Doo-Sabin Subdivision and its Reverse

CPSC 589

Final Project Report

Luke Olsen

Instructor: F.F. Samavati

April 19, 2004

Abstract

I implemented a robust, object-oriented application for performing Doo-Sabin subdivision and reverse Doo-Sabin subdivision on polygonal models having arbitrary topologies. Doo-Sabin subdivision is a technique for increasing the resolution of an arbitrary surface; the limit surface has C^1 continuity. Reverse Doo-Sabin subdivision reduces the resolution of a model, but by capturing additional information during the reversing process we are able to completely reconstruct the original model after reverse subdivision.

Subdivision schemes have wide applicability in both real-time and offline rendering paradigms. For instance, Pixar uses subdivision to ensure that the resolution of the models used in its feature films is greater than the resolution of the final rendering, to reduce shading artifacts. Interactive games could exploit subdivision surfaces to render low resolution models when the camera is far from the model, and progressively increase the resolution as the camera moves closer, much like mip-mapping of texture maps, to yield a progressive increase in detail.

1 Overview

1.1 Forward Doo-Sabin Subdivision

Doo-Sabin subdivision is a method for increasing the resolution of an arbitrary polygonal model. In two dimensions, Doo-Sabin is analogous to Chaikin (quadratic b-spline) subdivision.

Doo-Sabin subdivision places no restriction on the types of faces (triangles, quadrilaterals, etc.) in the model, but produces the best results with quadrilaterals. Doo-Sabin subdivision also has the property that most of the new faces created during subdivision are quadrilaterals.

Consider an arbitrary model *M*, consisting of faces, vertices, and edges. Subdividing *M* using the Doo-Sabin method produces a higher-resolution model *M'* with three types of faces:

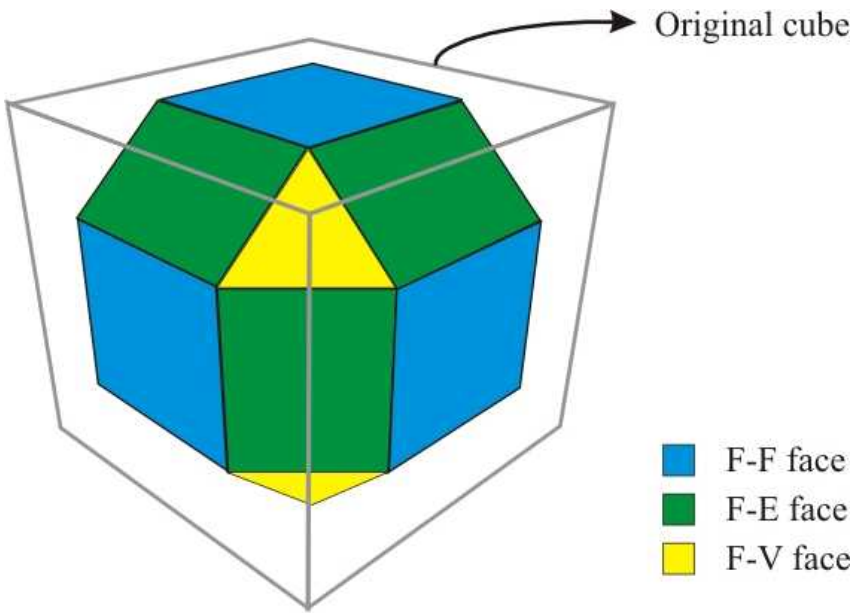


Figure 1: The three face types created by Doo-Sabin subdivision.

i. F-F faces:

Each face is contracted about its centroid to create a new face in *M*'. Contraction is based on the mask values derived by Doo and Sabin.

Let k = number of vertices in face $x \in M$

v_i = vertex i of face $x \in M$, for $i = 0, 1, \ldots, k\text{-}1$

v_i' = vertex i of face $x \in M$'

Then, the subdivision mask values are given by:

$$\alpha_0 = \frac{1}{4} + \frac{5}{4k}$$

$$\alpha_j = \frac{3 + 2\cos(2j\pi / k)}{4k}, \qquad j = 1, \ldots, k-1$$

and the vertex v_i' in *M*' is related to vertex v_i in *M* by:

$$v_i' = \alpha_0 v_i + \alpha_i v_{i+1} + \ldots + \alpha_{k-1} v_{(i+k-1)\%k}$$

ii. F-E faces:

For all non-boundary edges in *M*, there is a face on each side of the edge. Before

subdivision, these faces share an edge; after subdivision, the faces have been contracted and thus they no longer share an edge. An F-E face bridges the gap created between faces when faces are contracted.

To create an F-E face, no subdivision mask values are required. We only need to know which faces shared the edge in *M*, and then connect the new vertices of the faces together.

iii. F-V faces:

A vertex $v \in M$ of degree d has d neighboring vertices. Each vertex-neighbor pair defines an edge. As shown on page 2, each edge produces an F-E face in *M'*. Thus, v will be surrounded by d F-E faces after subdivision. Edges of these d F-E faces will form a connected component; this component defines an F-V face. Thus, each vertex in *M*, of degree d, produces a face in *M'* with d sides.

The various types of faces created by Doo-Sabin subdivision is best illustrated with an example. Consider a unit cube. A cube consists of: eight vertices, all of which are of degree 3; six quadrilateral faces; and, twelve edges. The discussion above tells us that one level of Doo-Sabin subdivision should produce a total of 26 faces: six F-F faces, eight F-V faces, and twelve F-E faces. Figure 1 shows a cube after one level of subdivision, and though it is difficult to verify that there are 26 faces from a single isometric point of view, this is indeed the case.

1.1.1 Forward Boundary Mask

As mentioned previously, Doo-Sabin subdivision is a three-dimensional extension of the Chaikin corner-cutting technique for two-dimensional curves. It should come as no surprise, then, to find that the Doo-Sabin boundary mask is exactly Chaikin subdivision. In Chaikin subdivision, each line segment along a polyline, defined by endpoints v_i and v_{i+1}, is contracted about the centroid c (analogous to contraction of faces in Doo-Sabin) to create new vertices w_j and w_{j+1}. Then, the gaps between the contracted line segments are bridged by additional line segments (analogous to F-E faces in Doo-Sabin). The contraction of the line segments is defined as:

$$w_j = \frac{1}{2} v_i + \frac{1}{2} c$$

$$w_{j+1} = \frac{1}{2} v_{i+1} + \frac{1}{2} c \,.$$

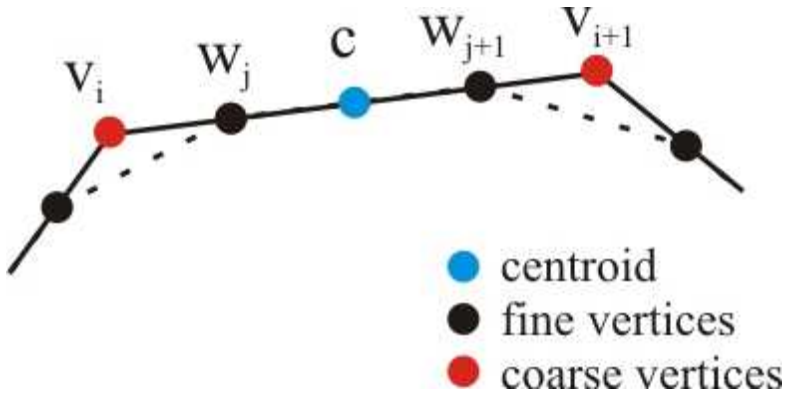


Figure 2: Boundary case for forward subdivision.

See Figure 2 for an illustration of boundary subdivision in the forward direction.

1.2 Reverse Doo-Sabin Subdivision

When a coarse model is subdivided to produce a higher-resolution fine model, the storage requirements and rendering complexity of the model increases accordingly. In many cases, the extra detail afforded by the fine model would be wasted; consider a model in an interactive game that occupies only 8 pixels on the screen.

As such, we would like a way to both increase and *decrease* the resolution of a model. Forward subdivision takes care of the former; reverse subdivision introduced here will accomplish the latter.

Forward Doo-Sabin subdivision is accomplished by contracting faces about their centroid, and then bridging the gaps created with additional faces (F-E and F-V faces). Reverse subdivision is the logical opposite of this process: F-F faces in *M*' are expanded about their centroid to create the original faces of *M*, while F-E and F-V faces are discarded.

Reverse subdivision is complicated by the fact that forward subdivision is a *one-to-many* operation in terms of vertices. Consider a vertex $v \in M$ that is contained by faces *i*, *j*, and *k*: each of these faces will create their own 'copy' of *v* when it is contracted; thus, vertex *v* effectively produces three vertices in *M*'. In general, a vertex of degree *d* will produce *d* new vertices in *M*'.

Because reverse subdivision is the inverse of forward subdivision, reverse subdivision is a *many-to-one* operation in terms of vertices. The reverse algorithm must therefore take into account the many vertices that are trying to merge into one. This merge process is done by local averaging [3].

Using the same notation as in Section 1.1, we can define the subdivision mask values. Let v_i' be the vertex of some k-sided face in model M^i, v_i be the same vertex in M^{i-1} (i.e. after reverse subdivision), and c be the centroid of the face. Then,

$$v_i = \frac{1}{\alpha} v_i' - \frac{1-\alpha}{\alpha} c, \qquad i = 0,1,\ldots,k-1,$$

where α is determined by the forward mask values (here, $\alpha = {}^1/_2$).

This process will determine the expanded vertices for each face. As mentioned previously, several of these per-face vertices should be merged into a single vertex. Section 2.5 will discuss how the *candidate* vertices for merging are determined. For now it suffices to note that the process must be carried out.

The merging or local averaging of candidate vertices essentially discards information: rather than having d distinct vertices, we discard a factor of ${}^{d-1}/_d$ of the information contained in the vertices and retain only the average. However, this additional information need not be discarded. By storing information about the relationship between each candidate vertex and the average, we can completely reconstruct the fine model that was reverse-subdivided. This information is captured in a *details* structure, described in Section 2.2.3 and Section 2.5.

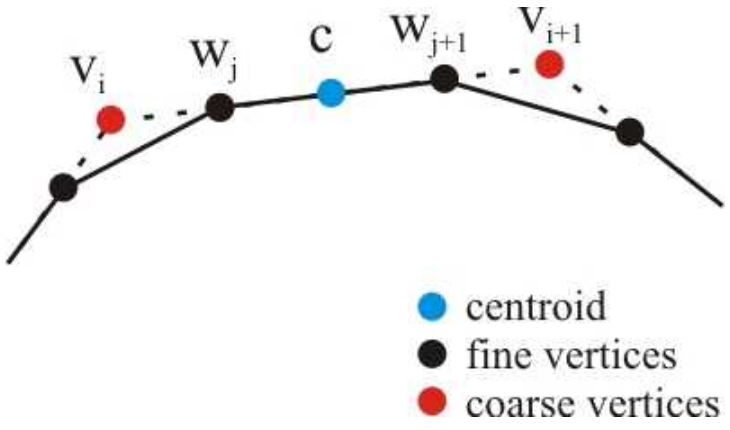


Figure 3: Boundary case for reverse subdivision.

1.2.1 Reverse Boundary Mask

Chaikin subdivision is used on the boundaries for forward Doo-Sabin subdivision. Chaikin subdivision contracts each segment of a polyline about its centroid, and then bridges the gaps between the polylines with additional line segments.

Reverse subdivision on the boundary, then, is the expansion of a line segment about its centroid. Consider Figure 3: v_i and v_{i+1} are the vertices of the coarse edge, while w_j and w_{j+1} are the vertices in the contracted edge; c is the centroid of both line segments. By definition of Chaikin subdivision, it is the case that $w_j = \frac{1}{2}v_i + \frac{1}{2}c$ and $w_{j+1} = \frac{1}{2}v_{i+1} + \frac{1}{2}c$. Thus, to reverse-subdivide a boundary (knowing w_j and w_{j+1}), the mask follows directly from algebra: $v_i = 2w_j - c$, and $v_{i+1} = 2w_{j+1} - c$.

2 Implementation Details

In this section, we will describe the work that was done over the term to implement an application program capable of demonstrating forward and reverse Doo-Sabin subdivision. In the discussion, the `Courier` font is used to denote code fragments or variable and class names used in the actual implementation.

2.1 System Architecture

The goal of the project is to implement a robust application program for performing forward and reverse Doo-Sabin subdivision. To accomplish this task, I wrote an application using C++, Qt, and OpenGL.

The application can be broken into three main components: the user interface (GUI); the renderer; and, the polygonal models being subdivided. The class diagram (Figure 4) shows the relationship between the classes in the system.

- `Window` class:
  This class is responsible for opening an application window and accepting/processing user input. It uses Qt's myriad classes to handle all functionality.
  The `Window` class owns an instance of the `Renderer` class. Qt's built-in message-passing system facilitates communication between the user interface and the renderer. For

example, to load a model the user selects ‘Open a model’ from the application window’s menu. The application opens a file dialog that allows the user to select the desired file. The application then passes on the name of the selected file to the renderer, which loads the model and draws it to the screen.

- `Renderer` class:

  This class is responsible for drawing a polygonal model to the screen, and for processing user input that occurs within it. For example, mouse events are used to rotate and zoom in on the model; the `Renderer` class receives the mouse events, decides what action to take, and updates the displayed image.

  The renderer also receives user input that is passed on from the `Window` class. For instance, when the user requests a subdivision by clicking on the ‘Subdivide’ button, the window passes on this request to the renderer, and the renderer in turn passes this on to the model itself.

  The `Renderer` class owns an instance of the `DooSabinModel` class, but knows very little about it. The renderer simply passes on requests to forward or reverse subdivide the model to the `DooSabinModel` class, and reports the success or failure back to the `Window` class so that the user interface can be updated accordingly. The renderer is also responsible for updating the on-screen version of the model according to the subdivision level (after subdivision, the renderer requests a new display list from the model) and user-requested render mode (when the display mode changes, the renderer gets a new display list from the model).

- `DooSabinModel` class:

  This class is the core of the system. The `Renderer` class owns one instance of this class to represent the loaded model.

  The `DooSabinModel` class makes several functions publicly available to facilitate user interaction with it. For changing the resolution, there are methods `subdivide()` and `reverse()`. For manipulating vertices of the model, there are the methods `closest_vertex()`, `get_vertex_pos()`, and `update_vertex_pos()`.

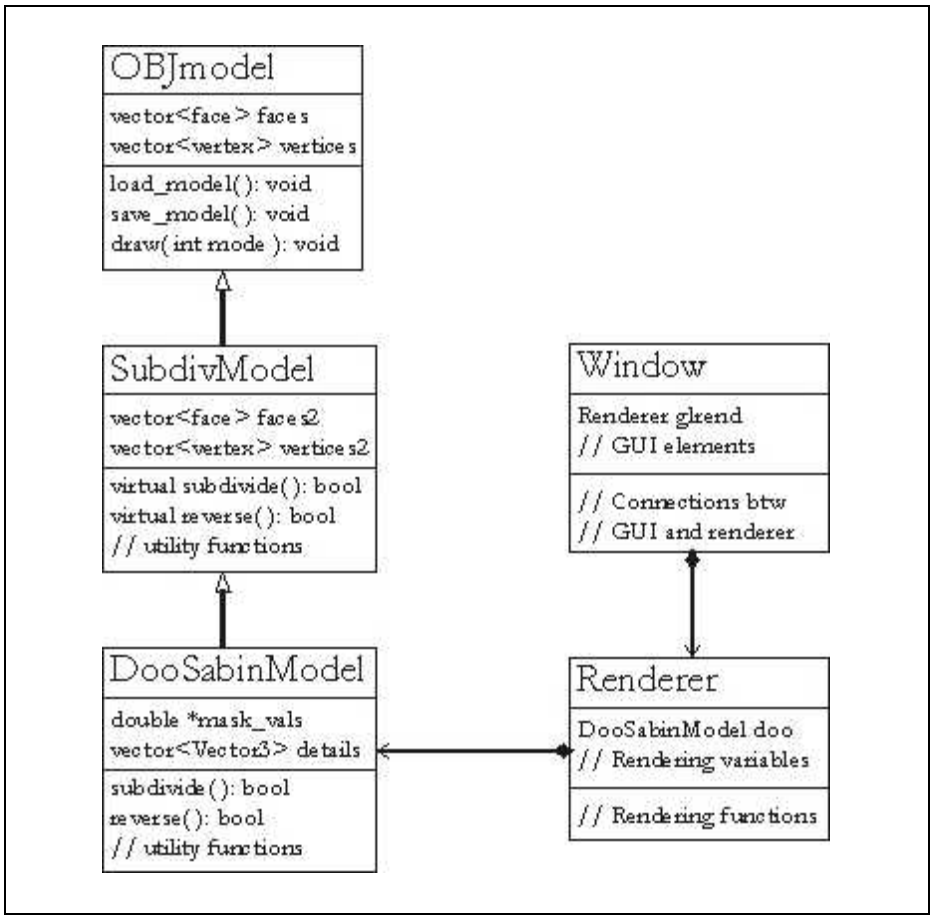


Figure 4: Class diagram.

2.2 Data Structures

The `Window` and `Renderer` class are fairly straightforward; the data structures contained within them do not need elaboration beyond the inline documentation. The other three main classes (see Figure 4) are more complicated, and are discussed in detail below.

2.2.1 OBJmodel Data Structures

The data structures used in the `OBJmodel` class are largely dictated by the file format that the class is designed to read, export, and render. A minimal OBJ-format file contains only faces and vertices. Vertices are specified by a three-dimensional coordinate. Faces are defined by a series of vertex indices; for example, a quadrilateral face is defined by the four vertices at the corners of the faces. A complete description of the OBJ format is beyond the scope of this report; it will suffice to say that the `OBJmodel` class is able to load and render most files.

The two main data structures of an OBJmodel are summarized in Figure 5.

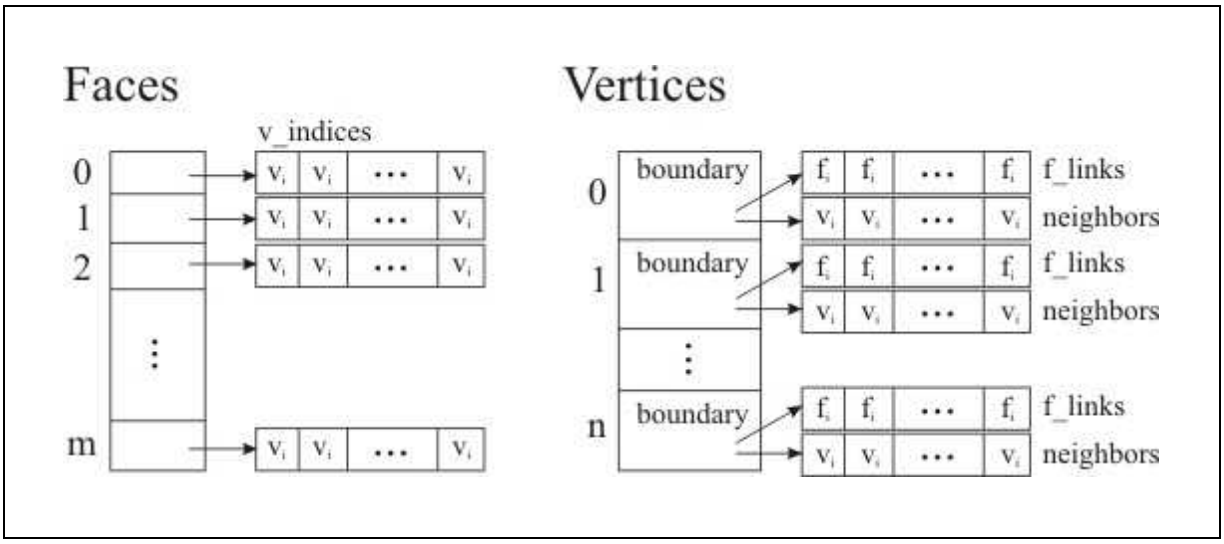


Figure 5: Main data structures of the OBJmodel class.

The `faces` structure is an STL vector of `OBJface` structs. An `OBJface` contains a list of vertex indices for the face (`v_indices`), along with some rendering information (normals, material, texture coordinates). A *k*-sided face requires *k* vertex indices, of course. The order that vertices are listed in the `v_indices` vector is important for non-triangular faces; each adjacent pair of vertices in the `v_indices` vector defines an edge.

The `vertices` structure is an STL vector of `OBJvertex` structs. An `OBJvertex` structure contains the coordinates of the vertex, a flag (Boolean) indicating whether the vertex is on the boundary, a list of faces that contain the vertex (`f_links`), and a list of vertices that are connected by an edge to the vertex (`neighbors`).

There are several other data structures in `OBJmodel` that are not relevant to performing subdivision. These structures are the `face_normals`, `vertex_normals`, `texture_coords`, `materials`, and `groups`. These structures exist for rendering purposes, and are peripheral to a discussion of the subdivision process.

2.2.2 SubdivModel Data Structures

`SubdivModel` inherits from `OBJmodel` (see Figure 4). As such, it already has a set of faces and vertices, the ability to load and save files, and the ability to draw a model to the screen.

With the 'middle' `SubdivModel` (middle in the sense that it inherits from `OBJmodel` and `DooSabinModel` inherits from it), the goal is to encapsulate functionality that is extraneous to a basic polygonal model, but that is not specific to Doo-Sabin subdivision.

To this end, SubdivModel creates a second set of faces, `faces2`, and a second set of vertices, `vertices2`. The majority of subdivision methods are not *in-place* operations, in the sense that faces and vertices can't be modified in place without affecting later stages of the subdivision. For this reason, the majority of subdivision methods will need a second set of faces and vertices; when a new face or vertex is created during subdivision, it can be added to the *inactive* set of faces or vertices.

To avoid modifying every function to check which set of faces and vertices is active, `SubdivModel` also creates a set of pointers: `curr_faces` and `curr_vertices` point to the active set and are used when drawing the model to screen; `next_faces` and `next_vertices` point to the inactive set, and are used during forward and reverse subdivision to store new faces and vertices. Thus, at the end of `subdivide()` or `reverse()`, the pointers must be swapped.

2.2.3 DooSabinModel Data Structures

The major data structures required to support subdivision are inherited by `DooSabinModel` from `SubdivModel` and `OBJmodel`. These include the dual sets of faces and vertices, and the normals, materials, etc. related to rendering.

As described in Section 1.1, Doo-Sabin subdivision works by contracting faces according to the subdivision mask values. Because of the costly cosine term in the mask value formula, we do not want to calculate the mask values repeatedly. In the constructor, `DooSabinModel` pre-computes a certain number of mask values. At the minimum, we wish to cache the mask values for the most common face types: triangles and quadrilaterals. If we ever encounter a face for which the mask values haven't yet been computed, then we compute the mask values for that size of face and cache the results. The data structure used to cache the mask values is an array of double-precision floating point pointers, called `mask_vals`. The first element of the array points to an array of mask values for triangles, the second element of `mask_vals` points to an array of mask values for quadrilaterals, and so on. Thus the shape of `mask_vals`

is like a two-dimensional array, except that each row has one more column than the preceding row.

The data structure used to capture the details created during reverse subdivision (thus enabling complete reconstruction of a fine model from a reverse-subdivided coarse representation) is a bit ugly. Recall from Section 1.2 that the goal is to retain information that is filtered out by the local averaging of candidate vertices in reverse subdivision. In general, there will be d candidates for a given vertex of degree d; call these candidate vertices $c_1, \ldots, c_d$, and let $\bar{c}$ denote the average of the candidate vertices. By storing the difference between each candidate c_i and $\bar{c}$, we can return the original candidate vertices.

Every time a model is reverse-subdivided, a new set of details are necessary. Thus, the overall structure of `details` is an array (actually, an STL vector), and each time a model is reverse-subdivided a new set of details is appended. If the reversed model has m faces, then the details for that level will be an array or vector of m entries. Each element corresponding to a k-sided face is a vector of k difference vectors. Thus, `details` is a three-dimensional vector, and each entry can be represented by a triple of indices: (*level*, *face*, *difference vector*).

2.3 Design Decisions

Early in the lifespan of this project, I decided to write the code with an emphasis on readability and maintainability over raw speed and efficiency. In particular, though C/C++ offer so much opportunity for low-level twiddling, I chose to exploit the Standard Template Library (STL) of C++ for several data structures.

With the exception of a few arrays that are easy to manage, most large and dynamic data structures were implemented as STL vectors. The STL vector class offers random access similar to simple arrays, but offers several nice features over arrays, such as random insertion, appendage, and deletion, as well as automatic garbage collection (as much as possible in C++, at any rate). All the features translate into ease of coding at the expense of some object-oriented overhead.

Despite the reliance on vectors, efficiency was not forgotten. Wherever vectors had to be processed sequentially – such as when we loop over all faces to create F-F faces in forward

subdivision – iterators were used to avoid the linear traversal of a linked list when standard array-like indexing is used.

Consistent with the goal of maintainability and ease of coding, the choice was made to use Qt over more efficient but less powerful APIs such as GLUT. The built-in message passing system of Qt is extremely valuable for building a good user interface within an object-oriented framework. Contrasted with GLUT, which does not allow class member functions to be used as event handlers, Qt is definitely the best choice for writing user interfaces in conjunction with C++ and OpenGL.

In terms of user interface, I chose to keep the focus on the subdivision process rather than rendering. Early on, I had many options for configuring material properties, light positions, and so on. But in reality such options are unnecessary and may even detract from the real goal of the application. So, I instead define a default material for objects without a material defined, and have a set of static lights that work well for most models.

The user interface components that do exist, I think, exist for good reasons. Of course, rendering quality is important, and different tasks require different renderings. To that end, I tried to implement all of the rendering methods available with OpenGL, save for texture mapping. The critical rendering option that I wanted to implement is referred to as ‘Outline’ in the application, and refers to a wireframe rendering with hidden face removal. I had never implemented this particular rendering before, and it turned out to be somewhat problematic. Fortunately, the ubiquitous ‘red book’ has a good algorithm for it that exploits the stencil buffer; less fortunately, many graphics cards lack either hardware or driver support for the stencil buffer, and this rendering mode results in degraded performance as the effect is emulated in software. Nonetheless, several of the images in the results of Section 3 were rendered with the ‘Outline’ style, to nice effect.

The other major interface component, after rendering options, is the editing of vertices. While my implementation is not up to par with professional packages such as Maya, it suffices for some rudimentary vertex editing. To translate a mouse click that is received in screen coordinates into correct world coordinates, I turned to the GLUT library function `gluUnproject()`. This function, though reportedly buggy, has thankfully served its purpose well for this application.

2.4 Forward Subdivision

With the data structures described above, how does one actually implement Doo-Sabin subdivision in the forward case? Like most subdivision schemes, there is more to an implementation than plugging in the mask values; much bookkeeping must be done to ensure that an arbitrary model is handled properly.

Forward subdivision can be broken into two disjoint operations. First, the faces in *M* are contracted to produces the F-F faces of *M*'. Second, the F-E and F-V faces are created to bridge the gaps between the F-F faces.

2.4.1 F-F Face Creation

Creating F-F faces is a relatively simple task. We loop over all the faces in the model, and for each vertex of the face we use the mask values to compute a new vertex. The following code fragment describes the process of creating F-F faces.

```
for each face f in faces do
   create new face fnew
   for each vertex v in f do
      if v is on boundary then
         // compute vnew from boundary mask
      else
         // compute vnew from regular mask
      endif
      add index of vnew to indices of fnew
      add vnew to inactive set of vertices
   endfor
   add fnew to inactive set of faces
endfor
```

Note that, if there are *m* faces in *M*, then the first *m* faces of *M*' will be F-F faces. In particular, the *first* face of *M*' will always be an F-F face. This fact will be exploited in the reverse subdivision procedure.

2.4.2 F-E Face and F-V Face Creation

The creation of F-E and F-V faces is done concurrently. By looping over all vertices in *M*, we can be sure to get all F-V faces. Because each edge, and therefore each F-E face, is connected to some vertex in *M*, each edge will also be visited in the tour of the vertices.

The process is best described visually. Figure 6 traces the processing of a single vertex in a simple model of a cube. Beginning from the cube model in (a), contraction of the faces to create F-F faces produces (b). In (c), we select a single vertex *v*, shown in red. We use the `neighbors` structure of *v* to process every edge connected to it. In (d), the first such edge, *e*, is shown in red. Using information from the `faces` and `vertices` structures, we can determine which F-F faces need to be connected by *e*'s F-E face; (e) shows the F-E face (in beige) that is created. In (f), two edges of the F-E face are highlighted in blue, because these edges will also be used to make an F-V face. In particular, the rightmost blue edge is needed for *v*'s F-V face. The processing of edges connected to *v* continues in (g) through (k). In (i), all edges of *v* have been processed. At this point, we note that the blue edges have formed a cycle about *v*; these edges define the F-V face that corresponds to *v*. The last step in processing *v* is to create a new face using these edges.

Note that we only need to process each edge exactly once. However, each edge will have the opportunity to be processed twice: once for each endpoint vertex. To ensure each edge is only processed once, we exploit the unique index associated with each vertex: for each vertex-neighbor pair, we only process the edge if the index of the vertex is greater than the index of the neighbor.

One important issue to consider during F-E and F-V face creation is the face winding. Face winding refers to the order in which the vertices of a face are listed; reversing the winding also flips the normal of the face, and hence has ramifications on the rendering of the model. When we create F-E faces by joining vertices of its neighboring faces, there is no guarantee that the winding will be correct; in other words, the normal may point in the wrong direction, resulting in rendering artifacts. Similarly, the order in which the edges of the F-V faces are specified is not known, and the normal may point in the wrong direction.

Thankfully, the normals of the F-F faces is not changed by subdivision. So, if we assume that F-F face normals are correct, we can cascade the normal down to F-E and F-V faces. When creating an F-E face, we take the dot product of the F-E face and one of the neighboring F-F faces: if the dot product is less than zero, then the normal needs to be flipped, or, the face winding needs to be reversed. Similarly, we can compute the dot product of an F-V face normal and the normal of an adjacent F-E face, and reverse the face winding if needed.

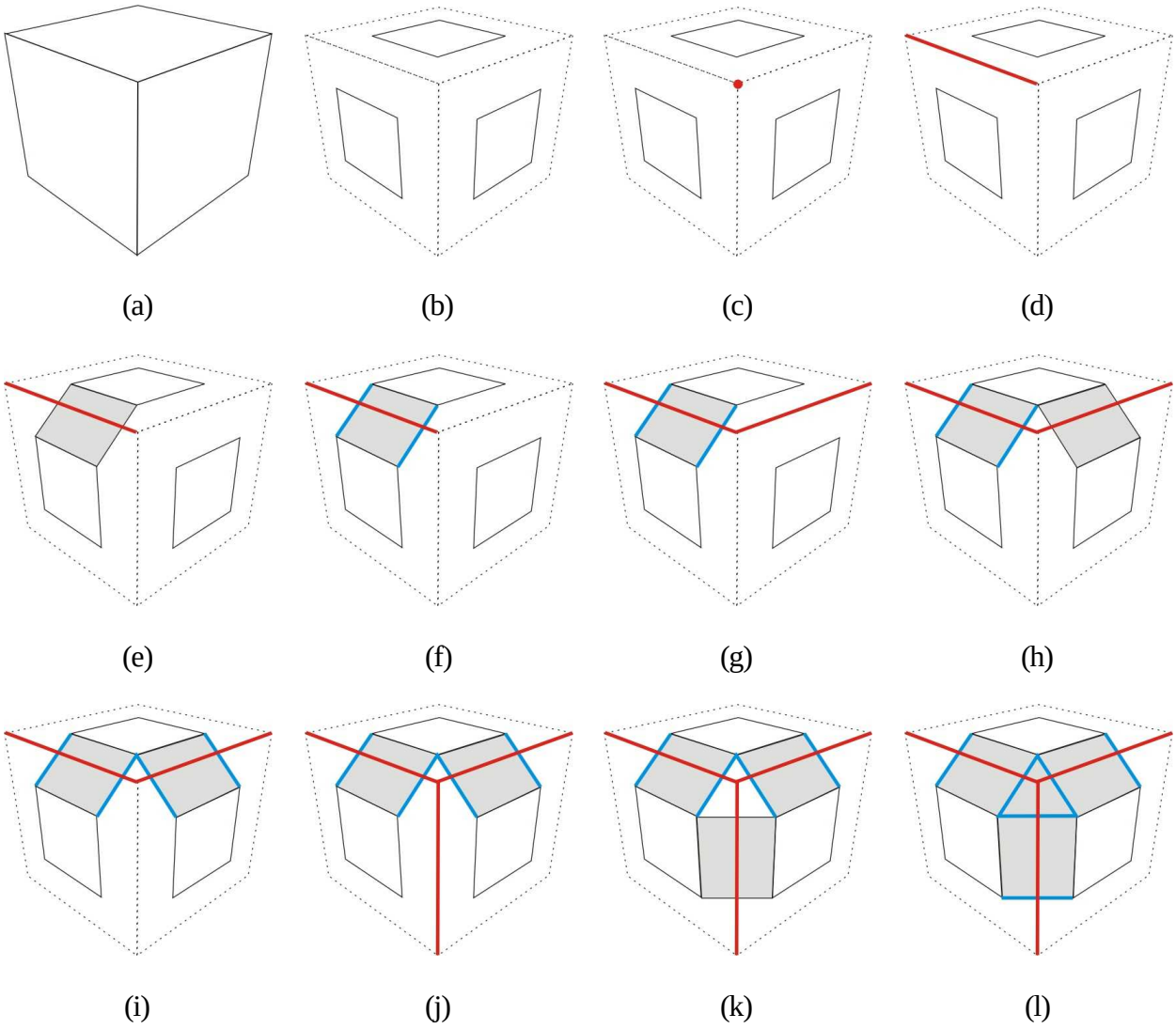


Figure 6: Creation of F-V face and adjacent F-E faces for a sample vertex. The vertex being processed is indicated by a red circle in (c). The red edges are edges connected to the vertex that are being/have been processed. The blue edges are used to make the F-V faces. See text for full discussion.

2.5 Reverse Subdivision

The algorithm for reverse subdivision is described fully in [3]. Here I will discuss the algorithm from an implementation standpoint.

The basic idea of reverse subdivision is to expand the F-F faces and discard the F-E and F-V faces. But how do we differentiate between the different types of faces? After subdivision, all faces are part of one large `faces` structure; no distinction is made between F-F, F-V or F-E

faces. However, it is possible to determine a complete set of only F-F faces provided we know the location of *one* F-F face.

As mentioned in Section 2.4.1, the first face of any set of faces will always be an F-F face, at least in the implementation of forward subdivision given in Section 2.4. We can also make the observation that an F-F face will only share edges with F-E faces; this follows immediately from the definition of F-E faces. Combining these two observations, we can visit all of the F-F faces as follows:

1. Start at a known F-F face (the first face of the current model).
2. For each edge *e* in the current face:
   a. Find the neighboring F-E face N_{FE}.
   b. Find the edge *e'* of N_{FE} that does not share any vertices with *e*; because all F-E faces are quadrilaterals, there will be exactly one such edge.
   c. Find the face that shares *e'* with N_{FE}. This is also an F-F face, call it C_{FF}.
   d. Recursively carry out Step 2 on child face C_{FF}.

As noted earlier, all F-E faces are quadrilaterals, and two edges are shared with F-F faces. This fact allows us to visit all the F-F faces in a graph traversal. What about the other two edges of an F-E face? They will be shared with F-V faces. As proved in [3], these edges form a graph, call it G_C, with the property that the connected components of G_C define all of the candidate vertices for a particular vertex in the reverse-subdivided model.

So, the reverse subdivision procedure is this: visit all F-F faces, expanding the faces according to Section 1.2 along the way; when an F-E face is traversed, keep track of the edges that are shared with F-V faces in a structure G_C; once all F-F faces have been visited, find the connected components of G_C and average all vertices in the component to get the final vertex.

Figure 7 illustrates a simple execution of this procedure. We start with a cube after one subdivision in (a). Next, we select the first F-F face and expand it, as in (b). For each edge, we find the connected F-E face (see blue face in (c)), and the child F-F face (gray face in (c)); the red edges in (c) denote the edges of the F-E face that are added to G_C. This process continues for the remaining edges of the face. In (f), we recurse on a child face, and repeat the process in (g). In (h), we again recurse on an F-F face. Finally, we arrive at (j) with all of the F-F faces

expanded, and the red edges of G_C. In (k), we find the connected components of G_C, and after averaging the vertices in each component, we reach the model in (l), the original cube.

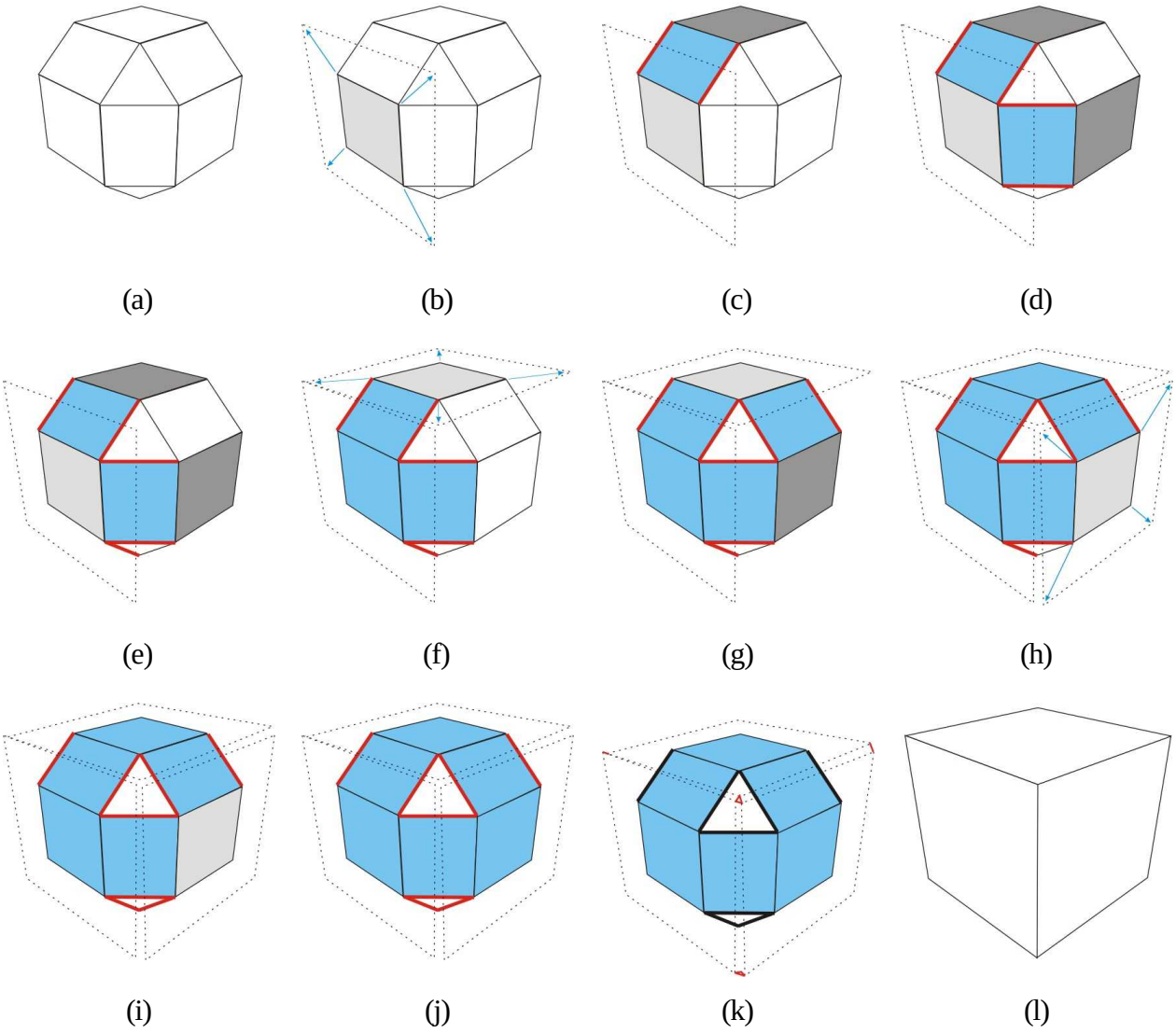


Figure 7: Reverse subdivision walkthrough. White faces haven't been processed. Beige faces denote the current F-F face being processed. Gray faces are children of the active F-F face. Blue faces are faces that have been processed. Red edges are the edges of G_C that are used to perform local averaging.

2.5.1 Multiresolution

Reverse subdivision results in a decrease in resolution. It may also result in the loss of information, because when the F-F faces are expanded, their vertices may not properly coincide. Local averaging of the candidate vertices solves the problem when our only goal is to

reduce the resolution and achieve a seamless model. However, if we wish to reconstruct the high resolution model from the low resolution version, we need to store additional information.

Section 2.2.3 describes a data structure called `details`. This structure stores a difference vector for each vertex of each face created by reverse subdivision. When we expand the F-F faces during reverse subdivision, every vertex of a face will be part of some component of G_C and will therefore be averaged. To allow us to return to the original vertex of a face, we store the difference between the original vertex (achieved by expanding the face about its centroid) and the average vertex of the component. Each face has its own details list, such that the indices of the details correspond to the indices of the face's vertex list. For example, if a face has vertex indices `[a b c d]` and details `[w x y z]`, then the true location of the face's third vertex is `vertices[c]+y`.

3 Results

The nice aspect of implementing a subdivision scheme is that testing the application is very easy: if the algorithm is incorrect, it will be blatantly obvious from the mangled results. Put another way, it is very unlikely for an incorrect algorithm to somehow produce the correct set of faces and vertices after subdivision (forward or reverse).

3.1 Cube

I used a simple cube for all the preliminary testing of the program, because it is the easiest object to debug visually. Figure 9 below show the results of forward and reverse Doo-Sabin subdivision on a cube. After subdividing twice to reach the model in Figure 9(c), the application was used to edit several vertices to reach the elongated cube shown in (d). As shown in (e) and (f), reverse subdivision loses much of the detail in the elongated model, returning to a very subtly different cube model in (f). However, with the multiresolution details, we can completely reconstruct (d) from (f).

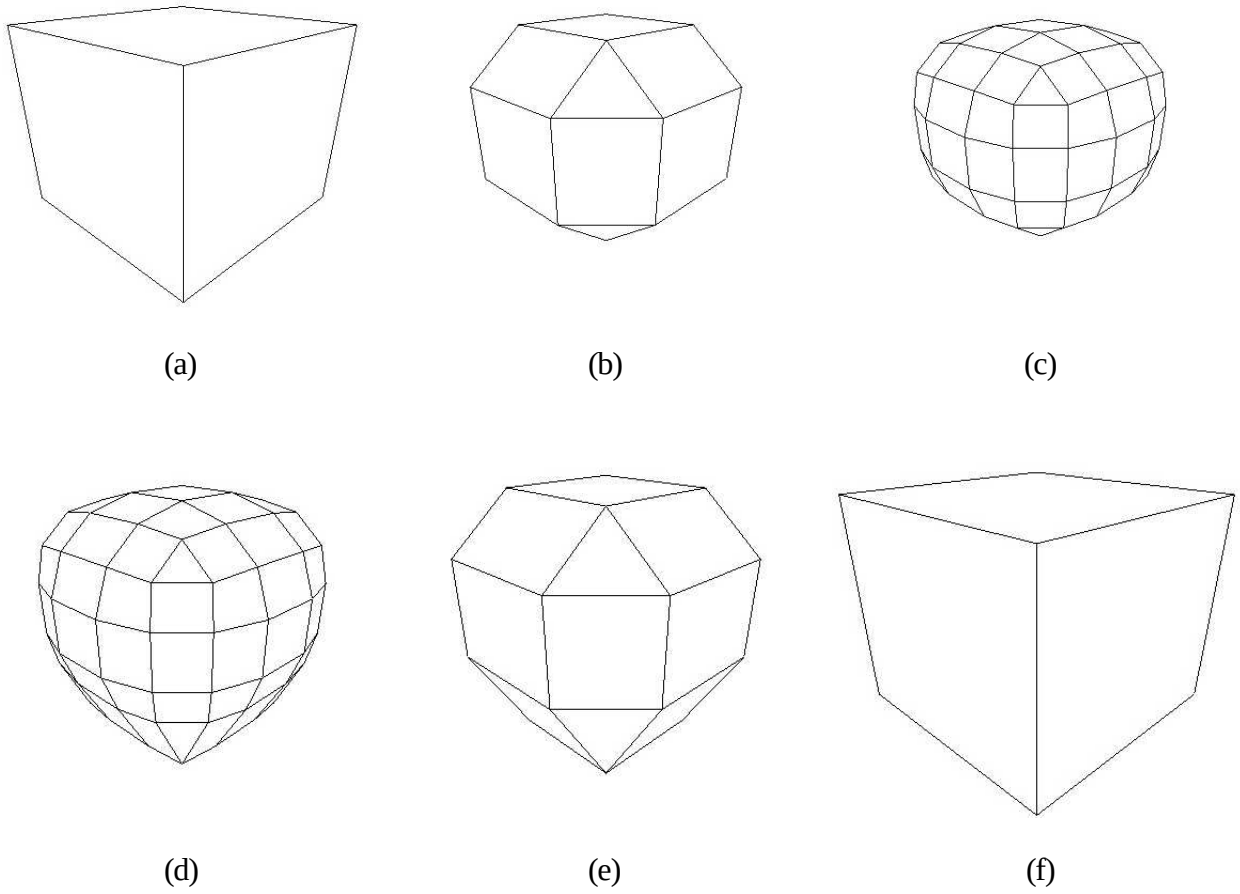


Figure 9: Forward and reverse subdivision of a cube. (a) Initial cube, M^0. (b) Cube after one subdivision, M^1. (c) Cube after two levels of subdivision. (d) Editing of vertices at subdivision level two. (e) M^2 reverse subdivided to M^1. (f) M^1 reverse-subdivided to M^0; restored to a cube.

3.2 Pawn

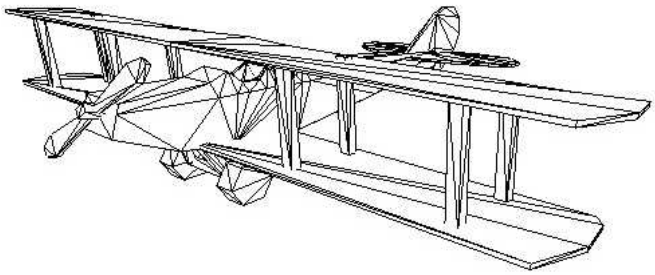
The pawn model is more complex than the cube – 304 faces compared to only six faces in the cube – but is an easy model to anticipate the result of subdivision for. Figure 10 below show the pawn model at one, two, and three levels of subdivision. At three levels of subdivision, the number of faces is exponentially greater than the original model (14594 compared the initial 304) and the increase in resolution and continuity is apparent.

(a) (b) (c) (d)

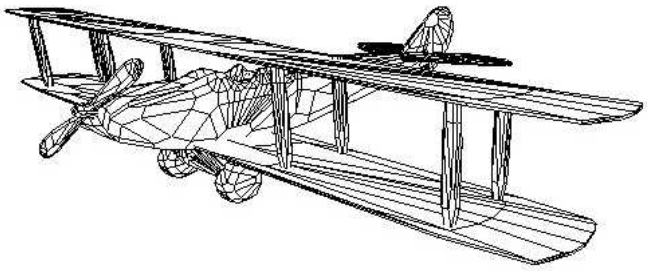
Figure 10: Forward subdivision of a chess piece. (a) Initial model, M^0. (b) After one level of subdivision, M^1. (c) After two levels of subdivision, M^2. (d) After three levels of subdivision, M^3.

3.3 Biplane

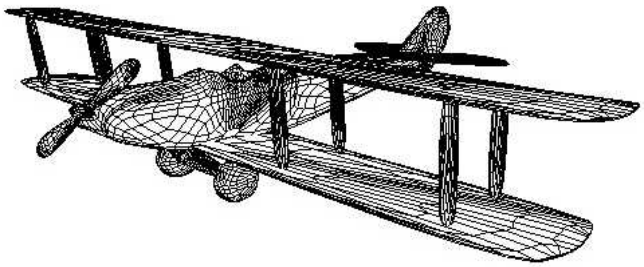
The biplane model is quite complex. The initial model has 354 vertices and 612 faces. After two levels of subdivision, the model has increased to 7376 vertices and 7420 faces.



(a)



(b)



(c)

Figure 11: Forward subdivision of a biplane. (a) Initial model. (b) After one level of subdivision. (c) After two levels of subdivision.

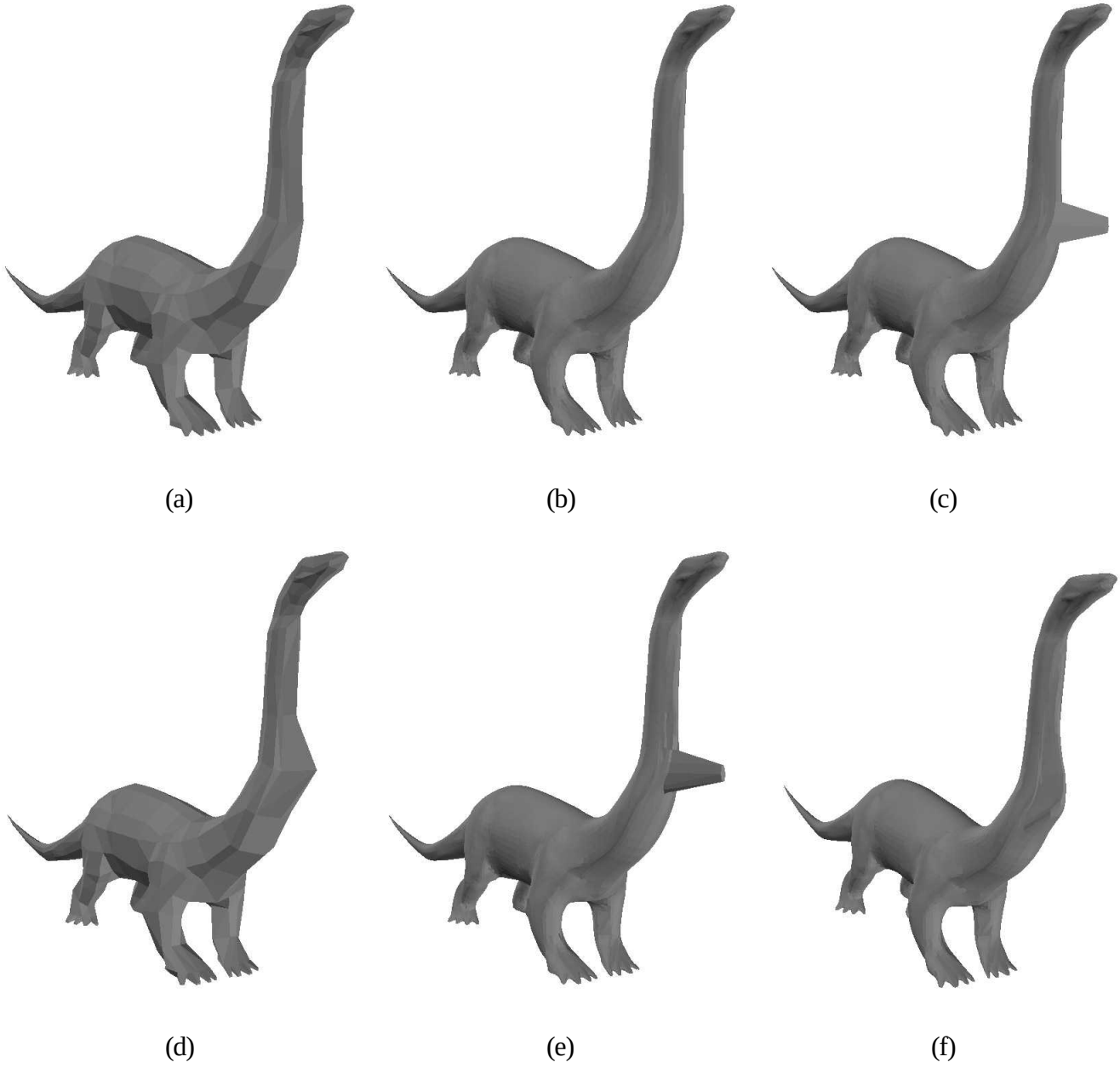
One thing of note about the biplane, and the reverse subdivision algorithm in general, is that it cannot be reverse-subdivided. This is because there are several disjoint groups that make up the plane: the hull, the landing gear, the wings, the propeller, and so on. Because these groups are disjoint, we cannot reach all of the F-F faces with our graph traversal algorithm employed in reverse subdivision.

3.4 Shark

The shark model is not terribly complex in its overall geometry, but it does have a lot of variation in the degree of its vertices. Doo-Sabin subdivision worked nicely on the model, increasing the original model's 263 vertices and 518 faces to 6216 vertices and 6220 faces after two levels of subdivision.

(a) (b) (c)

Figure 12: Forward subdivision of a shark. (a) Initial model, flat-shaded. (b) One level of subdivision, flat-shaded. (c) Two levels of subdivision, flat-shaded.



(a) (b) (c)

(d) (e) (f)

Figure 13: Forward and reverse subdivision of a dinosaur. (a) Initial model, M^0. (b) After two levels of subdivision, M^2. (c) M^2 with some vertices changed. (d) M^2 reverse-subdivided to M^0 + details. (e) Restoration of the edited M^2 from M^0 + details. (f) M^2 reached from the edited M^0, without use of the details. All images are flat-shaded.

3.5 Dinosaur

Here we use a dinosaur model to demonstrate the reconstruction of a model from a low-resolution representation and associated details. The initial model in (a) is comprised of 617

vertices and 849 faces. The model after two levels of subdivision has 11712 vertices and 11714 faces.

After subdividing the model twice, the application's interface was used to change several vertices on the neck of the model, giving it a bulge as in Figure 13(c). Reverse-subdividing the edited model twice to the original resolution yields (d); the bulge is visible, but clearly less prominent. When the multiresolution details are used during forward subdivision, we can completely reconstruct the edited model, as in (e); if the details are ignored, we lose the sharp bulge in the neck and reach a model (f) that is only slightly different that (b).

4 Future Work

The application program here is quite robust, but as in most software there is room for improvement.

Superficially, the application does have some issues with memory usage. There is an unfortunate tendency to crash when the operating system doesn't accommodate the application's hunger for memory. Thus, more robust error handling would be a good place to start modifying the system.

In Section 3.3, it was noted that reverse subdivision fails on models that have disjoint groups. This is not a problem whose solution is apparent, but certainly does not seem intractable. By starting at any face, we can traverse its neighbors recursively; by traversing the local neighborhood of all faces, one could determine the connected components of the model and then treat each disjoint component as a separate model. Though this process would no doubt be costly, subdivision of arbitrary topology models is typically not real time anyway.

Another useful extension of the program would be to export any existing details along with the face and vertex information when an OBJ file is exported. Because the OBJ format is quite loose, the details could be embedded in the same file as the face and vertex information. Then, our application and any other that desired to could load the details with a low resolution model and have the full multiresolution data; most other applications that deal with OBJ files are conditioned to just ignore unfamiliar tags (due to the fluid nature of the format), and so the additional information would simply enhance the low resolution model.

5 Conclusion

I think I succeeded in my initial goal of writing a system that is at once readable, maintainable, and extensible. In particular, a different subdivision scheme could easily be implemented as a child class of `SubdivModel`, just as `DooSabinModel` was, and plugged into the system with little or no modifications to the other classes.

In terms of the results achieved, the nice aspect of subdivision schemes is that there is not much subjectivity in judging the results. The algorithm either works, or produces garbage. The results of Section 3 show that the implemented system is a correct implementation of Doo-Sabin subdivision and its reverse. In that sense, this project could be judged a success.

References

[1] Amresh, A., G. Farin, and A. Razdan. *Adaptive Subdivision Schemes for Triangle Models*. Arizona State University

[2] DeRose, T., L. Kobbelt, A. Levin, and W. Sweldens. *Subdivision for Modeling and Animation*. SIGGRAPH Course Notes, 2000.

[3] Samavati, F. F., N. Mahdavi-Amiri, and R. H. Bartels. *Multiresolution Surfaces Having Arbitrary Topologies by a Reverse Doo Subdivision Method.* Computer Graphics Forum, Nov. 2001.

[4] University of California Davis, Computer Science Dept. *On-line Geometric Modeling Notes*. http://graphics.cs.ucdavis.edu/CAGDNotes/Doo-Sabin/Doo-Sabin.html

[5] Xu, X. and K. Kondo. *Adaptive Refinements in Subdivision Surfaces.* Eurographics 99 Proceedings, 1999.