Table of Contents

Reconciling OOD with DOD

syntax updated to the most recent c++ standard proposal
An evolution of this model can be found here

Introduction

Data Oriented Design is getting more and more attention as a mean to match the actual capability of current hardware either multicore CPU or GPGPUs. It is not difficult to prove that a Structure of Arrays matches much better the cache structure and the vector engine w.r.t. the corresponding Array of Structures. This is very well illustrated for instance by these two presentations by professional consultant for the game industry:

- Pitfalls of Object Oriented Programming
- introduction to data oriented design

On the other hand modern compilers have developed powerful capability of code analysis that makes them able to fully inline and even vectorize code that at first sight looks involving very inefficient operations such as copy of structures or long nested function calls. It becomes therefore possible to use at the same time highly optimized and user friendly Object Oriented model for the basic construction block and use a Data Oriented model for the storage classes and the organization of the high level algorithms.

A first simple example

Here is a modest exercise to show how a full DOD model can still use the full power of SoA and DOD (thanks to the power of recent compilers, namely gcc 4.7).

The exercise consists in computing the crossing of a set of lines with a plane and, conversely of a single line with many planes. The line,plane model is fully Object Oriented in its most naive and classical way: one could even use any of the existing libraries for that.

OO Basic Blocks

This is the the OO model

```
// an "oriented" plane is defined by
// nx*(x-xp)+ny(y-yp)+nz*(z-zp)=0 (or in vector notation: n*(x-p) * here is the dot-product)
// for a generic point x,y,z this equation gives the distance of the point (with its sign) from t

// a line is reprersented by the "vector" equation x=xl+c*t (t is scalar and represent the path a

template<typename T>
struct Vect {
  using value = typename std::remove_reference<T>::type;
  using ref = typename std::add_lvalue_reference<T>::type;

  Vect() {}
  Vect(T ix, T iy, T iz) : x(ix),y(iy),z(iz){}
  template<typename V>
  Vect(V v) : x(v.x), y(v.y), z(v.z) {}
  template<typename V>
  Vect& operator=(V v) { x=v.x; y=v.y; z=v.z; return *this; }

  T x,y,z;
  void normalize() {
    value in = value(1.)/std::sqrt(x*x+y*y+z*z);
    x*=in; y*=in; z*=in;
  }
```

```
};

template<typename T> using Point =  Vect<T>;

template<typename T1, typename T2>
inline
Vect<float> operator+(Vect<T1> const & a,Vect<T2> const & b) { return Vect<float>(a.x+b.x,a.y+b.y

template<typename T1, typename T2>
inline
Vect<float> operator-(Vect<T1> const & a,Vect<T2> const & b) { return Vect<float>(a.x-b.x,a.y-b.y

template<typename T>
inline
Vect<float> operator*(float s,Vect<T> const & a) { return Vect<float>(s*a.x,s*a.y,s*a.z);}

template<typename T>
inline
Vect<float> operator*(Vect<T> const & a,float s) { return Vect<float>(s*a.x,s*a.y,s*a.z);}

template<typename T1, typename T2>
inline
float dot(Vect<T1> const & a,Vect<T2> const & b) { return a.x*b.x+a.y*b.y+a.z*b.z;}

template<typename T1, typename T2>
inline
Vect<float> cross(Vect<T1> const & a,Vect<T2> const &b) {
  return Vect<float> (a.y*b.z-a.z*b.y,
                      a.z*b.x-a.x*b.z,
                      a.x*b.y-a.y*b.x
                      );
}

template<typename T>
struct Plane : public Surface {
  Plane(): p(),n(){}
  Plane(Point<T> ip, Vect<T> in) : p(ip),n(in){}
  Point<T> p;
  Vect<T> n;
};

template<typename T>
struct Line final : public Trajectory {
  using value = typename std::remove_reference<T>::type;
  using ref = typename std::add_lvalue_reference<T>::type;

 Line(): p(),c(){}
 Line(Point<T> const & ip, Vect<T> const & ic) : p(ip),c(ic){}
 template<typename L>
   Line(L l) : p(l.p),c(l.p){}
 template<typename L>
   Line& operator=(L l) {p=l.p; c=l.p; return *this;}
  Point<T> p;
  Vect<T> c;

  Point<value> go(float t) const { return p + c*t; }
};


// return t for the point of plane-line crossing
template<typename T1, typename T2>
inline
float cross(Plane<T1> const & plane, Line<T2> const & line) {
  return dot(plane.n,plane.p-line.p)/dot(plane.n,line.c);
}

// return distance of a point from a plane
```

```
template<typename T1, typename T2>
inline
float distance(Plane<T1> const & plane, Point<T2> const & point) {
  return dot(plane.n,point-plane.p);
}

// return distance of a point from a plane
inline
float distance(Plane const & plane, Point const & point) {
  return dot(plane.n,point-plane.p);
}
```

with its little OO test program

```
inline
Plane<float> makePlane(float offr=0) {
  // a generic plane
  double nx = -7./13., ny=3./7., nz=17./23.;
  double nor = 1./std::sqrt(nx*nx+ny*ny+nz*nz);
  return Plane<float>({float(3./7.+offr),float(4./7.+offr),13./17.},{float(nx*nor),float(ny*nor),
}

inline
Line<float>  makeLine(float r, float z) {
  // a generic line (almost parallel to the plane...)
  double cy = -7./13.+r, cx=3./7.-r, cz=17./23.+z;
  double nor = 1./sqrt(cx*cx+cy*cy+cz*cz);
  return Line<float> ({3.3/7.,4.4/7.,13./17.},{float(cx*nor),float(cy*nor),float(cz*nor)});
}

void testCross() {
  auto pl = makePlane();
  auto  ln = makeLine(0,0);

  auto d0 = distance(pl,ln.p);
  printf("d0 = %e, %a\n",d0,d0);
  float t = cross(pl,ln);
  printf("t  = %e, %a\n",t,t);
  auto x1 = ln.go(t);
  auto d1 = distance(pl,x1);
  printf("d1 = %e, %a\n",d1,d1);
}
```

DO storage model

DOD enters in data organization and in the high level algorithms. Collections of vectors, lines and planes are implemented as SoA, still both filling and retrieving goes through real (at least in code) objects of types Vector, Line, Plane: the only place where breaking of encapsulation occur in the implementation of the Collection classes that shall match the implementation of the basic classes.

Here is the DOD SoAs

```
struct Vects {
  static constexpr int nValues=3;
  using VV = Vect<float>;
  using RV = Vect<float&>;
  using CV = Vect<float const & >;

  Vects() : mem(nullptr),m_size(0){}
  Vects(int s, float * imem) : mem(imem),m_size(s){}
  float * __restrict__ mem;
  int m_size;

  float & x(int i) { return mem[i];}
```

```
  float & y(int i) { return mem[i+size()];}
  float & z(int i) { return mem[i+size()+size()];}

  float const & x(int i) const { return mem[i];}
  float const & y(int i) const { return mem[i+size()];}
  float const & z(int i) const { return mem[i+size()+size()];}


  CV operator[](int i) const {
    return CV(x(i),y(i),z(i));
  }
  RV operator[](int i) {
    return RV(x(i),y(i),z(i));
  }

  int size() const { return m_size;}

};



using Points =  Vects;

struct Lines {
  static constexpr int nValues=2*Vects::nValues;

  using VL = Line<float>;
  using RL = Line<float&>;
  using CL = Line<float const & >;

  Lines(){}
  Lines(int s, float * mem):p(s,mem),c(s,mem+s*Vects::nValues){}
  Vects p, c;

  CL operator[](int i) const { return CL(p[i],c[i]);}
  RL operator[](int i) { return RL(p[i],c[i]);}
  int size() const { return p.size();  }

};
```

with the DOD test program

```
void makeLines(Lines & lines) {
  float epsr = 0.5/13., epsz=1./23.;
  float r=0, z=0;
  int n = lines.size();
  for (int i=0; i!=n; ++i) {
    r+=epsr;
    z += (i<n/2) ? epsz : -epsz;
    lines.set(makeLine(r,z),i);
  }
}

void loopCross() {
  int N=256;
  float arena[N*Lines::nValues];
  Lines lines(N,arena);
  makeLines(lines);

  Plane pl = makePlane();
  int n = lines.size();
  float t[n];
  for (int i=0; i!=n; ++i) {
    t[i] = cross(pl,lines[i]);
  }
```

```
  printf("\n");
  for (int i=0; i!=n; ++i)
    printf("%e,%a ",t[i],t[i]);
  printf("\n");
}
```

when compiled with

```
c++ -std=c++1y -Ofast  CrossingPlane.cpp -Wall -fopt-info-vec
```

one can read

```
Vectorizing loop at CrossingPlane.cpp:148

148: LOOP VECTORIZED.
CrossingPlane.cpp:139: note: vectorized 1 loops in function.
```

where the function al line 139 is loopCross. To further confirm the full optimization one can verify that the only symbols generated are those of the three not-inlined test functions

```
nm ./a.out ç
00000001000019c0 T loopCross()
0000000100001870 T makeLines(Lines&)
0000000100001760 T testCross()
0000000100001700 T start
```

and evetually inspect the generated assembly code.

A more complex example

We can make the thing a bit more complicated assuming to have "bounds" on the Plane such as Rectangles and Trapezoids and we wish to check if the track crossing the plane is inside or outside the Bound and, say, update its position only if inside

the top level "kernel" may look like this:

```
template<typename S>
inline
void loopInsideKernel(S const & r) {
  int N=256;
  float arena[N*Lines::nValues];
  Lines lines(N,arena);
  makeLines(lines);

  int n = lines.size();
  int in[n];
  for (int i=0; i!=n; ++i) {
    float t = cross(r,lines[i]);
    auto x2 = lines[i].go(t);
    in[i] = inside(r,x2);
    x2 = in[i] ? x2 : decltype(x2)(lines[i].p);
    // if (in[i])
    lines[i].p = x2;
  }

  printf("\n");
  std::cout << typeid(S).name() << std::endl;
  for (int i=0; i!=n; ++i)
    std::cout << ( in[i] ? 'i' : 'o');
  std::cout << std::endl;
  for (int i=0; i!=n; ++i)
    std::cout <<  lines[i].p.x <<",";
```

```
  std::cout << std::endl;
}
```

specialized then in

```
void loopInsideR() {
  Rectangle r = makeRectangle(0.01f);
  loopInsideKernel(r);
}
void loopInsideLT() {
  LeftTrapezoid r = makeLeftTrapezoid(0.01f);
  loopInsideKernel(r);
}
```

and the whole inheritance and overloaded functions will look like this:

```
template<typename T>
struct Quadrangle : public Plane<T> {
  Quadrangle() {}
  Quadrangle(Point<T> const & ip, Vect<T> const & iu, Vect<T> const & iv, T iul, T ivl) :
    Plane<T>(ip,cross(iu,iv)),u(iu),v(iv), ul(iul), vl(ivl){}
  Vect<T> u, v;
  T ul, vl;
};

template<typename T>
struct Rectangle final : public Quadrangle<T> {
  template<typename ...Args>
  Rectangle(Args &&...args) :  Quadrangle<T>(std::forward<Args>(args)...){}
};


template<typename T>
struct BaseTrapezoid :  public Quadrangle<T> {
  BaseTrapezoid(){}
  BaseTrapezoid(Point<T> const & ip, Vect<T> const & iu, Vect<T> const & iv, T ius, T iul, T ivl)
    Quadrangle<T>(ip,iu,iv, 0.5f*(ius+iul), ivl), alpha((iul-ius)/(ius+iul)) {}
  T alpha;
};

template<typename T>
struct Trapezoid final : public BaseTrapezoid<T> {
  template<typename ...Args>
  Trapezoid(Args &&...args) :  BaseTrapezoid<T>(std::forward<Args>(args)...){}
};


// p is not the baricenter: is on the "straight" vertical side
template<typename T>
struct LeftTrapezoid final : public BaseTrapezoid<T> {
  template<typename ...Args>
  LeftTrapezoid(Args &&...args) :  BaseTrapezoid<T>(std::forward<Args>(args)...){}
};

// p is not the baricenter: is on the "straight" vertical side
template<typename T>
struct RightTrapezoid final : public BaseTrapezoid<T> {
  template<typename ...Args>
  RightTrapezoid(Args &&...args) :  BaseTrapezoid<T>(std::forward<Args>(args)...){}
};

template<typename T1, typename T2>
inline
void locCor(Quadrangle<T1> const & r, Point<T2> const & p,float & u,float & v) {
  Vect<float> d = p-r.p;
  // printf("d= %e,%e,%e\n",d.x,d.y,d.z);
```

```
  u = dot(d,r.u);
  v = dot(d,r.v);
  // printf("u,v= %e,%e\n",u,v);
}

template<typename T1, typename T2>
inline
bool inside(Rectangle<T1> const & r, Point<T2> const & p) {
  float u,v;
  locCor(r,p,u,v);
  return  (r.ul>std::abs(u) ) &  ( r.vl>std::abs(v) ) ;
}


template<typename T1, typename T2>
inline
bool inside(Trapezoid<T1> const & r, Point<T2> const & p) {
  float u,v;
  locCor(r,p,u,v);
  float ul = r.ul + v * r.alpha;
  // printf("u,v= %e,%e\n",u,v);
  return  (ul>u) &  ( r.vl>std::abs(v) ) ;
}

template<typename T1, typename T2>
inline
bool inside(LeftTrapezoid<T1> const & r, Point<T2> const & p) {
  float u,v;
  locCor(r,p,u,v);
  float ul = r.ul + v * r.alpha;
  // printf("u,v= %e,%e\n",u,v);
  return  (v>0.f ) & (ul>u  ) &  ( (-r.vl)<v ) ;
}

template<typename T1, typename T2>
inline
bool inside(RightTrapezoid<T1> const & r, Point<T2> const & p) {
  float u,v;
  locCor(r,p,u,v);
  float ul = r.ul + v * r.alpha;
  // printf("u,v= %e,%e\n",u,v);
  return  (u<0.f) & (ul>u ) &  ( r.vl>v ) ;
}
```

and with gcc 4.7 does vectorize!

Polymorphism

What about polymorphism? A truly Object Oriented design would call for functions such as `inside(Surface const &, Point const &)` and `cross(Surface const &, Trajectory const &)`. Besides the usual issues raised by the fact that `cross` is a *multi-method* and its implementation based on *double dispatch* creates problems of all sorts including cross dependencies and eventually performance inefficiency even in sequential programming, what real profit one can make of those polymorphic functions depends on use cases i.e. in which contexts we will find generic pointers to `Surface` and `Trajectory`.

A "reusable" library

Compile time polymorphism (templates) can be exploited for such a use-case as done above for `loopInsideKernel`

Polymorphic Collections

User extensions and override

-- VincenzoInnocente - 24-Sep-2011

This topic: LCG > VIOODvsDOD
Topic revision: r12 - 2014-05-18 - VincenzoInnocente