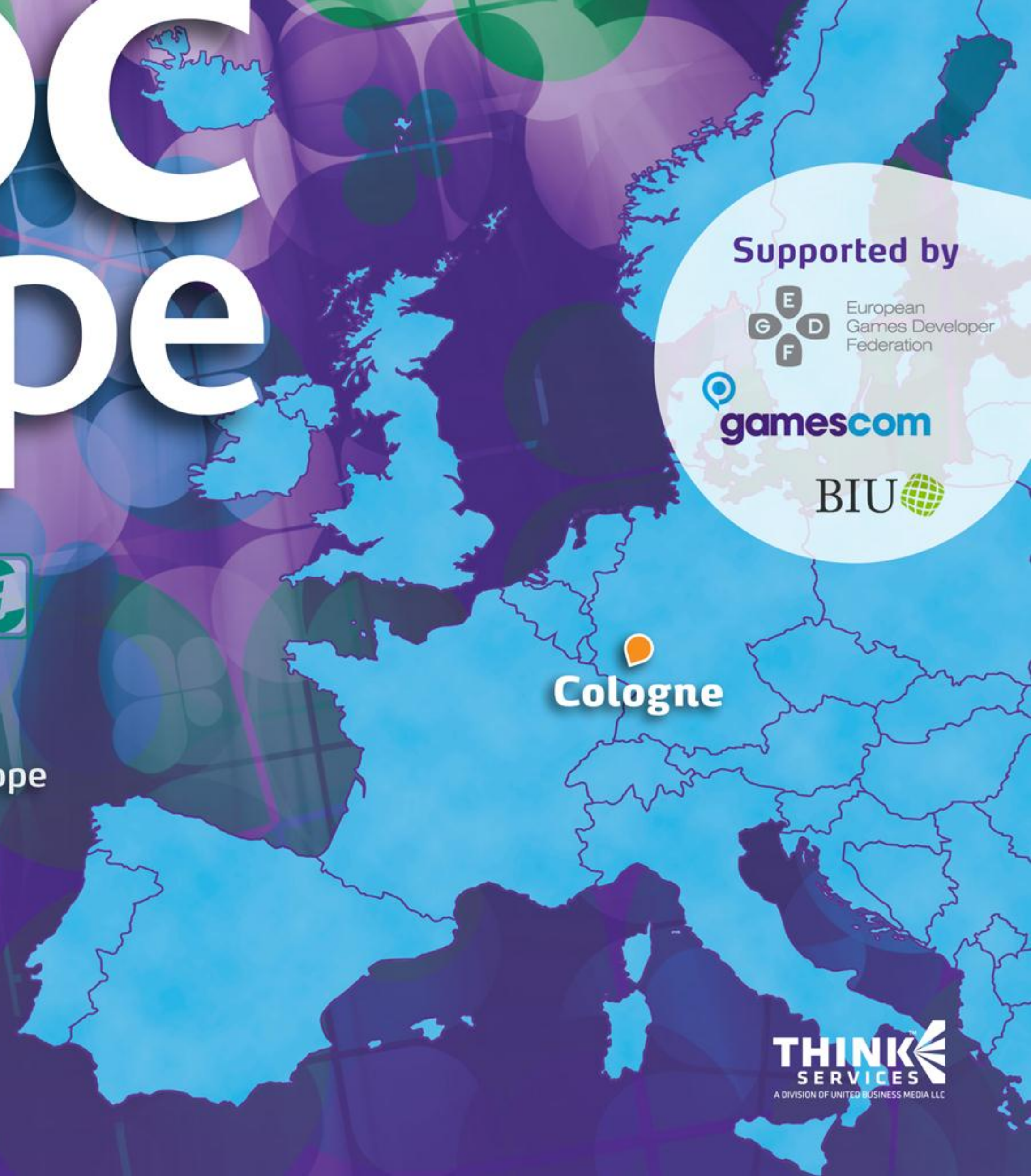
GDC 09 Europe

www.GDCEurope.com

August 17–19, 2009

Game Developers Conference® Europe
Cologne Congress Center East
Cologne, Germany



Supported by

European Games Developer Federation

gamescom

BIU

THINK SERVICES
A DIVISION OF UNITED BUSINESS MEDIA LLC

Dual Numbers: Simple Math, Easy C++ Coding, and Lots of Tricks

Gino van den Bergen
gino@dtecta.com

Introduction

- Dual numbers extend the real numbers, similar to complex numbers.
- Complex numbers adjoin a new element i, for which $i^2 = -1$.
- Dual numbers adjoin a new element ε, for which $\varepsilon^2 = 0$.

Complex Numbers

- Complex numbers have the form

$$z = a + b\,i$$

where a and b are real numbers.
- a = real(z) is the real part, and
- b = 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.
- $a = \mathrm{real}(z)$ is the real part, and
- $b = \mathrm{dual}(z)$ 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.

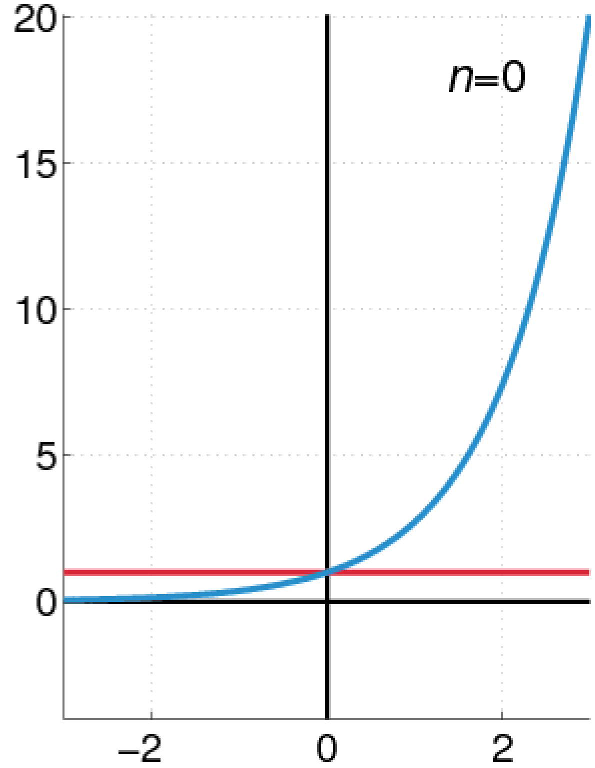
Taylor Series

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

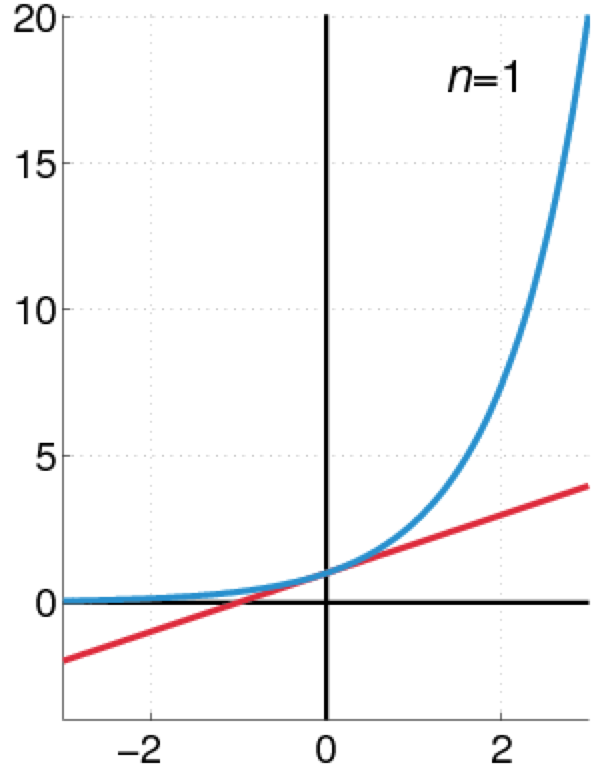
$$f(a+h) = f(a) + \frac{f'(a)}{1!}h + \frac{f''(a)}{2!}h^2 + \cdots$$

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

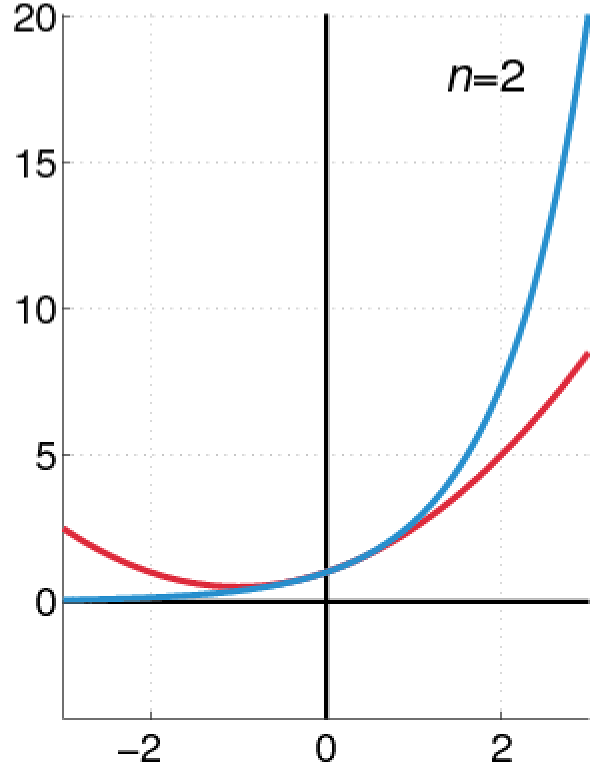
Taylor Series Example



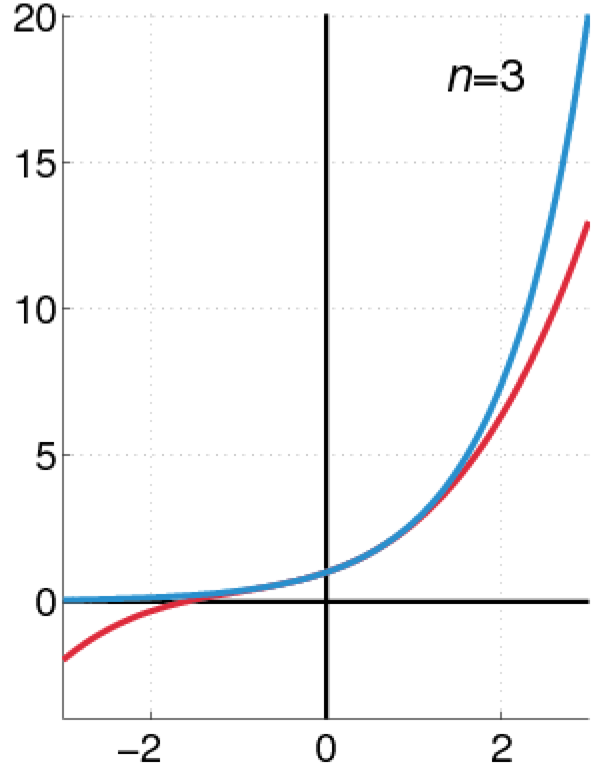
Taylor Series Example



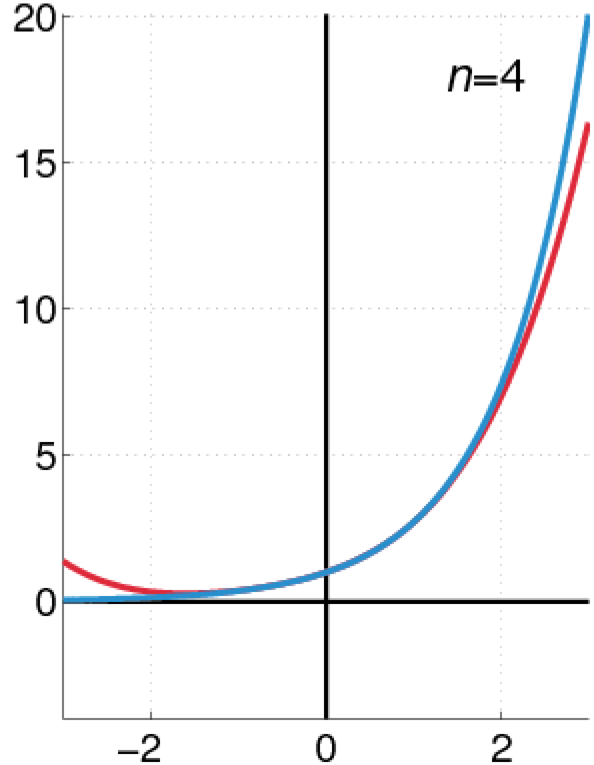
Taylor Series Example



Taylor Series Example



Taylor Series Example



Taylor Series and Dual Numbers

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

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

- 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 can be extended to dual numbers:

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

- For example,

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

Compute Derivatives

- 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:

$$\frac{d}{dx}(f(x)\cdot g(x)) = f(x)\cdot g'(x) + f'(x)\cdot g(x)$$

  Notice the "cross" product of functions and 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 (conversion)
  - Overloading
  - Traits class templates

Abstract Scalar Type

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

```
typedef float Scalar;
```

Constructors

- Primitive 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 primitive types can be overloaded in hand-baked classes, e.g. `std::complex`.
- Primitive types use 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, e.g. machine epsilon, are obtained through a traits class defined in `<limits>`.
- Use `std::numeric_limits<T>::epsilon()` rather than `FLT_EPSILON`.
- Either specialize this traits template for hand-baked classes or create your own traits class template.

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++ stdlib has a 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
{
public:
…
T real() const { return m_re; }
T dual() const { return m_du; }
…
private:
	T m_re;
	T m_du;
};
```

Dual<T>: Constructor

```
template <typename T>
Dual<T>::Dual(T re = T(), T du = T())
    : m_re(re)
    , m_du(du)
{}
```

…

```
Dual<float> z1; // zero initialized
Dual<float> z2(2); // zero dual part
Dual<float> 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>: operators (Cont’d)

- We also need these

```
template <typename T>
Dual<T> operator*(Dual<T> a, T b);

template <typename T>
Dual<T> operator*(T a, Dual<T> b);
```

since template argument deduction does not perform implicit type conversions.

Dual<T>: Standard Math

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

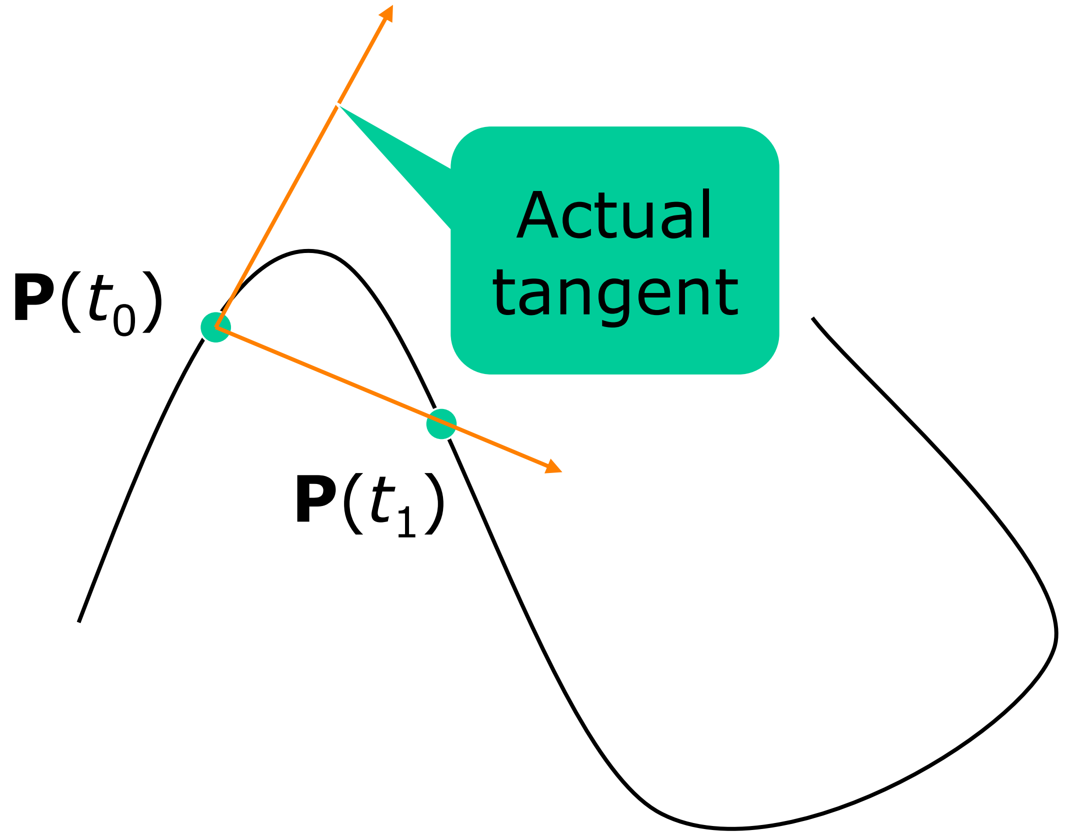
Curve Tangent Example

- Curve tangents are often computed by approximation:

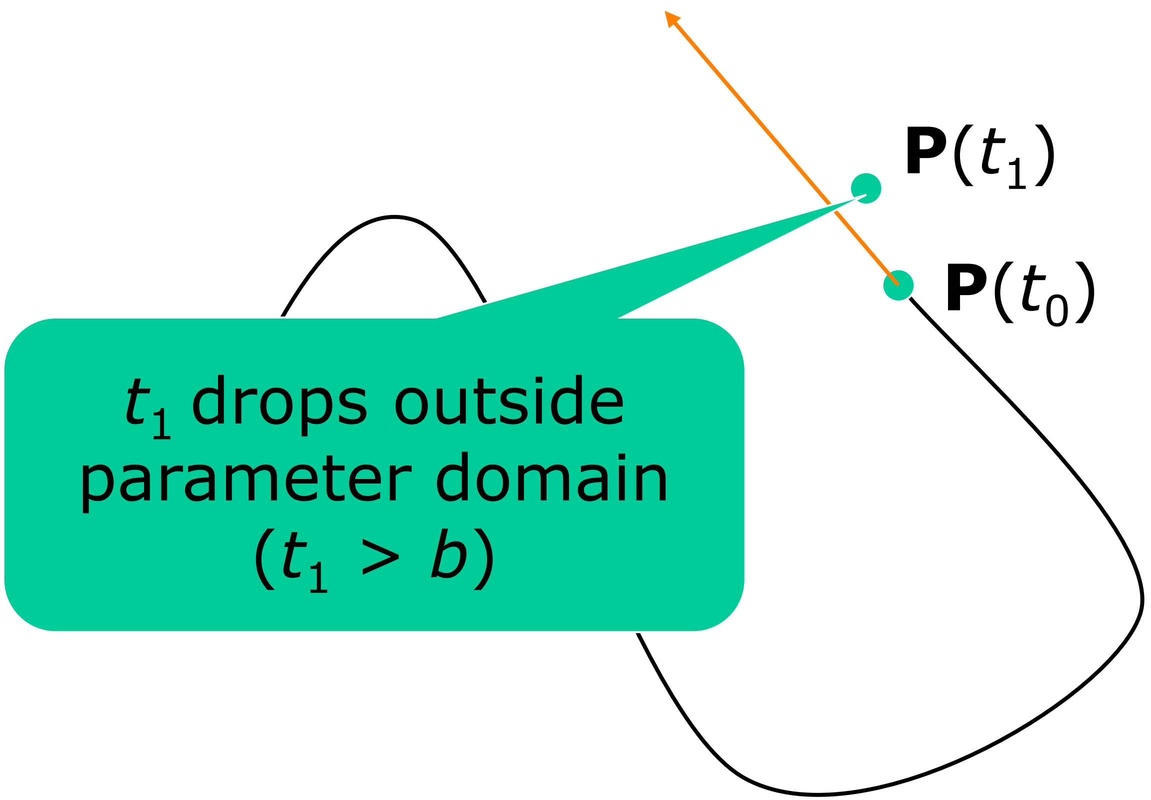
$$\frac{\mathbf{p}(t_1)-\mathbf{p}(t_0)}{\left\|\mathbf{p}(t_1)-\mathbf{p}(t_0)\right\|}, \quad where \quad t_1 = t_0 + h$$

for tiny values of *h*.

Curve Tangent Example: Approximation (Bad #1)



Curve Tangent Example: Approximation (Bad #2)



Curve Tangent Example: Analytic Approach

- For a 3D curve

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

the tangent is

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

Curve Tangent Example: Dual Numbers

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

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

- Call the curve function on `Dual<Scalar>(t, 1)` rather than `t`:

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

Curve Tangent Example: Dual Numbers (Cont'd)

- The evaluated point is the real part of the result:

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

- The tangent at this point is the dual part of the result after normalization:

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

Line Geometry

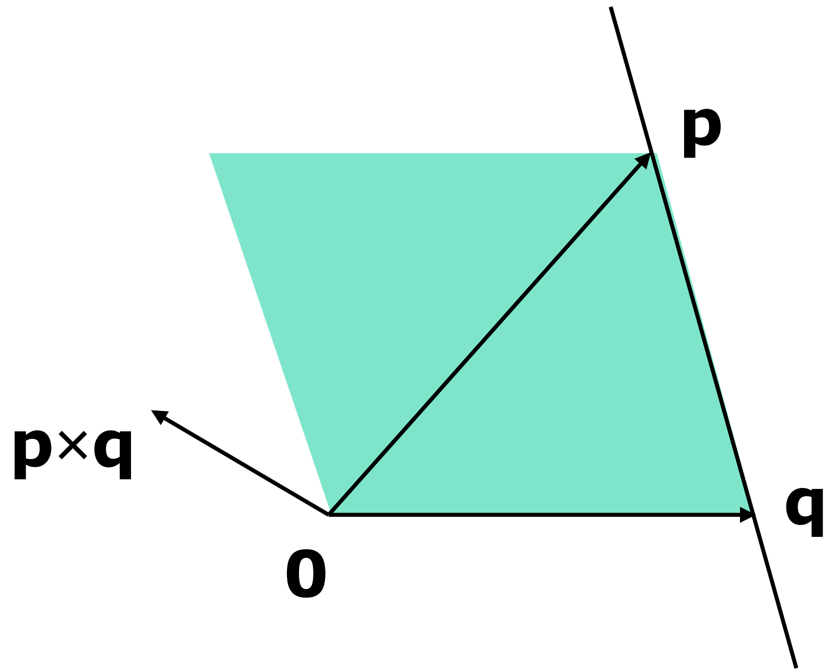
- The line through points **p** and **q** can be expressed:
- Explicitly,

$$\mathbf{x}(t) = \mathbf{p}\,t + \mathbf{q}(1 - t)$$

- Implicitly, as a set of points **x** for which:

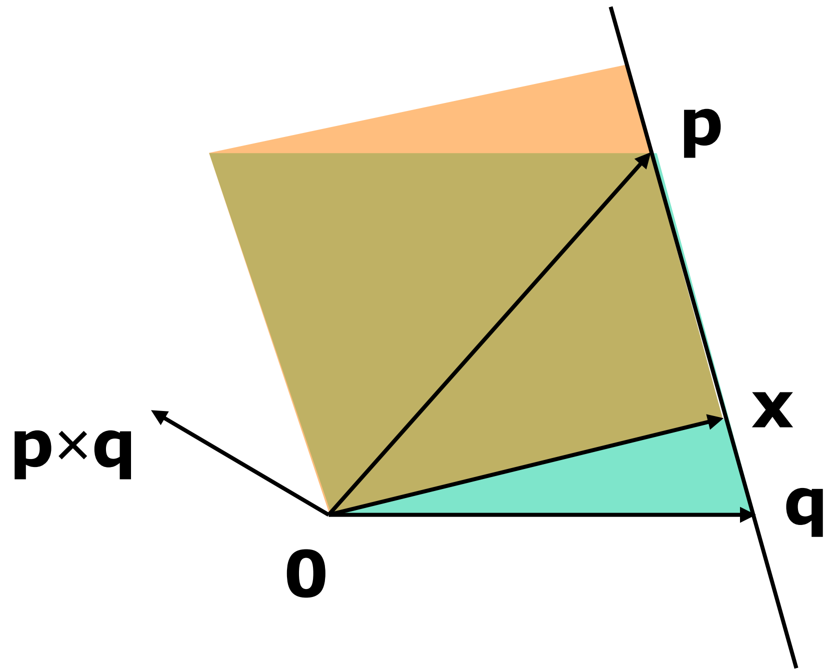
$$(\mathbf{p} - \mathbf{q}) \times \mathbf{x} = \mathbf{p} \times \mathbf{q}$$

Line Geometry



- **p** × **q** is orthogonal to the plane **opq**, and its length is equal to the area of the parallellogram spanned by **p** and **q**.

Line Geometry



- All points **x** on the line **pq** span with **p** – **q** a parallellogram that has equal area and orientation as the one spanned by **p** and **q**.

Plücker Coordinates

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

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

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

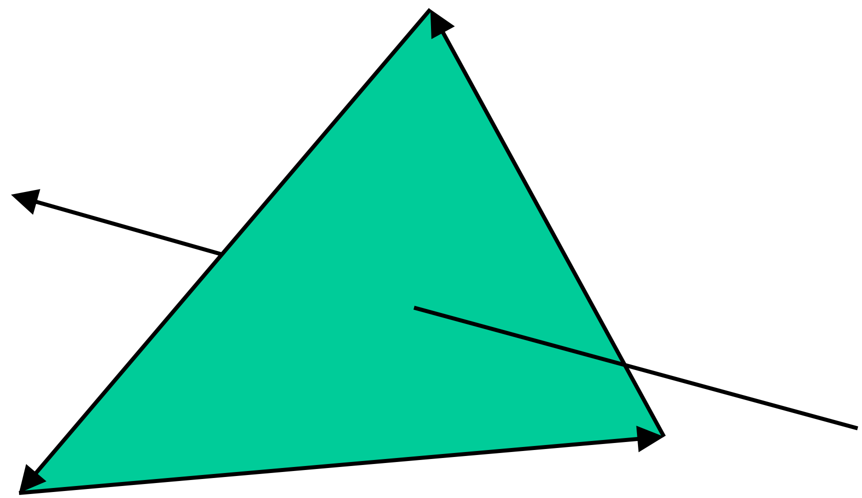
Plücker Coordinates (Cont'd)

- Main use in graphics is for determining line-line orientations.
- 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 the edges are all equal, then the ray intersects the triangle.

Plücker Coordinates and Dual Numbers

- Dual 3D vectors conveniently represent Plücker coordinates:

```
Vector3<Dual<Scalar> >
```

- For a line (**u**:**v**), **u** is the real part and **v** is the dual part.

Plücker Coordinates and Dual Numbers (Cont'd)

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

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

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

- The dual part is the permuted dot product.

Translation

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

Translation (Cont'd)

- Create a dual 3×3 matrix **T**, for which

  real(**T**) = **I**, the identity matrix, and

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

- Translation is performed by multiplying this dual matrix with the dual vector.

Rotation

- Real and dual parts are rotated in the same way. For a matrix $\mathbf{R}$:
- Real: $\mathbf{Rp} - \mathbf{Rq} = \mathbf{R}(\mathbf{p} - \mathbf{q})$
- Dual: $\mathbf{Rp} \times \mathbf{Rq} = \mathbf{R}(\mathbf{p} \times \mathbf{q})$
- The latter is only true for rotation matrices!

Rigid-Body Motion

- For rotation matrix **R** and translation vector **c**, the dual 3×3 matrix **M** = [**I**:-[**c**]$_\times$]**R**, i.e.,

  real(**M**) = **R**, and

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

  maps Plücker coordinates to the new reference frame.

Further Reading

- **Motor Algebra**: Linear and angular velocity of a rigid body combined in a dual 3D vector.
- **Screw Theory**: Any rigid motion can be expressed as a screw motion, which is represented by a dual quaternion.
- **Spatial Vector Algebra**: Featherstone uses 6D vectors for representing velocities and forces in robot dynamics.

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

Conclusions

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

Thank You!

- Check out sample code soon to be released on:

  `http://www.dtecta.com`