CS554 Geometric Modeling

Project2 Report

Tianle Yuan (933946576)

February 6, 2021

1. (Smoothing) Implement four weighting schemes for surface smoothing: uniform, cord, mean curvature flow, and mean values coordinates. This amounts to doing the following:
For I = 1 to N

For each vertex V_i:

$$V_i.x = V_i.old\_x + dt \sum_{j \in N(i)} w_{ij}(V_j.old\_x - V_i.old\_x)$$
$$V_i.y = V_i.old\_y + dt \sum_{j \in N(i)} w_{ij}(V_j.old\_y - V_i.old\_y)$$
$$V_i.z = V_i.old\_z + dt \sum_{j \in N(i)} w_{ij}(V_j.old\_z - V_i.old\_z)$$

N(i) records the set of vertices that are adjacent to V_i, and dt is the timestep. Note:

when dt is too large, the above process may "blow up". Also, for cord, mean curvature flow, and mean value coordinates schemes, the weights change after each iteration. Try recomputing them after each iteration and compare that to the case when you always use the weights derived from the initial mesh configuration（Answered at the last）. Which method is faster? Which method provides a higher quality? Use tables and figures to support your findings. For each model and weighting scheme, you will probably need to run smoothing N times. Try different N's.

For all of those schemes, I tried the combination between time step with 0.3, 0.6, 0.9 and N with 5, 10, 15, 20. From **Table 1** to **Table 4** below, we can clearly see the calculation speed and smooth results of each method. My conclusion is that the speed sequence is: **Uniform** faster than **Cord** than **Mean Curvature Flows** than **Mean Value Coordinates** by calculating the average operating time. For the smoothing quality, I'd like to define it in the following ways:

(1) Correctly flatten the surface of the shape without adding other non-existent graphic features. **Cord** scheme obviously violates this point (see the feline wings in the last column of **Table 2** and **Figure 9**).

(2) When the image features are being blurred, the smoothing results can keep the original features of the graph as far as possible. By comparing the result between **Uniform** and **Mean Value Coordinates** schemes, the later one keeps more character of feline such as wing shapes and legs figure (see **Table 1 & 4** and **Figure 8 & 11**).

(3) When the iteration step size is large, as the number of iterations increases, the image needs to continue to support iteration rather than breakage. **Mean Curvature Flows** scheme's stability is obviously influenced by iteration step size (see lower triangular matrix in **Table 3**).

Thus, **Mean Value Coordinates** preserves the basic features of the original graph while maintaining the stability of the iteration. The **Mean Value Coordinates** provides highest quality**.**

(1) Uniform (Note: we are using GL_FLAT over here) :

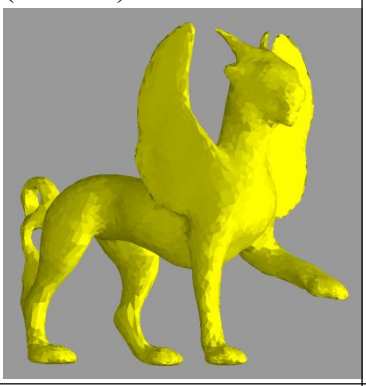
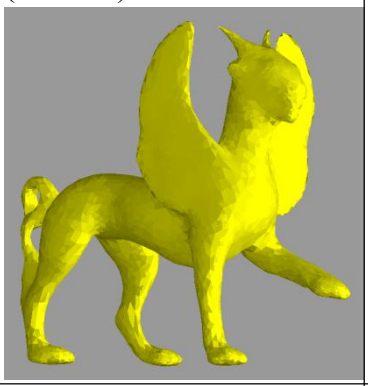
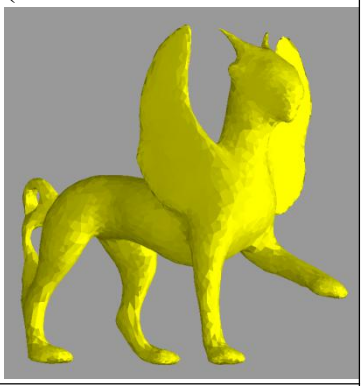
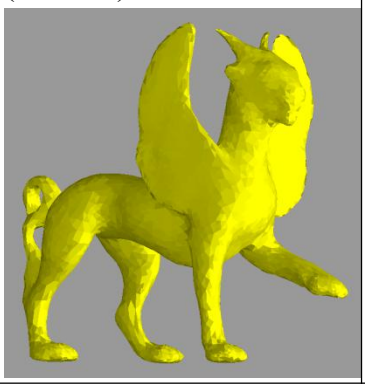
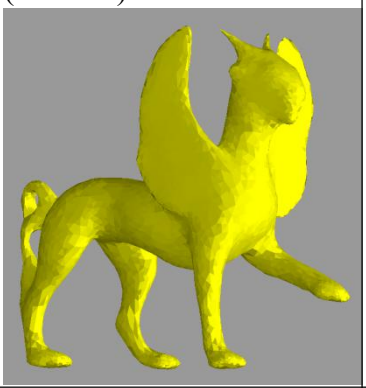
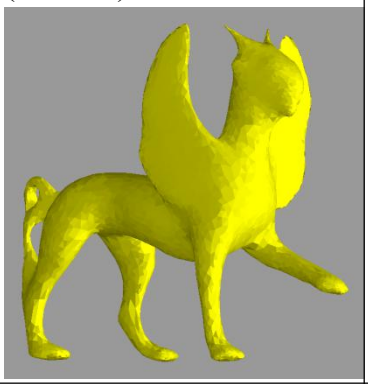
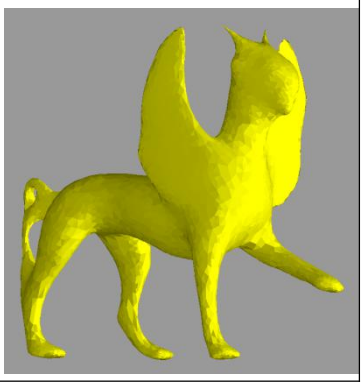
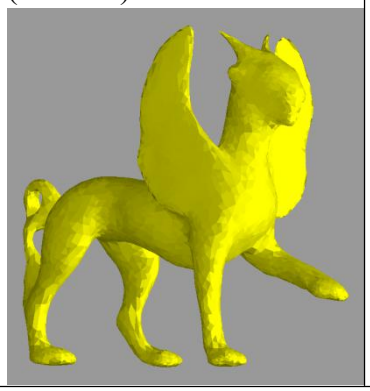
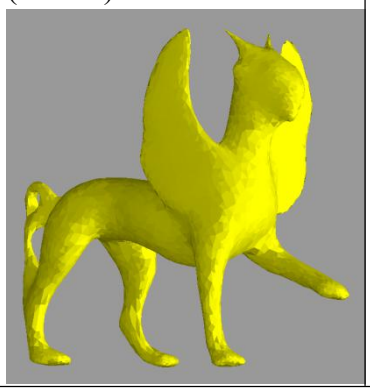
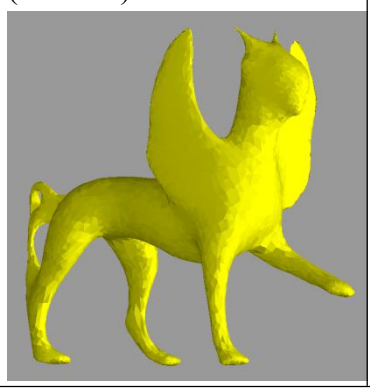
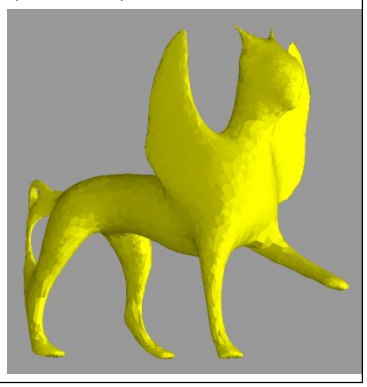
	5	10	15	20
.3	(+1.527s) 	(+1.583s) 	(+1.603s) 	(+1.591s 
.6	(+1.592s) 	(+1.532s) 	(+1.546s) 	(+1.579s) 
.9	(+1.574s) 	(+1.57s) 	(+1.511s) 	(+1.609s) 

Table 1. Uniform scheme experiment

(2) Cord (Note: we are using GL_FLAT over here) :

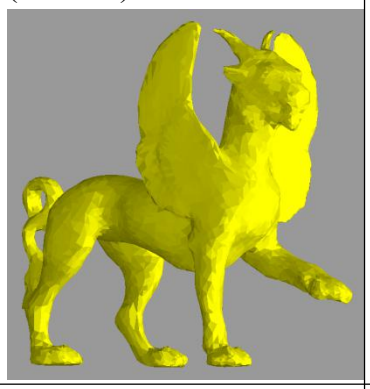
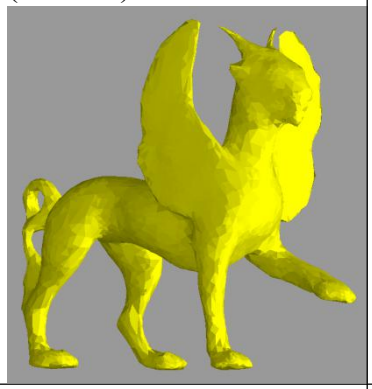
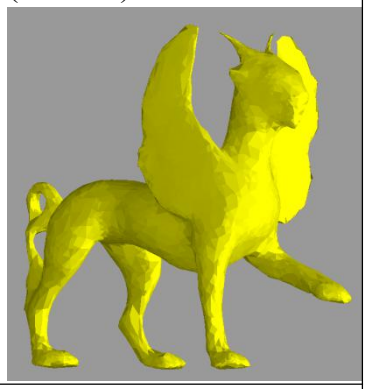
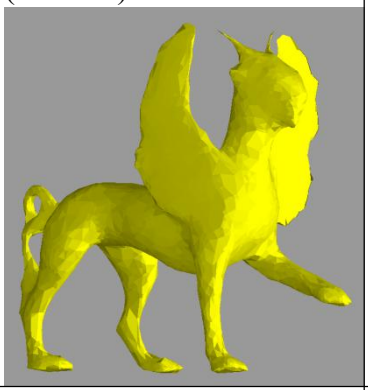
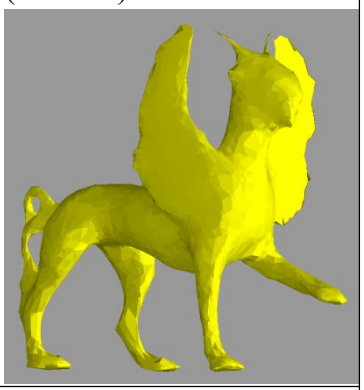
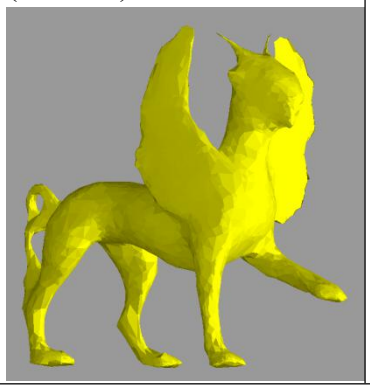
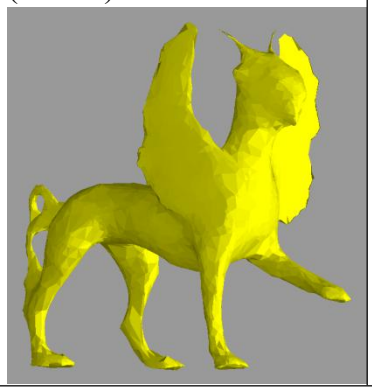
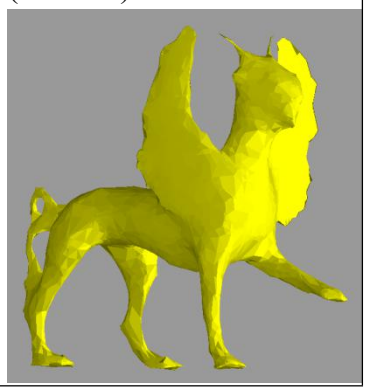
	5	10	15	20
.3	(+1.556s) 	(+1.592s) 	(+1.566s) 	(+1.536s) 
.6	(+1.576s) 	(+1.558s) 	(+1.483s) 	(+1.566s) 
.9	(+1.588s) 	(+1.536s) 	(+1.57s) 	(+1.595s) 

Table 2. Cord scheme experiment

(3) Mean Curvature Flow (Note: we are using GL_FLAT over here) :

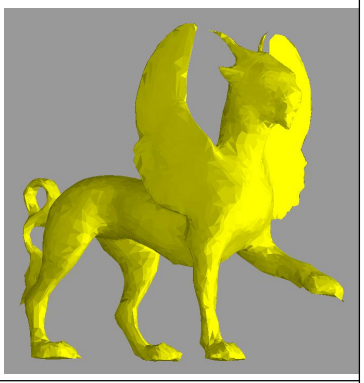
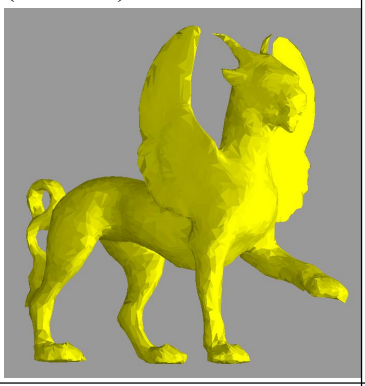
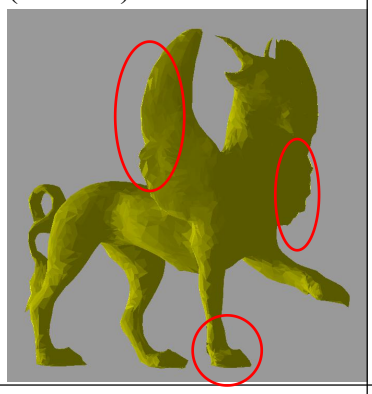
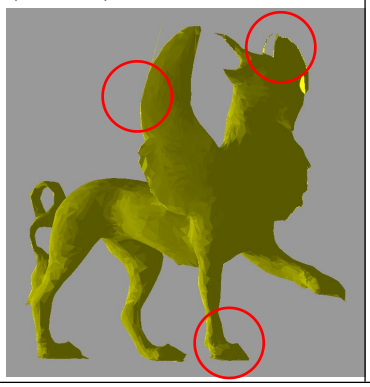
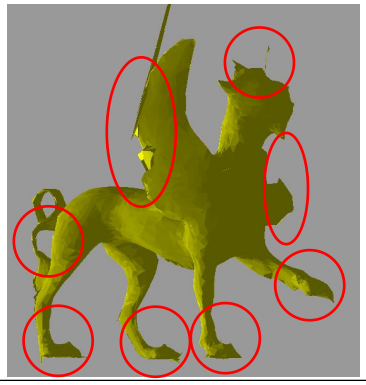
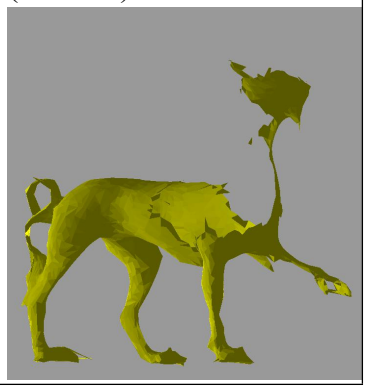
	5	10	15	20
.3	(+1.608s) 	(+1.592s) 	(+1.574) 	(+1.635s) 
.6	(+1.558s) 	(+1.579s) 	(+1.592s) 	(+1.552s) 
.9	(+1.586s) 	(+1.59s) 	(+1.588s) 	(+1.553s) 

Table 3. Mean Curvature Flow scheme experiment (Red circle with singular points)

(4) Mean Values Coordinates (Note: we are using GL_FLAT over here) :

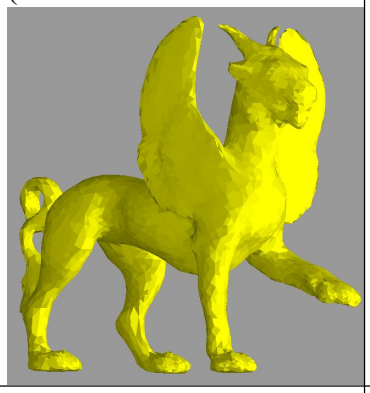
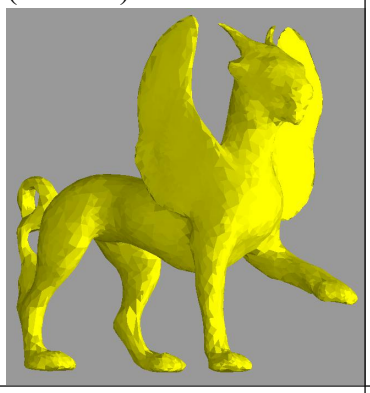
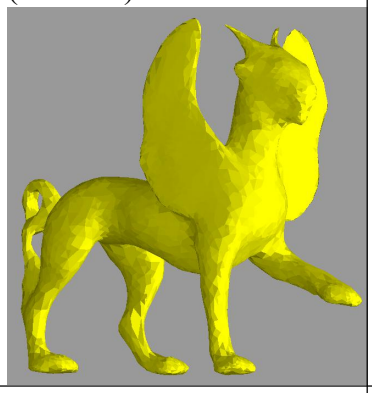
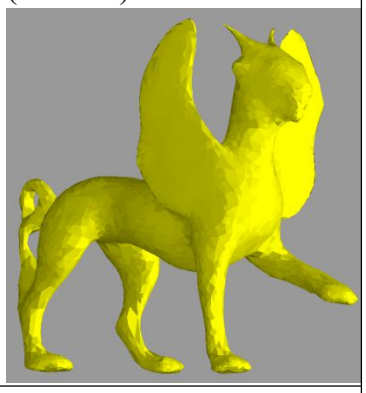
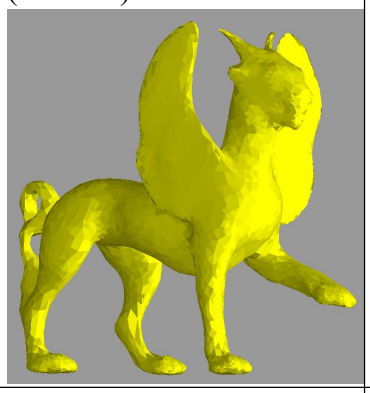
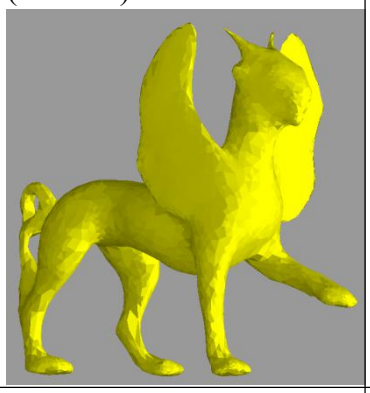
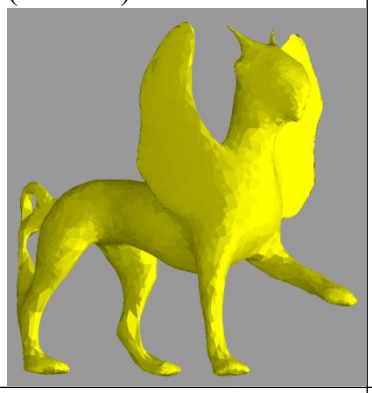
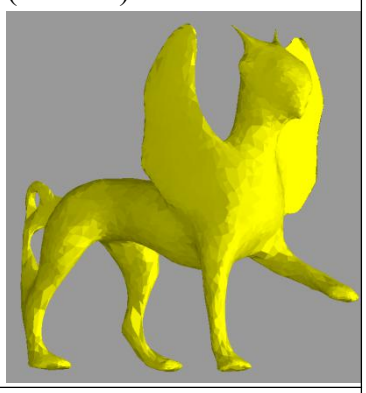
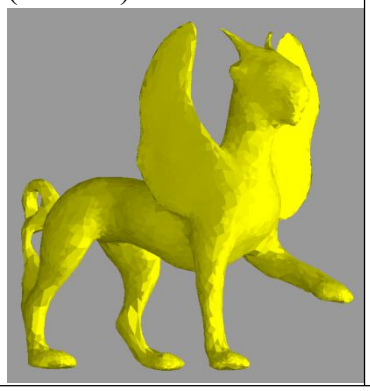
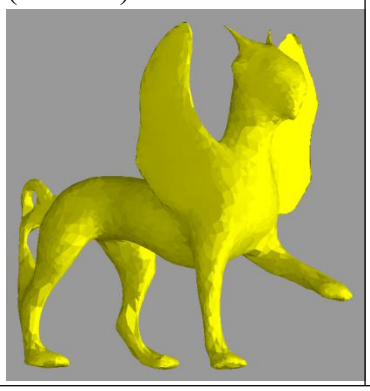
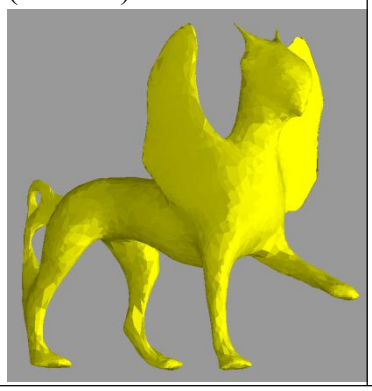
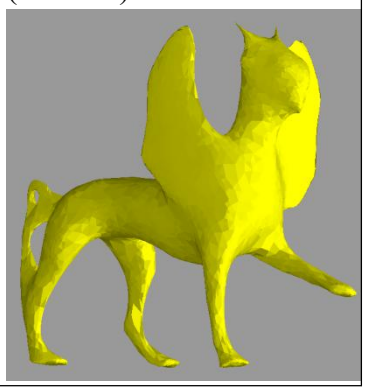
	5	10	15	20
.3	(+1.74s 	(+1.619s) 	(+1.6762) 	(+1.606s) 
.6	(+1.635s) 	(+1.578s) 	(+1.627s) 	(+1.649s) 
.9	(+1.589s) 	(+1.687s) 	(+1.625s) 	(+1.633s) 

Table 4. Mean Value Coordinates scheme experiment

Explanation of the strange situation in Table 3 of Mean curvature flow (MCF) experiment:

For the several strange results, whose surface has strange normal and erosion, in the last three columns, I have given explanations below:

(1) ***The limitation of numerical error:*** As we all know, a double number in C++ can only be saved in the range of 0~ 2.22e-16. However, with a large step, the coordinates of the vertices of the model will shrink really fast like the picture shown in **Figure 1** and the data of the corner structure will finally go into the meaning less as the picture shown in **Figure 2** :

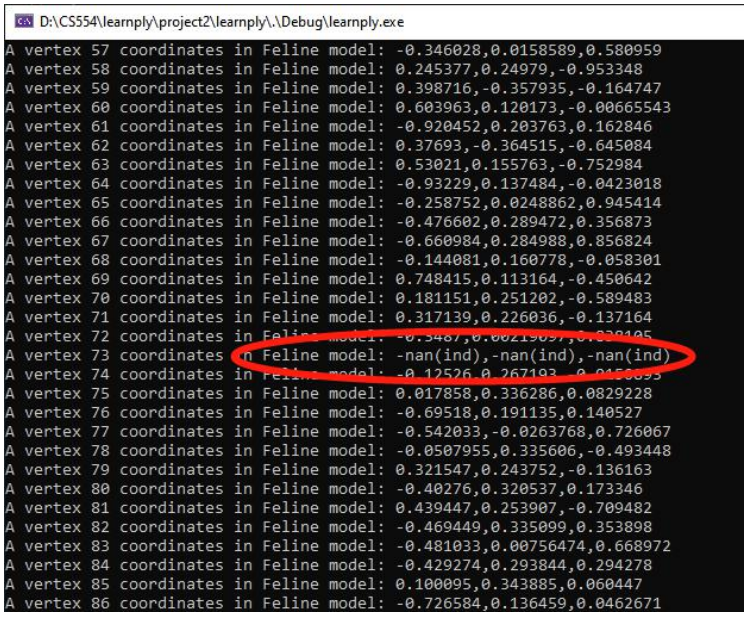


Figure 1. The 15th iteration of MCF with dt = 0.9 (Feline model)

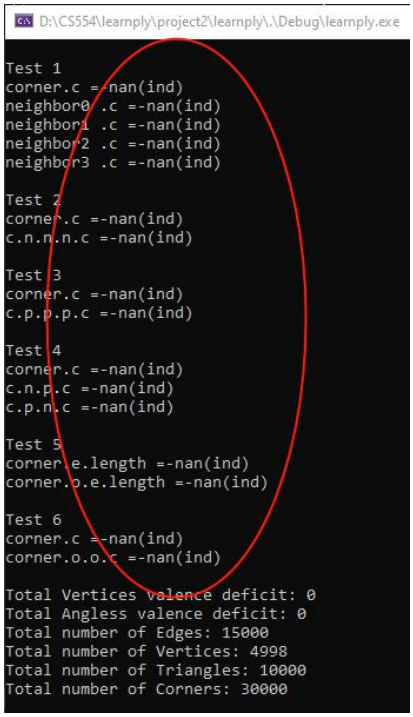


Figure 2. The 20th iteration of MCF with dt = 0.9 (Feline model)

In my algorithm of smoothing, I choose to construct a new polygon for each smooth iteration, which means all the parameters of smoothing will be calculated in the updated class Polygons (connect with new Vertex, Edges, Corners, Triangles). In each triangle mesh, each Edge's length and each Corner's angle are all saved as the double value. Those two variables are always connected with each other.

Firstly, when we set value of Corner's angle we need to use dot product to calculate the angle. However, in program, to get the angle we have to calculate like this:

```
Angle = acos(dot(vector1, vector2)/(length(vector1)*length(vector2)))
```

The "length" comes from the member of Edge, so if the length of an edge is too small (smaller than 2.22e-16), the final angle will be ∞, which means out of memory (equals to -nan(ind)). This is the reason of **Figure 2** red circle.

Then, when we need to update the length of each Edge, we have to use Corner to locate the new coordinates of each Vertex. So if the Corner's angle is too small (tend to be 2.22e-16) or two Vertices's distance are too close, the final length of edge will goes into meaningless tend to zero (equals to -nan(ind)).This is the reason of **Figure 1** red circle.

If the value of those two variable is -nan(ind), the shape erosion will then shows up.

(2) ***The limitation of cot() method:*** The method of cot() itself is really sensitive, if we set the time step too big, the value of cot() will change much bigger (unless near PI/2). Then the value will go into a singular point. Thus the sensitive, jump will impact the stability of the solution of the smooth equation.

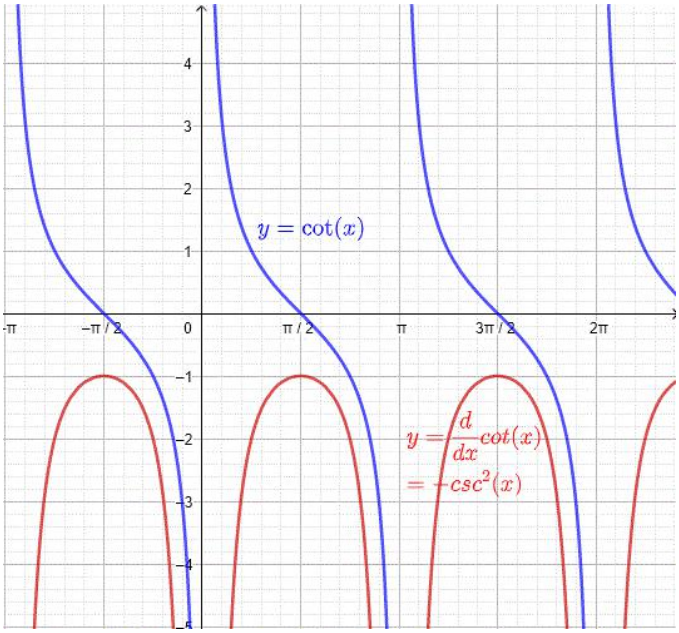


Figure 3. The graph of Cot() and its derivative

(3) ***The limitation of explicit method:*** The explicit method it self is not stable when the time step is large (according to the course notes: explicit update scheme is Unstable and Requires small step-size). Thus for the cases with dt = 0.3, 0.6, 0.9, compared with the case of dt = 0.1,

their time steps are too large. If we use dt = 0.1 (as **Figure 10**), we can do at least 60 times iterations.

(4) ***The limitation of model itself:*** The model shape will also affect the result of the times of final smoothing iteration. Some surface vertices with big curvature will be easier leading to singular points. For the model Feline and Sphere, although they almost have the same number magnitude of vertices, the Sphere can do as much as possible iterations by using MCF scheme if the M.M. allows, as **Figure 4** shown below:

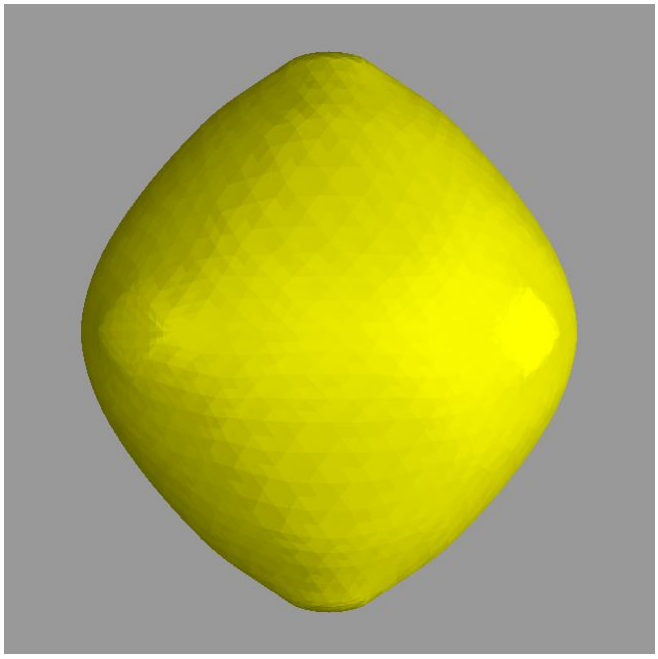


Figure 4. MCF's result of sphere smoothing after iterating 30*10 = 300 times (dt =0.9)

Try recomputing them after each iteration and compare that to the case when you always use the weights derived from the initial mesh configuration.

Actually, this part is really interesting. If we always use the weights derived from the initial mesh configuration, for **Figure 8,** it will not change because **Uniform** scheme's weight always equals to 1/N(j). For the uniform scheme, we can iterate as much as possible times that we want.

But for **Cord**, **Mean Curvature Flow**, and **Mean Values Coordinates**, compared with **Figure 9**, **10** and **11**, we can see the different limitation in **Figure 5**, **6** and **7** when we always use the weights derived from the initial mesh configuration.

Comparing **Figure 5** and **7** we can make a conclusion that: Except that the back width of **Cord** Feline is somewhat different, the appearance of the three are all similar to the result of **Uniform**. Another point is that the number of iterations of **Cord** and **Mean Values Coordinates** becomes the same as that of **Uniform**. I guess the reason is because, as we set weight always the same as initial, the weight can actually be seen as "constant weight number". As the iteration of **Cord** and **Mean Values Coordinates** schemes with the original "constant weight number" is

stable, then the iteration is no different than **Uniform** scheme, essentially.

Also if we compare **Figure 5** or **7** with **Figure 6**, we can find that even with the original "constant weight number", the **Mean Curvature Flow** scheme is not stable. The reason should be that because of the sensitivity of Cot() function, as a iteration changes the gradient, the weight should also adapt to gradient changes. Only simply reducing the time step size cannot sufficiently offset the effect of gradient change on Cot()'s sensitivity.

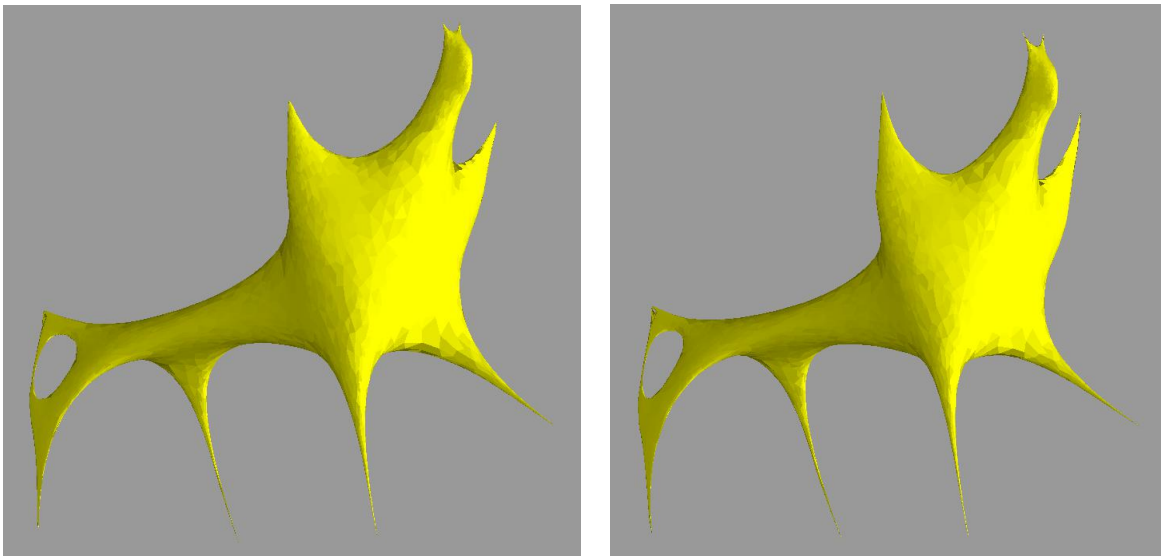


Figure 5. Uniform (left) & Cord (right) scheme's limitation with initial weight (iterating 200 times, dt =0.9)

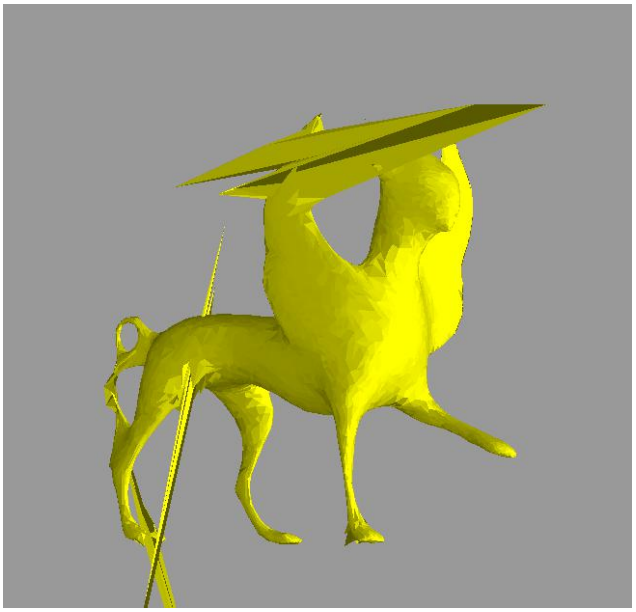


Figure 6. MCF scheme's limitation with initial weight (iterating 30 times, dt =0.1)

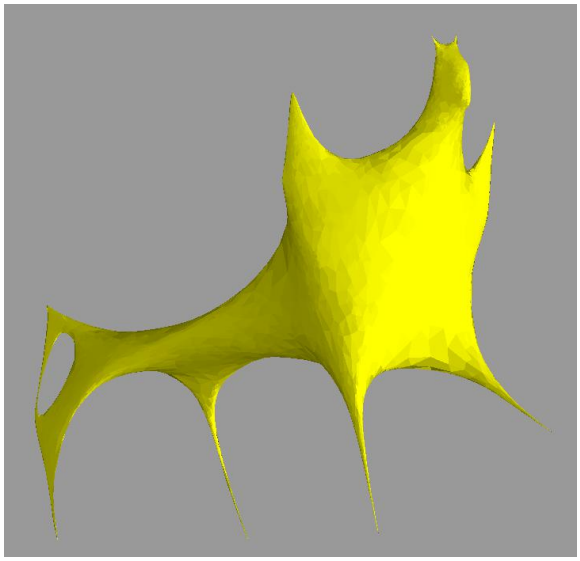


Figure 7. MVC scheme's limitation with initial weight (iterating 200 times, dt =0.9)

a. Compare these schemes and discuss their strengths and weakness. Which one would you use in the future, and why?

Based on the result of **Table 1-4** and iterating result of **Figure 8-11**, At the first, I want to talk about the strengths and weakness of each schemes.

For the **Uniform** scheme, I think this is the safest for doing smoothing. It always set the weight as 1/N(j), which can be seen as a constant. So it can withstand very many iterations. However, if we compare **Figure 8** and **Figure 11**, it is not hard to find that, this kind of smoothing scheme can not maintain the original features of the shape, that is to say, "generic but not specialized".

For the **Cord** scheme, it is a "compromise" method that guarantees the number of iterations but does not guarantee the original graphic features. Even worse, it will damages the shape of the original model and creates unnecessary noise, as **Figure 9** shown below. Guaranteeing the number of iterations does not mean it can iterate as many as **Uniform** scheme. Because the weight format is a reverse number, the scheme is easy to create the singular points. However, if we want to add some characters that the original shape don' t have (like an art creation) while smoothing, this method is a good idea.

For the **Mean Curvature Flow** scheme, as we discussed for the **Table 3** before (The four Limitations), this method is greatly influenced by time step, systematic error, number of iterations and model curvature. So it is a "sensitive" method, which means it can only do smoothing in a really little range of conditions. But once the method stay in the safe region, it can provide a good smoothing result (in **Figure 10**) like the **Mean Values Coordinates** scheme.

This means the scheme can keep good character of original shape.

For the **Mean Values Coordinates** scheme, it is the method that I will use in the future. Firstly, it satisfies the high Smoothing Quality requirement I defined at the first page of this report, which means **MVC** is a stable, high-fidelity, efficient way to preserve the original character of the image. By comparing **Figure 8** and **Figure 11**, it is not hard to understand this point. Secondly, compared with **Uniform** scheme, with time step between low and middle size, MVC can still be able to quickly approach the optimal smoothing result. However, this method still exists some weakness. The largest weakness is the time cost. It would spend more time than the **Uniform** one because every iteration have to calculate the new weight in the gradient direction.

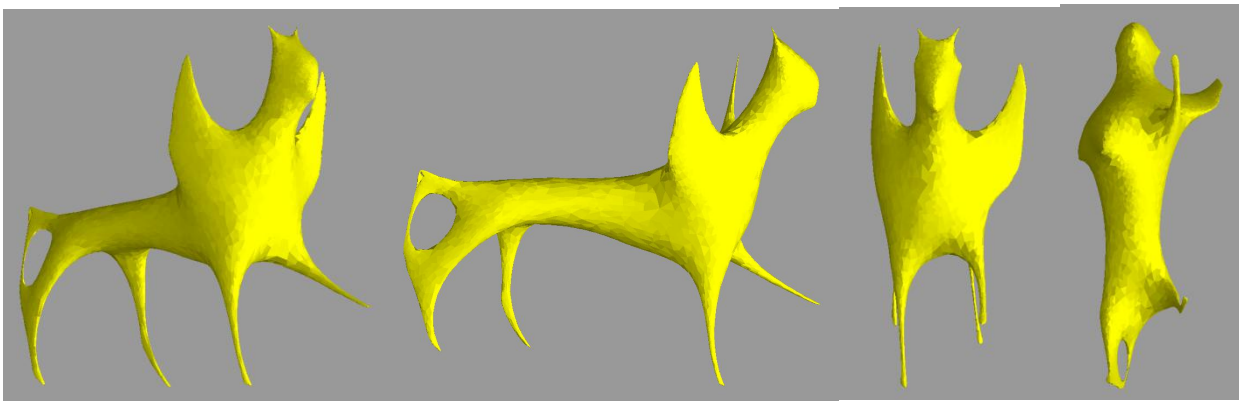


Figure 8. Uniform scheme's result
(after iterating 14*10 = 70 times, dt =0.9) (can be iterate more)

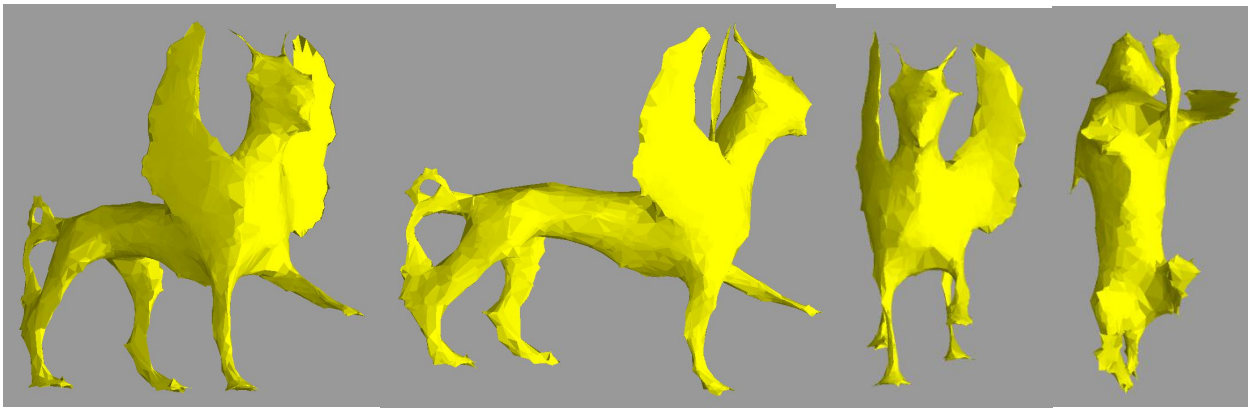


Figure 9. Cord scheme's result after iterating 10*5 = 50 times (dt = 0.9)

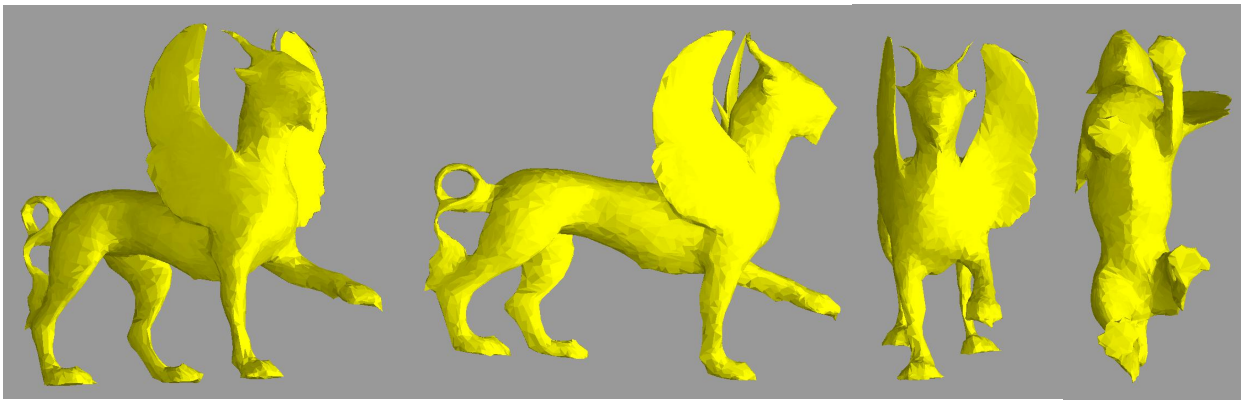


Figure 10. Mean Curvature Flow scheme's result after iterating 8*10 = 80 times (dt = 0.1)

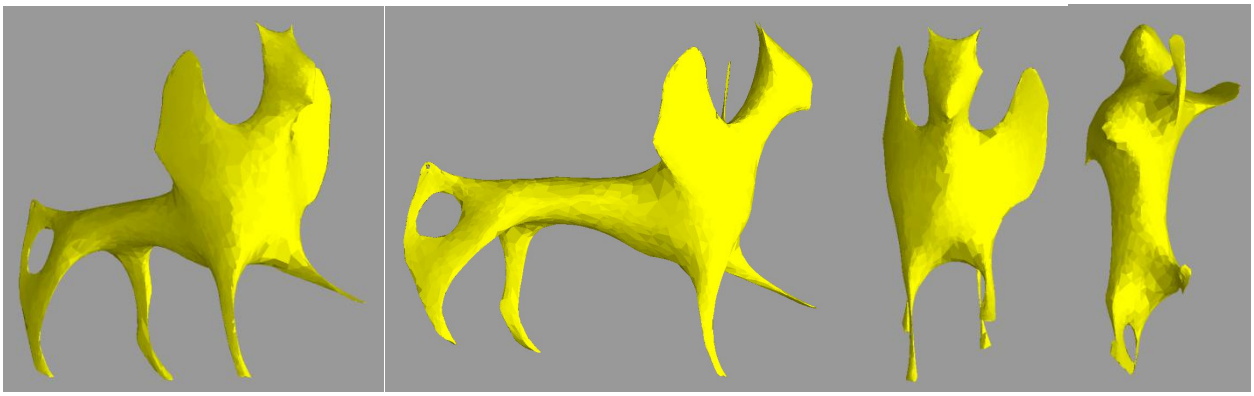


Figure 11. Mean Value Coordinates scheme's result (iterating 14*5 = 70 times, dt = 0.9)

b. Discuss the impact of smoothing on visualizing 3D checkerboard pattern from homework 1.

For the 3D checkerboard pattern, taking the Feline model as an example, I compared the four schemes when dt = 0.9 and N = 5 with the 3D checkerboard pattern mapping on original mesh, which can be seen in pictures below:

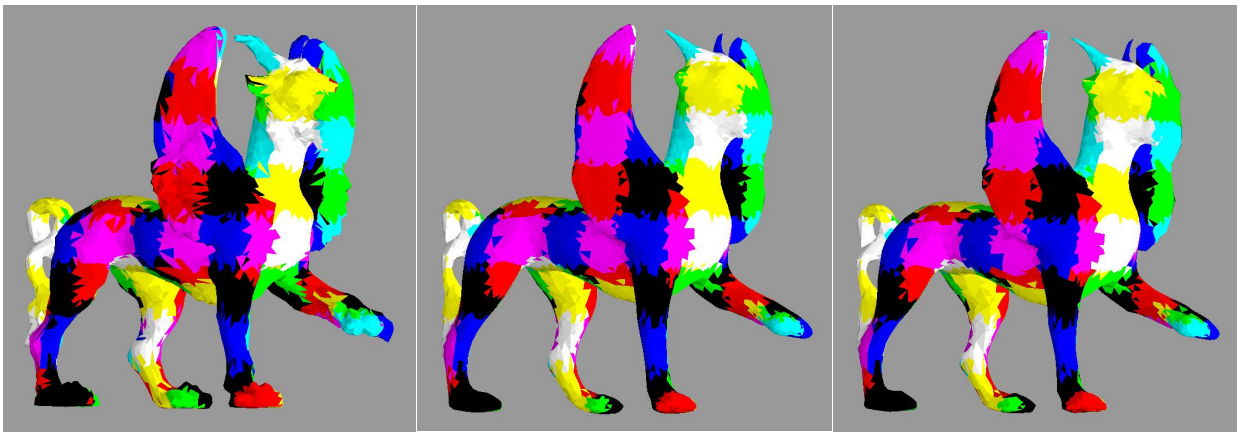


Figure 12. 3D Checkerboard under Original, Uniform & Cord schemes (From left to right) (GL_FLAT)

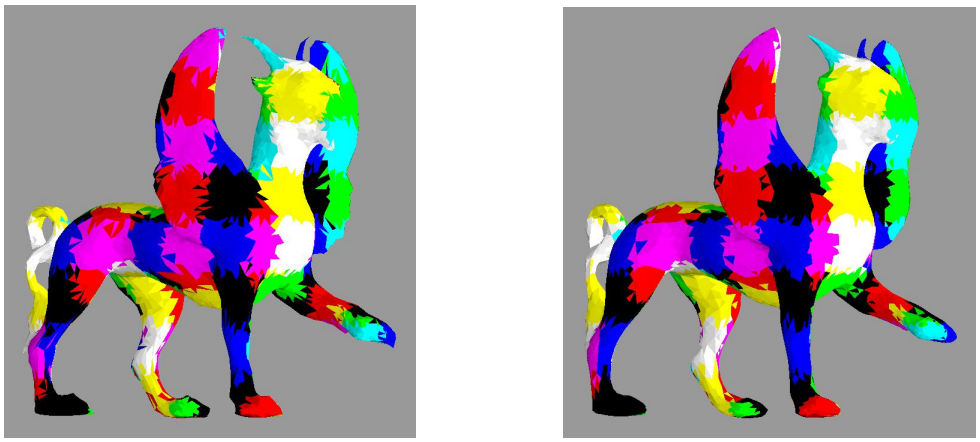


Figure 13. 3D Checkerboard under Mean Curvature & Mean Value schemes (From left to right) (GL_FLAT)

Base on the result of **Figure 12** and **13**, it is not hard to find that when doing smoothing on visualizing 3D checkerboard pattern, the **Cord** and **Mean Curvature Flow** scheme will increase the noise of the checkerboard pattern. Both of those two method are using the reverse variable (1/length and 1/tan()) as weight, so they are more easier to create some noise vertices. For the **Uniform** and **Mean Value Coordinates** schemes, they make the original pattern a little bit more acceptable because they blends some of the jagged edges into the color that belongs to it.

However, no matter which scheme has been chosen, the final visualizing of 3D checkerboard pattern will not change too much. The fundamental reason is that the original triangular mesh is not dense enough. Smoothing can only change the position of the mesh triangles, but does not increase the number of meshes. Only by increasing the number of grids can a higher quality 3D checkerboard pattern be achieved, which means Subdivision should be the most suitable patter-quality-improving way rather than smoothing.

2. (Morse design: graduate students only) Write a graphical user interface that allows the creation of a surface function h by specifying local maxima and minima vertices. You can currently select a triangle, so just picking the first vertex of that triangle should suffice. A specified maximum will always have a value one, and a minimum a value zero. Determine the values for other vertices through constrained heat diffusion. Use your favorite weighting scheme for determining the coefficients in the linear system.

For the user interface, my user can choose the maximum and minimum value by using their mouse clicking as **Figure 14** showing. Also, I give the keyboard instructions as shown below:

Keyboard guidance:
1=> Original shape with yellow color
2=> Regular subdivision
3=> Irregular subdivision
4=> Smooth
5=> Heat diffusion
6=> Original shape with white color
7=> Shape with 3D checkerboard color
or 3D checkerboard coordinates texture
s=> Swap smooth and flat
t=> Triangle mesh

My favorite weighting scheme for determining the coefficients in the linear system is **Mean Value Coordinates**. Thus, all the experiment below are based on the **MVC** scheme. Most of the time, I would set the time step **dt** as 0.9.

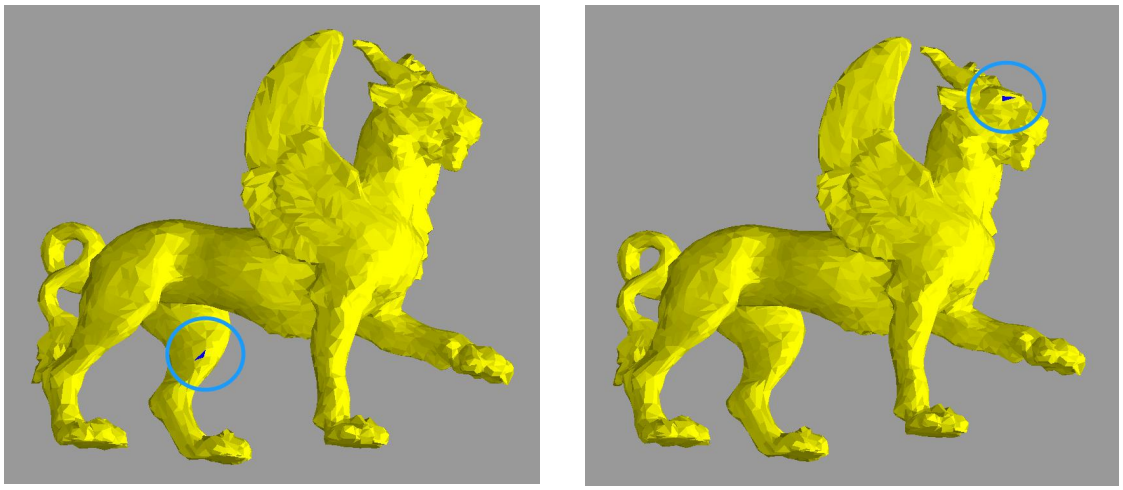


Figure 14. UI mouse clicking for choosing Max/Min point

a. Display the function. Note the function is defined at the vertices only, and will be between one and zero, everywhere.

After we chosen the maximum and minimum points by using our mouse middle button clicking. Then the system will catch the corresponding triangle, and give the triangle index to Polygon member "seed". For the two triangles, we use the first vertex in the "vlist". Those two vertices will be given constant number 0 and 1. Other points will be given initial temperature number 0. Then push button "5", final result can be seen in **Figure 15**.

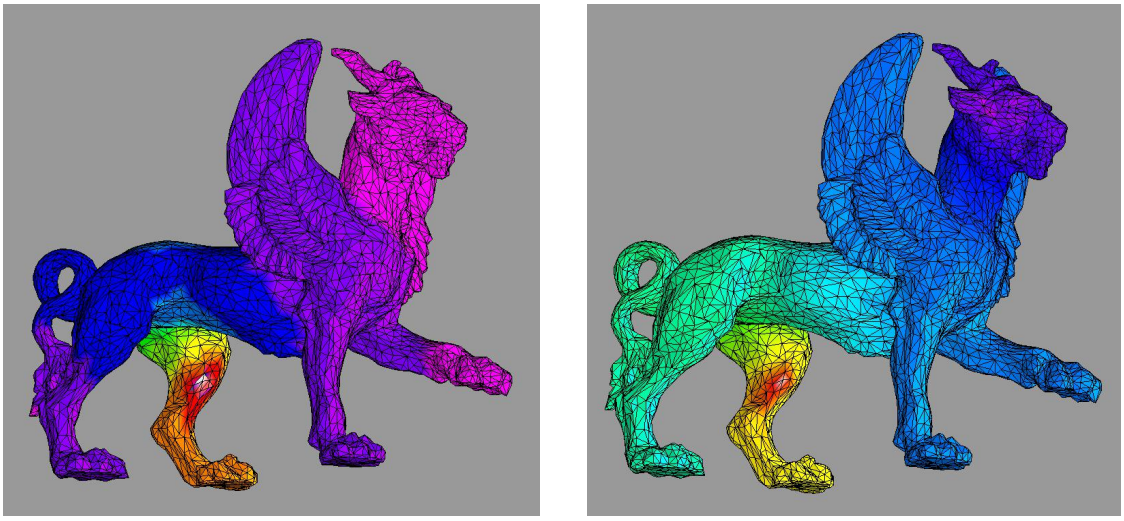


Figure 15. Heat diffusion visualization
(Discontinuous color palette left (set by myself),
Continuous color palette right ("rainbow" method))

To realize the heat diffusion visualization, I have used 3 Update Schemes. They are: **Explicit**, **Implicit** and **Gauss-Seidel**:

The simplest method to realize is **Gauss-Seidel** scheme, which has the formula shows below:

$$T^{m}(v_i) = T^{m-1}(v_i) + \lambda \Delta t \sum_{j \in N(i)} w_{ij}(T^{current}(v_j) - T^{m-1}(v_i))$$

For calculating the new value of each vertex (traverse all the vertices in Polygon), we can directly cite the current value from the vlist[i]->neb[j] by using **MVC**. Then update the value of the vertex. The sudo code has been shown below:

```
for (int i = 0; i < poly->nverts; i++)
    {
        if (poly->vlist[i] != maxima & minima)
        {
        for (int j = 0; j < n; j++)
            {
                p_sum += wij * (poly->vlist[j]->heatvalue - poly->vlist[i]->heatvalue);
            }
        }
        poly->vlist[i]->heatvalue = poly->vlist[i]->heatvalue + dt * p_sum;
    }
```

The second simplest method to realize is **Explicit** scheme, which has the formula shows below:

$$T^m(v_i) = T^{m-1}(v_i) + \lambda \Delta t \sum_{j \in N(i)} w_{ij}(T^{m-1}(v_j) - T^{m-1}(v_i))$$

For calculating the new value of each vertex (traverse all the vertices in Polygon), we need to cite the value from the vlist[i]->neb[j] and save it in a storage of the vlist[i]. Once we finished save all the data need to be updated in the storage, update the value of all the vertices in vlist. The sudo code has been shown below:

```
for (int i = 0; i < poly->nverts; i++)
    {
        if (poly->vlist[i] != maxima & minima)
        {
        for (int j = 0; j < n; j++)
            {
                p_sum += wij * (poly->vlist[j]->heatold - poly->vlist[i]->heatold);
            }
        }
        poly->vlist[i]->heatvalue = poly->vlist[i]->heatold + dt * p_sum;
    }
```

The hardest method to realize is **Implicit** scheme, which has the formula shows below:

$$T^m(v_i) = T^{m-1}(v_i) + \lambda \Delta t \sum_{j \in N(i)} w_{ij}(T^m(v_j) - T^m(v_i))$$

For calculating the new value of each vertex, we can not directly traverse all the vertices, because each vertex's update need the updated data. Thus, here we use Matrix to help us solve this problem. Firstly, we can change the formula in the clear expression like below:

$$
\begin{aligned}
& v^m{}_i = v^{m-1}{}_i + dt \sum_{j \in N(i)} w_{ij} (v^m{}_j - v^m{}_i) \\
\Rightarrow\ & v^m{}_i - dt \sum_{j \in N(i)} w_{ij} (v^m{}_j - v^m{}_i) = v^{m-1}{}_i \\
\Rightarrow\ & (1+dt) v^m{}_i - dt \sum_{\substack{j \in N(i) \\ v_j \notin \max \\ v_j \notin \min}} w_{ij} v^m{}_j = v^{m-1}{}_i + dt \sum_{\substack{j \in N(i) \\ v_j \in \max \\ v_j \in \min}} w_{ij} v^m{}_j \\
\Rightarrow\ & (1+dt) v^m{}_i - dt \sum_{\substack{j \in N(i) \\ v_j \notin \max \\ v_j \notin \min}} w_{ij} v^m{}_j = v^{m-1}{}_i + dt \sum_{\substack{j \in N(i) \\ v_j \in \max \\ v_j \in \min}} w_{ij} v^m{}_j \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\text{if } v_j == \max \text{ or } \min, v^m{}_j = 1 \text{ or } v^m{}_j = 0)
\end{aligned}
$$

Then, we can separate the formula in the format of matrix:

$$
\begin{pmatrix}
1+dt & \dots & \dots & -dt \cdot w_{0j} & \dots & \dots \\
\dots & 1+dt & \dots & \dots & \dots & \dots \\
-dt \cdot w_{i0} & \dots & \dots & \dots & \dots & \dots \\
\dots & \dots & \dots & \dots & \dots & \dots \\
\dots & \dots & -dt \cdot w_{nj} & \dots & \dots & 1+dt
\end{pmatrix}
\begin{pmatrix} v^m{}_1 \\ v^m{}_2 \\ \dots \\ v^m{}_{n-1} \\ v^m{}_n \end{pmatrix}
=
\begin{pmatrix} v^{m-1}{}_1 \\ v^{m-1}{}_2 \\ \dots \\ v^{m-1}{}_{n-1} \\ v^{m-1}{}_n \end{pmatrix}
+
\begin{pmatrix} dt \cdot w_{0 j_{\max}} \\ dt \cdot w_{2 j_{\max}} \\ \dots \\ dt \cdot w_{n-1 j_{\max}} \\ dt \cdot w_{n j_{\max}} \end{pmatrix}
$$

If we set the time step as the length of the total iteration time, then we can get the matrix iteration like the equations shows below. Theoretically, this kind of approximation can be accepted because of the large step stability of implicit method:

$$
\begin{pmatrix}
1+t_n & \dots & \dots & -t_n \cdot w_{0j} & \dots & \dots \\
\dots & 1+t_n & \dots & \dots & \dots & \dots \\
-t_n \cdot w_{i0} & \dots & \dots & \dots & \dots & \dots \\
\dots & \dots & \dots & \dots & \dots & \dots \\
\dots & \dots & -t_n \cdot w_{nj} & \dots & \dots & 1+t_n
\end{pmatrix}
\begin{pmatrix} v^m{}_1 \\ v^m{}_2 \\ \dots \\ v^m{}_{n-1} \\ v^m{}_n \end{pmatrix}
=
\begin{pmatrix} v^0{}_1 \\ v^0{}_2 \\ \dots \\ v^0{}_{n-1} \\ v^0{}_n \end{pmatrix}
+
\begin{pmatrix} t_n \cdot w_{0 j_{\max}} \\ t_n \cdot w_{2 j_{\max}} \\ \dots \\ t_n \cdot w_{n-1 j_{\max}} \\ t_n \cdot w_{n j_{\max}} \end{pmatrix}
$$

$$
\Leftrightarrow
\begin{pmatrix}
1+t_n & \dots & \dots & -t_n \cdot w_{0j} & \dots & \dots \\
\dots & 1+t_n & \dots & \dots & \dots & \dots \\
-t_n \cdot w_{i0} & \dots & \dots & \dots & \dots & \dots \\
\dots & \dots & \dots & \dots & \dots & \dots \\
\dots & \dots & -t_n \cdot w_{nj} & \dots & \dots & 1+t_n
\end{pmatrix}
\begin{pmatrix} v^m{}_1 \\ v^m{}_2 \\ \dots \\ v^m{}_{n-1} \\ v^m{}_n \end{pmatrix}
=
\begin{pmatrix} \dots \\ 1 \\ \dots \\ 0 \\ \dots \end{pmatrix}
+
\begin{pmatrix} t_n \cdot w_{0 j_{\max}} \\ t_n \cdot w_{2 j_{\max}} \\ \dots \\ t_n \cdot w_{n-1 j_{\max}} \\ t_n \cdot w_{n j_{\max}} \end{pmatrix}
$$

Then we can construct the matrix A, X, boundary matrix, and initial matrix for the equation system, which should be looks like:

$$A \cdot X = I + b$$

The sudo code can be shown as below:

```
Push back Matrix A
Construct Inital Vector I
for (int i = 0; i < poly->nverts; i++)
    {
        if (poly->vlist[i] != maxima & minima)
        {
        for (int j = 0; j < n; j++)
            {
                p_sum += wij * (poly->vlist[j]->heatold - poly->vlist[i]->heatold);
                ...
                Set the data of coefficient triple of Matrix A (push_back())
                Set boundary variable's data to the right side b
                ...
            }
        }
    }
X = Solver A(I+b);
for (int i = 0; i < poly->nverts; i++)
    poly->vlist[i]->heatvalue = x[i];
```

For testing those three scheme, we tried 10,000 times iteration, by using the Feline model, the result has been shown in **Figure 16**, **17**, and **18**. From the result, we can safely get two conclusions below:

(1) In terms of computational speed, matrices are much faster than simple nested code loops. That is the reason, why the **Implicit** method is approximate 60 times faster than **Explicit** and **Gauss-Seidel** scheme.

(2) Take the calculation efficiency as an example, to measure the efficiency of each method, we can observe the temperature diffusion results after 10,000 iterations. Obviously, **Gauss-Seidel** scheme is the most efficient diffusion method. The second efficient one would be the **Explicit**. The lowest efficient scheme is the **Implicit** method. Actually the reason should be simple, the diffusion efficiency depends on the vertex value updating period. For **Gauss-Seidel** scheme, it grab the current value that vertex have once it need some data. We can treat it as "immediately". For **Explicit** scheme, the vertex need to be updated should wait until the later iteration has been done. We can treat it as "wait a second". For **Implicit** scheme, the vertex need to be updated should wait until the whole iterations have been done. We can describe it as "hold on".

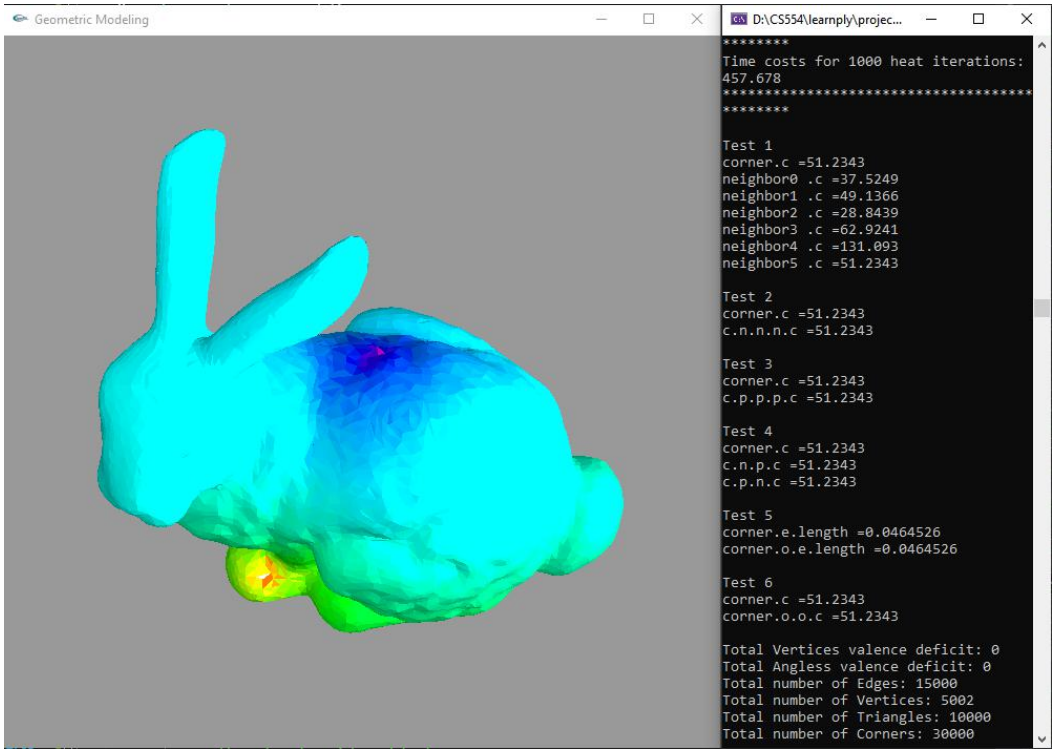


Figure 16. Gauss-Seidel result with 10000 iterations (457.678s) (GL_FLAT)

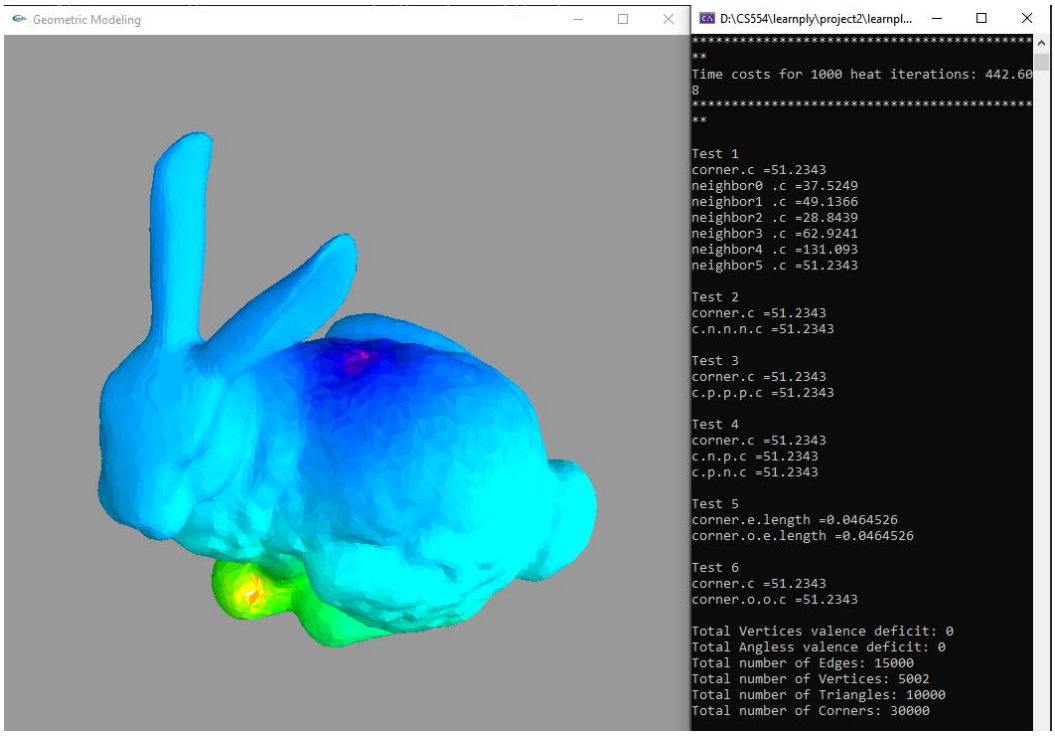


Figure 17. Explicit result with 10000 iterations (442.608s) (GL_FLAT)

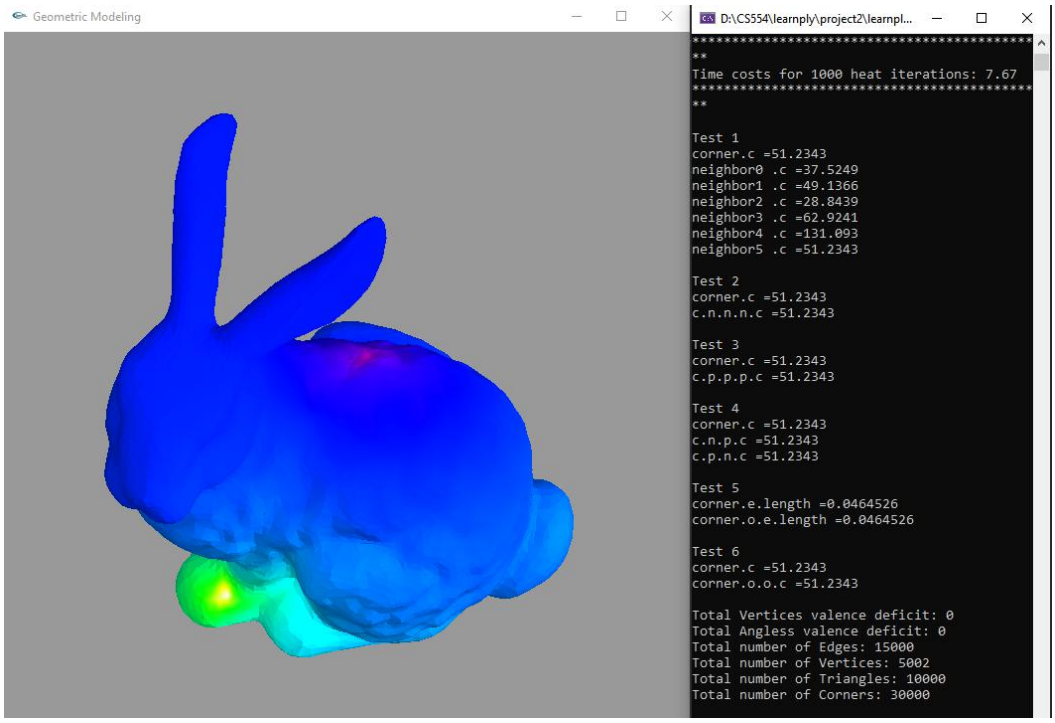


Figure 18. Implicit result with 10000 iterations (7.67s) (GL_SMOOTH)

b. Identify local maxima, local minima, and saddles using the following description, and comment on where they tend to happen, i.e., how are they related to the local features in the shape? A local maximum is a vertex whose value is larger than any of its neighbors, while a minimum has a smallest value in the neighborhood. For a saddle vertex v_0 , denote its neighboring vertices in a counterclockwise pattern $\{v_1, v_2, ..., v_n\}$. We assign a symbol '+' to vi if $h(v_i) > h(v_0)$. Otherwise, we assign a symbol '-'. Then the sequence of v_i 's turns into a series of pluses and minuses. Define $P(v_0)$ as the number of times in the sequence when there is a change from a plus to a minus sign. If $P(v_0) = 2$, this is a regular saddle point, also called 1-saddle. When $P(v_0) = 3$, the saddle is a monkey saddle or 2-saddle. In general, an N-saddle satisfies $P(v_0) = N + 1$. Compute M = # max + # min - # saddle. For an Nsaddle, count its multiplicity as N-1. For example, for each monkey saddle, you need to subtract two instead of one when computing M. Compare M to the Euler characteristic of the surface. What do you find? Provide an intuition behind your finding.

Firstly, I want to answer the question about how the local maxima, local minima, and saddles are related to the local features in the shape.

Local maxima is the value of the vertex is larger than value of all its distance-1 neighbor. Local minima is the value of the vertex is larger than value of all its distance-1 neighbor. So if

we draw the 3D shape by using the height to represent the value of all vertices, we can get the result in **Figure 19**. Local maxima would be the "top of the hill" and local minima would be the "bottom of the basin".

According to the definition form the question notes, the saddle points should looks like the shape in **Figure 20**. The figures tells us the saddle point happens only if it is surrounded by 2+i higher-height points and 2+i lower-height points ($i \geq 0$) and the higher and lower points are staggered.

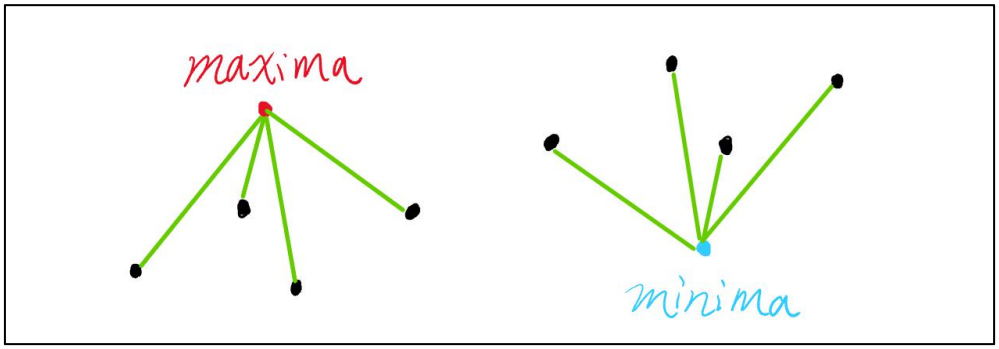


Figure 19. Visible height of each vertices value (Maxima left, Minima right)

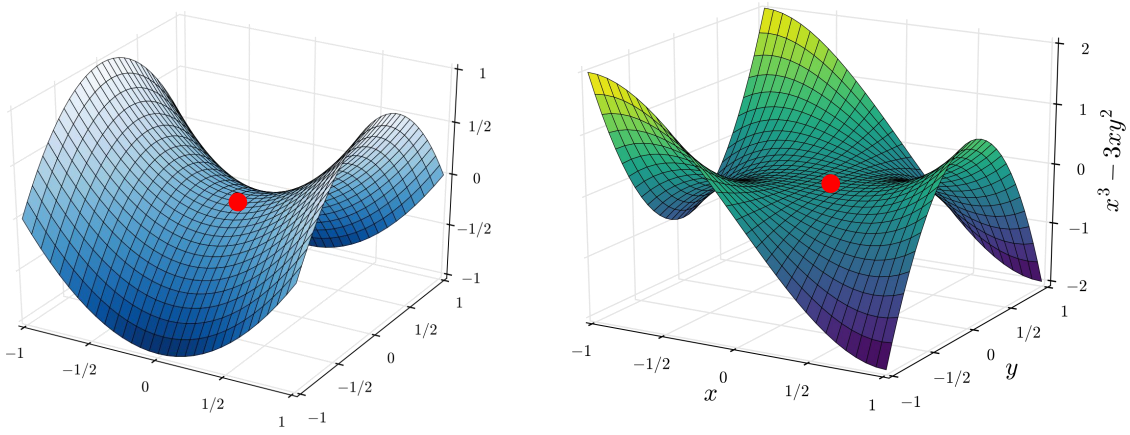


Figure 20. Saddle points
(Regular saddle point left, Monkey saddle point right)
(source from: https://en.wikipedia.org/wiki/Saddle_point)

Then, let's talk about the relationship between **M = # max + # min - # saddle** and **Euler characteristic**. Set maxima point and minima point by using mouse clicking. After a lot of iterations for letting the heat spreads out every place of the surface, the value of the surface will tend to be unchanged. At this point, the data of **M** is stable, which can be treated as the final

result. For the experiment, we choose the best pair of maxima and minima points on the surface for temperature diffusion. **Table 5** shows the final result.

Model	V-E+F	Result	Max+Min -Saddle:	Stable Iterate Times
bunny	5002-15000+10000	2	**2**	**5000 (Implicit Matrix)**
dodecahedron	20-?+12	/	/	/
dragon	10000-30000+20000	0	**0**	**60000 (Implicit Matrix)**
feline	4998-15000+10000	-2	**-2**	**90000 (Implicit Matrix)**
happy	9990-30000+20000	-10	**-10**	**1000000 (Implicit Matrix)**
hexahedron	8-?+6	/	/	/
icosahedron	12-30+20	2	**2**	**40 (Gauss-Seidel Loop)**
octahedron	6-12+8	2	**2**	**20 (Gauss-Seidel Loop)**
sphere	4098-12288+8192	2	**2**	**3000**
tatrahedron	4-6+4	2	**2**	**7 (Gauss-Seidel Loop)**
torus	4608-13824+9216	0	**0**	**3000**

Table 5. Final M value with it corresponding iterate times

The conclusion is that **M = # max + # min - # saddle** would be the same as the property value of **Euler characteristic**. Actually, as far as I think, the value of **M** should be the same as **E** character number. The reason can be seen below:

(1) For the shape with no tunnel, finally, for each vertices except maxima point and minima points, they will always have a neighbor higher than it and a neighbor lower than it. So there is no points should be saddle points. If we see the definition of maxima and minima points, they should also not be the saddle points. So **# saddle** equals to 0. Thus, **# max + # min** should only be two if we only do two mouse clicks. In this case **M = # max + # min - # saddle = 1+1-0 =2**.

(2) For the shape with more than one tunnel, we can think the case with one tunnel then add δ value for add tunnels. At the stable time, local maxima and minima points would still only be the two we set by using mouse. So **# max + # min = 2**. For the saddle point, for each tunnel, we can treat the shape as a torus shape. As **Figure 21** showing, if we connect maxima and minima points with a blue line, there are only two saddle points of the shape, which has been marked as the black point. Those two points are all Regular saddle point. Thus, for a tunnel we count two numbers of saddle. In this case **M = # max + # min - # saddle = 1+1- 2*Number of tunnels**.

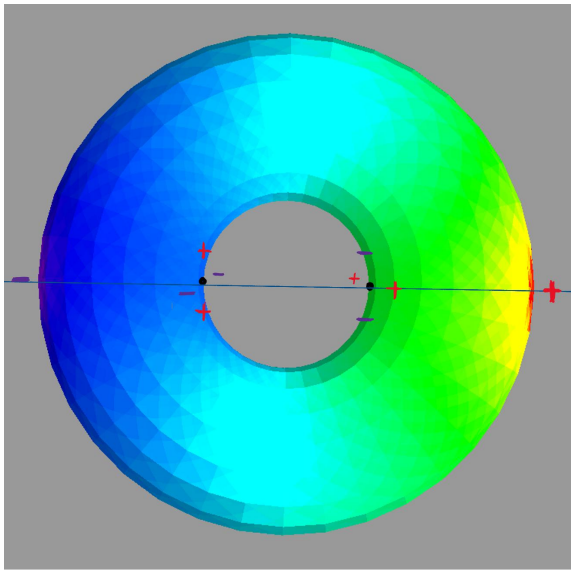


Figure 21. Get the saddle points in a torus

c. Use the design tool to construct two vertex-based functions (F, G). You should try to construct the functions so that their levelsets are nearly perpendicular to each other. Next, take a periodic example texture and apply the texture to the surface by treating (F, G) as the texture coordinates. Are you satisfied with the synthesis results? Propose and perhaps implement additional steps that may lead to higher quality synthesis.

For constructing the two vertex-based functions (F, G), my algorithm is try to construct a two heat diffusion levelsets system. Setting each pair of maxima and minima vertices on a line and making sure those two lines are parpenticular to each other is the basic idea. The idea can be shown in **Figure 22**. The green line shows the texture coordinates on model's surface.

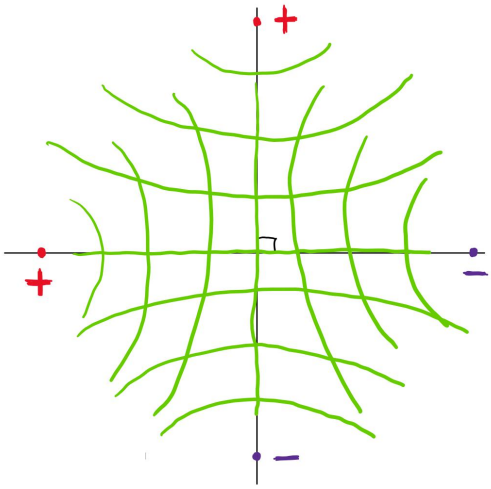


Figure 22. The schematic diagram of heat levelset system

Take the Sphere model as an example, the implementation can be seen in **Figure 23** and **24**. At this time, we do not do the periodic texture, but just for testing the sample.

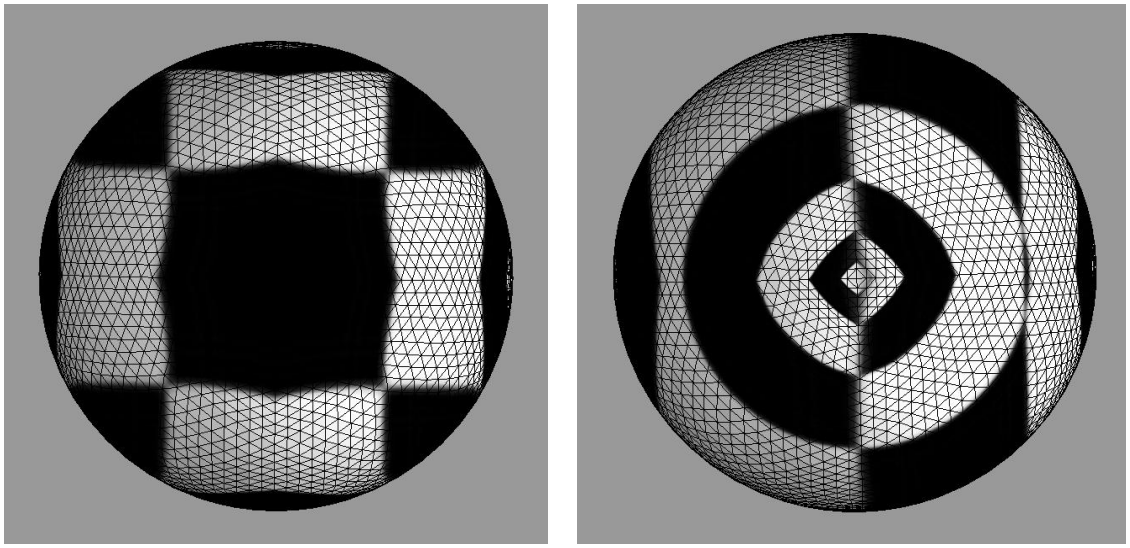


Figure 23. The perpendicular texture system texturing on regular shape (without periodic)

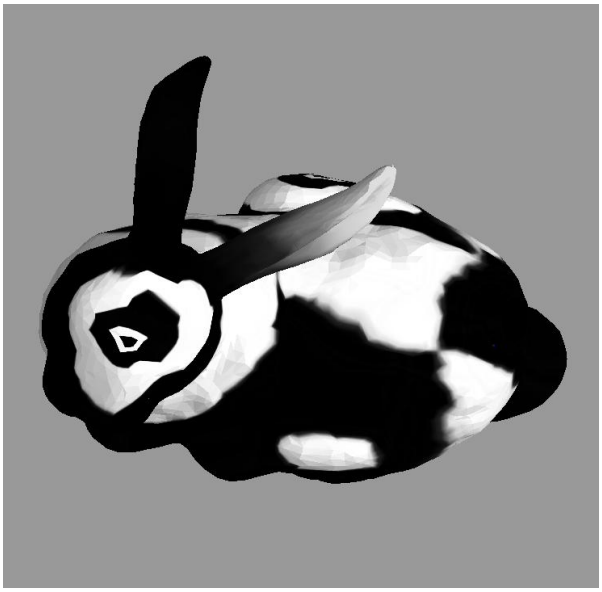


Figure 23. The perpendicular texture system texturing on irregular shape (without periodic)

Then, after set the perpendicular texture coordinates, let's do the periodic texture, here if we set the heat value 20 times larger than before, the texture will have 19 more periodic. The example can be seen in **Figure 24**.

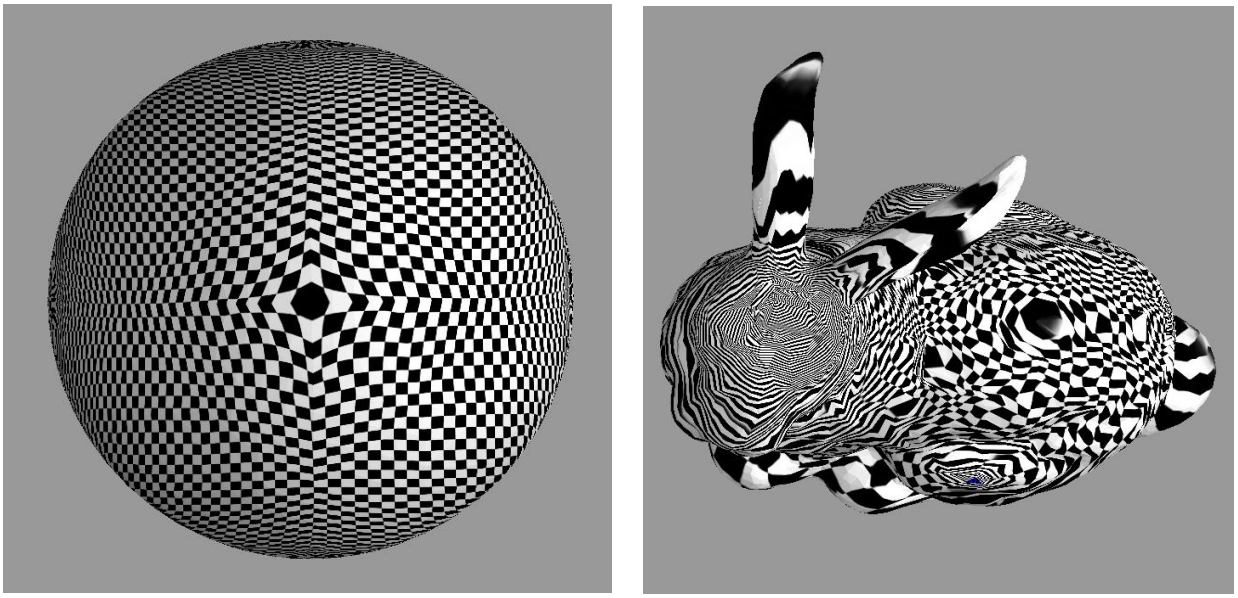


Figure 24. The perpendicular texture system texturing
(Left Regular shape, Right Irregular shape)
(without periodic)

However, I am not satisfied with the synthesis results. In my mind, if the texture picture is a square, a perfect texture on the surface should be square by square. For regular shape, it is okay, as shown at the left side of **Figure 24**. However, for the irregular shape, the texture result is really bad (as the right one of **Figure 24**), some place they are "approximate square" but some place they are definitely not the square texture. The reason is because: when the 2D texture coordinates is square grids, but when mapping it on the 3D space, the 2D coordinates grid will distort corresponding how curved the plane is, as **Figure 25** shows. Then even we look the shape in the normal direction, the texture picture will still be distorted.

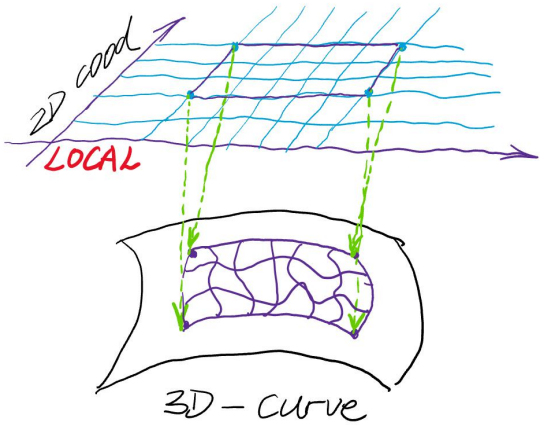


Figure 25. **Reason of texture distortion.**

Thus, I tend to design a global 2D coordinates, which is a projection of some direction of 3D axis. Here, “global” means the coordinates’ value will not be affected by the 3D surface. The idea can shown in **Figure 26**.

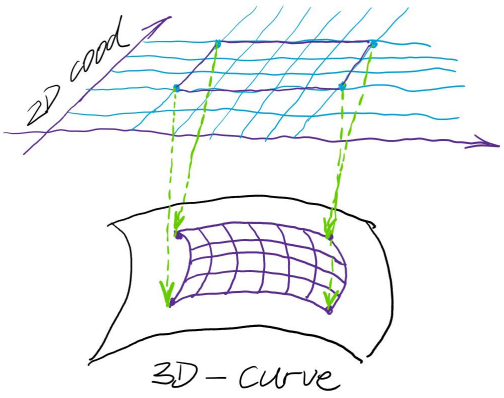


Figure 26. Idea to implement higher quality synthesis.

In order to get a perfect 3D texturing on the surface, the 2D coordinates should be mapped 3 times, which means in x-axis, y-axis, and z-axis direction. Luckily, the 3D check-board’s RGB can satisfy the requirement. Here, I do transformation between r, g, b to x, y, z, they can be 1:1 changed for non-periodic texturing as **Figure 27** and **28** shows:

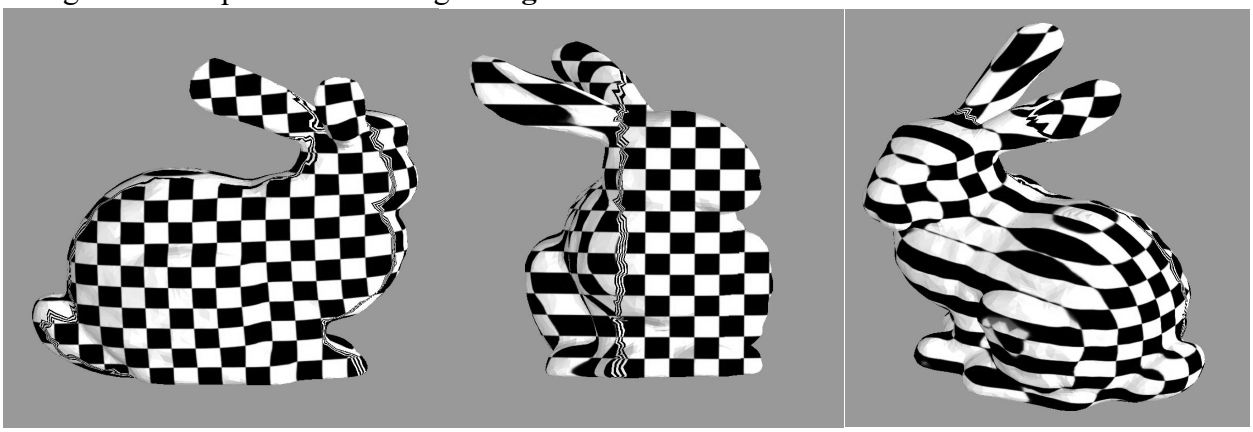


Figure 27. Non-periodic texturing (L = 1).

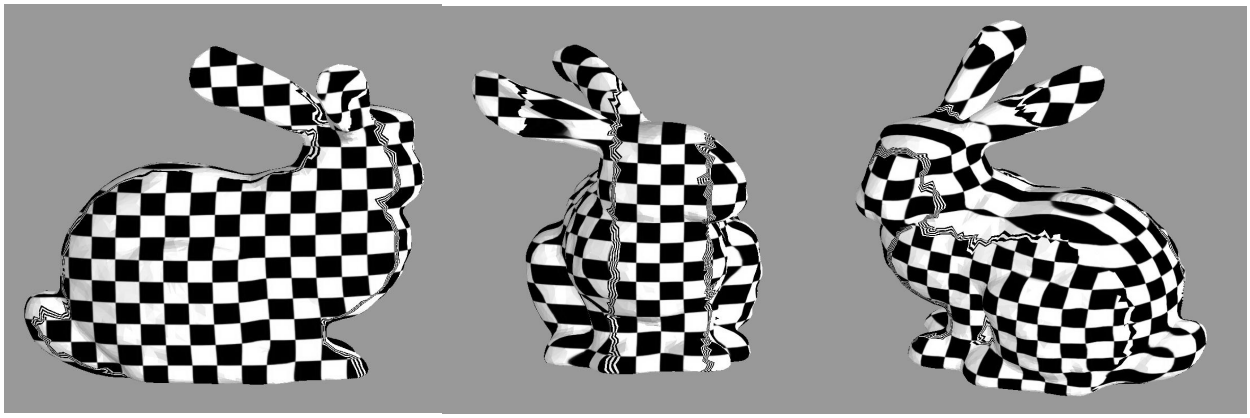


Figure 28. Non-periodic texturing (L = 0.5).

Now, let's do some periodic textures! Please enjoy **Figure 29** and **30**!

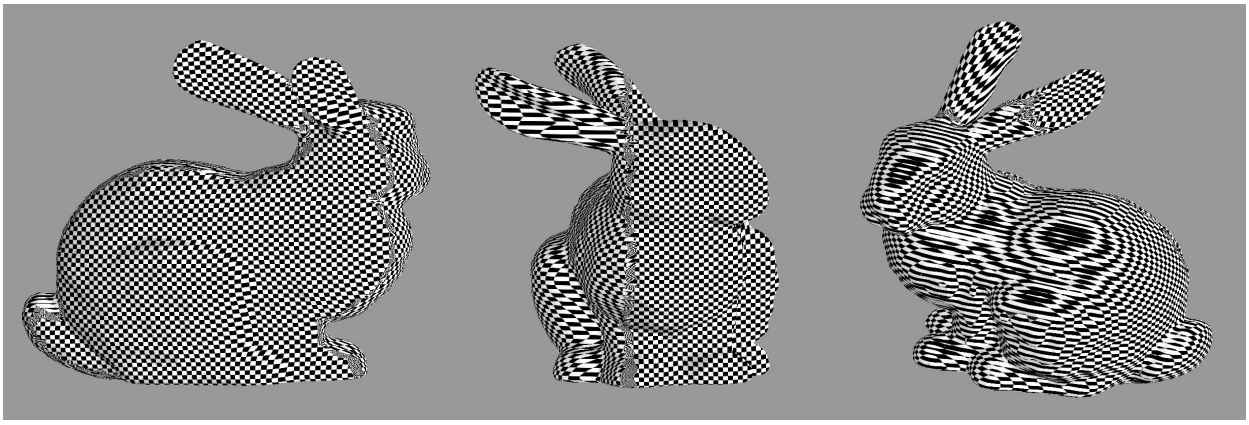


Figure 29. **Periodic texturing (L = 1, loop = 5).**

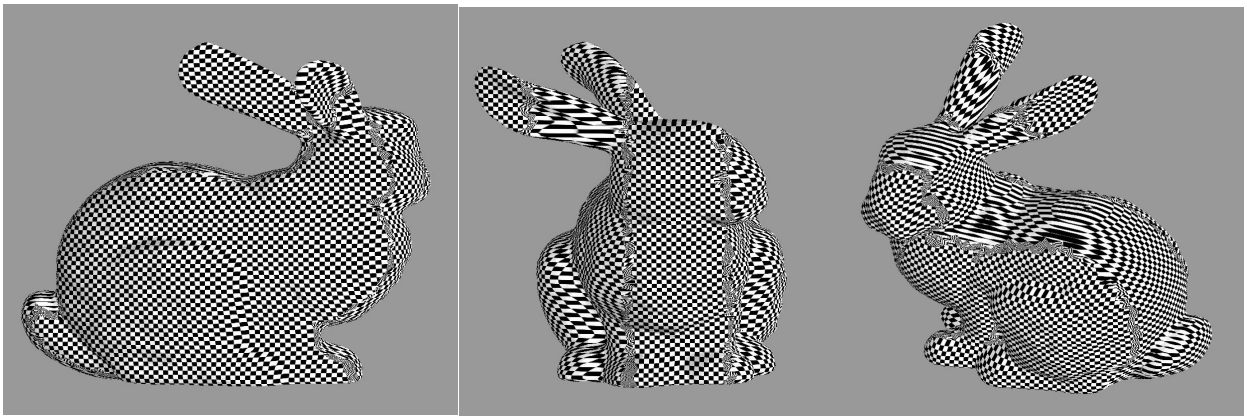


Figure 29. **Periodic texturing (L = 0.5, loop = 5).**

Finally, let's try some texture with no periodically joined tiles at the edges. Please enjoy **Figure 30** to **34**.

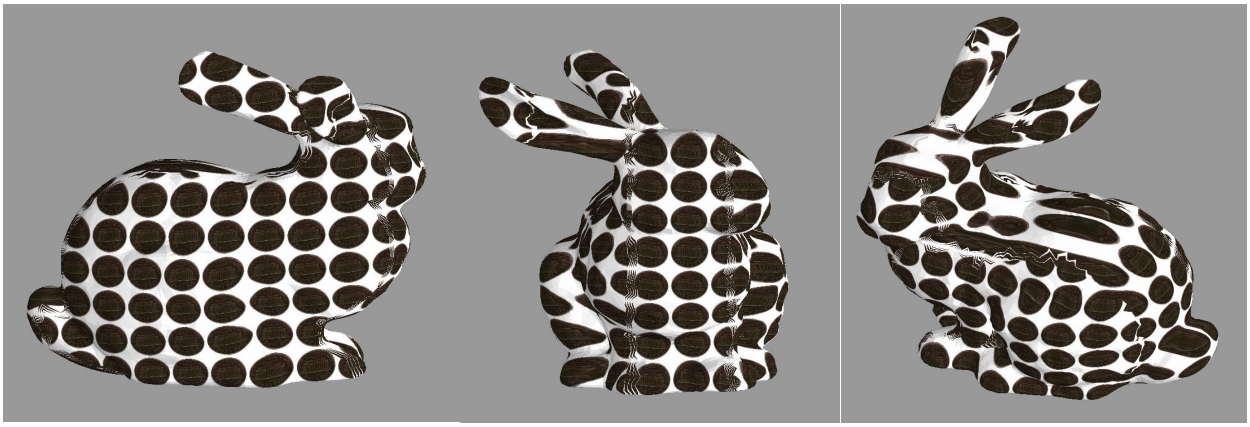


Figure 30. **Periodic texturing with coins (L = 0.5, loop = 5).**

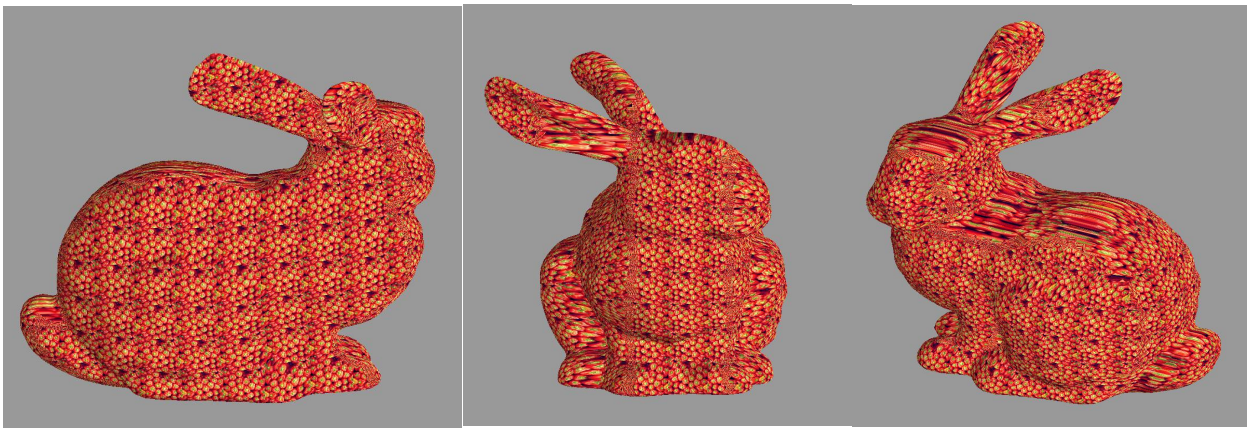


Figure 31. Periodic texturing with red pepper (L = 0.5, loop = 5).

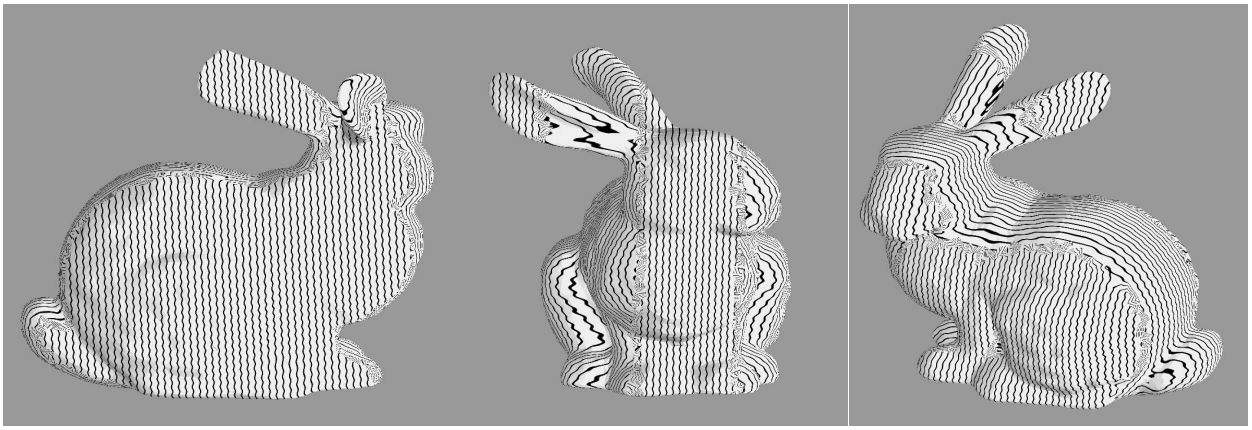


Figure 32. Periodic texturing with wiggle-grating (L = 0.5, loop = 5).

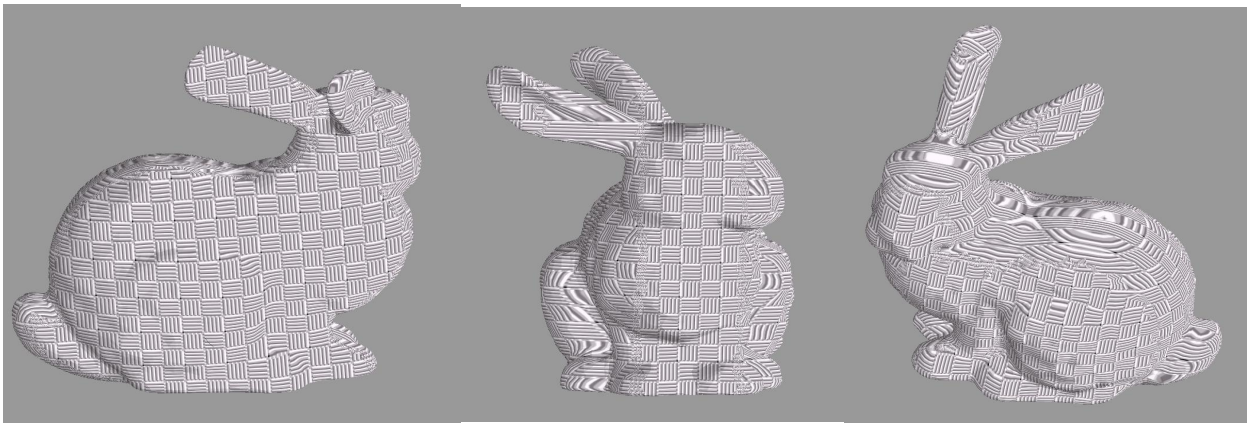


Figure 33. Periodic texturing with metal (L = 0.5, loop = 5).

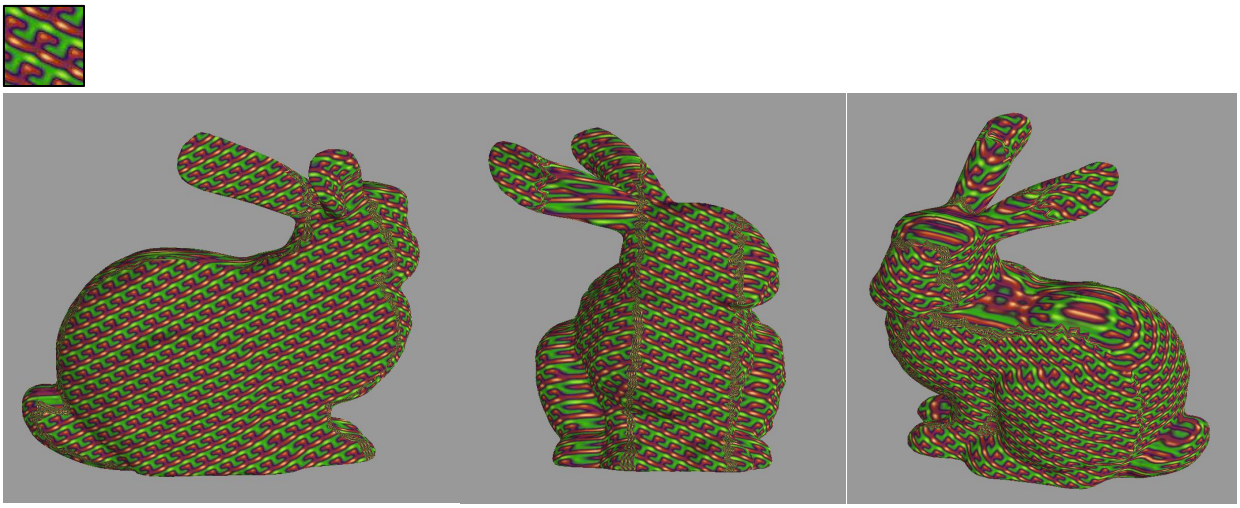


Figure 34. **Periodic texturing with slop line (L = 0.5, loop = 5).**

Some explanation fot the weird stretching of the top of bunny:

As we can see from the pictures above, the best location to observe the bunny is in the direction of x axis and z axis. However, you cannot see the correct texture in y axis direction. This is the limitation of the 3D shape's degree of freedom. For texture mapping, we can only choose **s** and **t** for corresponding mapping, but 3D space has 3 parameters **x**, **y** and **z** for each vertex. Thus texture space with 2 degree of freedom can not share to 3D space with 3 degree of freedom, which means there will also a degree cannot get the texture matching. Here I set **s** and **t** for **x** and **z** (see **Figure 35**), so if you look the texture in y axis direction, it will be stretched.

```
//-------------------------------------Tianle TEXTURE-------------------------------------
if ((xf % 2) != 0)
{
    glTexCoord2f(temp_v->y* loops, temp_v->z * loops);
    if ((zf % 2) != 0) glTexCoord2f(temp_v->y * G * loops, temp_v->z * loops);
    if ((yf % 2) != 0) glTexCoord2f(temp_v->y * loops, temp_v->z * B * loops);
    if ((yf % 2) != 0&& (zf % 2) != 0) glTexCoord2f(temp_v->y * loops, temp_v->z * loops);
}
else
{
    glTexCoord2f(temp_v->x * loops, temp_v->z * loops);
    if ((zf % 2) == 0 && (yf % 2) != 0) glTexCoord2f(temp_v->x * R * loops, temp_v->z * loops);
    if ((yf % 2) == 0 && (zf % 2) != 0) glTexCoord2f(temp_v->x * R * loops, temp_v->z * loops);
    if ((yf % 2) == 0 && (zf % 2) == 0) glTexCoord2f(temp_v->x * loops, temp_v->z * loops);
}
//----------------------------------------------------------------------------------------
```

Figure 35. **Degree of freedom is distributed to x and z**

Note: This idea comes from Tianle Yuan. Any quotation please identify this source.