Bi-Cubic B-Spline Donut Editor with Bump Mapping

1 Requirements

Goal: Implement an interactive “donut editor” as in the demo video.

Set only one directional light with its setting as follows.

- direction: $(0, 0, 1)$ (in camera coordinate system)
- ambient intensity: $(.1, .1, .1)$
- diffusive intensity:$(1, 1, 1)$
- specular intensity: $(1, 1, 1)$

Use the gold material.

- ambient: $(0.24725, 0.1995, 0.0745)$
- diffusive: $(0.75164, 0.60648, 0.22648)$
- specular: $(0.628281, 0.555802, 0.366065)$
- shininess: 128.0×0.4

Create the bumpmap texture (size 1024×512) yourself using an image editor like Gimp.

2 Bi-Cubic B-spline Surface

2.1 Parametric Surface

A parametric surface is defined as

$$\mathbf{r}(u, v) = \begin{bmatrix} x(u, v) \\ y(u, v) \\ z(u, v) \end{bmatrix}.$$

where $0 \le u, v \le 1$, called **parameters**, are real numbers. In other words, $\mathbf{r}(u, v)$ “maps” (u, v) in the unit square $[0, 1] \times [0, 1]$ to a point in $\mathbb{R}^3$. (Figure 1) For any point $\mathbf{P} = \mathbf{r}(u_0, v_0)$ on the surface $\mathbf{r}(u, v)$, two **tangent vectors** are defined as

$$\mathbf{t}_u := \frac{\partial \mathbf{r}}{\partial u}(u_0, v_0) = \begin{bmatrix} \frac{\partial x}{\partial u}(u_0, v_0) \\ \frac{\partial y}{\partial u}(u_0, v_0) \\ \frac{\partial z}{\partial u}(u_0, v_0) \end{bmatrix} \text{ and } \mathbf{t}_v := \frac{\partial \mathbf{r}}{\partial v}(u_0, v_0) = \begin{bmatrix} \frac{\partial x}{\partial v}(u_0, v_0) \\ \frac{\partial y}{\partial v}(u_0, v_0) \\ \frac{\partial z}{\partial v}(u_0, v_0) \end{bmatrix}$$

both of which are tangent to the surface at $\mathbf{P}$ (Figure 1). Then, the **normal vector N** at $\mathbf{r}(u_0, v_0)$ is defined as the cross product of $\mathbf{t}_u$ and $\mathbf{t}_v$ as

$$\mathbf{N} := \mathbf{t}_u \times \mathbf{t}_v.$$

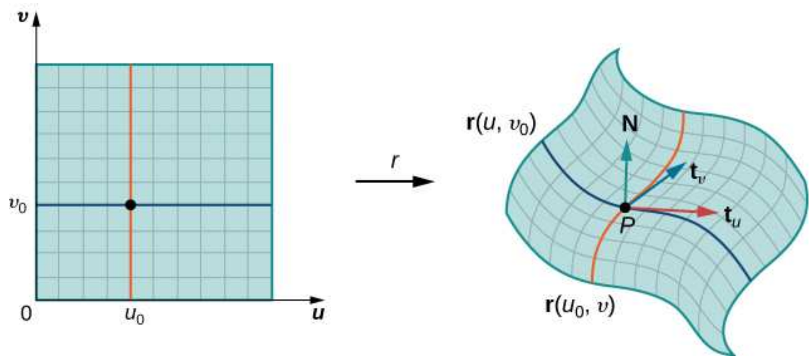


Figure 1: Parametric surface [3]

3 Bi-cubic B-Spline Surface

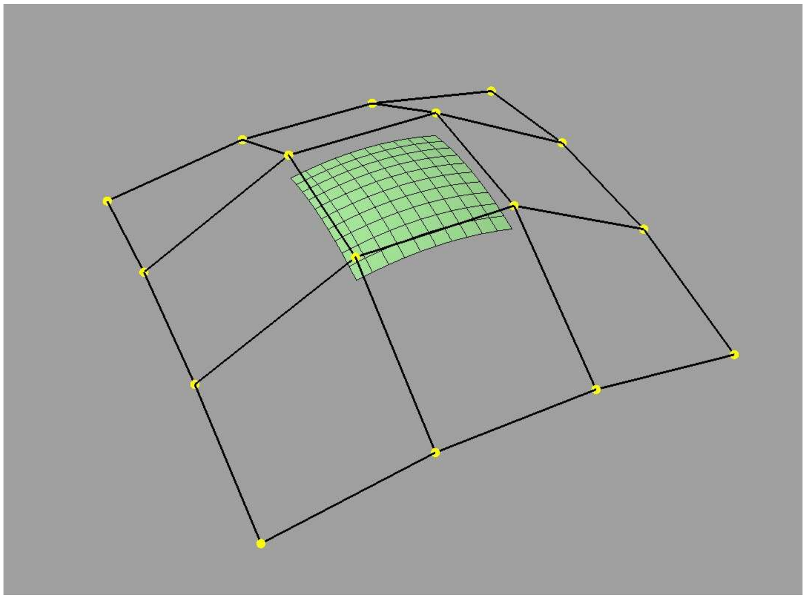


Figure 2: Bi-cubic B-spline patch [2].

A bi-cubic B-spline surface patch $\mathbf{p}(u, v)$ (Figure 2) is defined by 4×4 control points $\{\mathbf{c}_{ij}\}_{0\le i,j\le 3}$ as (Figure 3(b))

$$\mathbf{p}(u, v) = \sum_{i=0}^{3}\sum_{j=0}^{3} B_i^3(u)B_j^3(v)\mathbf{c}_{ij} \tag{1}$$

where (Replace t with u or v to derive $B_i^3(u)$ and $B_j^3(v)$ above.)

$$
\begin{aligned}
B_0^3(t) &= \frac{1}{6}(1-t)^3 \\
B_1^3(t) &= \frac{1}{6}(3t^3 - 6t^2 + 4) \\
B_2^3(t) &= \frac{1}{6}(-3t^3 + 3t^2 + 3t + 1) \\
B_3^3(t) &= \frac{1}{6}t^3.
\end{aligned}
$$

Two **tangent vectors** at the point $\mathbf{p}(u_0, v_0)$ on the patch is defined as

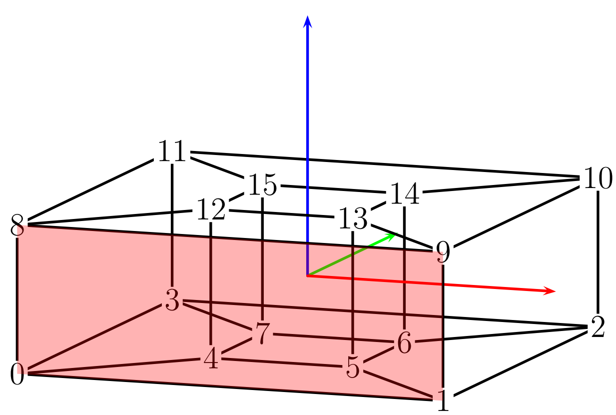
$$\mathbf{t}_u = \frac{\partial \mathbf{p}}{\partial u}(u_0, v_0) = \sum_{i=0}^{3}\sum_{j=0}^{3} \frac{dB_i^3}{du}(u_0) B_j^3(v_0)\mathbf{c}_{ij} \tag{2}$$

$$\mathbf{t}_v = \frac{\partial \mathbf{p}}{\partial v}(u_0, v_0) = \sum_{i=0}^{3}\sum_{j=0}^{3} B_i^3(u_0) \frac{dB_j^3}{dv}(v_0)\mathbf{c}_{ij} \tag{3}$$

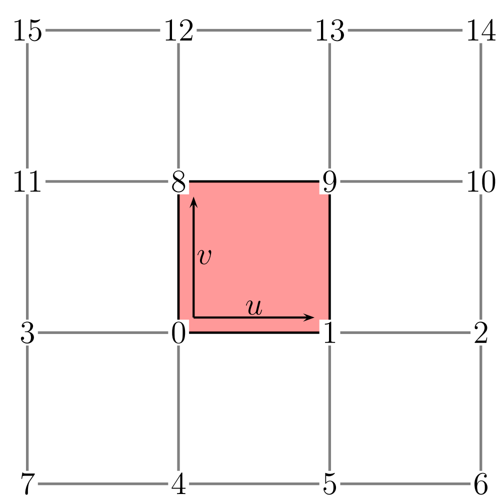
3.1 Bi-cubic B-spline Dobut

The donut is composed of 16 bi-cubic B-spline patches each associated with a quad face of the **control net** (Figure 3(a)). You can examine the control net by loading `donut.blend` file with Blender. (tested on version 4.0) The coordinates of the 16 points are

$$(\pm 2, \pm 2, \pm 2) \text{ and } (\pm 6, \pm 6, \pm 2).$$



(a) Control net of the donut.



(b) Control net of the bi-cubi B-spline patch associated with the pink quad in (a).

3.2 How to render the donut surface

- Assume that the control points are stored in `vec4 control_points[16]`. (You have to store them in a texture (might be tricky), uniform variables, or a uniform block because their position are updated interactively. Refer to `uniform-verts.html` example. (More explanation will be given in the class.)

- For each quad face, you have to extract the 4×4 **control net** as in Figure 3(b). While the positions of the 16 control points change interactively, the connectivity information (Figure 3(b)) do not change. Therefore, you can store the connectivity information in the a global variable of the vertex shader. Note that there are 16 quad faces. Let

  `int connectivity[16][4][4]`

  contains the connectivity information for each quad. (But you cannot define a multi-dimensional array in shaders. So you have to 'flatten' it in your implementation.)

- When rendering the donut, render 16 instances of one $N \times N$ grid mesh using

  `gl.drawElementsInstanced` function. (Refer to `uniform-verts.html` example.)

- In the vertex shader, compute the position of the patch using the formula (1). You can access the control point c_{ij} as

  `control_points[connectivity[gl_InstanceID][i][j]]`

  where `gl_InstanceID` is the instance id of the current vertex.

- Two tangent vectors should be computed in the vertex shader, too, using the formulas (2) and (3). (Try to figure out the formulas of $\frac{dB_i^3}{du}(u,v)$ and $\frac{dB_j^3}{dv}(u,v)$ yourself.)

- Transform the position (in `gl_Position`) and tangent vectors appropriately and output them. (tangent vectors as varying variables)

  - How should we transform the tangent vectors?

- In the fragment shader,

  1. Compute the normal as $\mathbf{N} = \mathbf{t}_u \times \mathbf{t}_v$.
  2. Fetch the bumpmap texture and compute $\frac{\partial T}{\partial u}(u_0, v_0)$ and $\frac{\partial T}{\partial v}(u_0, v_0)$ using a finite-divided-difference formula as in the project #2.
  3. Modify the normal vector using the formula (4). Don't forget to normalize the result before light computation.

4 Bump Mapping

Bump mapping was introduced by Blinn [1].

Let $\mathbf{t}_u := \frac{\partial \mathbf{r}}{\partial u}(u_0, v_0)$ and $\mathbf{t}_v := \frac{\partial \mathbf{r}}{\partial v}(u_0, v_0)$ be the tangent vectors of the parametric surface at $\mathbf{P} = \mathbf{r}(u_0, v_0)$ (stored in varying variables). Then, in the fragment shader, the normal vector is computed as

$$\mathbf{N} := \mathbf{t}_u \times \mathbf{t}_v$$

In addition, let $T(u_0, v_0)$ be the texture value of the bump texture at (u_0, v_0). Then, the normal vector at (u_0, v_0) should be perturbed as follows.

$$\underbrace{\mathbf{N}}_{\text{vector}} + \underbrace{\frac{\partial T}{\partial u}(u_0, v_0)}_{\text{scalar}} \underbrace{\frac{\mathbf{N} \times \mathbf{t}_v}{\|\mathbf{N} \times \mathbf{t}_v\|}}_{\text{normalized vector}} - \underbrace{\frac{\partial T}{\partial v}(u_0, v_0)}_{\text{scalar}} \underbrace{\frac{\mathbf{N} \times \mathbf{t}_u}{\|\mathbf{N} \times \mathbf{t}_u\|}.}_{\text{normalized vector}} \tag{4}$$

Note that $\frac{\partial T}{\partial u}(u_0, v_0)$ and $\frac{\partial T}{\partial v}(u_0, v_0)$ should be approximated using a finite difference rule, as in project #2. Also note that the above vector should be **normalized** to compute lighting/shading.

References

[1] James F. Blinn. Simulation of wrinkled surfaces. *SIGGRAPH Comput. Graph.*, 12(3):286–292, aug 1978. ISSN 0097-8930. doi: 10.1145/965139.507101. URL `https://doi.org/10.1145/965139.507101`.

[2] Pixar. Subdivision surfaces, 2023. URL `https://graphics.pixar.com/opensubdiv/docs/subdivision_`

[3] Gilbert Strang and Edwin Herman. *Calculus Volume 1*. OpenStax, 2020.