Computer Science 112
Art & Science of Computer Graphics
The College of Saint Rose
Fall 2015

# Topic Notes: Defining Custom Objects

Our next topic introduces a number of additional Ambrosia constructs that will allow us to build more complex objects.

Our primitive Mead objects can only get us so far. At some point, you will want something beyond spheres, cubes, cones, cylinders, and transformations of them. We will consider a number of new mechanisms that will allow us to build more complex objects with Mead.

---

## Mesh Objects

We will look next at a way to define our own objects out of polygons – a *mesh* object.

We begin with an example – our own custom definition of a cube:

**On the Wiki:** Meshcube

In this example, we define a list of points that form one of the sides of our cube:

```
front = [(-50,-50,-50), (50,-50,-50), (50,50,-50), (-50,50,-50)]
```

These are the four corners of the "front" of the cube.

Now, anywhere from here on in our program, the name `front` can be used when we want to have that list of 4 points.

This is an example of a *polygon* and will come up in many of our custom object mechanisms.

For our purposes, the points of our polygon should be *co-planar*.

We next define the box out of 6 polygons:

```
box = Mesh()
box.addPoly(front)
box.addPoly(xRot(90).mapPoly(front)) # top
box.addPoly(xRot(180).mapPoly(front)) # back
box.addPoly(xRot(-90).mapPoly(front)) # bottom
box.addPoly(yRot(90).mapPoly(front)) # left
box.addPoly(yRot(-90).mapPoly(front)) # right
```

We are creating an instance of a new class called `Mesh`, to which we will add a series of polygons that define an object.

Notice that we have a different message, `addPoly`, used to add polygons to a `Mesh`.

We could have defined separate polygon names for each of the 6 sides, *e.g.*:

```
back = [(-50,-50,50), (50,-50,50), (50,50,50), (-50,50,50)]
```

but instead we will take the `front` polygon and apply transformations to it! The syntax looks a little different here but the ideas are similar to how we transformed our primitive objects.

Let's look at one:

```
xRot(90).mapPoly(front)
```

sends a message to the transformation that rotates about the x-axis by 90 degrees to map a polygon using this transformation, and we send it our polygon `front`.

We can see the results of this transformation interactively in Python:

```
from ambrosia import *
front = [(-50,-50,-50), (50,-50,-50), (50,50,-50), (-50,50,-50)]
xRot(90).mapPoly(front)
```

We get back the coordinates of the polygon we add as the "top" of our box:

```
[(-50.0, 50.0, -50.0), (50.0, 50.0, -50.0), (50.0, 50.0, 50.0), (-50.0, 50.0
```

Redefining objects we already have (if we want a cube, we'd just use `cube` or `Cube()`, of course) is not very exciting. The big benefit is that we can now define objects other than those we could already get, such as a pyramid:

**On the Wiki:** Pyramid

Note that when the polygons added to a `Mesh` completely enclose some space, a solid object is produced. Otherwise, it is just the surfaces.

---

## Predefined Meshes

Ambrosia includes a few predefined `Mesh` objects that you might find interesting:

- The Newell Teapot
- The Stanford Bunny
- A Cow

- A Horse
- A Buddha
- A Dragon
- A Hand

Several of these are standard reference objects in computer graphics. Teapots show up in unexpected places in computer-generated animated films, for example.

A simple model that includes Ambrosia's definitions of these:

**On the Wiki:** Predefinedmeshes

Note that we need additional import statements to tell Python we will be using them. They do not come to use as standard definitions when we import Ambrosia's core functionality.

---

## Extruded Mesh Objects

There are a few Ambrosia methods that help us generate specific kinds of `Mesh` objects in a more convenient manner. The first of these that we will consider is the `extrude` method, which is used to build solid objects.

The idea of the `extrude` is to take a polygon that represents the "front face" of the object we wish to construct. We then apply a transformation of that polygon, which results in the "back face" of the object. Then the corresponding segments of the front and back faces are connected by other polygons (note they will be quadrilaterals).

A very simple application of this idea allows us another way to build a `Mesh` of our $100 \times 100 \times 100$ cube.

```
from ambrosia import *
front = [(-50,-50,-50), (50,-50,-50), (50,50,-50), (-50,50,-50)]
box = extrude(front, translate(0,0,100))
```

Again, that's not interesting. The interesting part is that we can take this idea and define any polygon we wish and use any transformation we wish. In this example, we define a polygon which is in the shape of a 4-point star in the xy-plane, and use an `extrude` to give it thickness to become a solid object.

**On the Wiki:** Extrudestar

There are a few things of note in this example. First, for simplicity, the polygon is defined in two dimensions, not three:

```
starFront2D = [(0,50), (10,10), (50,0), (10,-10), (0,-50), (-10,-10),
               (-50,0), (-10,10)]
```

However, since our `extrude` method expects a polygon defined in three-dimensional space, we need to tack on the third coordinate. Assuming we want them all to be z=0 (as we do here), there is another Ambrosia function that does just this:

```
starFront = raisePoly(starFront2D)
```

If we enter the above two statements into Python interactively, then type the name "`starFront`" to see its value, we see exactly that: our polygon with the same x- and y-coordinates, and z=0 in each:

```
[(0, 50, 0), (10, 10, 0), (50, 0, 0), (10, -10, 0), (0, -50, 0),
 (-10, -10, 0), (-50, 0, 0), (-10, 10, 0)]
```

With the polygon in three-space now in hand, we can use `extrude` to construct the `Mesh` object:

```
star = extrude(starFront, translate(0,0,100))
```

This `Mesh` consists of the two polygons that form the "ends": our original polygon and the one with the same x- and y-coordinates but now z=100 in each (*i.e.*, the result of the `translate`). It also has the 8 quadrilateral polygons connecting up corresponding pairs of edges of the two "end" polygons.

Before we look at our next example, we introduce another Ambrosia method, called `polygon`, that will help us build some of the common polygons we will often want to use: the regular polygons. The function works as follows:

```
polygon(4)
```

This one would print out the coordinates of a regular 4-sided polygon, better known as a square:

```
[(50.0, 0.0), (3.061616997868383e-15, 50.0), (-50.0,
6.123233995736766e-15), (-9.184850993605149e-15, -50.0)]
```

Yikes! That's messy until we point out that those ugly numbers like "3.061616997868383e-15" should really be read as "$3.061616997868383 \times 10^{-15}$, which is for all intents and purposes, 0. So this is the square with corners along the x- and y-axes at 50 and -50.

We can try these out with larger numbers of sides, and in each case we get a set of points that form a polygon with equal length sides and equal angles.

In the next example, we use `polygon(5)` to generate a polygon representing a regular pentagon. And then it is used in an `extrude`. But in this case, we also add a rotation to the transformation to introduce a "twisting" effect in the resulting `Mesh` object.

**On the Wiki:** Extrudepenta

We can take this to somewhat of an extreme by using a 100-sided polygon as our starting polygon and rotate by a large amount during the extrude to create an "hourglass" object.

**On the Wiki:** Hourglass

This is a pretty flexible construct and we will see more interesting examples later.

---

## Swept Mesh Objects

We can create mesh objects by "sweeping" a polygon as well.

We begin with a polygon – the *cross section* of our desired object, which must be in the xy-plane with all points having non-negative x coordinates:

```
littleRectangle = raisePoly([(0,10),(200,10),(200,-10),(0,-10)])
```

We can then "sweep" this polygon about the y-axis to generate our `Mesh`:

```
platform = sweep(littleRectangle, 12)
```

The result of the `sweep` operation is a `Mesh` object consisting of 12 copies of the original polygon, evenly spaced in a circle about the y-axis (in this case, one every 30 degrees), with neighboring copies connected to each other by solid material.

If we increase the 12, the object begins to look more and more round, but the `Mesh` object created is more and more complex, resulting in a more expensive rendering by Ambrosia.

**On the Wiki:** Sweptplatform

If we do not have an edge of our cross section polygon aligned along the y-axis, we can create an object with a hole:

**On the Wiki:** Holeyplatform

---

# Spindle Objects

While the sweep operation can approximate a round object (with a sufficiently large second parameter), a `Spindle` object will produce a truly round object:

```
washer = Spindle()
washer.profile(littleRectangle)
```

**On the Wiki:** WasherUnlike the results of `extrude` and `sweep`, a `Spindle` object is not a `Mesh`. It is a smoothly turned object, as turned on a lathe or thrown on a potter's wheel, determined by a polygonal profile. The rotation is always about the y-axis, and the profile is, as with sweep, determined by the profile polygon drawn in the xy-plane (but without negative x coordinates).