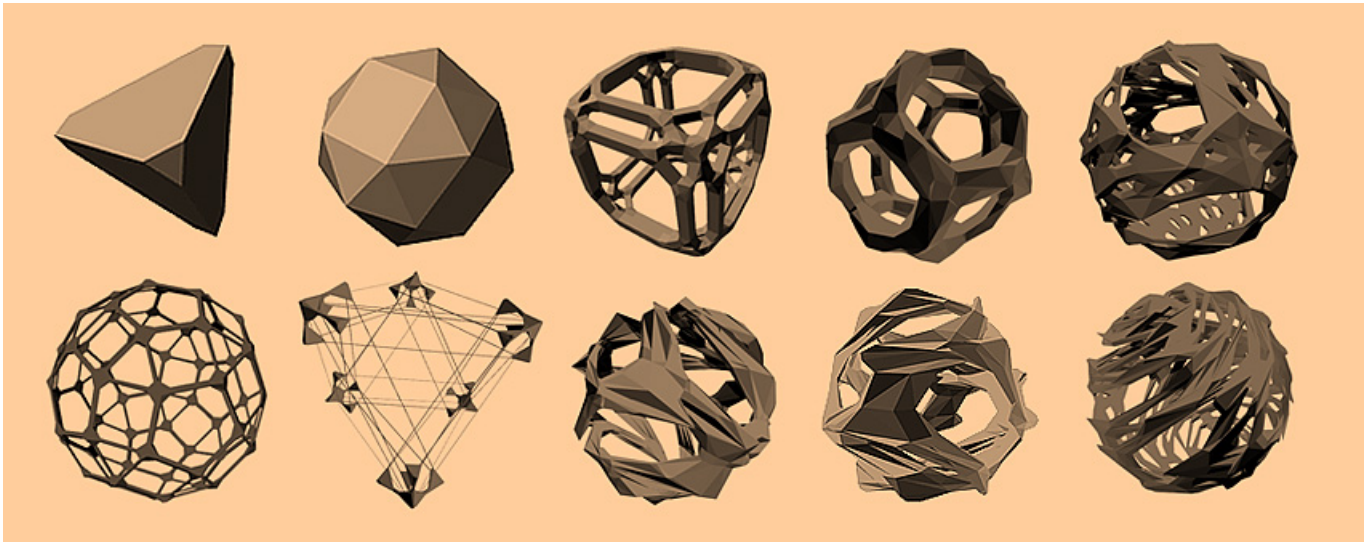
Hemesh Part 2

« Previous / Next »

By mark webster / January 19, 2015 / Learning / No Comments



Evolution of shape

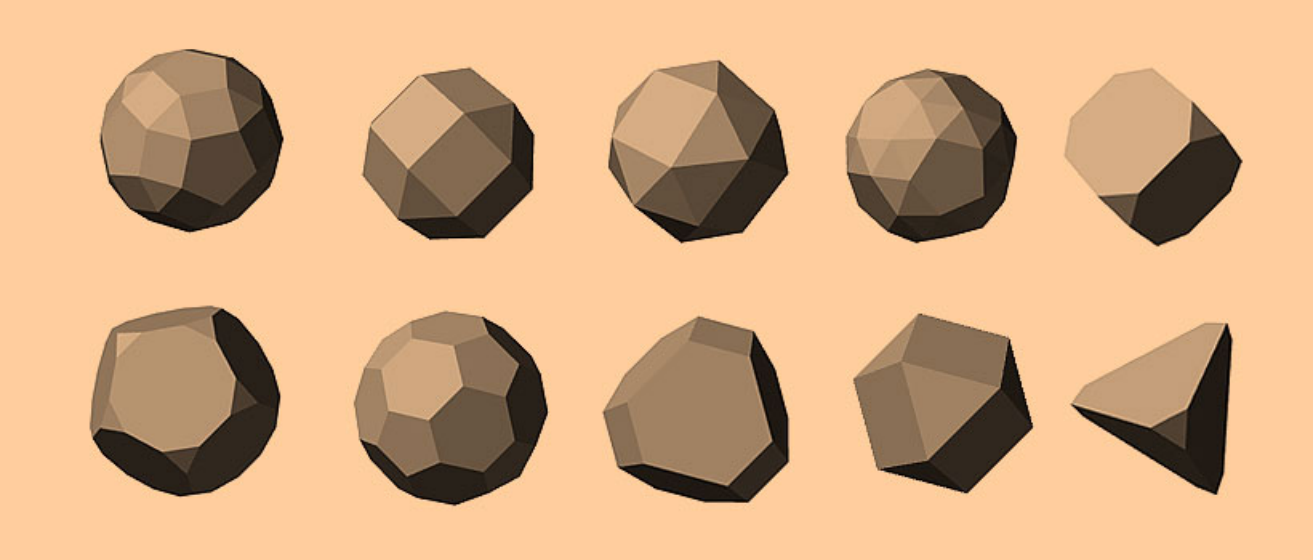
//
—— **HEMESH Library by Frederik Vanhoutte** ———-
//
Frederik Vanhoutte's website

Source Code :
Sketches Processing
Hemesh Library

//

INTRODUCTION

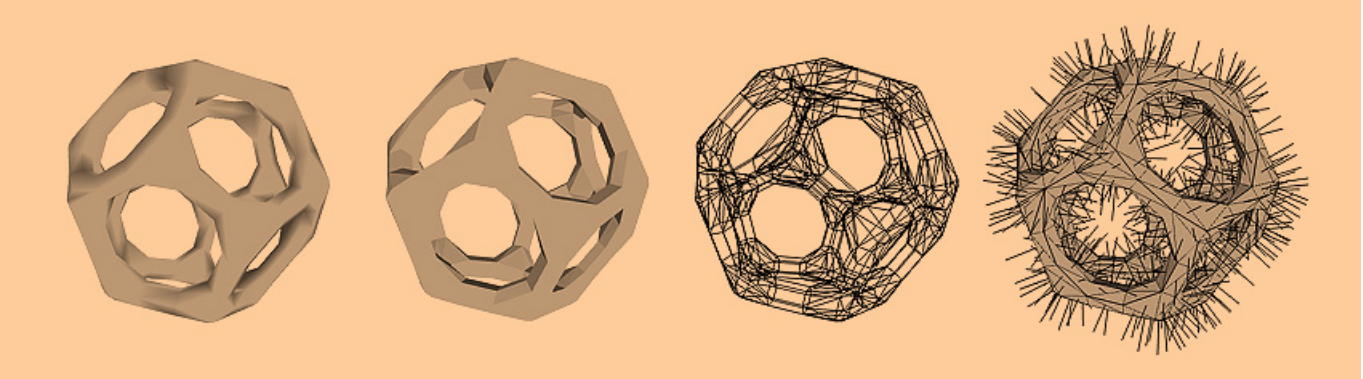
In the following examples, we are going to look at how we can implement key input with Hemesh to modify our shapes on the fly. It's not really practical to have to re-execute code constantly when working with systems for exploring form. Ultimately, we want to aim for flexible systems that enable us to visualise modifications in real time. In order to do this, we are going to develop a few sketches that use key inputs. These will serve as development for later sketches that implement the ControlP5 library. It's worth following the evolution of each, simply because new functionalities are added step by step and we begin to structure the code a little differently from those introduced in the first tutorial.



10 of the 13 Archimedean Solids

We are also going to discover some new shapes with our creator classes in HE_MESH. In the first sketch we work with the dodecahedron form which is a polygon made up of 12 regular pentagons. In the second, we use an Archimedean solid which is slightly more complicated and allows for 13 different polyhedra. For more info on this read the following article.If you want to know a little more about platonic solids then Paul Bourke's excellent website has a brief introduction in the following article.

Open the sketch entitled *A_key_interface_01*. In this first example we introduce different renderings for a dodecahedron, each applied with key input. What is important to note is that all methods for drawing mesh faces, normals and edges, belong to the WB_Render class. To access each of these methods with different keys, we use Processing's switch() structure which is another way of coding if/else conditions.



Various render methods

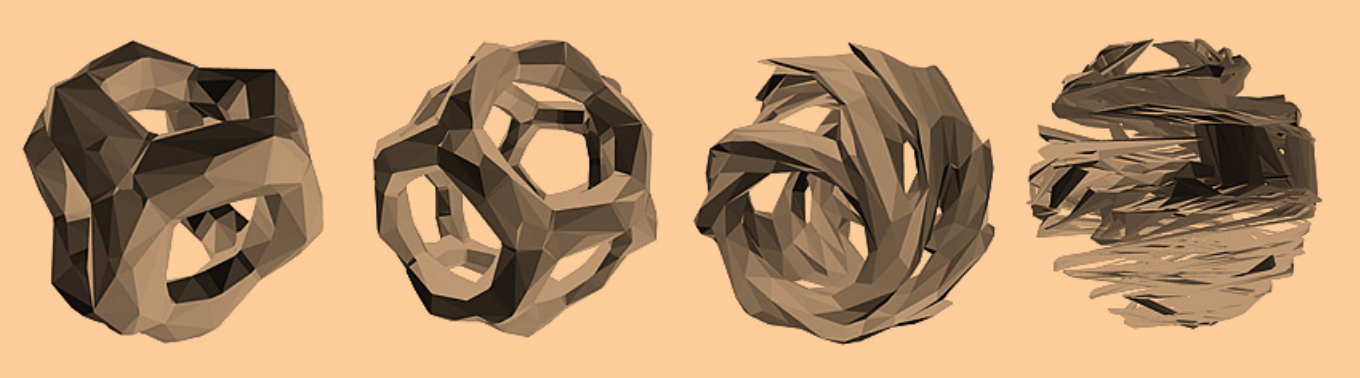
```
01  //Some examples of the various rendering methods
02  //that are contained in the WB_Render class.
03  ...
04    RENDER.drawFaces( MESH );
05  ...
06    RENDER.drawFacesSmooth( MESH );
07  ...
08    RENDER.drawEdges( MESH );
09  ...
10    RENDER.drawFaces( MESH );
11    RENDER.drawFaceNormals( 20, MESH ); // first param is the length
12  ...
13    RENDER.drawHalfedges( 20, MESH );
14  ...
15    RENDER.drawVertexNormals(30, MESH );
16
```

In our next sketch, *B_key_interface_02*, there are a number of modifications to the structure of the code that are worthy of note. Firstly, two custom functions have been added; createMesh() & createModifiers(), both of which

have been put in a new tab entitled INIT_. The reason for this is that our code is getting a little more complicated and as we add more modifiers and interactions to the sketch, it's a good idea to separate these two steps into their own functions. Remember that the logic to HE_MESH in making shapes is first to declare a creator with one of the many HEC classes. This is added to our mesh class declared as global. Next, modifiers are declared using the HEM set of classes and are applied to the mesh. Having two separate functions for these operations is basically good house keeping, making for clearer code and better accessibility to the necessary parts in the event of debugging and adding more shapes and modifiers in later developments. Notice therefore that we call these two functions both in setup() and in draw(), respecting the order of construction: > mesh & creator > modifier > render.

As for the modifiers used in this example, we've added HEM_Wireframe and the HEM_Twist class. The twist class is a neat modifier that gives interesting visual results and it's really simple to implement. In this example we have an angle variable and an axis which is constant. However, you could probably play around with this and add variables for shifting the axis to generate more complex shape modifications.

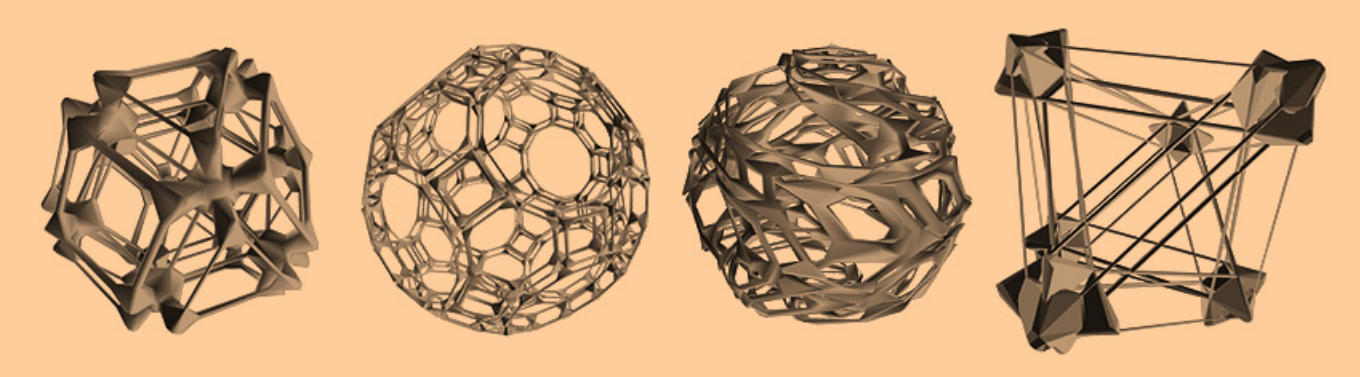
```
01  void createModifiers() {
02    HEM_Wireframe modifier = new HEM_Wireframe();
03    modifier.setStrutRadius(50);
04
05  //These two methods will have the greatest effect
06  //and should be relatively low values depending on your memory
07    modifier.setStrutFacets(6).setMaximumStrutOffset( STRUTOFFSET );
08
09  //Finally add our modifier to our mesh
10    if(STRUTOFFSET != 0) { // check we don't reach zero to avoid crashing our program
11    MESH.modify(modifier);
12   }
13
14   //Twist class takes an angle & an axis which is set with
15   //Hemesh's WB_Line class (x1,y1,z1,x2,y2,z2);
16     HEM_Twist twist = new HEM_Twist().setAngleFactor(TWISTANGLE);
17     L = new WB_Line(0, 0, 400, 0, 0, -400);
18     twist.setTwistAxis(L);
19     MESH.modify(twist);
20  }
21
```



Iterations of twisted shapes

Perhaps a little note on the HEM_Wireframe class that was seen in sketch *E_cube_modif_strut* in Part 1 of this tutorial. The HEM_Wireframe modifier converts the segments or edges of a shape into cylindrical struts with optional joint polyhedra at the vertices.It's main use is to create structural geometry based on the mesh topology, or as an alternate method to achieve a rendered wireframe effect. Two methods will have the most effect visually on your form : modifier.setStrutFacets(10).setMaximumStrutOffset(15); Keep the StrutFacets to a limit of 10 otherwise your computer will begin to struggle. The StrutOffset value will increase the amount of offset on the structure and hence increase the modifier effect.

In the last example, *C_key_interface_03*, a few more key interactions are added with extrusion & lattice modifiers to give a bit more variation. Notice that three different creators can be chosen using keys 1 - 3 and the UP/DOWN keys select the 13 Archimedean solids. However, what quickly becomes apparent with adding more and more keys is that we often forget which key does what. Equally it gets annoying having to look each time in the code to see which key modifies what. What we need is visual feedback that will make our system much easier to use. Or, to use the correct term, we need a GUI - a graphic user interface.



Iterations combining various modifiers to our shapes

ADDING A GRAPHIC USER INTERFACE

ControlP5 is an additional library dedicated to creating sliders, buttons, radial dials and text input interfaces. Apologies should be made here as I use an older version of ControlP5 for this tutorial, simply because we can create a second frame and have all the graphical user elements in one window whilst keeping the other window for our visual. I will get around to posting an updated version for this in the future. In the meantime, you can download this earlier version at our github link.

In the first example, *A_CP5_interface_01* we implement a very simple interface that modifies a cube with chamfer edges and chamfer corners. Chamfer is just a posh term for beveling. As you will notice, all interface elements have been placed in a function, controlInit() which is called in setup() at the beginning of our program.

```
01  //In globals
02  //ControlP5
03    import controlP5.*;
04
05  //Declare our various CP5 objects & variables
06    ControlP5 INTERFACES;
07    ControlWindow CW;
08    int CHAMFERDIST, CHAMFEREDGE;
09
10  ...
11
12  void controlInit() {
13    INTERFACES = new ControlP5(this);  // initialise ControlP5
14    INTERFACES.setAutoDraw(false);
15
16  //Create a new window (doesn't work in Processing 2.0+)
17  //The addControlWindow method creates a new window/frame
18    CW = INTERFACES.addControlWindow("controlP5window", 10, 10, 200, 300, 30);
19
20  //Create sliders
21    Slider chamfDist = INTERFACES.addSlider("CHAMFERDIST")
22    .setPosition(10, 20).setRange(1, 50).setValue(10);
23
24  //Some examples of additional methods
25  //for customising the slider object
26    chamfDist.getCaptionLabel().setColorBackground(color(255, 20, 30));
27    chamfDist.getCaptionLabel().getStyle().setPadding(4, 2, 2, 2);
28    chamfDist.getCaptionLabel().getStyle().setMargin(-5, 0, 0, 0);
29
30    Slider chamfEdge = INTERFACES.addSlider("CHAMFEREDGE")
31    .setPosition(10, 40).setRange(1, 20).setValue(5);
32
33  //Add elements to new window
34    chamfDist.setWindow(CW);
35    chamfEdge.setWindow(CW);
36  }
```

What should be noted when making interface objects is that we only need to add the necessary variable name for each and ControlP5 takes care of the rest. So in the above example, we have two variables, CHAMFERDIST and CHAMFEREDGE. These are added to each slider with .addSlider("CHAMFERDIST"). The methods that follow set the position of the slider, the range for the variables and the initial value at start up. To make sure we see our sliders in the separate window, make sure to add them to the controlWindow object using the setWindow method,

```
chamfDist.setWindow(CW);
```

In *B_CP5_interface_02* we develop on the above sketch adding global variables and the desired interface objects where necessary. Step by step it would be possible to implement all the key interactions we saw in the above sketches into a more user friendly and intuitive system. And that is exactly what we'll be finishing off with in part 3 of this tutorial next time along with a few 3D printed examples of course.

Tags: 3D, 3Dprinting, geometry, Hemesh, learning, Processing, tutorial