Meshes

You all know what vertices and edges are in the context of graphs. In computer graphics, a *mesh* is a specify type of graph. Vertices are points in R^n where $n = 2$ or 3. Edges are line segments. Polygons are cycles in the graph of length greater than 2, specifically, minimal cycles with non-repeating vertices (except the first and last). Polygons (also known as faces) are assumed to be planar.

To keep things simple, we'll assume that each edge of our mesh belongs to at most two triangles. There may be edges that belong to only one triangle. Such edges are said to be on the boundary of the surface. Restricting ourselves to edges that belong to at most two triangles means that we can talk about one surface. If an edge were to belong to more than two triangles, then we would be talking about multiple surfaces that are stuck together. (This is allowed in general, but we're not dealing with it.) Technically, one uses the term *manifold mesh* for the cases we are considering. (The term "manifold" is not something you are typically exposed to as an undergraduate. You would learn about it if you studied differential geometry.)

We'll typically be talking about meshes made out triangles today, although in general one might like to have meshes made out of quads (each polygon has four edges) or be more general and have a mix of triangles, quads, etc.

Polyhedra

A *polyhedron* is a polygonal mesh that is the boundary of some 3D volume. A *regular polyhedron* is a polyhedron whose faces are congruent i.e. identical except for a rotation and translation. There exist many regular polyhedra. For example, you may be familiar with the convex regular polyhedra (called the Platonic solids): tetrahedron, cube, octahedron, dodecahedron, icosahedron.

Some polyhedra such as a torus or a coffee cup model have *holes*. For the coffee cup, the hole is in the handle. For a polyhedron with no *holes* the number of vertices, faces, and edges satisfies Euler's formula:

$$V + F - E = 2.$$

Verify this formula for a tetrahedron (pyramid) and cube. For objects with one hole,

$$V + F - E = 0$$

and for objects with n holes,

$$V + F - E = 2 - 2n.$$

ASIDE: the number of holes is called the *genus*.

Data structures for polygons

A typical way to represent a polygon mesh is to use tables. (The term table is very general here. It could be an array, or a hashtable, etc.)

- *vertex table*: each entry points to a point in $\Re^3$, $v_i = (x_i, y_i, z_i)$
- *edge table*: each entry points to a pair of vertices, $e_j = (v_k, v_l)$
- *polygon table (face table)*: each entry points to a list of vertices that define the polygon.

This is sometimes called a *polygon soup* representation of a mesh, since it says nothing about which polygons are connected to which.

It would be nice to have a richer representation of the mesh, since this would allow you to answer certain questions about connectivity. For example, suppose you are given a face and you wished to know which other faces are connected to that face. The above tables don't allow you to do that efficiently. If, however, you were to augment say the edge table so that it had a list of the faces that the edge belonged to, then you could answer this query. You could examine each edge of the given face (via the list of vertices of that face, stored in the face table) and then lookup in the edge table which other face that edge belongs to.

Similarly, you might also wish to represent the set of polygons that a vertex belongs to. This could be useful, for example, in an animation in which you are moving the vertex and you want to know which polygons are affected. For each vertex in the vertex table, we could have a list of pointers to the polygon (or edge) table.

There is a tradeoff, of course. The more information that is made explicit, the easier it is to reason about the connectivity of the mesh and the neighborhood relationships, but the larger is the representation.

[ASIDE: In OpenGL, surface meshes are typically represented as a polygon soup, but there are a few specialized mesh structures you can use as well, called strips or fans, for example, `GL_TRIANGLE_STRIP` or `GL_TRIANGLE_FAN`. This allows you to use slightly less memory.]

Parameterizing the points on a mesh surface

A mesh is a 2D surface in R^3. How to parameterize this surface ? Specifically, we know the vertices and edges of the mesh, but what about the points inside the faces? How do we refernence these positions? These points lie on a plane that contains the face. We will show how to reference points on the face by a linear combination of the vertices of the face.

Before doing so, we ask a more basic question. A triangle (v_1, v_2, v_3) is a 2D surface in R^3. How to parameterize points in the triangle ? Assume v_1, v_2, v_3 are linearly independent (so the three points don't lie on a line). Consider set of points:

$$\{av_1 + bv_2 + cv_3\}$$

where a, b, c are real numbers. This set spans R^3. If we restrict this set to

$$a + b + c = 1$$

then we get a plane in R^3 that contains the three vertices v_1, v_2, v_3, namely (a, b, c) is $(1, 0, 0), (0, 1, 0), (0, 0, 1)$ respectively. We can further restrict to points within the triangle by requiring

$$a = 1 - (b + c)$$

and

$$0 <= a, b, c <= 1.$$

These are called the "convex combinations" of v_1, v_2, v_3. Note that this constraint can be derived by saying we are writing a vertex in the triangle as

$$v = v_1 + b(v_2 - v_1) + c(v_3 - v_1)$$

and grouping the terms that involve v_1.

[ASIDE: In mathematics, a triangle in 3D mesh is called a "2-simplex" and a, b, c are called "barycentric coordinates of points in the triangle.]

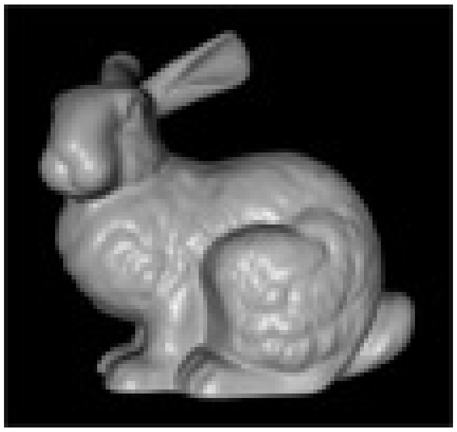
Finally, suppose a mesh has V vertices and F triangles (faces). Consider R^V, with one dimension for each vertex. For each triangle in the mesh, we have a corresponding 2-simplex in R^V. These F simplices define a mesh in R^V, whose points are in 1-1 correspondence with the mesh in R^3.

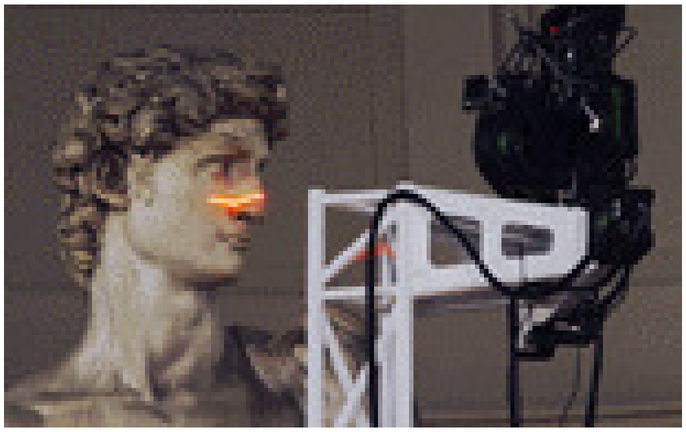
For example, suppose that v_1, v_2, v_3 is a triangle in the mesh. Then a point near the center of the triangle could be paramterized by say $(.3, .4, .3, 0, 0, 0, 0, \dots) \in \Re^V$.

Where do mesh models come from ?

You can build mesh models by hand. For example, the original Utah teapot that is provided in OpenGL was designed by a computer graphics researcher named Martin Newell in the mid-1970's. See story: `http://www.sjbaker.org/wiki/index.php?title=The_History_of_The_Teapot`

These days, one typically uses a 3D imaging device which yields a set of vertices (x, y, z). Often these devices come with software that also deliver a mesh in a standard format. See `http://graphics.stanford.edu/data/3Dscanrep/` for several recent large polygonal meshes that were popular to use in computer graphics research about 10 years ago, and a discussion of how they were obtained, for example, the "Stanford bunny" (shown below) and "Michelangelo's David".





Range scanning devices can be based on light (laser, photography) or lower frequency waves (radar). The models mentioned above were obtained using laser range finders. The scanning devices often use computer vision techniques. In a nutshell, these techniques combine images taken by two cameras, and use the slight differences in the image to infer the depth at each point. The basic principle is similar to how your brain combines the images seen by your left and right eyes to perceive depth. Typically these system also project a pattern onto the surface (using a laser or conventional light projector) to make the matching task easier. Details are omitted since this is more a topic for a computer vision course than a computer graphics course.

What's important is that the scanning devices typically define a depth map $z(x, y)$ on a regular grid of image coordinates (x, y). For each 3D position of the scanner/camera, you get a set of points $(x, y, z(x, y))$ in 3D space. There is a natural triangulation on these points, namely it is a height function defined over an (x, y) grid. (See leftmost figure in the "terrain" figures below.)

For solid surfaces, and especially surfaces with holes, one needs to make many such scans from different positions of the scanner i.e. you need to scan the back and front. This requires piecing together the scans. It is a non-trivial computational problem. Details are again omitted (since it is more computer vision than graphics).

The models require alot of data! For example, if we were to represent an object by having a vertex spaced every 1 mm (say), then we would need 1,000 vertices for a 1 meter curve on the surface, and so we would need about one million vertices for a 1 m^2 surface area. Several of the models in the Stanford database are of that size. The model of the entire David statue, which was scanned at about .25 mm resolution, has about two billion triangles! Many different scans were needed especially to capture the details within the concavities and these scans needed to be pieced together.

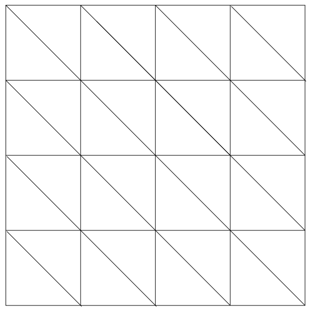
If you are interested in learning more, see the slides for links to a few examples of current state of the art, including the light stage work by Paul Debevec.

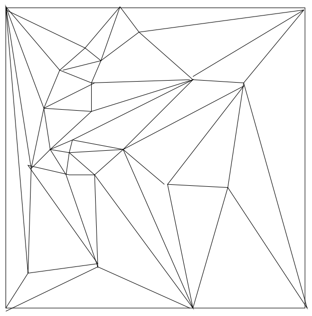
Terrains

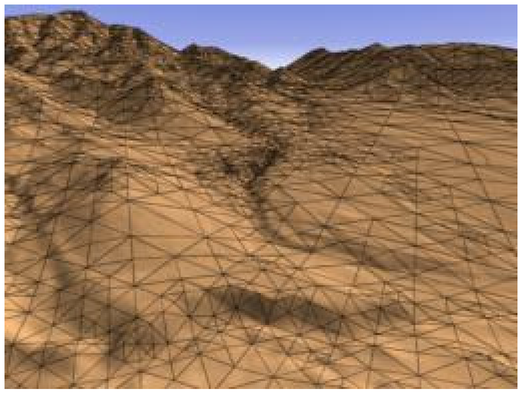
A *terrain* (or height field) is a polygonal mesh such that one coordinate (say z) is a function of the other two coordinates (say (x, y)) i.e. there is at most one z value for each x, y value. The points on the terrain can be written $(x, y, z(x, y))$.

Terrains can be defined over a *regular* polygonal mesh in the (x, y) plane e.g. integer coordinates, or over a *non-regular* polygonal mesh in the (x, y) plane. (See below.) Terrains are popular and useful, since all scenes have a ground and in most natural scenes the ground is non-planar. See the example [1] below on the right of a mesh, which was texture mapped and rendered. (Later in the course we will say what we mean by "texture map".)

Note that scanning devices mention above often compute terrain meshes even though the objects being scanned are not themselves terrains, since they have front sides and back sides and holes. This is why multiple scans of each object are needed and the terrain meshes must be combined into non-terrain meshes.


regular (x, y)


non-regular (x, y)


Example

[1]The image was taken without permission from Peter Lindstrom's Georgia Tech web page.

Level of Detail (LOD)

We use the term *resolution* to refer to the size of polygons in the model, for example, the length of the shortest edge. As we discussed last class in the context of fractals, if we limit ourselves to a given smallest edge size (say 1 mm) then there may be details in the surfaces geometry at a smaller size that we ignore when modelling the object.

The resolution at which an object should be displayed in some image might be much coarser then the resolution of the model in the database. For example, if a mudpile or rockface were viewed from up close then each triangle in the model might project to multiple pixels and so the highest (finest) resolution of the mesh would be needed to draw the object accurately. But if the same object were seen from a great distance, then each triangle might occupy only a small fraction of a pixel and so the fine resolution of the model would not be needed. One often refers to the *level of detail* (LOD) of a model when describing the resolution that is being represented.

One approach to having multiple levels of detail is to *precompute* several models that differ in resolution. For example, a low resolution model might have 10^2 polygons, a medium resolution model might have 10^4 polygons, and a high resolution model might have 10^6 polygons, etc.

How do you choose which model to use at any time? One idea is to consider, when drawing an object, roughly how many pixels the object would occupy in the image (using a bounding box for the object, say) and then decide which resolution model to use based on this number of pixels.

For example, suppose you wanted to use the Stanford model of Michelangelo's David in an image, such that whole David is viewed from a distance of 20 meters and occupies only a small fraction of the image. Obviously it would be wasteful to process every one of the two billion polygons in the model. Instead, you would use a lower resolution model.

A second approach is to represent the model using a data structure that allows you to vary the number of polygons by simplifying the polygonal mesh in a near-continuous way (see below). For example, you could let the resolution of the model vary across the object in a *viewpoint dependent* way. In an outdoor scene, the part of the ground that is close to the camera would need a higher LOD, whereas for the part that is further from the camera a lower LOD would suffice. Thus, we could use a coarse resolution for distant parts of the scene and a fine resolution for nearby parts of the scene. For the example of the terrain the previous page, the surfaces in the upper part of the image which are way off in the distance would not need to be represented as finely (in world coordinates i.e. $\Re^3$) as the polygons at the bottom of the image which are relatively close to the camera.

Mesh Simplification

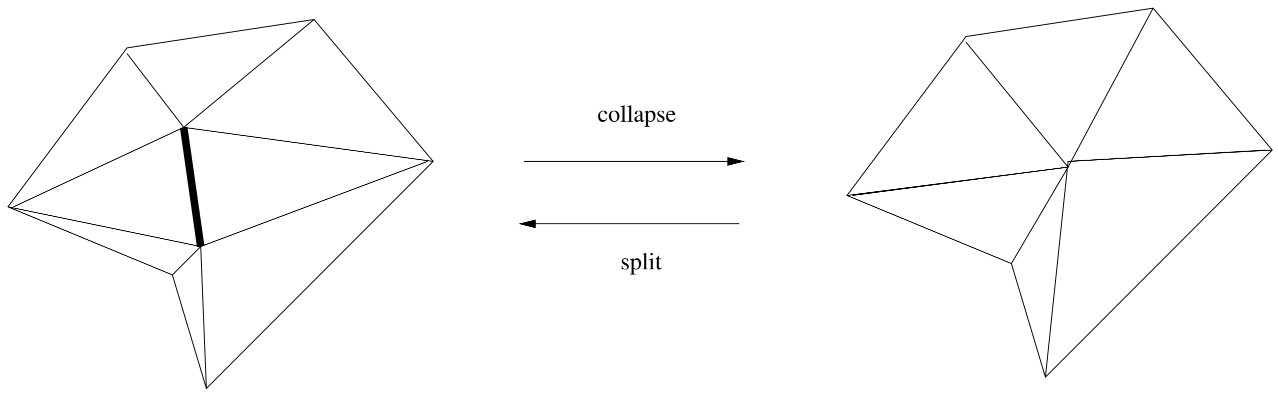
How do we obtain models with different level of detail ? For example, suppose we are given a triangular mesh having a large number of triangles. From this high LOD mesh, we would like to compute a lower level of detail mesh. This process is called *mesh simplification.*

Edge collapse and vertex split

Suppose we are given a triangulated mesh with V vertices, and we wish to simpify it to mesh having $V - 1$ vertices. As shown in the example below, we can collapse one of the edges (defined by a pair of vertices) to a single vertex. There are two ways to do this. One way is to delete one of the vertices of this edge. The other is to replace both vertices of the edge by one new vertex. Either way, a new triangulation must be computed (locally) to take account of the fact that vertices have been deleted or replaced.

Notice that when you do an edge collapse, you lose two triangles, one vertex, and three edges, and so $V + F - E$ is unchanged.

Also notice that an edge collapse is invertible. To go the other direction, you can split a vertex into two (essentially adding a new vertex) and join the two vertices by an edge. The vertex you are splitting may belong to several edges, and so when you split the vertex you need to decide whether these edges remain with the vertex or whether they instead join the new vertex.



Subtle issues can arise. For example, faces can be flipped. In the first example below, the boldface edge is collapsed and all edges that are connected to the upper vertex are deleted and/or replaced by an edge to the lower vertex. Consider one such new edge, namely the dotted one shown on the right. This dotted edge in fact belongs to *two* triangles: one whose normal points out of the page, and a second whose normal points into the page. The latter is thus flipped. This is not good. For example, if the V vertex surface were a terrain then the $V - 1$ vertex surface should also be a terrain.

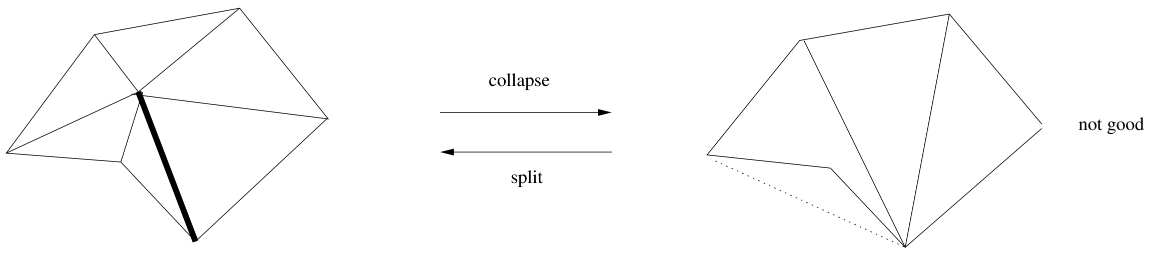
In the second example below, a different edge is collapsed and no such problem arises.

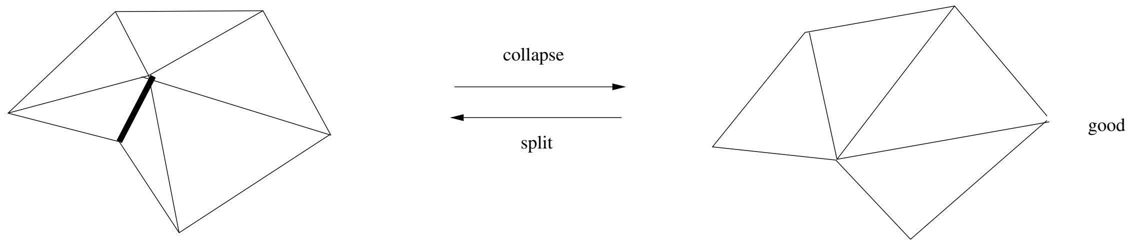
You can organize edge collapses using a forest of binary trees. The leaves of the trees are the vertices of the original mesh. Each edge collapse creates a sibling relationship between two nodes in the tree, such that the parent of these siblings is the vertex that the pair of vertices collapses to.

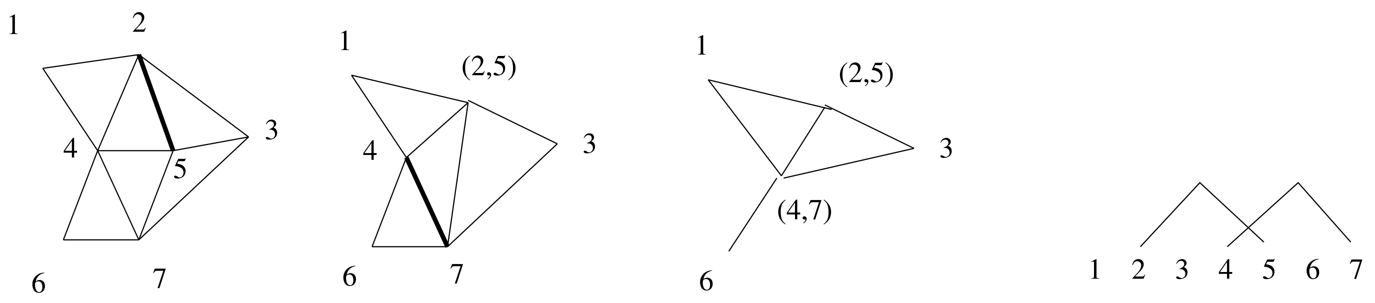
Start with the entire high LOD model, such that each vertex is its own tree (a trivial tree consisting of a root only). Then, if two vertices are connected by an edge and you choose[2] to collapse this edge, then a parent node is defined for these two vertices and the trees for these two vertices are merged. The representative vertex for the new parent node is either one of its children vertices, or some new vertex that replaces the two children vertices. This information must be stored at the parent vertex, so that it is possible later to *split* the parent vertex, that is, to recover the children vertices (and the edge between them). As many edges can be collapsed as desired, i.e. one can simplify the mesh from V vertices down to any V' vertices, where $V' < V$.

The example below shows the basic idea. Two edge collapses are shown. Although the rightmost "triangulation" doesn't appear to be a triangulation, you must keep in mind that only part of the mesh is shown. In particular, verticles 1, 3, 6 are each connected to other vertices not shown here.

[2]I am not specifying how you decide which edge to collapse.







Such a data structure can be used to choose the level of detail at render time. One can begin with a greatly simplified mesh, whose vertices are the roots of the trees in the forest. Once can then expand the mesh by "splitting" (or subdividing) vertices until the resolution of each part of the mesh is fine enough based on some criterion. [ASIDE: mesh simplification and LOD choice is a highly technical area in computer graphics, and my goal here is just to give you a feel for the sort of problems people solve. For an example, see `http://research.microsoft.com/en-us/um/people/hoppe/proj/pm/`]

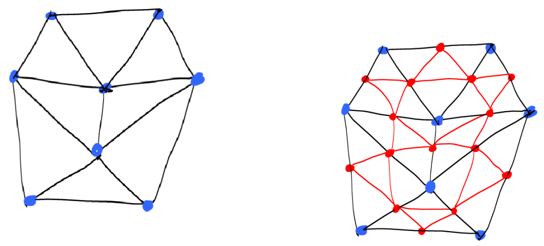
Subdivision surfaces

Last class we discussed how to interpolate between a sequence (or grid) of vertices, in order to define a smooth curve (or surface). We use cubics (and bicubics). There are many very nice methods such as those for modelling curves and surfaces in this way. However, there are limitations, for example, they have strong restrictions on the sampling of the points. Bicubics typically require that one piece together 4×4 grids. That was fine for Martin Newell to make the Utah teapot. But it typically non-trivial to make a large number of 4×4 grids from the scans that one obtains of objects such human bodies.

A different set of techniques, known as surface subdivision, were invented long ago which relaxed these requirements. I discussed one of these in class, namely Loop subdivision (which was named after Charles Loop, not `for/while` Loop). This scheme assumes the surface has been triangulated.

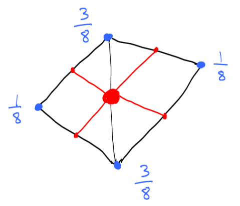
The general idea of subdivision is to make a smooth mesh with many vertices from a coarse mesh that has relatively few vertices. One iteratively "subdivides" the mesh, introducing more (smaller) triangles, such that the vertices of the triangles define a "smoothed" version of the surface at the higher level.

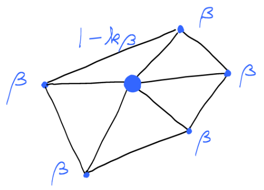
In Loop subdivision, every triangle is broken down into 4 subtriangles. There is a unique way to do this. Given a mesh (such as on the left below), insert a new vertex on the midpoint of each edge and then make three new edges inside each face, namely joining the three new vertices (see red vertices and edges in figure on the right).



To make the new surface smoother than the original one, the vertices need to be moved. For the new (red) vertices that we added, there are four natural vertices to consider when choosing the position of the new vertex. Each new (red) vertex sits on an old (black) edge, and there are the two old (blue) vertices that define this edge. There are also two old (blue) vertices opposite this edge in the face of the two old triangles to which this edge belongs. Loop lets the weights be each $\frac{3}{8}$ for the adjacent vertices, and he lets the weights be each $\frac{1}{4}$ for the opposite vertices.

We also move the original vertices. Each of the original vertices is connected to k other vertices in a ring. (Hence we sometimes need to know which are the vertices that v is connected to be an edge.) We want to give the original vertex'es position some weight. We also want to give some weight (say β) to all the other vertices that surround it. If we give each of the surrounding vertices a weight β, then the total weight is $k\beta$ which means the weight of the original vertex is $1 - n\beta$. What is β ? There is no strict rule. One simple scheme is to let the original point keep about half its weight ($1 - k\beta = \frac{1}{2}$) so $\beta = \frac{2}{k}$.


$\frac{3}{8}$
$\frac{1}{8}$
$\frac{1}{8}$
$\frac{3}{8}$


$1-k\beta$
β
β
β
β
β