Automatic Differentiation, C++ Templates and Photogrammetry

Dan Piponi
ESC Entertainment
To be published in The Journal of Graphics Tools

September 13, 2004

Abstract

Differential calculus is ubiquitous in digital movie production. We give a novel presentation of *automatic differentiation*, a method for computing derivatives of functions, that is not well known within the graphics community and describe some applications of this method. In particular we describe the implementation of a photogrammetric reconstruction tool used on the post-production of *Matrix Reloaded* and *Matrix Revolutions* that was built using automatic differentiation.

1 Introduction

Differential calculus has many applications in digital movie production. Its applications include rendering (anti-aliasing [Ige99], motion-blurring [Duf02]), animation (inverse-kinematics, match-moving), modelling (determination of normals, photogrammetry [TMHF00]), simulation (implicit integration of equations of motion [BW98]) and other miscellaneous applications (lens and colour calibration and other kinds of model fitting problem, optical flow estimation [HS81]). There are a number of methods for writing software to compute derivatives of functions and their approximations: deriving expressions analytically by hand that are subsequently implemented as expressions in code, computer algebra and finite differences. However, there is also another technique: *automatic differentiation*. Unfortunately this approach is not well known within the graphics community though there are some scattered references ([MH92],[Gle97],[HS98]). It provides a way to compute multiple derivatives of highly complex functions efficiently and accurately. In this paper we will present one approach to automatic differentiation and describe one application that was used with considerable success during the post-production of *Matrix Reloaded* and *Revolutions*.

2 Theory

By definition, the derivative of a function f, written $f'(x)$ or $\frac{df}{dx}$ is defined by

$$f'(x) = \lim_{\delta \to 0} \frac{f(x+\delta) - f(x)}{\delta} \tag{1}$$

Probably the most popular approach to computing derivatives within software is to derive a symbolic expression for the derivative manually and then directly code the resulting expression. This process can be generalised to line by line differentiation of code. For example, suppose we know how to compute the derivatives of the functions f and g. Then given the pseudocode

$$\begin{array}{lcl} y & \leftarrow & f(x) \\ z & \leftarrow & g(x,y) \end{array}$$

we can use the chain rule [Kap03] to transform it to the pseudocode

$$\begin{array}{lcl} y & \leftarrow & f(x) \\ y' & \leftarrow & f'(x) \\ z & \leftarrow & g(x,y) \\ z' & \leftarrow & \dfrac{\partial g(x,y)}{\partial x} + y' \dfrac{\partial g(x,y)}{\partial y} \end{array}$$

that computes both the original y and z as well as their derivatives with respect to x.

If the derivatives of f and g are unknown then we must implement functions that compute f' and g' in terms of the definitions of f and g using a similar transformation. In this way we ultimately append to every line of code containing a differentiable expression a new line to compute the derivative.

This method generalises naturally to longer code and to computing derivatives with respect to multiple variables. However, differentiating code by hand is a tedious and error prone process. Every modification to the code must be repeated in differential form. Consider, for example, using the above process to differentiate, with respect to 1000 variables, a function that requires 1000 lines of code to compute. One approach to this is to post-process our source code with another tool that automatically differentiates code line by line. This complicates the build process and puts constraints on coding styles.

A different approach is through the method of finite differences. In Equation (1) we can choose δ to be a small value, rather than take the limit as it approaches zero, and compute $(f(x+\delta) - f(x))/\delta$. This is an approximation to $f'(x)$ known as the forward difference. There is some difficulty associated with choosing a suitable value for δ. Too large and the approximation to the definition of the derivative is poor. Too small and we can expect the result to be dominated by rounding errors. We can improve the accuracy using central differences (i.e. $(f(x+\delta) - f(x-\delta))/2\delta$) but the errors can still be large. Note also that this requires two separate evaluations of f. If we generalise this method to compute n partial derivatives of a function then we can expect to require $n+1$ separate computations of f. See also Press at al. [PFTV92] for details of more sophisticated approximations to the true derivative.

3 Dual Objects

There is however, in a language like C++ which supports the introduction of new types and operator overloading, an elegant alternative method to compute derivatives of a function. It is automatic, requires far fewer evaluations of the function than finite differencing, is much more accurate than finite differencing, does not require post-processing of the code and results in code that is easy to maintain and modify. Although the presentation will be different, the implementation we will describe is very similar to that in [Jer89].

Return again to Equation (1) with the example $f(x) = 2x^2 + 1$. If $\delta = d$, a small non-zero number, then the approximation to $f'(x)$ is given by

$$\frac{f(x+d) - f(x)}{d} = (4dx + 2d^2)/d = 4x + 2d$$

The exact derivative is $4x$ so we have an error term, $2d$, with the same order of magnitude as d. Note that this error is coming from the $2d^2$ term in the numerator. If d^2 is small compared to d then the error becomes small and the approximation approaches the true value of the derivative. Ultimately, if d^2 were zero then we could compute the derivative exactly, but unfortunately the only choice for a real number d such that $d^2 = 0$ is $d = 0$ and this renders the whole computation meaningless. If we could find a non-zero number d such that $d^2 = 0$ then our problems would be solved. But where does one find such a number?

Consider a related problem: When we meet the equation $x^2 + 1 = 0$ for the first time it appears that there is no solution. It turns out that we *can* find a solution if we add new numbers to our number system. By convention we add i with the property that $i^2 = -1$. We may do the same with our problem. We extend the real numbers by adding a new variable, d, with the property that $d^2 = 0$. As with the definition of i there are some additional properties we require. For example *commutativity* so that $ad = da$ and $a + d = d + a$ for any real number a. Assuming the existence of such a number we can rewrite Equation (1) as

$$f(x + d) = f(x) + df'(x)$$

Consider, for example, $f(x) = x^n$.

$$f(x+d) = (x+d)^n = x^n + nx^{n-1}d + d^2(\frac{n(n-1)}{2}x^{n-2} + \ldots)$$

All of the terms after the first two are zero. So we can read off the derivative of $f(x)$ as the coefficient of d finding that $f'(x) = nx^{n-1}$ as expected.

We now have a strategy for computing derivatives: perform all of our function evaluations over the real numbers extended by d and read off the derivative from the final coefficient of d. We will call these extended numbers *dual numbers*. (These numbers were originally introduced by Clifford in 1873 [Cli73].)

4 Generic programming, polymorphism and operator overloading

At first sight it may appear that it requires substantial work to extend mathematical functions to operate on this new number type. However, one of the strengths of modern programming languages such is C++ is the potential for generic programming [MS89]. This is the methodology by which software is written so as to be independent of the underlying datatypes used. This style is typified by the Standard Template Library which employs the C++ template mechanism to define abstract datatypes and algorithms independently of the types of their contents and arguments. In fact, there are now a number of template based mathematical libraries for C++ that are generic [Vel98]. Frequently this is used to enable programmers to write code that is independent of information such as the machine representation of floating point numbers (e.g. whether `float` or `double` is used.)

However, we can also exploit such libraries to write code that can make use of more radical replacement of the underlying number type. The task now is to implement this replacement class and ensure that it exports an interface similar enough to that of the more usual number types that it can replace them. Because our new class is defined algebraically (through $d^2 = 0$) the implementation is in fact very similar to the implementation of complex numbers as a class. In fact we can now make a very general statement: computing derivatives of a wide class of functions is no more difficult that computing those functions applied to complex numbers.

We will consider examples in C++. For ease of exposition we will initially make our underlying datatype `float` though we will return to this point later. We now extend the `float` type by adding in the new elements implied by the existence of d. Every element of our new class can be written in the form $a + bd$ for real a and b. We call a the *real* part and b the *infinitesimal* part. We can now define the class by

```
class Dual {
public:
    float a; // Real part
    float b; // Infinitesimal part
    Dual(float a0,float b0 = 0.0f) : a(a0), b(b0) { }
};
```

For brevity of exposition we have chosen not to use accessor methods and instead make the members of this class public.

Consider the summation of two objects of type `Dual`.

$$(a_0 + b_0 d) + (a_1 + b_1 d) = (a_0 + a_1) + (b_0 + b_1)d$$

Similarly the product of two `Dual` objects is given by:

$$(a_0 + b_0 d)(a_1 + b_1 d) = a_0 a_1 + (a_0 b_1 + a_1 b_0)d$$

Using these expressions we can implement addition and multiplication via operator overloading as follows:

```
Dual operator+(const Dual &x,const Dual &y) {
    return Dual(x.a+y.a,x.b+y.b);
}
Dual operator*(const Dual &x,const Dual &y) {
    return Dual(x.a*y.a,x.a*y.b+x.b*y.a);
}
```

We are now in a position to start computing derivatives. As an example we consider the function $f(x) = (x+2)(x+1)$. We implement this in generic form as a polymorphic function:

```
template<class X> X f(X x) {
    return (x+X(2.0f))*(x+X(1.0f));
}
```

An important point to note is that we have cast the constants in this expression to the template parameter type so that the arguments to the `+` operator are of the same type. An alternative is to overload the operators so that they can accept arguments of differing type.

We now define:

```
Dual d(0.0f,1.0f);
```

and finally we can compute the derivative of $f(x)$ at $x = 3$, say, by computing

```
f(X(3.0f)+d)
```

and extracting the infinitesimal part of the result, in other words looking at the `b` member of the value of this expression.

If we compile and run this example we should find that f returns

$$(3 + d + 2)(3 + d + 1) = (5 + d)(4 + d) = 20 + 9d$$

giving the derivative as 9. A more conventional path gives $f'(x) = (x + 2)1 + 1(x + 1) = 2x + 3$ and this evaluated at $x = 3$ gives the identical result 9.

So now we can summarise the method: in order to compute the derivative of a function in C++ we must make the code generic and evaluate the function with d added to the argument. The derivative is given by the infinitesimal part of the returned value.

So far we have only considered addition and multiplication. Subtraction is straightforward but division is a little more subtle. Using the generalised binomial expansion of $(1 + x)^{-1}$ we find:

$$\begin{aligned}
\frac{a_0 + b_0 d}{a_1 + b_1 d} &= \frac{a_0 + b_0 d}{a_1} \frac{1}{1 + (b_1/a_1)d} \\
&= \frac{a_0 + b_0 d}{a_1}(1 - (b_1/a_1)d + d^2(\ldots)) \\
&= \frac{a_0}{a_1} + \frac{a_1 b_0 - a_0 b_1}{a_1^2} d
\end{aligned}$$

Note that division by any dual number of the form $0 + bd$ is not defined. This will pose no problems for us.

It is straightforward now to generalise to transcendental functions. Any differentiable function f defined over the real numbers can be extended to the duals by

$$f(a + bd) = f(a) + bf'(a)d$$

As an example we present the implementation of `cos`:

```
Dual cos(const Dual &x) {
    return Dual(cos(x.a),-b*sin(x.a));
}
```

With these functions defined we are now in a position to differentiate a large proportion of the differentiable code that is typically in use. In combination with a suitably generic vector and matrix library we can automatically differentiate vector and matrix expressions. We can even implement complex code such as fluid dynamics simulation and determine how the output parameters depend on an input parameter. (For a closely related example see [TMPS03]).

Note that d is similar to the infinitesimals introduced by Newton and Leibniz. It also shares some similarities with the infinitesimals of non-standard analysis [Rob74],[Ber92].

5 Partial Derivatives

So far we have illustrated how to differentiate with respect to a single variable. The above method generalises to partial derivatives. Instead of introducing a single variable d such that $d^2 = 0$ we introduce a set of variables d_i with $i \in I$ an index set. We stipulate that the d_i commute with all real numbers (e.g. $ad_i = d_i a$ and $a + d_i = d_i + a$ for all real a.) and that $d_i d_j = 0$ for all $i, j \in I$. A general dual number of this type may now be written as $x = a + \sum_{i \in I} b_i d_i$. We can represent members of this dual number class as a pair consisting of the real number a (the *real* part) and a vector of reals, $(b_i)_{i \in I}$ (the *infinitesimal* parts). The above implementation generalises naturally. We can now compute partial derivatives by reading the coefficients of the d_i. For example, suppose we have a function f mapping a pair of reals to a real and wish to compute partial derivatives at (x, y). Then we compute $f(x + d_0, y + d_1)$. The required partial derivatives appear as the coeficients of d_0 and d_1.

$$f(x + d_0, y + d_1) = f(x, y) + \frac{\partial f}{\partial x}(x, y)d_0 + \frac{\partial f}{\partial y}(x, y)d_1$$

Note that when we evaluate partial derivatives we obtain a significant performance gain over finite differencing. For example if we differentiate an expression containing the sine function with respect to N variables we need only compute the sine and cosine of the argument once each. When using finite differences we require $N + 1$ evaluations of the sine function, one for each evaluation of the expression.

6 Second Derivatives and Higher

We now take a different approach to Jerrell [Jer89] to compute second derivatives by a method similar to that used by the FADBAD library [BS96]. Let us return to the single variable case as this is easier to consider.

Thus far we have illustrated dual numbers over the class `float`. We must now generalise to duals defined over an arbitrary class:

```
template<class X>
class Dual {
public:
    X a; // Real part
    X b; // Infinitesimal part
    Dual(X a0,X b0 = 0) : a(a0), b(b0) { }
    static Dual<X> d() {
        return Dual<X>(X(0),X(1));
    }
};
```

We may now compute second derivatives by iterating the above method. If we wish to find the second derivative of a function f we write a C++ function g to compute the derivative of f in a generic manner and then use the same method to compute the derivative of g. We can best illustrate with some sample code:

```
template<class X>
X f(X x) {
    return ... // Compute some function of x
}

//
// Compute f'(x)
//
X g(X x) {
    return f(Dual<X>(x)+Dual<X>::d()).b;
}

//
// Compute f''(x)=g'(x)
//
... = g(Dual<float>(x)+Dual<float>::d()).b;
```

7 Differentiable Rendering

Frequently in ditigal movie post-production we are faced with the problem of inverse rendering: determining what input to a renderer will produce an output

that matches a given image. For example, as input to a 3D renderer we may have a number of parameters $(\theta_1, \theta_2, \ldots)$. These parameters may range from transformation parameters such as angle of rotation to shader parameters such as the exponent in a Blinn-Phong shader. We can consider a renderer to be a function

$$f : (\theta_1, \theta_2, \ldots) \to (I_{i,j,c})$$

where $I_{i,j,c}$ represents the color of the c channel of the (i, j)-th pixel of the rendered image. If $J_{i,j,c}$ is the desired output then we can write a sum of squares error term

$$e = \sum_{i,j,c} (I_{i,j,c} - J_{i,j,c})^2$$

If the rendering code is written genericaly enough that some parameters may be replaced by dual number objects then we may compute e as well as its derivative with respect to those parameters. Armed with the derivative information we may now use a black-box minimisation algorithm such as non-linear conjugate gradients to efficiently derive input parameters that result in an image I that best matches J.

More generally many problems in post-production can be solved by differentiating or inverting a subsystem of a 3D renderer. For example part of a ray-tracer takes as input a specification of a ray and returns the texture coordinates in the textures with which the ray intersects. By implementing this code generically we can automatically compute the derivative of the texture coordinates with respect to the ray specification. This is what is required to anti-alias texture mapping correctly and so automatic differentiation gives a straightforward means of implementing ray tracing differentials [Ige99].

8 Photogrammetry and Match-Moving

In this paper we will concentrate on the subsystem that transforms and projects 3D points to 2D screen coordinates. This will allow us to invert this operation to derive the 3D model that best fits a collection of projected 2D points. In other words we will demonstrate a method for implementing photogrammetry and match-moving. Our implementation was named *Labrador*.

Suppose we wish to reconstruct a set of 3-dimensional points P_i, $i \in I$, with positions (p_i). We have a set of images indexed by the set J in which some of the P_i appear projected. For each image j we have a camera projection function c_j such that $c_j(p_i)$ is the projection of the point P_i into the 2-dimensional screen space associated with camera j. We have an index set R such that $(i, j) \in R$ if and only if the point P_i appears projected in image j. For each (i, j) in R we have a 2D position $z_{i,j}$ at which the point P_i appears to be projected. In other words $z_{i,j} = c_j(p_i) + e_{i,j}$ where $e_{i,j}$ is the difference between actual and observed projections.

Suppose that the x and y coordinates of $e_{i,j}$ are independent normally distributed variables whose components have variance σ_j. We would like to find

the positions of the points P_i such that the deviations of the $e_{i,j}$ are minimised in the least squares sense. The maximum likelihood estimator of the positions of the P_i is given by the p_i that minimise this expression:

$$e = \sum_{(i,j)\in R} e_{i,j}^2 = \sum_{(i,j)\in R} (c_j(p_i) - z_{i,j})^2/\sigma_i^2$$

Using automatic differentiation we can find the derivatives of this expression and so minimise it using a black-box minimiser.

But there are considerable generalisations we can make. In practice we may have further information about the structure of the P_i, for example that some of the P_i are coplanar, while the details of the c_j may be unknown, for example the cameras may have unknown focal length and lens distortion or may be at an unknown position.

Our approach is as follows: users build an approximation to the scene that we are reconstructing within Alias|Wavefront *Maya*. We allow them to use transformations in the *Maya* scene graph to represent partially known information in the scene. For example if an object in the scene is known to be a cuboid but has unknown size then users can instantiate a cube object and scale it using a *Maya* scale transform. The user then marks variables in the scene whose values are unknown. In the cuboid example the user marks the three numbers defining the unknown scalings of the cube. There may be many unknown parameters marked in a scene: scalings, rotation angles, translations, camera focal lengths and so on. We then reimplemented the subsystem of *Maya* that computes the projections of points in this scene hierarchy in a generic way. We generically compute the error e for this scene in terms of the marked parameters and minimise this function using a black-box minimisation routine. As discussed in the next section we used a version of the Levenberg-Marquardt algorithm for minimisation as this is generally a good choice for nonlinear least-squares problems [PFTV92].

9 Implementation Details

Labrador uses an active set variation of the Levenberg-Marquardt algorithm [NW99],[PFTV92] suitable for bounded and unbounded minimisation. Rather than approximate the Hessian (the matrix of second derivatives) of e using a function of the Jacobian [PFTV92] we used automatic differentiation to obtain the full Hessian. At each iteration we used the conjugate gradient algorithm to solve the linear system involving Hessian. One slight difficulty is that the matrix of coefficients of this linear system isn't guaranteed to be positive definite as needed by the conjugate gradient algorithm. To some extent this problem is self-correcting because the Levenberg-Marquardt algorithm automatically adds terms to this matrix that tend to make it positive definite if the algorithm fails to descend towards a minimum. However, we also modified the conjugate-gradient algorithm to stop iterating and return the result of iteration so far if it detected a negative eigenvalue.

We considered sequences of live action images to be sets of independent images. This allowed *Labrador* to carry out both photogrammetry and match-moving. In fact, rather than compute match-moves from frame to frame incrementally we instead computed match-moves by minimising over all frames simultaneously. This often resulted in Levenberg-Marquardt minimisations in spaces of dimension greater than 10000. *Labrador* was able to work with such minimisations.

We gave users significant freedom of expression to describe known information. Any transformation parameters such as rotation angles or scalings could be marked for solving. *Labrador* supports an arbitrary number of cameras placed anywhere in the scene hierarchy and parameters for transforms above cameras may be marked. *Labrador* will reconstruct animated articulated geometry and even reconstruct cameras mounted on articulated geometry as with real camera rigs. If some of the images represent a sequence over time then the user indicates whether or not a parameter is to be considered animated or not. If it is animated then *Labrador* treats the value at each time step as an independent parameter for which it should solve.

Additionally *Maya* supports the connection of parameters in a scene by means of symbolic expressions. We implemented an interpreter to evaluate a subset of Maya's expression language in a generic and thus differentiable manner and allowed these expressions to enter into the expression for e. This meant that modellers and match-movers could express constraints that are difficult to express via transforms. For example users could use this facility to set the expression for the height of one building to be the same expression as that for the height of another building so that the buildings are constrained to have the same height. Similarly two cameras can be constrained to have the same unknown focal length or complex information about the relationships between the members of a truss can be entered. *Labrador* regularly finds global minima in the presence of complex scene graphs and expressions. (Compare with the CONDOR system [Kas92].)

The vector (b_i) described in Section 5 was represented sparsely. This allowed us to compute and represent the Hessian sparsely and use the sparse conjugate gradient method to solve for the iteration step. Exploiting sparsity is essential to obtaining good performance. Consider computing the N^2 second derivatives, with respect to $(x_1, \ldots, x_N)$, of

$$f = \sum_{i=1}^{L} f_i^2$$

where each f_i is a function of no more than M of the x_i. Then the Hessian has no more than LM^2 non-zero entries. For match-moving problems this is usually much less than N^2.

(Our approach to automatic differentiation is known as *forward mode* differentiation. Another approach to automatic differentiation is *reverse mode* automatic differentiation [GC91]. Often this is better suited to problems with large numbers of input variables. We also implemented this method but found

that we could achieve better performance by using the method described above with a sparse representation for the duals. A sparse variation of reverse mode differentiation may also be possible but this was not tried.)

10 Results and Discussion

Automatic differentiation has proved a highly effective method for computing derivatives. In particular it has provided an efficient solution to the problems of reconstruction of geometry and camera moves from film. *Labrador* was able to solve for both structured and unstructured geometry and it replaced commercial match-moving and photogrammetry software in-house entirely.

The most important point to note is that we were able to implement a highly effective tool merely by enforcing the rule that the evaluation of e was implemented generically allowing the use of a standard black-box minimiser. In other words once the reusable library components were written then the solver was no more complex to implement than the forward code that simply transforms and projects points. What is more, we were able to do this by the full Newton method rather than the Gauss-Newton approximation, a task that is often considered too complex to be implemented (see [TMHF00], Section 4.3). (Note that we also tested the Gauss-Newton method and found that in some scenarios it can perform better. Comparison between the two methods is a complex topic.) *Labrador* was able to generalise much of the functionality of existing photogrammetry tools such as Façade [DTM96] and replaced commercial tools in-house. There are no other hidden components to *Labrador*'s solver. It consists almost entirely of generic code to compute e and library code to optimise generic functions.

Without a standard set of benchmarks it is difficult to give an idea of timings for *Labrador*. In production, scenes are built incrementally in *Maya* using *Labrador* repeatedly to find the best local minimum using information entered so far. Used in this way, with a single CPU on a 2.8GHz Xeon, the time for *Labrador* to solve is an insignificant part of the workflow. In a scenario such as solving for a 600 frame match-move determining 6 camera transform parameters for each frame as well as approximately 100 parameters that don't vary in time (a minimisation in a 3700 dimensional space) *Labrador* might take $5 - 45$ minutes to solve. But this does not reflect real usage. In fact this example could be speeded significantly by using every tenth frame to obtain a good approximation that is subsequently used to initiate a full solve. (Note, most commercial match-moving tools will not use all frames simultaneously but instead typically solve one frame at a time or solve using a small batch at a time. When using *Labrador* in this way the time to solve is often significantly less than the time to communicate the results back to *Maya*.)

One surprising fact emerged from this work. The space of possible solutions is large, often having a dimensionality running into the thousands, and has a complex topology due to the presence of parameters such as rotation angles. But for real photogrammetry and match-moving problems that appeared dur-

ing production there is frequently a large 'basin' around the global minimum. Within this basin it was possible to start the Levenberg-Marquardt algorithm and arrive at the unique global minimum. The specification of information through the use of transforms and expressions removes many potential sources of ambiguity. In rare cases a local minimum was discovered that was not the global minimum. It was often a configuration that was approximately related to the global minimum by a simple symmetry operation, for example a building being inverted with respect to the axis pointing directly forward from the camera in which it was projected. (Cf. the 'bas-relief' ambiguity [BKY99].) In these cases it was an easy process to apply that operation and then restart the search. It is also worth noting that our minimiser was conservative in the sense that it would never take a step worsening the objective function. This was useful in ensuring that the solver never left the basin to become lost.

11 Acknowledgements

Thanks to JP Lewis and Doug Moore for their contributions to the development of *Labrador*.

References

[Ber92] Martin Berz. Automatic differentiation as nonarchimedean analysis. In *Computer Arithmetic and Enclosure Methods*, Amsterdam, 1992. Elsevier Science Publishers.

[BKY99] Peter N. Belhumeur, David J. Kriegman, and Alan L. Yuille. The bas-relief ambiguity. *Int. J. Comput. Vision*, 35(1):33–44, 1999.

[BS96] Claus Bendtsen and Ole Stauning. Fadbad, a flexible C++ package for automatic differentiation, 1996. `http://www.imm.dtu.dk/fadbad`.

[BW98] David Baraff and Andrew Witkin. Large steps in cloth simulation. In *Computer Graphics Proceedings, Annual Conference Series*, pages 43–54. SIGGRAPH, 1998.

[Cli73] W. K. Clifford. Preliminary sketch of bi-quaternions. *Proceedings of London Mathematical Society*, 4:381–395, 1873.

[DTM96] Paul E. Debevec, Camillo J. Taylor, and Jitendra Malik. Modeling and rendering architecture from photographs: A hybrid geometry- and image-based approach. *Computer Graphics*, 30(Annual Conference Series):11–20, 1996.

[Duf02] Thomas Duff. Motion blurring implicit surfaces, November 2002. US patent 6,483,514.

[GC91] Andreas Griewank and George F. Corliss, editors. *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*. SIAM, Philadelphia, Penn., 1991.

[Gle97] Michael Gleicher. Motion editing with spacetime constraints. In *Proceedings of the 1997 symposium on Interactive 3D graphics*, pages 139–ff. ACM Press, 1997.

[HS81] Berthold K. P. Horn and Brian G. Schunck. Determining optical flow. *Artificial Intelligence*, 17(1-3):185–203, August 1981.

[HS98] Wolfgang Heidrich and Hans-Peter Seidel. Ray-tracing procedural displacement shaders. In *Graphics Interface*, pages 8–16, 1998.

[Ige99] Homan Igehy. Tracing ray differential. In Alyn Rockwood, editor, *Siggraph 1999, Computer Graphics Proceedings*, pages 179–186, Los Angeles, 1999. Addison Wesley Longman.

[Jer89] M. E. Jerrell. Function minimization and automatic differentiation using C++. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 169–173. ACM Press, 1989.

[Kap03] Wilfred Kaplan. *Advanced Calculus*. Addison-Wesley, 2003.

[Kas92] Michael Kass. Condor: constraint-based dataflow. In *Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, pages 321–330. ACM Press, 1992.

[MH92] Don Mitchell and Pat Hanrahan. Illumination from curved reflectors. *Computer Graphics*, 26(2):283–291, 1992.

[MS89] Musser and Stepanov. Generic programming. In *ISSAC: Proceedings of the ACM SIGSAM International Symposium on Symbolic and Algebraic Computation (formerly SYMSAM, SYMSAC, EUROSAM, EUROCAL) (also sometimes in cooperation with the Symbolic and Algebraic Manipulation Groupe in Europe (SAME))*, 1989.

[NW99] J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer, 1999.

[PFTV92] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, Cambridge (UK) and New York, 2nd edition, 1992.

[Rob74] A. Robinson. *Non-Standard Analysis*. North-Holland, 1974.

[TMHF00] Bill Triggs, Philip McLauchlan, Richard Hartley, and Andrew Fitzgibbon. Bundle adjustment – A modern synthesis. In W. Triggs, A. Zisserman, and R. Szeliski, editors, *Vision Algorithms: Theory and Practice*, LNCS, pages 298–375. Springer Verlag, 2000.

[TMPS03] Adrien Treuille, Antoine McNamara, Zoran Popovic, and Jos Stam. Keyframe control of smoke simulations. *ACM Transactions on Graphics (TOG)*, 22(3):716–723, 2003.

[Vel98] Todd L. Veldhuizen. Arrays in Blitz++. In *Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'98)*, Berlin, Heidelberg, New York, Tokyo, 1998. Springer-Verlag.