**Evan MacHale**

N00150552 / Year 4 Creative Computing

Technical Review Document / 6th December 2018

Generative Jewellery Web Application Utilising a Combination of 3D Subdivision and Data Structure Algorithms.

**Table of Contents**

[**01 Introduction**](#_9n94vlxf8upb) **2**

[**02 Background Research**](#_4sdr1yx0gs5x) **3**

[02.1 Generative Design](#_aagdfi7di13c) 3

[02.2 Subdivision Surfaces](#_fdqxgntlkicr) 4

[02.2.1 Doo-Sabin](#_4x0ppqela15) 4

[02.2.2 Catmull-Clark](#_9lsjqf61y6q1) 6

[02.2.3 Loop](#_ohft02sd89oq) 8

[02.2.4 Half-Edge](#_jfgqbdkf8kdy) 9

[02.3 Three.js](#_lbzyugcl4ob1) 10

[02.3.1 Data structure](#_ttoehhdxck9m) 10

[02.3.2 Subdivision Algorithm](#_lgvkkrbidave) 11

[02.3.2.1 Generate Data Structure](#_vk753sgkf04n) 15

[02.3.2.2 Find Edge-Points](#_4z6h2i2givsp) 18

[02.3.2.3 Find Vertex Points](#_q87sr9j28e49) 19

[02.3.2.4 Define New Surface](#_o9tnrxomkrab) 20

[**03 Conclusion**](#_o2w9qgpc02cb) **21**

[**04 References**](#_4oo5044o1g1) **22**

[**05 Bibliography**](#_p9gvicvt5ic8) **23**

# 

# 

# 01 Introduction

The purpose of this technical review is to utilise an existing algorithm to allow a designer to create computer generated jewellery from basic shapes in a three-dimensional environment. This jewellery is structured and manipulated by a set of *weights* defined by constants in the algorithm equations. The shape of the jewellery is determined by the effect of these weights on the properties of the *source shape*.

A designer need only understand the relationship between these weights and the visible effect they have on the source shape to generate complex forms. Rather than having to understand the mathematics of the underlying algorithm, the designer is exposed to the power of computer-aided design by means of a number of simple weights.

Computed-aided design augments a designer’s ability to design. However, by specifying *custom* weights and allowing the computer to do all the work, designers may explore further possible generative designs. By introduction of *parametres* in the place of these weights, the potential design outcomes for the designer magnify.

Utilising parametric principles in concert with the algorithm produces a generative design approach to jewellery design, exposing designers to innumerable computer-augmented design possibilities of which the designer is composer.

# 

# 02 Background Research

Background research for this *Research & Analysis* document will take the form of a technical review. Included in this review is an investigation into the various algorithms and data structures that have been considered for implementing the project. So too is included a study of how such technical topics are implemented through the exploration of practical code development to-date.

## 02.1 Generative Design

Research began with the concept of *generative design* in the web. Generative design is the concept that design processes that utilise the power of the computer, i.e computer-aided design platforms like SolidWorks, are dramatically changing. It is to say that the designer shifts from being a performer of tasks to being a conductor, effectively orchestrating the decision making process of the computer. Generative design is in essence: iteratively developing different processes and then selecting those that produce the most visually compelling results. (Gross, Bohnacker, Laub, Lazzeroni 2018)

This heading took research to areas such as industrial design to Autodesk with *Project Dreamcatcher* and their generative design for architecture. This exploration of architecture spaces lead research into the realm of Zaha Hadid’s (Schumacher 2008) parametric design, a form of generative art that utilises mathematics to generate complex forms. The premise of complex shape generation then landed the research before a computational architect by the name of *Michael Hansmeyer,* whom’s work and methods were found in *Hartmut Bohnacker’s* (Gross, Bohnacker, Laub, Lazzeroni 2018) *Generative Design in p5.js* book. This phenomenal book, it’s resources and work of Hansmeyer’s algorithmic, mathematical form generation (Hansmeyer 2009) brought research into the realm of *algorithm-driven design*.

Algorithmic design is the concept of how algorithms can be used to augment the human ability to design, working almost like an exoskeleton. The classic question of *will robots replace designers* is addressed. The answer is no. Instead computers will become apprentices, carrying out the author’s vision by means of abstract needs and of course in a manner that cuts significant amounts of valuable time out of the traditional design process. The author’s vision is augmented by the computer and innumerable design possibilities are explored by the computer that no mere designer may cover in entirety. (Vetrov 2018)

So, what would happen if both worlds could collide? Looking into the work of Michael Hansmeyer’s algorithmic work, one finds that his work is also 3D printed and presented as an approach to *Avant-Garde* design. This idea of a deterministic decision processes passed to an algorithm complimented with principles of generative design such like *Perlin Noise,* as a simple example, settled the project theme on the unconventional and popular industry of 3D printed jewellery. A perfect niche to exploit the power of the computer in an apt creative sub-industry driven by the machine.

## 02.2 Subdivision Surfaces

A subdivision surface is a recursive method used in generating smooth, curved surfaces on three-dimensional meshes. These methods are a three-dimensional application of *Chaikin’s curv*e generation algorithm for arbitrary polygons.

Similarly to a Bezier curve, this algorithm takes a polygon and recursively *‘clips’* it’s edges to form a new smoother polygon. In place of the original polygon, in between each of the polygon’s source points, are generated quadratic curves, namely parabolas. Fundamentally, this algorithm pieces together parabolic *segments* to generate a curve.

This algorithm is defined by a simple rule. For each segment of the polygon, two new points are found at a distance ¼ and ¾ between the endpoints. Once the new points are found along each line segment, all points are then connected together to form new line segments and thus a new refined polygon. (Joy 1999) (UC Davis Academics 2009)



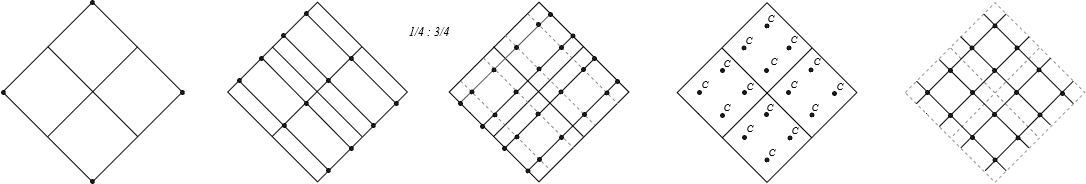
*Figure 1.1: Chaikin’s Curve.*

This process would be applied by others to an aptly named *Chaikin’s Surface*, whereby the surface represents an arbitrary polygonal mesh. This process similarly applies recursive refinements upon a given mesh’s array of source points, but with differing algorithmic weights.

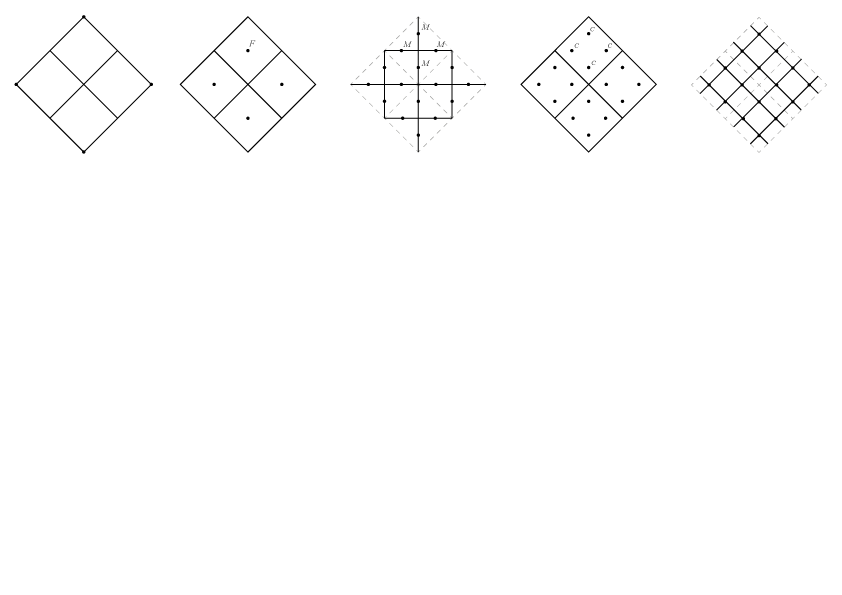
### **02.2.1 Doo-Sabin**

Presented in 1978, Doo-Sabin’s algorithm over an array of control points is an extension and generalisation of Chaikin’s algorithm that lifted mathematical restrictions placed on the topology of these data arrays to include any arbitrary topology. That is to say, a mesh constructed using a non-uniform array, or *net*, of vertex points may implement said subdivision surface algorithm. (UC Davis Academics 2009)

Elaborating on this extension, they utilised the corner clipping principles of Chaikin’s algorithm on this *control point net*, or *mesh*. To describe Doo-Sabin’s algorithm, the ¼ : ¾ ratio is applied to all edge segments of all faces, to generate new edge segments. These points are then connected up together to form a new grid of internal segments. Upon these new segments is found the new Chaikin’s points, which define the new subdivided faces of the mesh.

**

*Figure 1.2: Doo-Sabin implementing Chaikin’s algorithm.*

This is the extension of Chaikin’s algorithm, however in practice the rules defining the algorithm and the finding of the Chaikin’s points is far simpler. For each face is found a *face point,* defined as the centroid of the face. It may be observed that these new Chaikin’s points lie as the average of the face point and of the vertex, but also of the midpoints of the adjacent edges extending from the vertex as part of the given face. Once found, the new Chaikin’s points are connected up inside the original face. These control points are then connected to all adjacent face’s Chaikin’s points, defining the subdivided mesh net. (Joy 1997)**

*Figure 1.3: Doo-Sabin algorithm.*

The Doo-Sabin algorithm per face may be defined as finding firstly the *face point* of the face(this example face being a quad with four points):

Finding the midpoints:

*&*

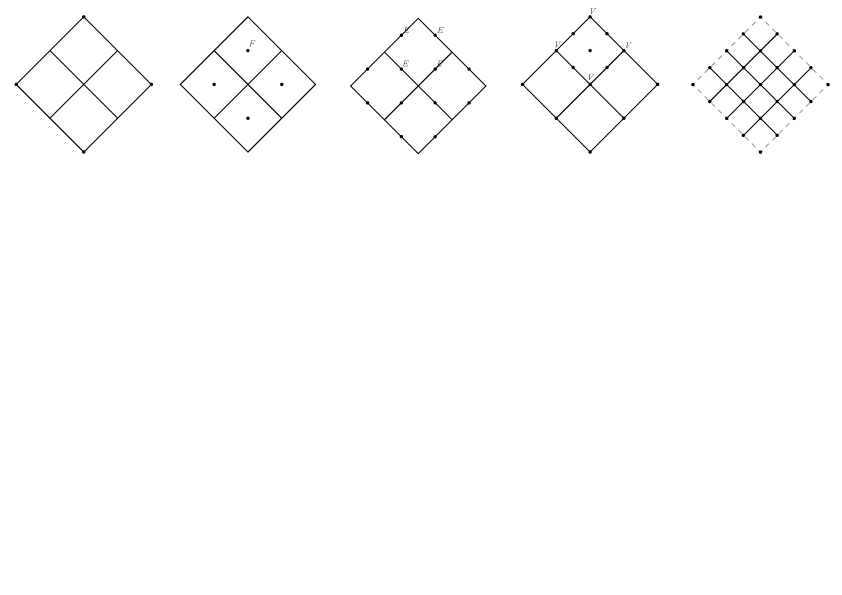
Finding the Chaikin’s point:

To then reiterate, all Chaikin’s points are found within the confines of each face and are connected internally, these Chaikin’s points for each face are then connected to all adjacent face’s Chaikin’s points to create a new subdivided array of control points.

### **02.2.2 Catmull-Clark**

After, but at roughly the same time as Doo-Sabin, in 1978 Catmull-Clark’s algorithm was presented for the same purposes as Doo-Sabin. However, rather than apply Chaikin’s specific algorithm to a surface, they made a slight variation. Their approach moved from biquadratic curve generation to bicubic. This decision was made with due to the lack of parabolic inflection points in the curve generation. In Chaikin’s original algorithm inflection points occurred only where each of the new control points connected to form a new quadratic line segment between three original line segments. In Catmull-Clark’s Curve, this cubic approach was determined as a more effective way to generate smoother curves. (UC Davis Academics 2009)

Revising Chaikin’s algorithm, Catmull-Clark created their own variation of Chaikin’s curve. Instead of the ¼ : ¾ formula, they introduced what are known as *edge points* and *vertex points.* For a given polygon, each line segment’s midpoint is defined as the edge point. Next is generated a new *vertex point* which corresponds to each point of the original polygon. These new vertex points are a repositioned polygon source point, defined as the average of it’s connecting edge’s edge points, shared face points and the original control point.

**

*Figure 1.4: Catmull-Clark algorithm.*

This method did not produce any greater degree of detail, however it’s resulting algorithm applied to a subdivision surface was more complex. As has been explained, using similar principles, edge points and face points are found for every face of an array of control points or mesh. Using these points for a face the final step is to reposition the original mesh points known as vertex points. (Loop 1987)

To find a vertex point, for each source control point of the mesh, take the average of all edge points that lie on connected edges to the source control point. Take also the average of all face points of adjoining faces that share the given source control point. Take also the control point itself and then apply the algorithm weights to find the new vertex point. Once all edge, face and vertex points are found, all are connected up, defining the subdivided mesh net. (Joy 1997)

Catmull-Clark algorithm may be defined mathematically as follows by firstly finding the face points of a given face:

Finding the edge points, taken as the average of the two end points of a line segment and the average of the two adjoining face’s face points:

Finally, the vertex point, defined as the average of all face points surrounding a control point, the average of all edge points surrounding a control point and some constant by the control point itself:

Notice this constant *C*, defined as where *n* is the number of edges extending out from the control point. This constant is generally considered as a rather curious element mathematically within the geometric modelling community. People looked at this algorithm weight and asked what if they added their own values in place of it? From this play, it was discovered that odd mathematical behaviours come as a result, which in turn created a subculture of people experimenting with this single constant. For example, if this constant is simply 1, in some places along the mesh surface appear spikes. (UC Davis Academics 2009) (Hansmeyer 2012)

The vertex point weights are thus defined as:

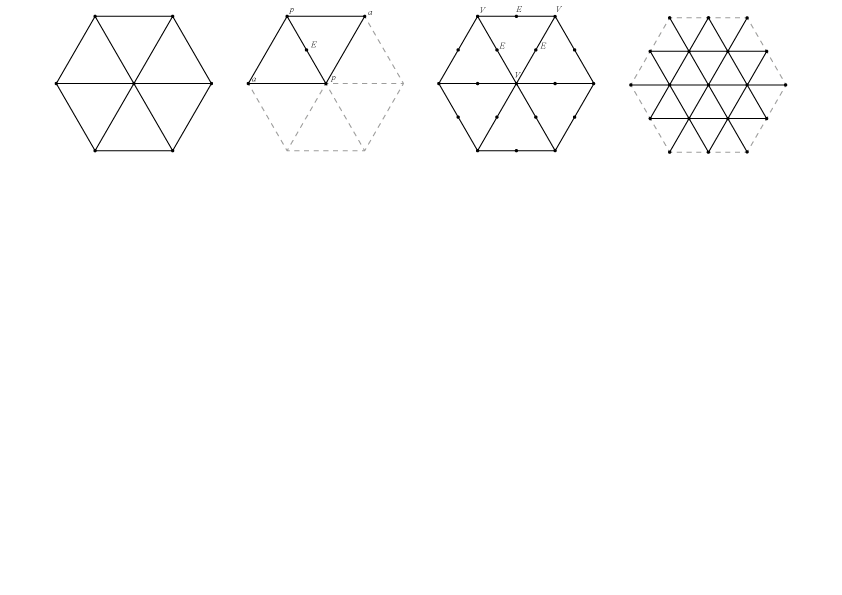
## 

### **02.2.3 Loop**

In 1987, Loop’s algorithm was presented as a generalised triangular subdivision surface, the aim of which was to further smooth and optimise subdivision methods. Loop subdivision caters strictly to triangular meshes where each face of the mesh is a triangle. At first, this appears restricting, however, when one takes into account the effectiveness and frequent use of triangles in computer graphics, Loop’s algorithm becomes efficient. Consider a sphere in computer graphics. Spheres are the smoothest primitive geometry; the only way to generate spheres is by beginning with an icosahedron, where all faces start as equivalent, equilateral triangles. (Loop 1987)

Loop’s algorithm is an abstraction of Catmull-Clark/Doo-Sabin algorithmic principles in the sense that it uses edge points and vertex points to redefine a geometry. It employs a *split* and *smooth* approach as opposed to the corner clipping procedures of Chaikin’s curve, where each triangle is split from a one to four faces and smoothed using algorithmic weight constants. Loop subdivision removes the need for face points, resulting in a less complex algorithm, however to reiterate, is not feasible on an arbitrary mesh.

Loop subdivision may then be characterised by firstly finding all edge points for each line segment defining a face. Each edge point is the average of both points on an edge segment and the average of the two adjacent triangles third points . These edge points when connected up inside a given face, define four new faces, showing this one to four methodology. New vertex points are found for all source points on the mesh, defined as the average of all other source points that lie on shared edges of that vertex and the vertex itself. Then are applied equation weights to these values. (UC Davis Academics 2009)



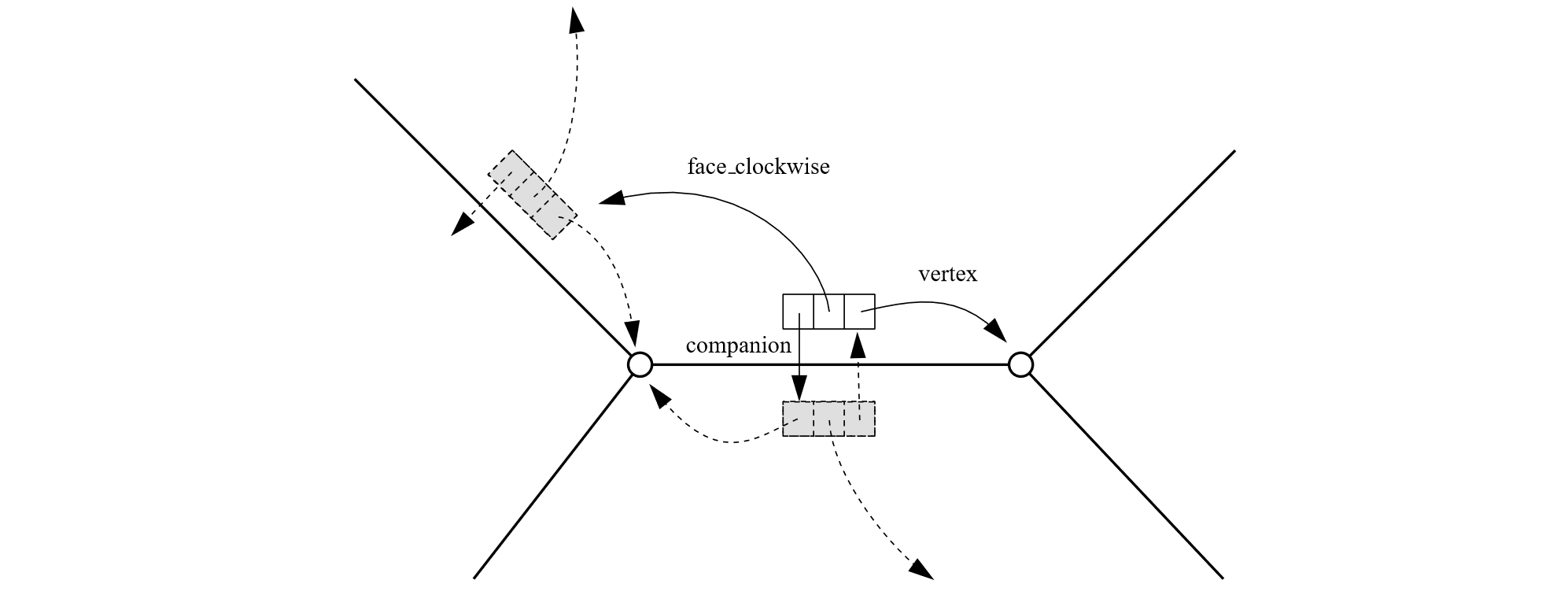
*Figure 1.5: Loop algorithm.*

Find the edge points as the average of the two edge vertices and adjacent vertices of the edge:

Then by taking a source point and the average of all it’s related edge points to find the new vertex point, applying the algorithm weights:

### **02.2.4 Half-Edge**

When working with subdivision algorithms, the majority of time is spent calculating and documenting both old and newly generated vertices. The efficiency of a subdivision implementation hinges on how effectively data is stored. The *half-edge* or *split-edge* is the most popular and effective data structure used for storing arbitrary meshes when implementing subdivision surfaces. In involves a procedure of *splitting* every edge of a mesh into two *halves*. Each half stores several pointers to, an edge vertex, the face, the next vertex clockwise around the face(the next half edge), and the opposite half edge. (UC Davis Academics 2009) (Joy, Legakis, MacCracken)

**

*Figure 1.6: Half-Edge data structure.*

This structure grants a more effective use of storage. Instead of storing all faces and the edges of each, thereby storing duplicates, we store split edges with extended references to the entire mesh structure, allowing for efficient traversal of the mesh.

## 

## 02.3 Three.js

The technology of choice for implementing the proposed algorithm is the *Three* JavaScript library. Three is the web’s most powerful cross-browser *WebGL* JavaScript library and API. It is the technology of choice for creating interactive 3D environments on the web.

Due to the emphasis of 3D space, the library is naturally built around many mathematical and geometric principles. To elaborate on the rationale, Three has an existing subdivision modifier in its current build and also has utilities for creating the data structures needed to store and efficiently manipulate meshes in a JavaScript object focused approach.

In this subchapter, the algorithms of choice will be analysed and so too the mesh data structures, in an exploration of how Three applies these mathematical geometrics in a JavaScript environment.

### **02.3.1 Data structure**

The data structure to be employed in this project is the half-edge data structure. In researching the Three library, discovered was *SubdivisionModifier.js* and *QuickHull.js,* two utility files that relate to the studied algorithms. These two files implement JavaScript variations of the half-edge data structure.

*QuickHull.js* is a utility file that implements the *quickhull* algorithm in Three. This file is used as a method of computing the convex hull of a set of points in a plane. What is useful here is that a quickhull algorithm too requires the use of a half-edge algorithm to structure its data. It holds classes that are used to create and define a new data structure for a mesh.

*See Generative-Jewellery\build\01\01.3\half-edge.js for code data structure.*

Aforementioned was that the half-edge data structure was the most effective way to support a subdivision algorithm implementation. Albeit true, the way in which such data structures are applied in any technology differ, yet of course, remain the same in principle.

Even though the half-edge data structure algorithm exists in Three it is built into the quickhull algorithm. The problem with its use lies in the fact that Three already has its own JavaScript object data structure for meshes that is pre-defined and ready to use and manipulate.

Every time the subdivision algorithm is iterated a new data structure must be constructed from the previous mesh. Therefore the argument to be made is that it would be inefficient to use and create a seperate data structure that differs from Three’s existing mesh structure, every time a subdivision is applied. To use it would result in poor optimisation and divert valuable project research time in deconstructing another 1217 lines of code.

In the *SubdivisionModifier.js* file is found a simpler method of setting up a variation of the half-edge system inside objects and arrays that correspond to the source mesh structure. In Three, a mesh object is found as a *child* property of the *scene* object:



Inside the mesh child object is found the geometry of the mesh in a similar fashion:



Inside this object are found the properties *vertices* and *faces*. These arrays are the only properties to be modified in the entire mesh. In *SubdivisionModifier.js* is created a cloned array of the vertices so to add properties that represent the relationships of each vertex, it’s shared edges and faces. This is the method of choice for this project and will be elaborated in the next section.

### **02.3.2 Subdivision Algorithm**

The subdivision algorithm chosen for this project is the *Catmull-Clark* subdivision. This algorithm has slightly more complex equations which will allow for more customisation. The algorithm also works on arbitrary meshes and not strictly triangular ones. However, the *SubdivsionModifer.js* file which is the root of this project uses Loop subdivision. It’s only difference is that there are no face points to be found, therefore, for the purposes of analysis we will investigate the Loop algorithm.

Upon the writing of this review, a Catmull-Clark algorithm had been attempted yet remains uncompleted, The structure of this code is what will be used as it does remain similar but was rewritten and altered in modern ES6 JavaScript. Furthermore, it should be noted that Three environment setup will not be discussed only code relating to the algorithms.

The *index.js* is our working Three environment where the scene is rendered with the camera, meshes are added to the scene and animation loops are implemented. *subdivisionModifier.js* is where the subdivision algorithm is implemented. However, before adding a mesh to the scene in the index, we pass it to the subdivision modifier and take into account whether a subdivision is to be implemented. This is defaulted and passed as 0 and spawns a primitive, unsubdivided mesh in the scene.

let subdivisions = 0;

When a key is pressed, jQuery is used to call the next subdivision:

// Modern jQuery key detection.

$(document).keydown(function(e) {

switch(e.key) {

case 'ArrowUp': nextSubdivision(1);

break;

case 'ArrowDown': nextSubdivision(-1)

break;

default: return; // exit this handler for other keys

}

e.preventDefault(); // prevent the default action (scroll / move caret)

});

Next is found the subdivision level, the minimum of which must be 0:

const nextSubdivision = (s) => {

subdivisions = Math.max(0, subdivisions + s);

generateSubdivision();

}

The generate subdivisions function is then called. This is where we destroy the previous subdivided mesh and subdivide anew from a source mesh. We create the new source mesh, create an instance of the subdivision class from *subdivisionModifier.js* and invoke the class methods on the geometry before adding the returned subdivided mesh to the scene:

const generateSubdivision = () => {

if ( mesh !== undefined ) {

// Removes The object from memory.

geometry.dispose();

smooth.dispose();

scene.remove(mesh);

}

// Instantiate new subdivision object.

const modifier = new SubdivisionModifier(subdivisions);

// New geometry to be added in place of the old

// N.B With new level of subdivision.

geometry = new THREE.OctahedronGeometry(1);

material = new THREE.MeshBasicMaterial({color:'red',wireframe:true});

// Scaling

const params = geometry.parameters;

if (params.scale) {geometry.scale( params.scale, params.scale, params.scale );}

// Smooth out the shape ~ adding more vertices.

smooth = modifier.modify(geometry);

const faceIndices = ['a','b','c'];

for (let i = 0; i < smooth.faces.length; i++) {

let face = smooth.faces[ i ];

for (let j = 0; j < 3; j ++) {

let vertexIndex = face[faceIndices[j]];

let vertex = smooth.vertices[vertexIndex];

}

}

mesh = new THREE.Mesh(smooth,material);

mesh.scale.setScalar(params.meshScale ? params.meshScale : 1);

scene.add(mesh);

}

The subdivision is implemented through the *SubdivisionModifier* class, consider the instantiation and method calling:

const modifier = new SubdivisionModifier(subdivisions);

smooth = modifier.modify(geometry);

*modifier* is created and initialised as an object of the *SubdivisionModifier* class, passing the number of subdivisions as a parametre:

class SubdivisionModifier {

constructor(subdivisions) {

this.subdivisions = (subdivisions === undefined) ? 1 : subdivisions;

}

}

The variable smooth is then used to pass the geometry to the modify method and return the subdivided geometry:

// Inheritance w/ prototype chain

// Class-ical inheritance.

SubdivisionModifier.prototype.modify = function(geometry) {

Geometry = geometry.clone();

geometry.mergeVertices(); // Remove duplicates

const iterations = this.subdivisions;

for (let i = 0; i < iterations; i++) {

this.subdivide(geometry);

}

geometry.computeFaceNormals(); // Orientation toward light source

geometry.computeVertexNormals();

return geometry;

}

Now consider *this.subdivide(geometry).* We use the *this* object to refer to the owner of the method we are executing, being *SubdivisionModifier.* We finally pass the geometry to another method that implements our algorithm on the geometry.

The subdivision algorithm is implemented in the *subdivision* method in four stages:

1. Generate data structure
2. Find edge-points
3. Find vertex points
4. Define new subdivided surface

#### **02.3.2.1 Generate Data Structure**

To generate the edge oriented data structure we must first define four variables:

const oldVertices = geometry.vertices;

const oldFaces = geometry.faces;

const metaVertices = new Array(oldVertices.length);

const sourceEdges = {};

The variable metaVertices will correspond to the source mesh vertices and will hold references to all edges that come out of a point, and by extension other end vertices and shared faces. The sourceEdges object will contain a list of edge properties and their edges points. We then pass these four variables to a function to process the geometry and populate our data structure:

const generateLookups = (oldVertices, oldFaces, metaVertices, sourceEdges) => {

for (let i = 0; i < oldVertices.length; i++) {

metaVertices[i] = {edges:[]};

}

for (let i = 0; i < oldFaces.length; i++) {

processEdge(oldFaces[i].a, oldFaces[i].b, oldVertices, sourceEdges, oldFaces[i], metaVertices);

processEdge(oldFaces[i].b, oldFaces[i].c, oldVertices, sourceEdges, oldFaces[i], metaVertices);

processEdge(oldFaces[i].c, oldFaces[i].a, oldVertices, sourceEdges, oldFaces[i], metaVertices);

}

}

We first populate *metaVertices* with arrays to keep track of every edge extending from the vertex forming the basis of our edge-based structure. Then, Using each face of the mesh, we process every edge on the mesh, pushing the relationships of each edge to the relevant vertex in *metaVertices*, this will also create duplicates of each edge, representing the split-edge data structure principles, albeit not perfectly. This is used later on when applying the algorithm weights:

const processEdge = (v1, v2, oldVertices, sourceEdges, currentFace, metaVertices) => {

const vertexIndexA = Math.min(v1, v2);

const vertexIndexB = Math.max(v1, v2);

const key = vertexIndexA + "\_" + vertexIndexB;

let edge;

if (key in sourceEdges) {

edge = sourceEdges[key];

} else {

const vertexA = oldVertices[vertexIndexA];

const vertexB = oldVertices[vertexIndexB];

edge = {

a: vertexA, // pointer reference

b: vertexB,

newEdgePoint: null,

faces: [] // pointers to faces

};

// Give the edge the info

sourceEdges[key] = edge;

}

edge.faces.push(currentFace);

// For both points on the edge, give them the edge info.

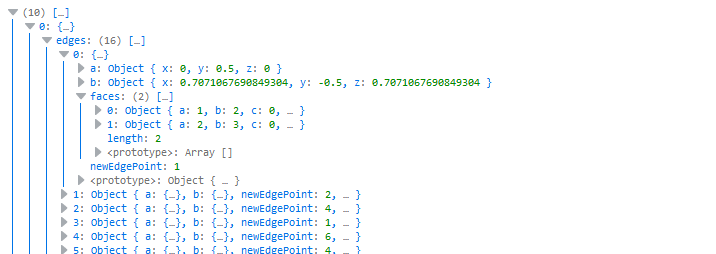
metaVertices[v1].edges.push(edge);

metaVertices[v2].edges.push(edge);

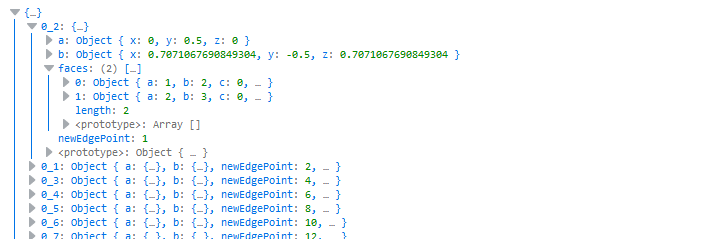
}

We also add each created edge to the *sourceEdges* object to keep track of the total amount of edges for the entire mesh. We check if an edge exists using sourceEdges, two vertices share the same edge so we push to both vertices in *metaVertices.* The *edge* object implements our half-edge formula with pointers to the vertex, it’s faces, the next vertex around the edge and a custom *edge-point.*

From this process we can log and observe the data structure created in *metaVertices*:



Also we can query the list of edges:



We can now use *metaVertices* as a pseudo half-edge query system that employs properties that point to the next vertex around an edge. We may also query the faces an edge shares and by extension the vertices of those shared faces. Then as an added custom variable, the edge-point of that edge, a valuable part needed in the subdivision algorithm.

#### 

#### **02.3.2.2 Find Edge-Points**

After the data structure is generated, we apply the first step of the Loop algorithm. We must find the edge-points of every edge and record them to be merged into the final geometry later. We will see our algorithm equation weights in action here. For Loop subdivision we find the average of the vertices and apply the weights:

const newEdgeVertices = [];

const adjacentVertexWeight = 1 / 8;

const edgeVertexWeight = 3 / 8;

We use the sourceEdges object to query each edge, then populate the newEdgeVertices array with the new edge-points for concatenation later into a new array of control points. For each edge we find the average of the two connected vertices, then we apply the weight:

const faceVertices = ['a','b','c'];

for (i in sourceEdges) { // For each edge create new edge point.

currentEdge = sourceEdges[i];

newEdgePoint = new THREE.Vector3();

connectedFaces = currentEdge.faces.length;

newEdgePoint.addVectors(currentEdge.a,currentEdge.b).multiplyScalar(edgeVertexWeight)

...

Next we find the average of the two adjacent points defined as the third point on each shared face, then we apply the weight:

vertexHolder.set(0, 0, 0);

// For both faces, for every point on given face, find the third point.

for (let j = 0; j < connectedFaces; j ++) {

face = currentEdge.faces[j];

for (let k = 0; k < 3; k ++) {

oppositePoint = oldVertices[face[faceVertices[k]]]; // a,b,c

if (oppositePoint !== currentEdge.a && oppositePoint !== currentEdge.b) break;

}

vertexHolder.add(oppositePoint);

}

vertexHolder.multiplyScalar(adjacentVertexWeight);

Lastly we merge the averages together to find the true vertex of the edge-point. We then push the new edge-point to the newEdgeVertices array:

...

// Combine values of vertex and opposite points to find new point.

newEdgePoint.add(vertexHolder);

// Keeps count of creation order.

currentEdge.newEdgePoint = newEdgeVertices.length;

newEdgeVertices.push(newEdgePoint); // List of edge points.

} // End for loop.

newEdgeVertices array now holds a list of the vertices that need to be connected up to the vertex points later to define the new subdivided mesh. Both metaVertices and sourceEdges also have values pushed to their pointers to correspond to the indexes of newEdgeVertices.

#### **02.3.2.3 Find Vertex Points**

Finding the vertex points involves no new point generation but simply takes the source control point mesh and applies the algorithmic weights. This step is largely formula work. We loop through every source vertex and apply the weights:

for (let i = 0; i < oldVertices.length; i++) {

oldVertex = oldVertices[i];

connectingEdges = metaVertices[i].edges;

const numberOfConnectingEdges = connectingEdges.length;

...

We apply the vertex weights defined in Loop’s algorithm to the given vertex:

beta = 3 / (8 \* numberOfConnectingEdges);

connectingVertexWeight = beta;

sourceVertexWeight = 1 - numberOfConnectingEdges \* beta;

newSourceVertex = oldVertex.clone().multiplyScalar(sourceVertexWeight);

...

Then we apply the same weight to the surrounding vertices by finding them along the edges connecting to the given vertex:

vertexHolder.set(0, 0, 0);

for (let j = 0; j < numberOfConnectingEdges; j++) {

connectingEdge = connectingEdges[j];

connectingPoint = connectingEdge.a !== oldVertex ? connectingEdge.a : connectingEdge.b;

vertexHolder.add(connectingPoint);

}

// Apply weight to connecting vertices.

vertexHolder.multiplyScalar(connectingVertexWeight); // β

newSourceVertex.add(vertexHolder);

newSourceVertices.push(newSourceVertex);

} // End for loop.

We also push these vertices to the newSourceVertices array to be concatenated to the newEdgeVertices next.

#### **02.3.2.4 Define New Surface**

Lastly in the algorithm we must now connect the *newEdgeVertices* and the *newSourceVertices* to form the new faces of the geometry. We concatenate the vertices and run through each face, subdividing it into four smaller triangles:

newVertices = newSourceVertices.concat(newEdgeVertices);

for (let i = 0; i < oldFaces.length; i++) {

face = oldFaces[i];

edge1 = getEdge(face.a, face.b, sourceEdges).newEdgePoint + sl;

edge2 = getEdge(face.b, face.c, sourceEdges).newEdgePoint + sl;

edge3 = getEdge(face.c, face.a, sourceEdges).newEdgePoint + sl;

newFace(newFaces, edge1, edge2, edge3);

newFace(newFaces, face.a, edge1, edge3);

newFace(newFaces, face.b, edge2, edge1);

newFace(newFaces, face.c, edge3, edge2);

} // End for loop

We then take the properties of the source geometry we originally passed in and replace them with the newly found subdivided control points:

geometry.vertices = newVertices;

geometry.faces = newFaces;

To the geometry is then returned to *index.js* to be added to the scene.

# 03 Conclusion

In review, discussed were the concepts of generative/algorithmic design and how algorithms may augment the designers processes, whilst being complimented by generative design may yield further creative results only a computer may have the ability to produce. So then were our algorithms of choice investigated, and the differences between them. Furthermore, was discussed data structures in tandem with the algorithm and how they would affect the design process. To then elaborate on the theory, we explored the research completed to date; how such theorems translate to modern JavaScript.

To conclude, of the algorithms explored, Catmull-Clark and Loop subdivisions are the two algorithms most beneficial to the project desirables. To implement these algorithms we must use split-edge data structures, however, not strictly. It is moreso beneficial to incorporate such a data structure into existing JavaScript object structures. Furthermore, code snippets from both Catmull-Clark and Loop exercises were elaborated on to show clarity of process.

It is important to again be reminded that the algorithmic weights are the key to the project investigation process. These weights will incorporate generative design principles in order to generate the complex forms, whilst the algorithm itself generates shape and explores the forms affect on shape. This is the integral concept to the project design problem.

To fully view the explored algorithm snippets in action, visit the GitHub project repository at:

[*https://github.com/larryzodiac/Generative-Jewellery*](https://github.com/larryzodiac/Generative-Jewellery)

Navigate to *build* to view source code and see the relevant readme for demo links. One may also view Kanban project progression by navigating to the *Projects* tab in the repository menu bar found below the page navigation bar.

# 

# 04 References

Gross, B. , Bohnacker, H. , Laub, J. , Lazzeroni, C. (2018). Generative Design, Revised and Updated Edition. Princeton Architectural Press; Reprint edition.

Hansmeyer, M. Computational Architecture. (n.d). Retrieved from: http://www.michael-hansmeyer.com/

Hansmeyer, M. (2012). Building unimaginable shapes. TED. Retrieved from: https://www.youtube.com/watch?v=dsMCVMVTdn0

Joy, K. (1999) Chaikin’s Algorithm For Curves. Visualisation & Graphics Group, University of California, Davis. Retrieved from: http://graphics.cs.ucdavis.edu/education/CAGDNotes/Chaikins-Algorithm.pdf

Joy K. (1997) Doo-Sabin Surfaces. Institute for Data Analysis and Visualization, University of California, Davis. Retrieved from: http://graphics.cs.ucdavis.edu/~joy/GeometricModelingLectures/Unit-9-Notes/Doo-Sabin.pdf

Joy K. (1997) Catmull-Clark Surfaces. Institute for Data Analysis and Visualization, University of California, Davis. Retrieved from: http://graphics.cs.ucdavis.edu/~joy/GeometricModelingLectures/Unit-9-Notes/Catmull-Clark.pdf

Loop, C. (1987). Smooth Subdivision Surfaces Based on Triangles. 21-33. Retrieved from: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/thesis-10.pdf

Joy, K. , Legakis, J. , MacCracken, J. (n.d). Data Structures for Multiresolution Representation of Unstructured Meshes. Retrieved from: http://graphics.cs.ucdavis.edu/~joy/GeometricModelingLectures/Unit-9/resources/Tahoe-Paper.pdf

Schumacher, P. (2008). Parametricism - A New Global Style for Architecture and Urban Design. Retrieved from: http://www.patrikschumacher.com/Texts/Parametricism%20-%20A%20New%20Global%20Style%20for%20Architecture%20and%20Urban%20Design.html

UC Davis Academics. (2009). Computer Graphics. Retrieved from: https://www.youtube.com/playlist?list=PL\_w\_qWAQZtAZhtzPI5pkAtcUVgmzdAP8g

Vetrov, Y. Algorithm-Driven Design. (n.d). Retrieved from: https://algorithms.design/

# 05 Bibliography

Vetrov, Y. (2017). Algorithm-Driven Design: How Artificial Intelligence Is Changing Design. Retrieved from: https://www.smashingmagazine.com/2017/01/algorithm-driven-design-how-artificial-intelligence-changing-design/

Autodesk. (n.d). Project Dreamcatcher. Retrieved from: https://autodeskresearch.com/projects/dreamcatcher

Autodesk. (2016). Generative Design for Architecture: Autodesk MaRS Office. Retrieved from: https://vimeo.com/193915345

Autodesk. (2015). Project Dreamcatcher with Erin Bradner and Michael Bergin. Retrieved from: https://www.youtube.com/watch?v=jvN5AVZW8r8&feature=youtu.be

Holmes3D.net. (2014). A Quick Introduction to Subdivision Surfaces. Retrieved from: http://www.holmes3d.net/graphics/subdivision/

Mode Lab. (2017). Parametric Design Fundamentals. Retrieved from: https://www.youtube.com/playlist?list=PLGV167zE8gnUF2JbcgpfRDQ-FYaYXH5se

McGuire, M. (2000). The Half-Edge Data Structure. Retrieved from: http://www.flipcode.com/archives/The\_Half-Edge\_Data\_Structure.shtml

Matt's Webcorner. (2014). Subdivision. Retrieved from: http://graphics.stanford.edu/~mdfisher/subdivision.html