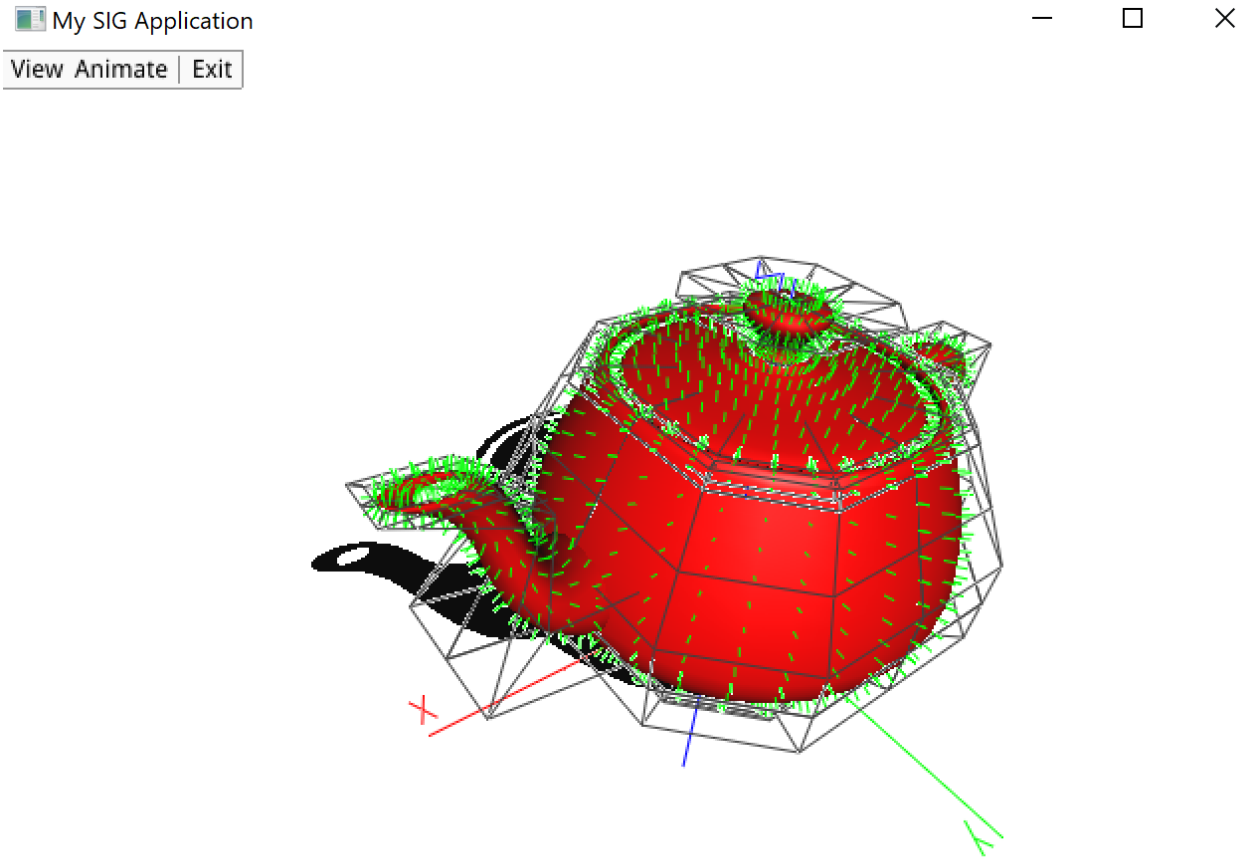
EECS 267 Lab 02 – Parametric Surfaces



Main Points

Create Patch

The code can be found under the function:

```
GsVec createPatch(vector<GsVec> controlPts, float u, float v)
```

To create a beizer patch, we just apply the regular Bezier curve in a double for loop, with 2 incrementors, one for u and one for v. The approach used in this lab was to use the 16 control points to create another curve that has 4 control points along the v direction by using the u parameter. Then we can evaluate this new curve with the v parameter to find the correct position of this u,v, point on this patch. Repeat this for all possible combinations of u,v, where each parameter ranges from [0-1]. (Incrementor = 1/resolution). This loop process is found in buildTeapot() (See below).

Calculating the Normals

The code can be found under the function:

```
GsVec findNormal(vector<GsVec> controlPts, float u, float v)
```

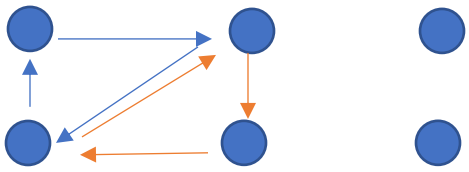
To calculate the normals of each point on the Bezier surface, we first need to compute the Bezier patches with the derivate version of the Bezier uCurve and the Bezier vCurve. (To find the derivate, just apply the chain rule). Once we find the derivate version of the Beizer uCurve and vCurve using the 16 control points and parameters (u,v), we need to perform a cross product. (duCurve cross dvCurve). This cross product will tell us the normal (direction) of the corresponding (u,v) point on the surface. To make sure it's a unit vector, we can normalize this vector.

Building the Teapot

The code can be found under the function:

```
void buildTeapot (int resolution)
```

This function creates/builds the teapot model. It first loads in the correct control points for each of the 32 patches. (16 control points loaded in, 32 times). After loading the 16 control points in, the next step is to create the patch. We calculated the u,v points with the createPatch function (See Above). Also, we found the normals of these u,v vertices with the getNormal function (See Above). The amount of points created in each patch is determined by the input resolution. (The total number of points in a patch is = resolution * resolution). The higher the resolution, the smaller the incrementor is for the Bezier curve (incrementor = 1/resolution). After finding all the vertices and normals, I saved them into a new model. Once they were loaded into the model, I created triangular faces using m->F().push.set(i, j, k). See Diagram below to see how the faces were created. The blue lines represent one triangle, and the orange lines represent another triangle (in one iteration of the for loop).



After the faces were set, I just added the teapot model to rootG. To create the cage, I simply created all 24 lines that connected the 16 control points, and then added them to rootG.

I utilized the SnLines class for the cage. To create the normal, I also utilized the SnLines class (The end points of the normal display line were: 1) The vertex point and 2) The vertex point + 0.1*normal)

Creating the Shadow

To create the shadow, I used the same matrix from the undergrad shadow PA. For this shadow matrix to work, the model must be placed on top of the plane that we want to project the shadow on. In this case, we want to project the shadow on the XY plane, since the teapot was oriented on the Z axis. I had to check if the teapot was above the XY plane, I did this by checking what the minimum z-coordinate was for the teapot. It turns out that the minimum z-coordinate was 0, so there was no translatation needed to get the teapot above the XY plane (since all z-points were greater than or equal to 0). See below for matrix details:

```
void MyViewer::buildTeapot(int resolution) { ... }

GsMat MyViewer::shadowMatrix() {
    float ground[4] = { 0,0.0, -0.5,0 };
    float light[4] = { xLight, yLight, zLight,0 };

    float dot = ground[0] * light[0] + ground[1] * light[1] + ground[2] * light[2] + ground[3] * light[3];

    GsMat shadowMat(dot - ground[0] * light[0], 0 - ground[1] * light[0], 0 - ground[2] * light[0], 0 - ground[3] * light[0],
        0 - ground[0] * light[1], dot - ground[1] * light[1], 0 - ground[2] * light[1], 0 - ground[3] * light[1],
        0 - ground[0] * light[2], 0 - ground[1] * light[2], dot - ground[2] * light[2], 0 - ground[3] * light[2],
        0 - ground[0] * light[3], 0 - ground[1] * light[3], 0 - ground[2] * light[3], dot - ground[3] * light[3]);

    return shadowMat;
}
```

Keyboard Events

Here's what each key does in this project:

n : Show Normals

c : Show Cage

s : Show shadow

e : Move shadow towards -x

r : Move shadow towards +x

o : Decrease resolution

p : Increase resolution

Evaluation

Topic

For this lab, I want to evaluate the following:

1. The creation time of the teapot with respect to it's resolution.
2. Memory allocation with respect to it's resolution.

(Note: Normals, Cage lines, and Shadows are still computed)

Purpose

The reason why I want to evaluate my project with regards to creation time, and memory allocation, is because I want to see what the optimal resolution is.

Statistics/Table

The Resolution was tested/changed via initialization in the constructor of myViewer.

(Note: I would have it change through keyboard input, but that may cause some problems such as memory leaks from past models, making my data inaccurate)

*(Note: Keep in mind that the resolution controls the total number of vertices. Total size of the vertex array = 32 * resolutions^2. Each patch has resolution^2 vertices)*

The Creation Time was measured using *gs_time()*

Memory Allocation was measured using the detailed Windows Task Manager. (I could of used the Diagnostics tool on Visual Studio 2017, but I preferred the task manager).

Resolution	Creation Time (seconds)	Memory Allocation (MB)
2	0.0038	32.9 MB
5	0.0240	38.4 MB
10	0.0823	33.4 MB
15	0.1815	36.2 MB
20	0.4107	40.4 MB
30	0.6504	37.5 MB
40	1.1360	45.3 MB
50	1.6214	47 MB

Analysis

From the data above, we can see that the creation time has a direct relationship with the resolution. As the resolution increases, so does the creation time. Also, Memory allocation increases as the resolution increases (most of the time but not always). Some guesses to why the memory allocation sometimes doesn't increase could be:

1) Initial cache setups for the application
2) Left over memory from the previous run

I don't have a perfect explanation for why this happens. (If you have any ideas, please let me know)

Conclusion

As usual, the optimal choice for resolution depends on how we are going to apply this project.

If we want to apply this project in a dynamic environment where our teapot changes (such as rotating or moving or transforming), we would want a teapot with a lower resolution, so it'll compute the transformations faster (since it will have less points to compute). An optimal choice for this would be something less than 15 (something that takes less than 0.2 seconds to build).

If we want to apply this project in a static environment, we won't have to worry about build time since we are building it only once. In this case, we can pick a higher resolution to have a smoother display of the static teapot. However, this will require some more memory. An optimal choice for this would be something between 25 - 50 (Resolution past 50 isn't too noticeable).