Introduction to OpenSCAD

Steve Graves

Teaching Goals

- Concepts Not Nuts and Bolts
- Ask questions. Classes are for active learning. There are probably many levels of experience in class. Don't be shy, everyone was a nubie once.
- If a question refers to something coming up, it is a good question. But I might defer it.
- I plan on laying the foundation first, but have several useful concepts that go a little further than the basics. Going to try to touch all of them.
- I expect that after this class you will feel confident to create a simple object.

Use the Manual

- I am not going to regurgitate the manual.
- Concepts are more important than details.
- In the real world, open books are allowed.
- Use help menu to open manual.
- Don't clutter mind with parameter details. It will happen automatically.

Concepts

- What is OpenSCAD?
- What is the 3D printing process and how does OpenSCAD fit in it?
- What are basic concepts for all programs that create 3D objects?
- What is OpenSCAD's syntax and how is it unique?
- What are some OpenSCAD user interface tips?
- What are some design patterns?
- Use Divide and Conquer.
- Don't reinvent the wheel.

What is OpenSCAD?

- Program to design 3D objects. It has become the defacto standard on thingiverse for parametric designs.
- Open source and free.
- Other free programs are Blender, SketchUp, FreeCad and HeeksCad
- OpenSCAD generates objects from programs(scripts?) and is mostly non-interactive.
- A program means parametric designs that are mathematically precise.

OpenSCAD User Interface

- Left hand side is text edit window for writing code. Other text editors can be used. Xemacs is a good example.
- Right hand side is window for inspecting rendered 3D object.
- Object in right hand window can be manipulated using standard OpenGL mouse moves.
- Left mouse allows one to rotate object.
- Right mouse pans.
- Middle mouse or wheel zooms.

What is the 3D Printing Process?

- 3D object is designed (OpenSCAD)
- An STL file is exported.
- A slicer program breaks STL file into layers and generates G code.
- G code is sent to printer.

General 3D Object Creation Concepts

- Key concept is “Transformed Shapes are Combined” into new more complex shapes.
- Complex objects are created from simple “Shapes” called primitives.
- Simple and complex shapes are modified with “Transformations” like translate, size, color.
- Primitive shapes are “Combined” in various ways using boolean logic and other more complicated functions.
- Auxiliary functions are used to do math, etc.

3D (and 2D) Primitives

- Cube
- Sphere
- Cylinder
- Polyhedron
- Square (2D)
- Circle (2D)
- Polygon (2D)

Making 3D from 2D

- 2D primitives are turned into 3D by extrusion
- Two basic types of extrusion
- linear_extrude
- rotate_extrude

Combining Shapes

- Union (All shapes added together)
- Difference (Shapes after 1st shape are subtracted)
- Intersection (Intersection of all shapes)
- Minkowski (Function with interesting properties to add two shapes)
- Hull (Enclose multiple shapes in minimum shape)

Transforming Shapes

- Scale
- Resize
- Rotate
- Translate
- Mirror
- Multmatrix
- Color

OpenSCAD 3D Object Creation

3D objects are created using the general pattern

```
Combine(){
    optTransform() optTransform() ... Shape();
    optTransform() optTransform() ... Shape();
    ...
}
```

Work from end of line toward front of line.

Work from innermost to outermost

Syntax

- Functions (modules) are overloaded.
- Variables have unusual scope.
- Variables do not store shapes. Shapes are implicit entities.
- Arrays [] are used a lot to represent points or vectors. Very often as parameters to functions. Beware! Remember to use [] as well as (). In most cases there is no syntax error if you don't.
- Lines are terminated with semicolon.
- A “line” of code has a shape just before the semicolon. That shape can be “transformed” by functions before it. These functions have no semicolons. The transformations are applied in reverse order (closest first and working back).

More on syntax

- We can learn more about the working syntax by looking at the syntax for a module.
- Modules implicitly return shapes.
- So a module must create a shape internally or get one “passed” to it.
- Shapes are “passed” as children, not parameters. Remember shapes are not stored in variables.

Module Syntax

```
Module name(parameter1, parameter2, ...){
    Code that creates a shape internally
        and/or
    Code that operates on children called child(0),
    child(1) up to child($children - 1)
}
```

Calling a module

- Three general forms. Each relates to earlier pattern of Combine, Transform and Shape

Calling a module as a Combine

```
Name(Para1,...){
    Child(0);
    Child(1);
    …
}
```

- Child is any construct that makes a shape
- i.e. line of optTransform() Shape();

  or the form of Combine(){ , call to another module, etc.

Calling a module as transform

Name(optPara1, optPara2, …) Child;

- Child is any construct that makes a shape.

Calling module as shape

Name(optPara1, optPara2, …);

- Note no children, ends with semicolon
- A shape is created internally to module
- Modules making shapes are a common form.
- For example, a module to create a bolt might be

  bolt(dia, len);

User Interface Tips

- Use “Thrown Together” view and “Compile” for a quick and dirty look at design
- Break large systems into files to best use above technique. Use .stl files to speed up rendering in complex designs.
- Use “Compile and Render” for more refined look (and longer wait for processing)
- There is a comment/uncomment function
- Set $fs high during development

Useful Design Patterns

- Identify all the parameters for object(s) being created
- Assign a variable to each parameter before creating design

More Design Patterns

```
difference(){
    union(){
        union child(0);
        union child(1);
    }
    difference child(1); (Union is difference child(0))
    difference child(2);
    …
}
```

More Design Patterns

```
module oneShape(){
    ...;
}

module shapeTwo(){
   ...;
}

union(){
   transformations() shapeOne();
   transformations() shapeTwo();
}
```

Divide and Conquer

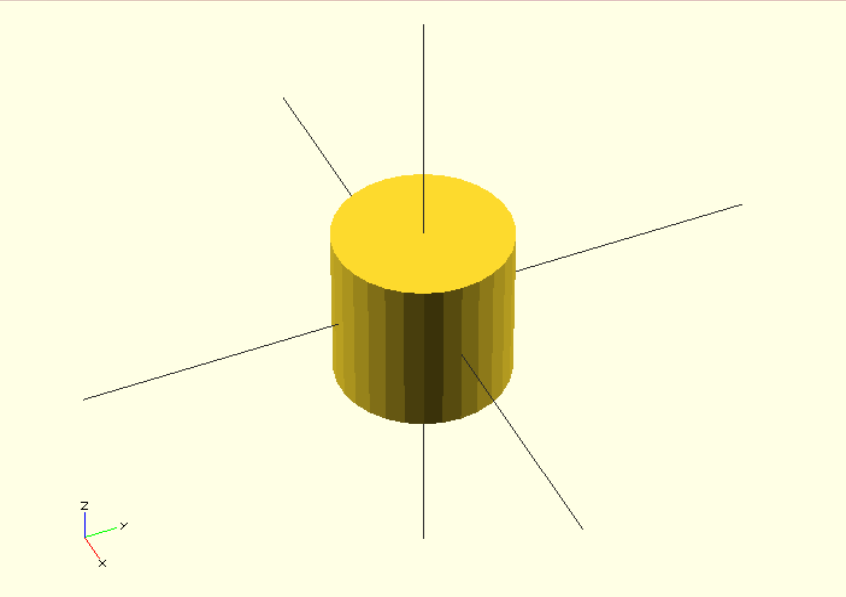
- Shapes are obvious
- Moves and rotates can also be divided. Create parts in local coordinate systems, then move them into position in global coordinate system.

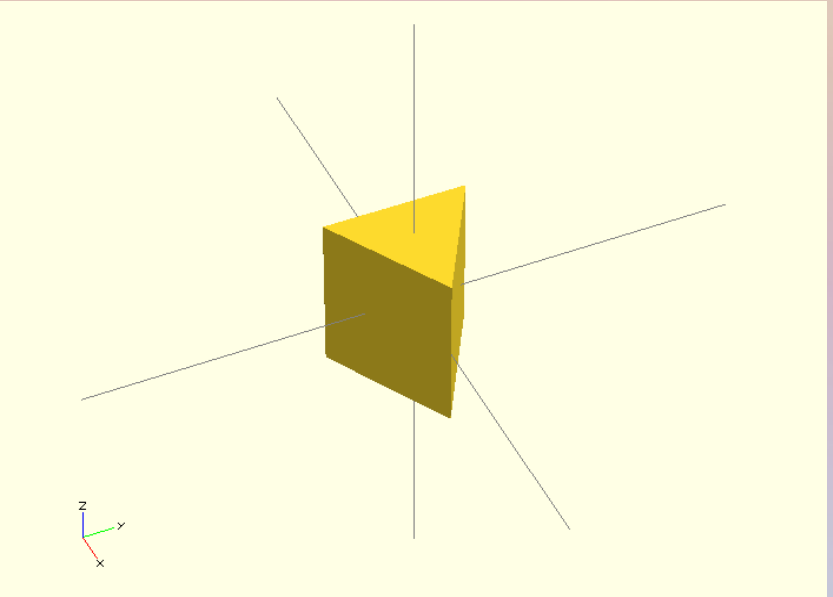
Don't Reinvent the Wheel

- Somebody has already written a “Wheel” module.
- Use builtin MCAD library
- Search thingiverse
- Search the Internet

Tip: Use $fn for more shapes

- The $fn parameter can be used with cylinder to create shapes with sides.

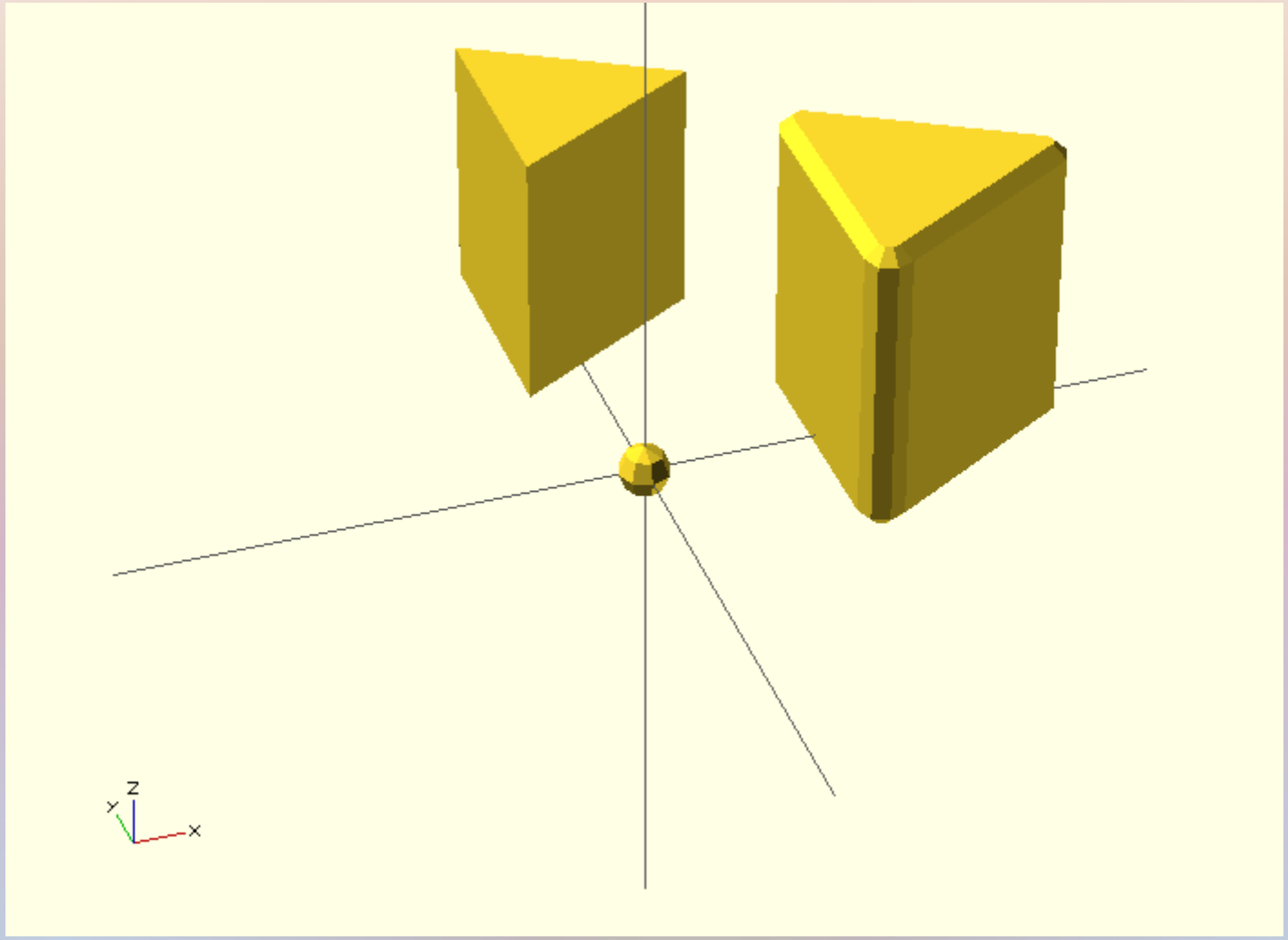




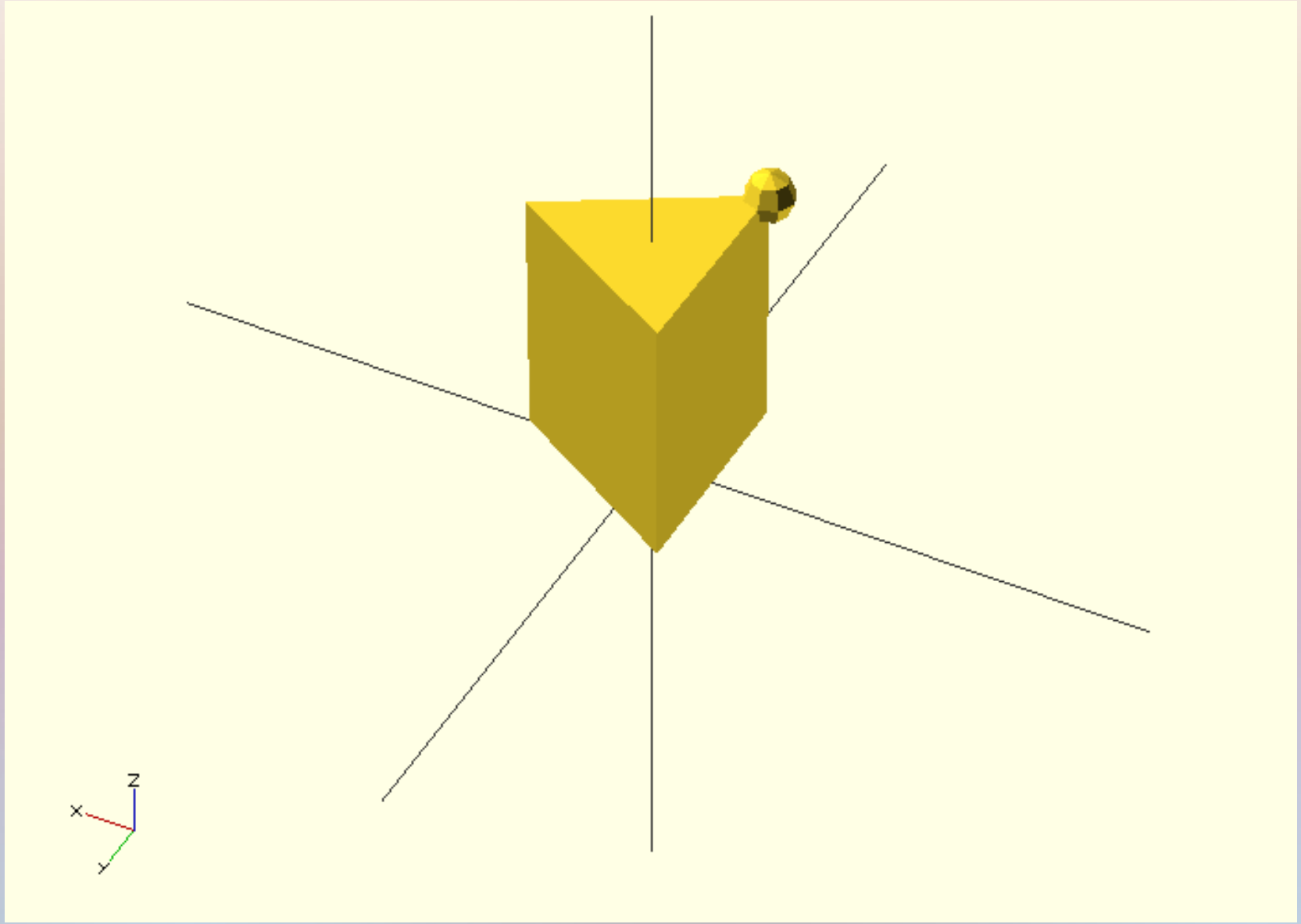
Minkowski (Operation!)

- Very useful but hard to understand.
- One shape is added to another shape at every point in the shape. Order doesn't matter.
- Similar to matrix addition.
- The additions at the surface are the only ones that have an effect.
- Imagine one shape being smeared onto the other shape.
- Warning: Minkowski is incredibly slow with spheres.

Minkowski Example



Minkowski Animated



Use module for shape “variables”

```
/*
 * This creates a shell of given thickness
 * around an arbitrary shape
 */
module shell(thickness){
    difference(){
        minkowski(){
            child(0);
            sphere(r=thickness, center = true);
        }
        child(0);
    }
}
shell(10){
    cylinder(r=25);
}
```

Creating manifold designs

- Manifold effectively means water-tight.
- Leaks happen at shared boundaries. Shared boundaries also slow down processing.
- Extend objects that subtract beyond surface.
- Either use “glue” objects or extend surfaces into other objects for unions.
- Spheres can sometimes fail to be manifold. Use circle and rotate_extrude instead.

More on manifold designs

- Use thrown together and compile to see questionable boundaries.