**Capabilities**

The PSystem plugin can load in and draw any number of objects you have saved as ".obj" files using only planar geometry. These files can be exported easily from Rhino. (However, oddly, Rhino cannot read them, so save the original in Rhino's native format too so you can edit it in Rhino again later.)

Once you have gotten your object in, you can attach particles of different types to the various layers you have set up in your file. That means different layers of the object can have different behaviors and interactions.

**Exporting**

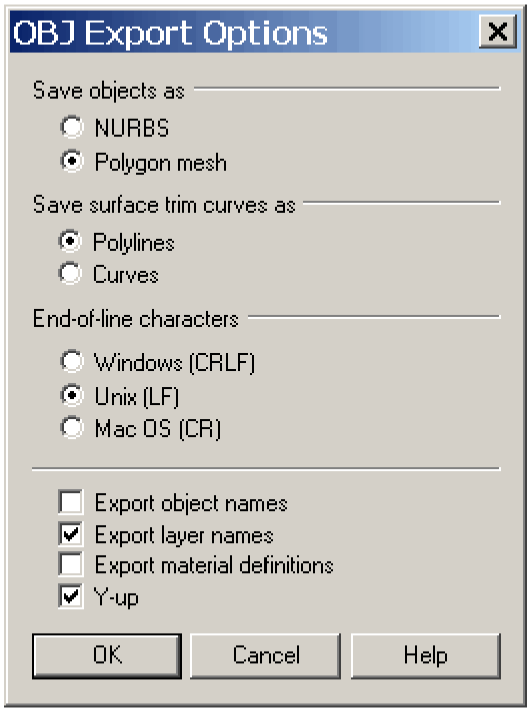
To prepare an object for export from Rhino, first divide your model into as many layers as you want. Place the pieces of your model that you want to manipulate individually on the different layers. Name your layers in a way that makes sense. You will be referring to the layers by name from MultiProcessing. (Any spaces that appear in your layer names will be converted to underscores "_" when you refer to that layer from MultiProcessing.)

Select either "Save as..." or "Export Selected..."

Choose "Wavefront (*.obj)" as the file format, and choose the data folder of your project, and hit "Save."

In the "OBJ Export Options" dialog, choose the options as shown to the right exactly.

Hit "OK"


OBJ Export Options

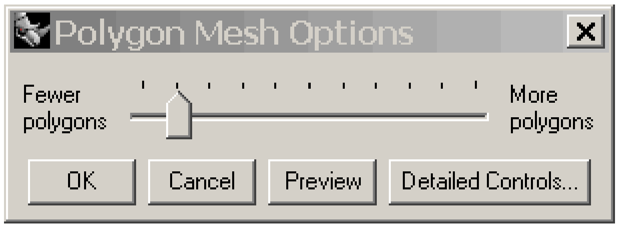
Save objects as
- ○ NURBS
- ● Polygon mesh

Save surface trim curves as
- ● Polylines
- ○ Curves

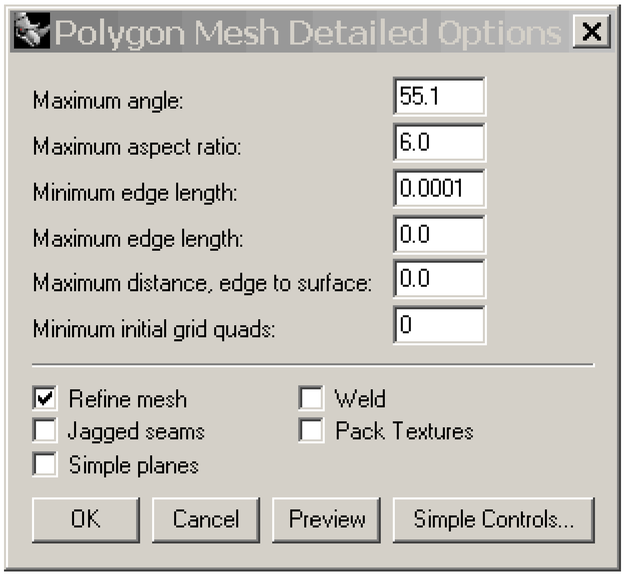
End-of-line characters
- ○ Windows (CRLF)
- ● Unix (LF)
- ○ Mac OS (CR)

- ☐ Export object names
- ☑ Export layer names
- ☐ Export material definitions
- ☑ Y-up

From the "Polygon Mesh Options" select the lowest number of polygons that looks reasonable when you preview it. This will really help out speed once you import it.



You can also select "Detailed Controls..." to get this dialog, which gives you a ton of control. It may well be worthwhile.



| | |
|---|---|
| Maximum angle: | 55.1 |
| Maximum aspect ratio: | 6.0 |
| Minimum edge length: | 0.0001 |
| Maximum edge length: | 0.0 |
| Maximum distance, edge to surface: | 0.0 |
| Minimum initial grid quads: | 0 |

- [✓] Refine mesh
- [ ] Jagged seams
- [ ] Simple planes
- [ ] Weld
- [ ] Pack Textures

Anyhow, save the ".obj" file, and you're ready to load it from MultiProcessing.

**Loading**

Loading geometry is quite easy. All you need to do is decare an object of type Surface, and use `ps.loadSurface("NAME");` load to the file. Remember, the file must be located in the "data" directory of your project.

```
Surface yourNameHere = ps.loadSurface("fileName.obj");
```

That's it. The object is now loaded into the variable named "yourNameHere." You can reference it anywhere that variable is in scope. That means that if you're going to be using it from multiple functions, you probably want to declare it globally.

The object is scaled up ten times automatically after it loads so that it is not so tiny, but if you want to change the scale on it, use the "scale(factor)" command:

```
yourNameHere.scale(factor);
```

**Displaying**

Again, could hardly be easier. Put this in your loop:

```
yourNameHere.draw();
```

You can also add any image as a texture to your object with the command. You can manipulate the texture coordinates in Rhino before exporting if you like.

```
yourNameHere.setTexture("imageName.jpg");
```

**Manipulating**

This gets slightly more complicated. Now that you've imported the geometry, you want it to change form. But it's just this crazy mess of data, so what to do?

Here's what: attach particles to each of the vertices in a specified layer, and then control those particles. So first, you need to define a particle type that you want to apply to these vertices. That could look something like this:

```
class RectParticle extends Particle {

  public RectParticle() {
  }

  void draw() {
    if (fixed()) {
      fill(255, 0, 0);
    }
    else {
      fill(100, 100, 100, 100);
    }
    super.draw();
  }

}
```

This defines a particle type that draws itself as a box (red if the particle is fixed, and gray if it isn't). It has no other behavior, so it will just be acted upon by forces.

Once you've defined your particle type, you use the "applyParticles" command to spread them onto a layer of your object:

```
yourNameHere.applyParticles("Layer_01", new RectParticle());
```

The layer name in the first argument must match one of the layers from your obj file exactly including capitalization. Remember that spaces are converted into underscores "_". Having done that, you'll end up with something like this:

Notice that the layer we have chosen is now sinking out of sight. That's because the particles are affected by gravity. So let's turn gravity off, which is more likely what we were after:

```
ps.setGravity(0);
```

Now these particles are useful for debugging because you can see them, but they may not be what you want in the long term. Here is a particle type that does absolutely nothing. No drawing, no moving. Nothing. The only thing this particle type does is to be affected by forces.

```
class NoShowParticle extends Particle {
  public NoShowParticle() {
  }
  void draw() {
  }
}
```

This is a useful template for the kinds of particles you may want to spead on a layer.

If you wanted to have a layer move slowly to the right, you could spread this kind of particle on it:

```
class RightParticle extends Particle {
  public RightParticle() {
    fix();
  }

  void draw() {
    pos[0] = pos[0] + 1;
  }
}
```

The particles are fixed, so they won't be affected by forces, and they are moved one unit to the right in their loop each frame, so the layer they are spread on moves to the right, retaining its original shape. You can use variations on this to have layers move or deform in any arbitrary way you choose.

## Keeping the original shape

Let's say you decided it would be fun to have all of the particles on a layer be attached by a spring to another particle outside the layer. That would be pretty easy to do:

But there's a problem with this in that the original shape of the layer has nothing to do with the final form. There is no pressure for the layer to retain its starting shape.

We can add such a pressure by applying springs at each vertex to the original position of that vertex. There is a utility function to do this called

```
yourNameHere.addSpringsToOriginalShape("Layer_01");
```

Again the layer name refers to a layer in the obj file. You can specify how strong the springs keeping the original shape are with this version of the function:

```
yourNameHere.addSpringsToOriginalShape("Layer_01", strength);
```

Here strength is the spring strength. For reference, a strength of 1.0 is a very strong spring.

**Too many forces**

It is easy to get carried away with these techniques and produce something intolerably slow. For instance, you may be using a particle type that adds a magnet to itself:

```
class MagnetParticle extends Particle {
  public MagnetParticle() {
    Magnet m = addMagnet(this);
    m.strength = -0.5;
  }

  void draw() {
  }
}
```

This would let you define a layer that other layers moved to avoid. But it will be disastrously slow because each particle has its own magnet, and each magnet applies to all particles.

The solution is to attach a single magnetic particle to the centroid of the layer. The centroid is the average vertex position, about where you'd expect. That way you can use a single strong magnet that will move with the layer and other layers will react to:

```
yourNameHere.attachParticleToCentroid("Layer_01",
particleType);
```