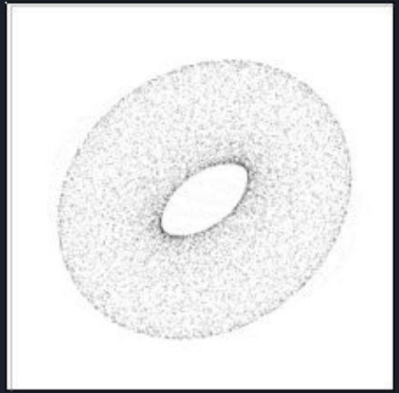
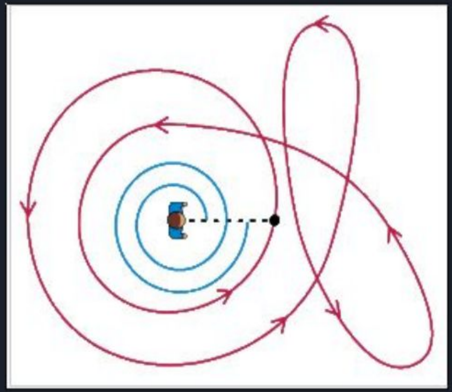
Fast Winding Numbers for Soups and Clouds

Felipe A. Moreno

Definitions

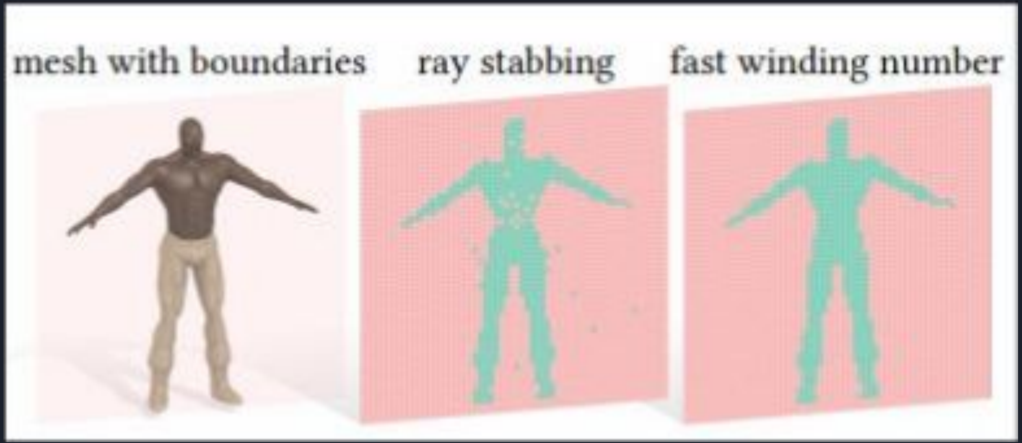
- Point Cloud: is a set of data points in space. Point clouds are generally produced by 3D scanners, which measure a large number of points on the external surfaces of objects around them.
- Winding Number: The winding number of the curve is equal to the total number of counterclockwise turns that the object makes around the origin.





Advantages

- The winding number in a variety of new applications: voxelization, signing distances, generating 3D printer paths, defect-tolerant mesh booleans and point set surfaces.
- Determines how many times a planar curve encircles a query point.
- For overlapping regions, the winding number measures how many times the region is inside the surface.



Surface Points

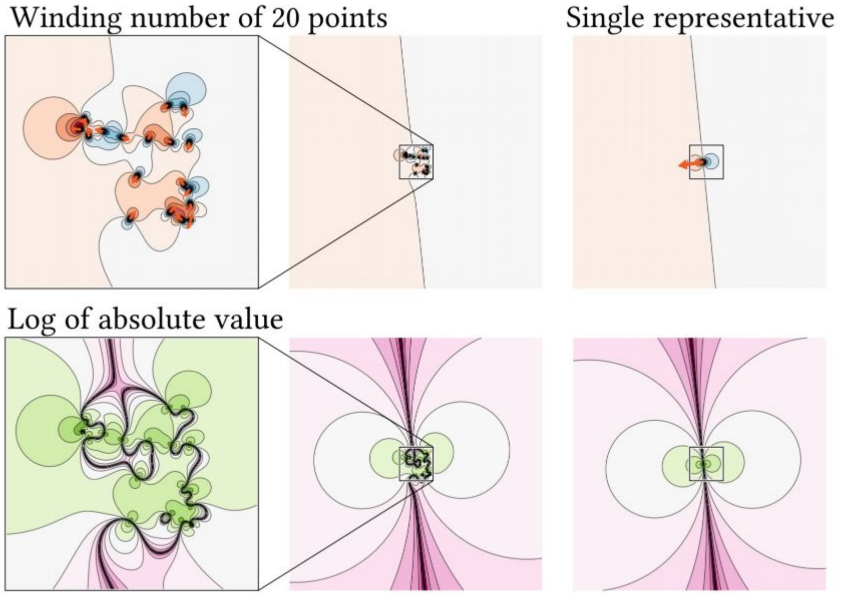
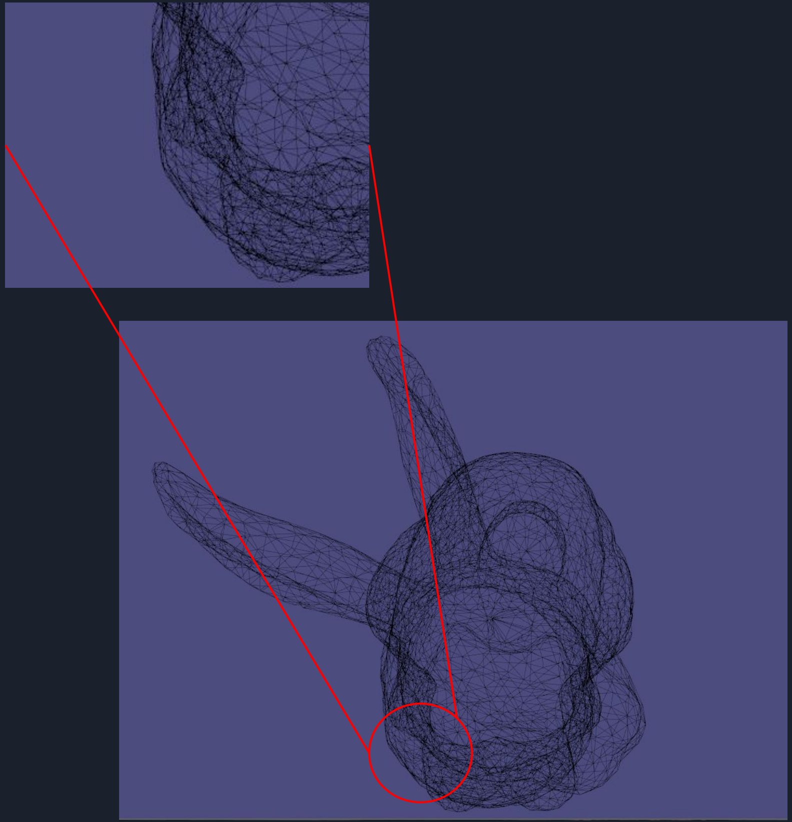


Fig. 6. A cluster of 20 dipoles has an intricate winding number field nearby (left), but far away their function is quite tame (middle) and well approximated by a single, *stronger* dipole (right).



Algorithm to calculate Winding Numbers

$$\frac{(\mathbf{x}-\mathbf{q})\cdot\hat{\mathbf{n}}}{4\pi\|\mathbf{x}-\mathbf{q}\|^3} = \nabla\left(\frac{-1}{4\pi\|\mathbf{x}-\mathbf{q}\|}\right)\cdot\hat{\mathbf{n}} =: G_{\hat{\mathbf{n}}}(\mathbf{q},\mathbf{x})$$

$$w(\mathbf{q}) = \sum_{i=1}^{m} a_i \frac{(\mathbf{p}_i-\mathbf{q})\cdot\hat{\mathbf{n}}_i}{4\pi\|\mathbf{p}_i-\mathbf{q}\|^3} \approx \frac{(\tilde{\mathbf{p}}-\mathbf{q})\cdot\tilde{\mathbf{n}}}{4\pi\|\tilde{\mathbf{p}}-\mathbf{q}\|^3} =: \tilde{w}(\mathbf{q})$$

$$\tilde{\mathbf{n}} = \sum_{i=1}^{m} a_i\hat{\mathbf{n}}_i, \quad \tilde{\mathbf{p}} = \frac{\sum_{i=1}^{m} a_i\mathbf{p}_i}{\sum_{i=1}^{m} a_i},$$

```
// Extract interior tets
MatrixXi CT((W.array()>0.5).count(),4);
{
  size_t k = 0;
  for(size_t t = 0;t<T.rows();t++)
  {
    if(W(t)>0.5)
    {
      CT.row(k) = T.row(t);
      k++;
    }
  }
}
// find bounary facets of interior tets
igl::boundary_facets(CT,G);
// boundary_facets seems to be reversed...
G = G.rowwise().reverse().eval();

// normalize
W = (W.array() - W.minCoeff())/(W.maxCoeff()-W.minCoeff());
```

Algorithm fast approximation

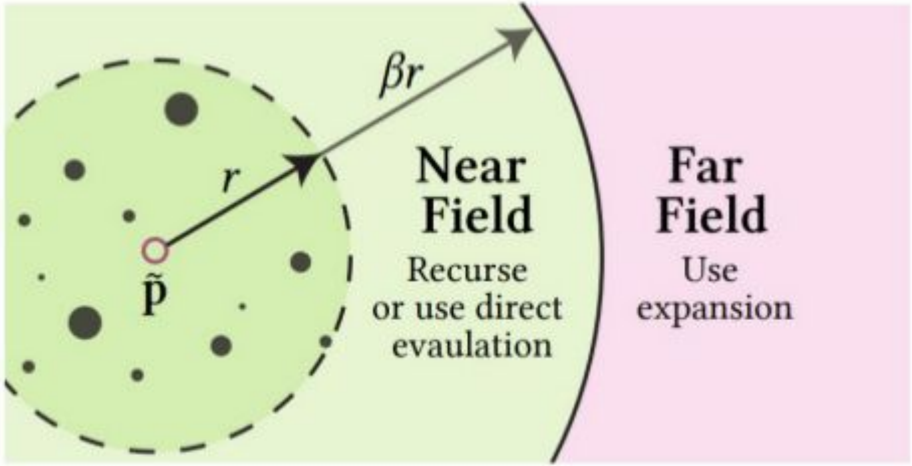


Fig. 7. Our spatial partitioning separates near and far fields, recursively.

```
Algorithm 1: Fast Approximation of Winding Number
FASTWN(q,tree)

Inputs:
  q      Query point in ℝ³
  tree   Root of bounding volume hierarchy for points/triangles
  β      accuracy parameter
Outputs: scalar winding number of all elements in tree at q
// tree.p: center of tree's winding number approximation, tree.ŵ
// tree.r: maximum distance from tree.p to any of its elements
if ||q − tree.p|| > β * tree.r then
  // q is sufficiently far from all elements in tree
  return tree.ŵ(q)
else
  val ← 0
  if tree has no children then
    // q is nearby; use direct sum for tree's elements
    for each point/triangle e in tree do
      // w_e: area-weighted dipole or solid angle
      val += w_e(q)
  else
    for each child of tree do
      // Recursive call
      val +=FASTWN(q, child)
  return val
```

Fast Algorithm to calculate Winding Numbers

$$\begin{aligned}
G_{\hat{\mathbf{n}}}(\mathbf{q},\mathbf{x}) &= \hat{\mathbf{n}} \cdot \nabla G(\mathbf{q},\mathbf{x}) \\
&= \hat{\mathbf{n}} \cdot \nabla G(\mathbf{q},\tilde{\mathbf{p}}) \\
&\quad + ((\mathbf{x}-\tilde{\mathbf{p}}) \otimes \hat{\mathbf{n}}) \cdot \nabla^2 G(\mathbf{q},\tilde{\mathbf{p}}) \\
&\quad + \tfrac{1}{2}((\mathbf{x}-\tilde{\mathbf{p}}) \otimes (\mathbf{x}-\tilde{\mathbf{p}}) \otimes \hat{\mathbf{n}}) \cdot \nabla^3 G(\mathbf{q},\tilde{\mathbf{p}}) \\
&\quad + \textit{ higher order terms},
\end{aligned}$$

$$\begin{aligned}
w(\mathbf{q}) \approx& \left(\sum_{i=1}^{m} a_i \hat{\mathbf{n}}_i\right) \cdot \nabla G(\mathbf{q},\tilde{\mathbf{p}}) \\
&+ \left(\sum_{i=1}^{m} a_i (\mathbf{p}_i - \tilde{\mathbf{p}}) \otimes \hat{\mathbf{n}}_i\right) \cdot \nabla^2 G(\mathbf{q},\tilde{\mathbf{p}}) \\
&+ \tfrac{1}{2}\left(\sum_{i=1}^{m} a_i (\mathbf{p}_i - \tilde{\mathbf{p}}) \otimes (\mathbf{p}_i - \tilde{\mathbf{p}}) \otimes \hat{\mathbf{n}}_i\right) \cdot \nabla^3 G(\mathbf{q},\tilde{\mathbf{p}}) \\
&=: \tilde{w}(\mathbf{q}).
\end{aligned}$$

```
HDK_Sample::UT_SolidAngle<float,float> solid_angle;

std::vector<HDK_Sample::UT_Vector3T<float> > U(V.rows());
for(int i = 0;i<V.rows();i++){
  for(int j = 0;j<3;j++){
    U[i][j] = V(i,j);
  }
}
solid_angle.init(F.rows(), F.data(), V.rows(), &U[0], order);

igl::parallel_for(T.rows(),[&](int q)
//for(int q = 0;q<T.rows();q++)
{
  HDK_Sample::UT_Vector3T<float>Qq;
  Qq[0] = T(q,0);
  Qq[1] = T(q,1);
  Qq[2] = T(q,2);
  Wapprox(q) = solid_angle.computeSolidAngle(Qq, accuracy_scale)/(4.0*M_PI);
}
,1000);
```

Fast Algorithm to calculate Winding Numbers

Algorithm 1: Fast Approximation of Winding Number FASTWN(q,tree)

Inputs:

- **q** Query point in $\mathbb{R}^3$
- tree Root of bounding volume hierarchy for points/triangles
- β accuracy parameter

Outputs: scalar winding number of all elements in tree at **q**

// tree.p: center of tree's winding number approximation, tree.$\tilde{w}$

// tree.r: maximum distance from tree.p to any of its elements

if $\|\mathbf{q} - tree.\mathbf{p}\| > \beta * tree.r$ **then**

  // q is sufficiently far from all elements in tree

  return tree.$\tilde{w}$(**q**)

else

  val ← 0

  if *tree has no children* **then**

    // q is nearby; use direct sum for tree's elements

    for *each point/triangle e in tree* **do**

      // w_e: area-weighted dipole or solid angle

      val += w_e(**q**)

  else

    for *each child of tree* **do**

      // Recursive call

      val +=FASTWN(**q**, child)

  return val

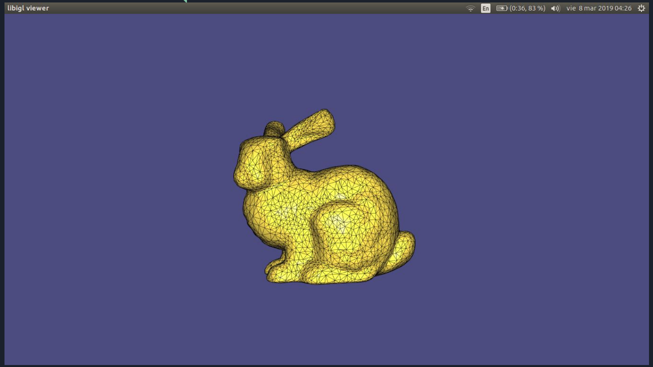
```
T sum = (descend_bits&1) ? child_data_array[0] : 0;
for (int i = 1; i < nchildren; ++i)
    sum += ((descend_bits>>i)&1) ? child_data_array[i] : 0;

*data_for_parent += sum;
```

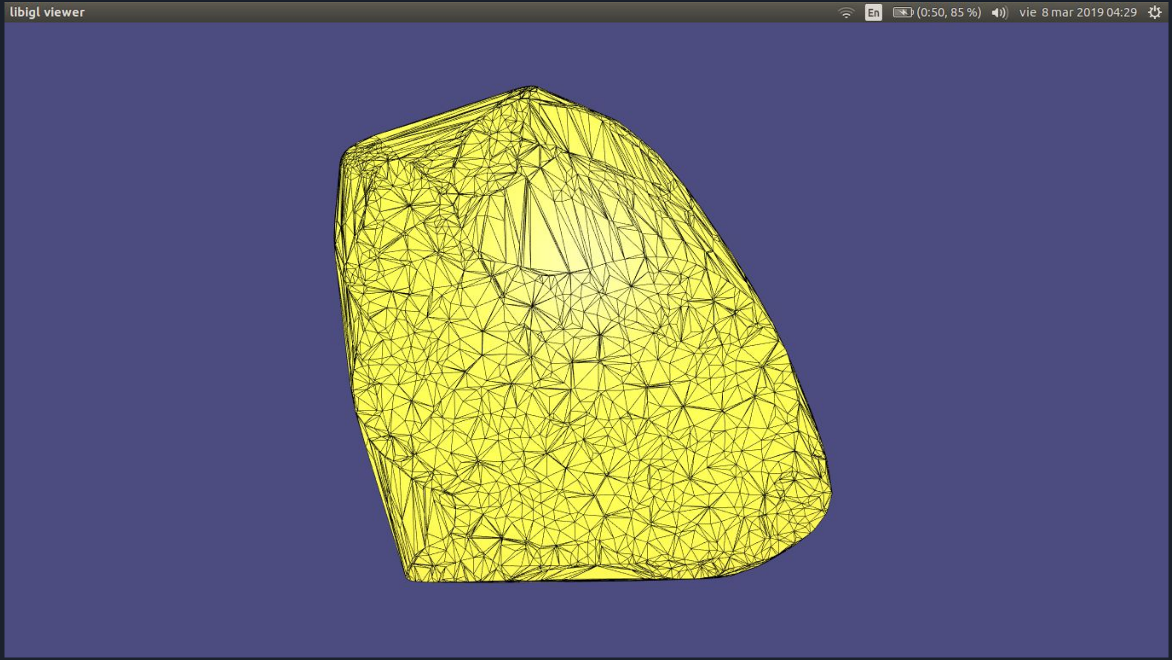
```
for (int i = 0; i < nchildren; ++i)
{
    const LocalData &child_data = child_data_array[i];
    UT_Vector3T<T> displacement = child_data.myAverageP - UT_Vector3T<T>(data_for_parent->myAverageP);
    UT_Vector3T<T> N = child_data.myN;

    // Adjust Nij for the change in centre P
    data_for_parent->myNijDiag += N*displacement;
    T Nxy = child_data.myNxy + N[0]*displacement[1];
    T Nyx = child_data.myNyx + N[1]*displacement[0];
    T Nyz = child_data.myNyz + N[1]*displacement[2];
    T Nzy = child_data.myNzy + N[2]*displacement[1];
    T Nzx = child_data.myNzx + N[2]*displacement[0];
    T Nxz = child_data.myNxz + N[0]*displacement[2];

    data_for_parent->myNxy += Nxy;
    data_for_parent->myNyx += Nyx;
    data_for_parent->myNyz += Nyz;
    data_for_parent->myNzy += Nzy;
    data_for_parent->myNzx += Nzx;
    data_for_parent->myNxz += Nxz;
```

```
// Adjust Nijk for the change in centre P
data_for_parent->myNijkDiag += T(2)*displacement*child_data.myNijDiag + displacement*displacement*child_data.myN;
data_for_parent->mySumPermuteNxyz += (displacement[0]*(Nyz+Nzy) + displacement[1]*(Nzx+Nxz) + displacement[2]*(Nxy+Nyx));
data_for_parent->my2Nxxy_Nyxx +=
    2*(displacement[1]*child_data.myNijDiag[0] + displacement[0]*child_data.myNxy + N[0]*displacement[0]*displacement[1])
    + 2*child_data.myNyx*displacement[0] + N[1]*displacement[0]*displacement[0];
data_for_parent->my2Nxxz_Nzxx +=
    2*(displacement[2]*child_data.myNijDiag[0] + displacement[0]*child_data.myNxz + N[0]*displacement[0]*displacement[2])
    + 2*child_data.myNzx*displacement[0] + N[2]*displacement[0]*displacement[0];
data_for_parent->my2Nyyz_Nzyy +=
    2*(displacement[2]*child_data.myNijDiag[1] + displacement[1]*child_data.myNyz + N[1]*displacement[1]*displacement[2])
    + 2*child_data.myNzy*displacement[1] + N[2]*displacement[1]*displacement[1];
data_for_parent->my2Nyyx_Nxyy +=
    2*(displacement[0]*child_data.myNijDiag[1] + displacement[1]*child_data.myNyx + N[1]*displacement[1]*displacement[0])
    + 2*child_data.myNxy*displacement[1] + N[0]*displacement[1]*displacement[1];
data_for_parent->my2Nzzx_Nxzz +=
    2*(displacement[0]*child_data.myNijDiag[2] + displacement[2]*child_data.myNzx + N[2]*displacement[2]*displacement[0])
    + 2*child_data.myNxz*displacement[2] + N[0]*displacement[2]*displacement[2];
data_for_parent->my2Nzzy_Nyzz +=
    2*(displacement[1]*child_data.myNijDiag[2] + displacement[2]*child_data.myNzy + N[2]*displacement[2]*displacement[1])
    + 2*child_data.myNyz*displacement[2] + N[1]*displacement[2]*displacement[2];
```

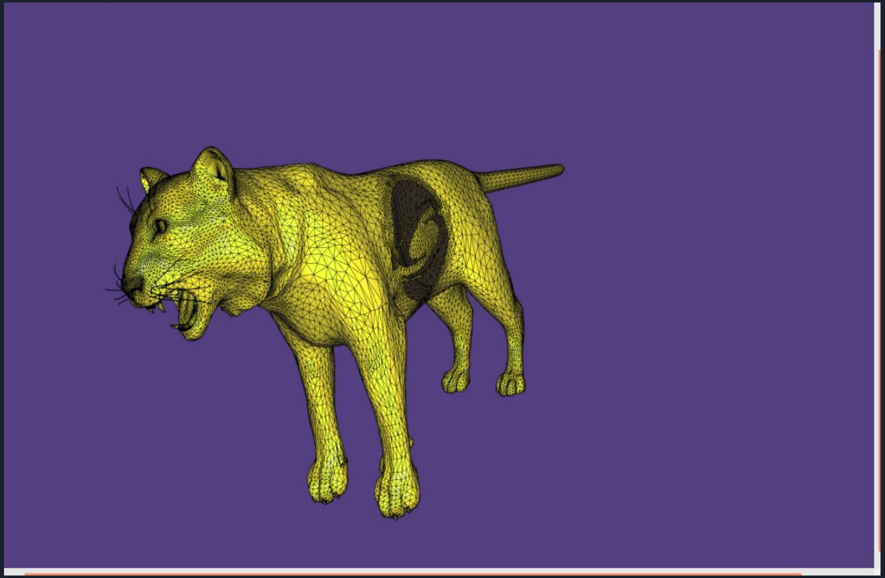
Object representation

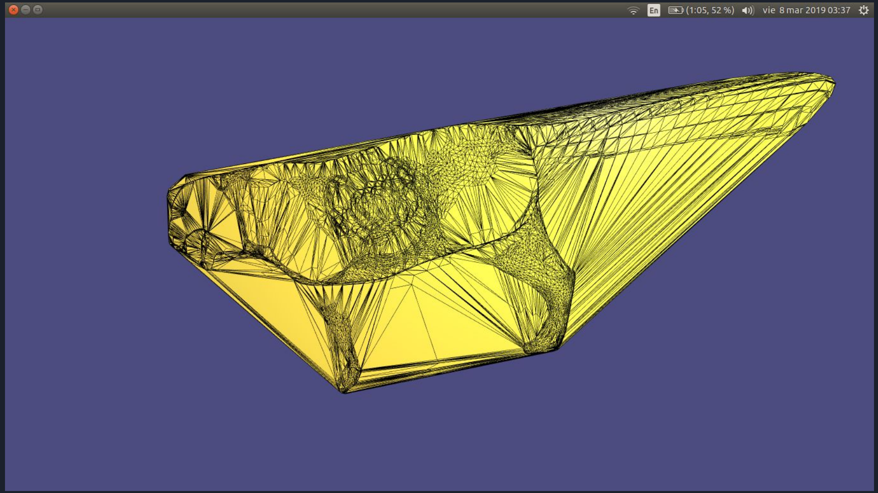
Results - Rabbit



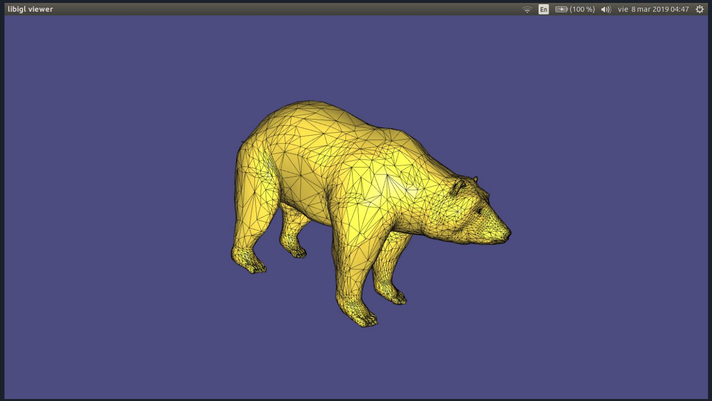


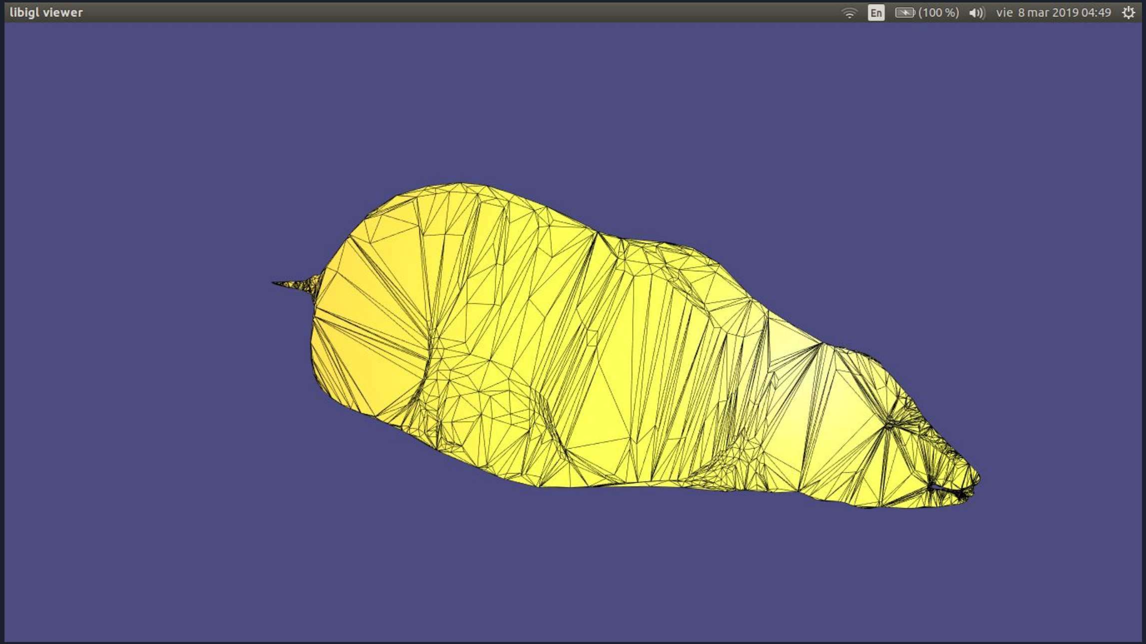
Results - Cat



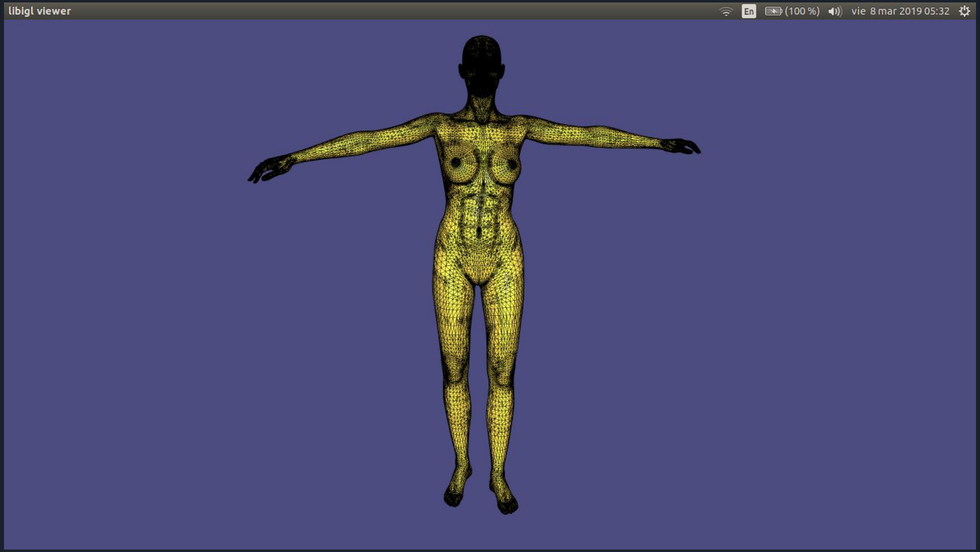


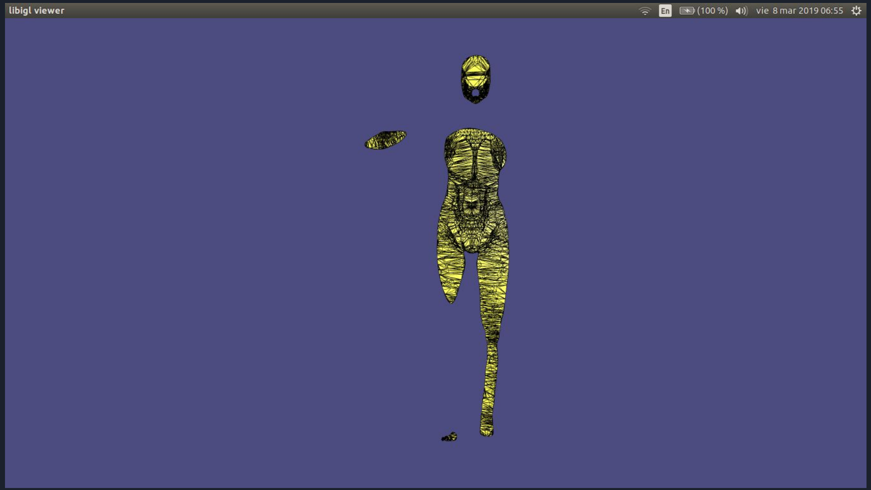
Results - Bear



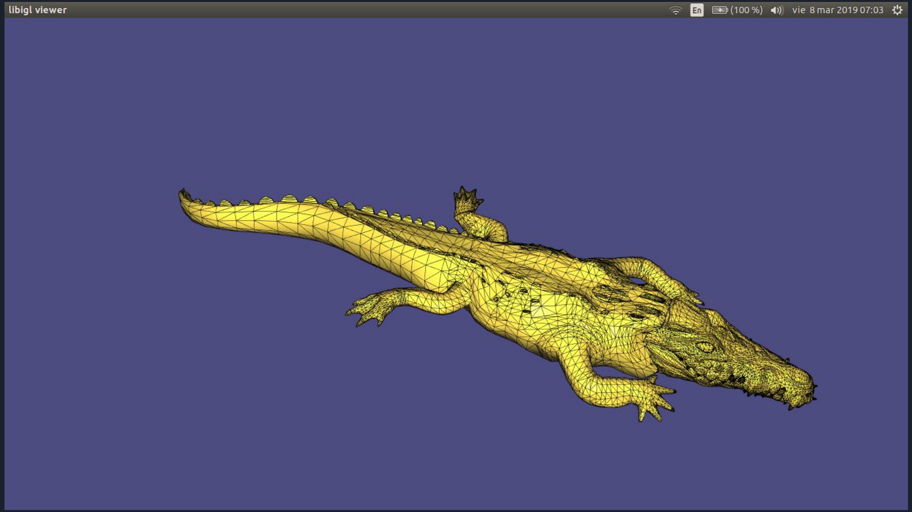


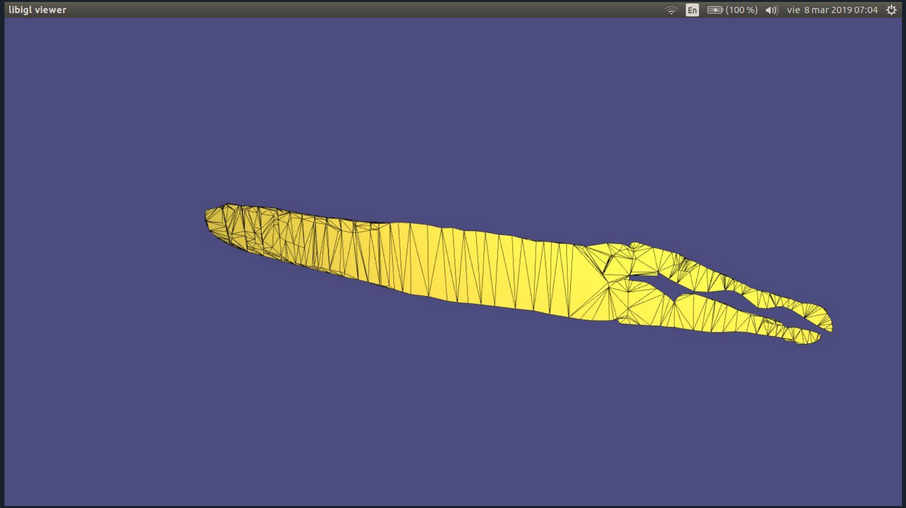
Results - Girl





Results - Crocodile





Results

Mesh	Winding Numbers	Calculate Winding Numbers (ms)	Calculate Fast Winding Numbers (ms)
Big-Sigcat	164916	107014	3242
Bunny	34055	3794	310
Bear	56605	65600	14769
Crocodile	98719	29242	149
Girl	615313	603822	4954
Beast	192613	93014	5836

Thanks