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

Topic Notes: Constructive Solid Geometry

We next move into our first set of tools to construct different kinds of objects. We will use *Constructive Solid Geometry (CSG)*, which allows us to form complex objects from simple objects using three basic operations: *union* (using Ambrosia's `Group`), *difference* (using Ambrosia's `Difference`), and intersection (using Ambrosia's `Intersection`).

Defining Groups

As we develop all but the most trivial models, it will be essential to break the more complex objects we wish to model down into simpler and simpler objects until (at least for now) we need only our primitives (*e.g.*, cubes, spheres, *etc.*). A fundamental idea in this process will be the `Group`.

A `Group` accumulates the objects added to it into a compound unit. We have been working with one `Group` all along: our `scene` is an Ambrosia `Group`.

But we will define out own `Group`s. The idea is that we will add one or (usually) more objects to our `Group`, then use that `Group` in just about any context where we'd use one of our primitive objects. We can add the `Group` to the `scene`, apply transformations and material properties to it, and more.

As an example, we will revisit the ice cream cone model, but this time, create a `Group` that represents an ice cream cone, then add multiple instances of that `Group` to the `scene`.

On the Wiki: Icecreamcones

Our `Group`, named `iceCreamCone` consists of the three objects: a `cone` and two `spheres`. We then add two instances of our `iceCreamCone` to the scene, and can apply additional transformations that get applied to the entire `Group`.

This is a very powerful tool to help us build more complicated models.

Next, let's consider a further enhancement of this idea:

On the Wiki: Icecreamcones2

Notice in this one that the component for the cone is given a material property (`yellowPlaster`), but the scoops of ice cream do not have material property specified. If we just added one of these to the screen, the ice cream would have a dark grey material (Ambrosia's "default material").

This allows us to specify a material property when we add each `iceCreamCone` to the `scene`. Notice that Ambrosia applies this material property to only those components of the `Group` that were not given a property earlier. So the scoops of ice cream get the material, but the cone remains `yellowPlaster`.

An additional example using `Groups`:

On the Wiki: Snowmen

Differences and Intersection

Intersections and differences work as you might expect from what the names suggest, mathematically.

- an object defined by the *intersection* of a set of objects is the portion that is in common to all objects
- an object defined by the *difference* of a set of objects is the portion of the first that is not also part of the others

Just using a few of our primitive objects, we can see how these work:

```
halfSphere = Difference()
halfSphere.add(sphere)
halfSphere.add(cube, translate(50,0,0))
```

The resulting object is the "left" half of the unit sphere (the part in negative x). The object specified by the second `add` is "subtracted" from the object specified by the first.

Note carefully that order matters for differences. If we reverse the order above, we get a cube with a half sphere scooped out.

For an intersection, order doesn't matter:

```
halfSphere = Intersection()
halfSphere.add(sphere)
halfSphere.add(cube, translate(50,0,0))
```

Here, we also get a half sphere, but only the half that is covered by the cube instead of the half that was not covered by the cube.

In each of these cases, we use the `add` message to include components in our CSG unit, even when the operation is more logically thought of as a "subtract" or "intersect" instead.

A simple example using an intersection:

On the Wiki: Arrows

And one using a difference:

On the Wiki: Cups

A much more complex example, mostly using groups but also a difference:

On the Wiki: Lamp

Note the use in these examples of names that we define in place of numbers. This is a good programming practice. When you have a number in your program that has a specific meaning in the context in which it's used, define a name for it and use that name instead. It will make your program easier to read and understand both for you and for others who might be trying to understand it.